Cooperación entre dominios de restricciones y estrategias de cooperación en el contexto $CFLP$

Tesis doctoral

Sonia Estévez Martín

Departamento de Sistemas Informáticos y Computación

Universidad Complutense de Madrid

Directores: Jesús Correas Fernández y Fernando Sáenz Pérez

Tesis doctoral presentada por la doctoranda Sonia Estévez Martín en el *Departamento de Sistemas Informáticos y Computación* de la *Universidad Complutense de Madrid* para la obtención del título de doctora en Ingeniería Informática.

Terminada en Madrid en mayo del año 2015.

Título:

Cooperación entre dominios de restricciones y estrategias de cooperación en el contexto $CFLP$

Doctoranda:

Sonia Estévez Martín (s.estevez@fdi.ucm.es)

Departamento de Sistemas Informáticos y Computación, Universidad Complutense de Madrid

Facultad de Informática, 28040 Madrid, España

Directores:

Jesús Correas Fernández (jcorreas@fdi.ucm.es)

Fernando Sáenz Pérez (fernan@sip.ucm.es)

Esta tesis doctoral ha sido realizada dentro del grupo de investigación Programación Declarativa (grupo 910502 del catálogo de grupos reconocidos por la UCM) y como parte de los proyectos de investigación: SICOMORo-CM (S2013/ICE-3006 - Comunidad de Madrid), CAVI-ART (TIN2013-44742-C4-3-R Ministerio de Economía y Competitividad), N-GREENS Software-CM (S2013/ICE-2731 - Comunidad de Madrid), ESTuDIo (TIN2012-36812-C02-01 - Ministerio de Economía y Competitividad), VIVAC (TIN2012-38137-C02 - Ministerio de Economía y Competitividad), PROMETIDOS (S2009/TIC-1465 - Comunidad de Madrid), FAST-STAMP (TIN2008-06622-C03-01 - Ministerio de Ciencia e Innovación), DOVES (TIN2008-05624 - Ministerio de Educación y Ciencia), MERIT-FORMS (TIN2005-09207-C03-03 - Ministerio de Educación y Ciencia), ENVISAGE (FP7 610582 - Comisión Europea), Ayuda a grupos de investigación UCM (GR3/14-910502 - Universidad Complutense de Madrid) y PROMESAS (S-0505/TIC-0407 - Comunidad de Madrid).

Agradecimientos

Quiero dar las gracias a mis directores de tesis, los profesores Fernando Sáenz Pérez y Teresa Hortalá González, que se jubiló y fue sustituida por el profesor Jesús Correas Fernández;

a mis compañeros del Departamento de Sistemas Informáticos y Computación y en general de la Universidad Complutense de Madrid, por su amistad y apoyo;

al profesor Francisco López Fraguas por sus acertados comentarios;

y por último, pero no por ello menos importante, a esas personas cercanas a mi y cómo no, a mi familia.

Resumen

Esta tesis es una aportación a la cooperación entre dominios de restricciones tomando como marco la programación lógico funcional con restricciones (CFLP, Constraint Functional Logic Programming*). En particular se centra en $\mathcal{TOY}$, un lenguaje y sistema bajo este marco que ha sido desarrollado por el Grupo de Programación Declarativa de la Universidad Complutense de Madrid. $\mathcal{TOY}$ resuelve objetivos por medio de una estrategia de estrechamiento perezoso guiada por la demanda que se combina con la resolución de restricciones.*

La cooperación entre dominios se establece mediante los llamados dominios mediadores. Estos dominios proveen unas restricciones híbridas denominadas puentes que contienen variables de distintos dominios (por ejemplo: números reales y dominio finito de números enteros). Los puentes sirven de mecanismo de comunicación y, además, permiten proyectar información entre dominios. La cooperación entre el dominio de Herbrand y el resto se realiza a través de la igualdad estricta reificada.

La cooperación entre dominios se formaliza con un cálculo de resolución de objetivos que combina el estrechamiento perezoso con la resolución de restricciones. Este cálculo incluye reglas específicas de cooperación y propagación de restricciones entre resolutores. En particular, en esta tesis se estudian dos instancias concretas de cooperación:

1. *La cooperación entre el dominio de Herbrand $\mathcal{H}$, el dominio de los números reales $\mathcal{R}$ y el dominio finito de los números enteros $\mathcal{FD}$.*
2. *La cooperación entre $\mathcal{H}$, $\mathcal{FD}$ y el dominio de conjuntos finitos de números enteros $\mathcal{FS}$.*

El modelo formal que se presenta en esta tesis proporciona además un mecanismo de detección de fallo en los dominios $\mathcal{FD}$ y $\mathcal{FS}$. En esencia, este mecanismo trata de detectar el fallo antes de enviar las restricciones a los resolutores.

Otra aportación de esta tesis, resultado del estudio de la cooperación entre resolutores, ha sido la creación de una nueva restricción en el dominio $\mathcal{FS}$, que impone que todos los elementos de un conjunto sean menores que los elementos de otro conjunto dado. Esta restricción, ausente en las clásicas bibliotecas de restricciones de conjuntos finitos, como por ejemplo en las del sistema ECL^iPS^e, aumenta la capacidad expresiva del lenguaje al permitir definir de forma más compacta problemas complejos.

Con respecto a la semántica declarativa provista por el marco CFLP, se demuestra que el cálculo de resolución de objetivos definido para la cooperación es correcto y completo con ciertas limitaciones.

Como complemento al marco teórico se ha extendido el sistema $\mathcal{TOY}$ con nuevos resolutores de restricciones y se ha implementado la cooperación de dos instancias de cooperación entre resolutores disponibles en este sistema. Esta nueva implementación confirma experimentalmente la validez del enfoque propuesto en esta tesis.

En la cooperación entre $\mathcal{FD}$ y $\mathcal{R}$ se han utilizado las bibliotecas del sistema SICStus Prolog, mientras que en la cooperación entre $\mathcal{FD}$ y $\mathcal{FS}$ se ha desarrollado una interfaz con el sistema ECL^iPS^e para utilizar su biblioteca de restricciones sobre conjuntos.

Otro resultado de esta tesis es la comprobación mediante resultados experimentales del beneficio del uso de la cooperación en problemas que contienen restricciones de distintos dominios. También se ha comprobado experimentalmente que el uso de las proyecciones o bien mejora la eficiencia de los programas o bien no produce una sobrecarga significativa.

En resumen, esta tesis desarrolla un estudio de la cooperación de dominios de restricciones en el contexto $CFLP$ de una forma teórica (definiendo un modelo computacional para la cooperación entre los dominios de restricciones), práctica (desarrollando la implementación de las dos instancias de cooperación presentadas en esta tesis) y experimental (probando mediante experimentos que la cooperación es viable y puede mejorar el rendimiento).

Abstract

This thesis is a contribution to the cooperation of constraint domains using Constraint Functional Logic Programming ($CFLP$) as a framework. In particular it focuses on $\mathcal{TOY}$, a language and system developed in this framework by the Declarative Programming Group of the Complutense University of Madrid. $\mathcal{TOY}$ solves goals by means of a demand-driven lazy narrowing strategy combined with constraint solving.

The cooperation between domains is established by the so-called mediatorial domains. These domains provide hybrid constraints called bridges which contain variables of different domains (for example: real numbers and finite domains of integer numbers). Bridges are used as a communication mechanism and, additionally, can project information between domains. Cooperation between the Herbrand domain and others is done through reification of the strict equality.

The cooperation among domains is formalized with a goal solving calculus which combines lazy narrowing with constraint solving. This calculus includes specific rules for cooperation and constraint propagation between solvers. In particular, two specific instances of cooperation are developed in this thesis:

1. *The cooperation between the Herbrand domain $\mathcal{H}$, the domain of real numbers $\mathcal{R}$ and the finite domain of integers $\mathcal{FD}$.*
2. *The cooperation between $\mathcal{H}$, $\mathcal{FD}$ and the domain of finite sets of integers $\mathcal{FS}$.*

The formal model presented in this thesis also provides a mechanism for anticipating failure in the domains $\mathcal{FD}$ and $\mathcal{FS}$. Essentially, this mechanism tries to detect failure before sending constraints to solvers.

Another contribution of this thesis, resulting from studying cooperation, is the definition of a new constraint on the domain $\mathcal{FS}$ that imposes that all elements of a set are smaller than the elements of another set. This constraint, which is not available in conventional integer set constraints libraries, as for example in the libraries of the ECL^{i}PS^e system, increases the expressiveness of the language because it allows the definition of complex problems in a compact form.

Regarding the declarative semantic provided by the framework $CFLP$, the goal-solving calculus defined for the cooperation has been proved to be sound and complete with some limitations.

In addition to the theoretical framework, the $\mathcal{TOY}$ system has been extended with new constraint solvers, and two instances of cooperation have been implemented with the available solvers in this system. This new implementation experimentally confirms the validity of the approach proposed in this thesis.

In the cooperation between $\mathcal{FD}$ and $\mathcal{R}$, the system libraries of SICStus Prolog have been used, whereas on the cooperation between $\mathcal{FD}$ and $\mathcal{FS}$ an interface with the system ECL^{i}PS^e has been developed to use its set constraint library.

Another result of this thesis is a corroboration, via experimental results, of the benefits of using cooperation on problems involving constraints of different domains. Also, it has been experimentally verified that the use of projections either improves the efficiency of programs or do not produce a significant overhead.

In summary, this thesis studies the cooperation of constraint domains in the $CFLP$ framework in three different ways: theoretically (defining a computational model for the cooperation of constraint domains), practically (developing an implementation of two instances of cooperation proposed in this thesis) and experimentally (showing by means of experiments that cooperation is feasible and can improve performance).

Índice general

Índice de figuras

Índice de tablas

Lista de términos

$ACon_{\mathcal{D}}$ conjunto de restricciones atómicas del dominio $\mathcal{D}$. 43

$APCon_{\mathcal{D}}$ conjunto de restricciones primitivas atómicas del dominio $\mathcal{D}$. 43

$Con_{\mathcal{D}}$ conjunto de restricciones del dominio $\mathcal{D}$. 43

$Exp_{\Sigma}(\mathcal{B})$ conjunto de Σ-*expresiones* sobre $\mathcal{B}$. 13, 38

$Exp_{\Sigma}(\mathcal{B}^{\mathcal{D}})$ conjunto de Σ-*expresiones* sobre $\mathcal{B}$ del dominio $\mathcal{D}$ (abreviado $Exp_{\mathcal{D}}$). 42

$Exp_{\mathcal{D}}$ conjunto de Σ-*expresiones* del dominio $\mathcal{D}$ (abrevia $Exp_{\Sigma}(\mathcal{B}^{\mathcal{D}})$). 42

$GExp_{\Sigma}(\mathcal{B})$ conjunto de Σ-*espresiones* básicas sobre $\mathcal{B}$. 13, 38

$GPat_{\Sigma}(\mathcal{B})$ conjunto de Σ-*patrones* básicos sobre $\mathcal{B}$. 13, 38

$Pat_{\Sigma}(\mathcal{B}^{\mathcal{D}})$ conjunto de Σ-*patrones* sobre $\mathcal{B}$ del dominio $\mathcal{D}$ (abreviado $Pat_{\mathcal{D}}$). 42

$Pat_{\Sigma}(\mathcal{B})$ conjunto de Σ-*patrones* sobre $\mathcal{B}$. 13, 38

$Pat_{\mathcal{D}}$ conjunto de Σ-*patrones* del dominio $\mathcal{D}$ (abrevia $Pat_{\Sigma}(\mathcal{B}^{\mathcal{D}})$). 42

$Sol_{\mathcal{D}}(\pi)$ conjunto de soluciones de una restricción π. 44

$Sol_{\mathcal{P}}(G)$ soluciones del objetivo G con respecto al programa $\mathcal{P}$. 80

$Sub_{\Sigma}(\mathcal{B}^{\mathcal{D}})$ conjunto de Σ-*sustituciones* sobre $\mathcal{B}$ del dominio $\mathcal{D}$ (abreviado $Sub_{\mathcal{D}}$). 42

$Sub_{\Sigma}(\mathcal{B})$ conjunto de *sustituciones* sobre $\mathcal{B}$. 40

$Sub_{\mathcal{D}}$ conjunto de Σ-*sustituciones* del dominio $\mathcal{D}$ (abrevia $Sub_{\Sigma}(\mathcal{B}^{\mathcal{D}})$). 42

$Val_{\mathcal{D}}$ valoraciones del dominio $\mathcal{D}$. 43

$WTSol_{\mathcal{D}}(\pi)$ conjunto de soluciones bien tipadas de una restricción π. 44

$WTSol_{\mathcal{P}}(G)$ conjunto de soluciones bien tipadas de G con respecto a $\mathcal{P}$. 80

$\Gamma = \{X_1 :: \tau_1, \ldots, X_n :: \tau_n\}$ entorno de tipos, las variables X_i tienen tipos τ_i. 39

$\Sigma, \Gamma \vdash_{WT} e :: \tau$ juicio de tipos. 39

$\bot$ valor indefinido. 11

$\gg_P$ relación de producción. 73

$\mathcal{B}$ conjunto de valores básicos. 13, 38

$\mathcal{M} : \mu \in Sol_{\mathcal{P}}(G)$ $\mathcal{M}$ es un testigo de $\mu \in Sol_{\mathcal{P}}(G)$. 80

$\mathcal{TV}ar$ variables de tipo. 11, 39

$\mathcal{U}_\Sigma(\mathcal{B})$ universo de valores sobre $\mathcal{B}$. 38

$\mathcal{U}_\mathcal{D}$ universo de valores del dominio $\mathcal{D}$ (abrevia $\mathcal{U}_\Sigma(\mathcal{B})$). 41

$\mathcal{V}ar$ variables de datos. 11

$\sqsubseteq$ orden entre expresiones. 38

$\star$ un tipo de sustitución. 41, 45, 75

$\restriction$ un tipo de sustitución. 41, 75

$cvar_\mathcal{D}(\Pi)$ conjunto de variables críticas. 62

$dvar_\mathcal{D}(\Pi)$ conjunto de variables demandadas por Π. 61

$e \preceq e'$ la expresión e' es una instancia de la expresión e o e es más general que e'. 40

$odvar(G)$ conjunto de variables obviamente demandadas por el objetivo G. 74

$odvar_\mathcal{D}(\Pi)$ conjunto de variables obviamente demandadas de Π. 62

$p^\mathcal{D}$ interpretación del símbolo de función primitiva p en el dominio $\mathcal{D}$. 41

$pvar(P)$ conjunto de variables producidas del conjunto de producciones P. 73

$vdom(\sigma)$ conjunto de variables que componen el dominio de la sustitución σ. 40

$vran(\sigma)$ conjunto de variables que forman el rango de la sustitución σ. 40

Capítulo 1

Introducción

La esencia de la programación declarativa consiste en proporcionar un alto nivel de abstracción de tal forma que la especificación o declaración de un problema sea suficiente para resolverlo. La programación declarativa está representada por distintos paradigmas entre los que se encuentran el paradigma lógico LP (*Logic Programming*) [SS94, vEK76, Llo84] el funcional FP (*Functional Programming*) [Pey87, Hud89] y, como amalgama de estos dos paradigmas, los lenguajes lógico funcionales FLP (*Functional Logic Programming*) [HK96, AH10, Han13]. Enmarcado en este paradigma FLP nació $\mathcal{TOY}$ [ACE$^+$07, LS99b], el lenguaje utilizado en esta tesis.

Por otra parte la programación con restricciones CP (*Constraint Programming*) resuelve problemas combinatorios particularmente difíciles, especialmente en las áreas de planificación y programación de tareas. La incorporación de las restricciones a los lenguajes declarativos se inició con el desarrollo del esquema CLP (*Constraint Logic Programming*) [JL87, JM94, JMMS98]. Este esquema proporciona un marco general para la programación lógica con restricciones que hereda la semántica y el estilo declarativo de la programación lógica. La combinación de CLP con FLP ha dado lugar a varios esquemas $CFLP$ (*Constraint Functional Logic Programming*), desarrollados desde 1991 con el objetivo de combinar la expresividad de los paradigmas de programación con restricciones, funcional y lógico. Tanto el esquema CLP como los esquemas $CFLP$ se construyen sobre un dominio de restricciones $\mathcal{D}$ que proporciona valores específicos de datos, restricciones basadas en operaciones primitivas específicas y un resolutor de restricciones apropiado. Por lo tanto, hay diferentes instancias $CLP(\mathcal{D})$ del esquema de CLP para diversos dominios $\mathcal{D}$, y también para $CFLP$, cuyas instancias $CFLP(\mathcal{D})$ proporcionan un marco declarativo para cualquier dominio $\mathcal{D}$.

$\mathcal{TOY}$ es un lenguaje incluido en el paradigma lógico funcional con restricciones ($CFLP$) [LF92, AGL94] que resuelve objetivos por medio de una estrategia de estrechamiento perezoso guiada por la demanda [LLR93, LRV04] combinada con la resolución de restricciones [LRV07]. $\mathcal{TOY}$ dispone de una serie de dominios de restricciones puros: $\mathcal{H}$, el dominio de Herbrand con restricciones de igualdad y desigualdad sintáctica; $\mathcal{FD}$, el dominio de los enteros con restricciones de dominio finito; $\mathcal{R}$, el dominio de los números reales; y $\mathcal{FS}$, el dominio de conjuntos finitos de enteros. Sin embargo, las aplicaciones prácticas a menudo requieren la utilización de más de un dominio "puro" y esto implica que las codificaciones de los problemas en un único dominio tengan que adaptarse artificialmente para ajustarse a dicho dominio.

Es decir, ciertos problemas no se resuelven naturalmente en un único dominio o simplemente se desean formular utilizando restricciones de distintos dominios de restricciones. Este es el problema que resuelve esta tesis en el lenguaje y sistema $\mathcal{TOY}$.

Para poder utilizar varios dominios de restricciones en un mismo programa se ha utilizado la idea de dominio de restricciones "híbrido", construido como una combinación de dominios simples "puros" y diseñado para soportar la cooperación entre sus componentes. De esta forma se ha abordado la cooperación entre dominios de restricciones desde el punto de vista teórico, desarrollando un modelo computacional para la cooperación, y desde el punto de vista práctico, desarrollando un prototipo.

El modelo computacional que se desarrolla en esta tesis para la cooperación entre dominios de restricciones en el contexto $CFLP$ está basado en el esquema y el cálculo de resolución de objetivos propuesto en [LRV04, LRV07]. Este modelo computacional ha sido adaptado y ampliado con nuevos mecanismos para modelar la cooperación entre dominios. En particular, se utilizan unas restricciones especiales llamadas *puentes*, que se encargan de comunicar diferentes dominios. Estas restricciones se utilizan para *proyectar* restricciones de un dominio a otro. Es decir, si una restricción pertenece a un dominio $\mathcal{D}_1$ y tiene todas sus variables relacionadas a través de puentes con otro dominio $\mathcal{D}_2$, entonces dependiendo de la semántica de la restricción se puede definir una nueva restricción en $\mathcal{D}_2$ equivalente a la primera. Esto es, se proyecta en este dominio toda la información posible de la restricción original. Esta idea de proyectar restricciones es similar al trabajo de Petra Hofstedt [Hof00b, Hof01, FHR05], y se adaptó al esquema $CFLP$ añadiendo restricciones puente como técnica novedosa. De esta forma, se consigue que las proyecciones sean más flexibles y compatibles con la disciplina de tipos, pues $\mathcal{TOY}$ es un lenguaje fuertemente tipado que utiliza un inferidor de tipos basado en el sistema de tipos polimórficos de Damas-Milner [DM82].

Para formalizar este modelo computacional de cooperación de dominios de restricciones se construye lo que se denomina un *dominio de coordinación* ($\mathcal{C}$) como una suma amalgamada de dominios ($\oplus$). $\mathcal{C}$ es un dominio híbrido construido como la combinación de varios dominios puros $\mathcal{D}_i$ más un dominio especial para la comunicación entre los dominios puros llamado *dominio mediador* $\mathcal{M}$:

$$\mathcal{C} = \mathcal{M} \oplus \mathcal{D}_1 \oplus \cdots \oplus \mathcal{D}_n$$

Sobre el dominio de coordinación $\mathcal{C}$ se establece un cálculo de resolución de objetivos $CCLNC(\mathcal{C})$ (Cooperative Constraint Lazy Narrowing Calculus over $\mathcal{C}$) que se demuestra que es correcto y completo (con algunas limitaciones) con respecto a la instancia $CFLP(\mathcal{C})$ del esquema genérico $CFLP$ [LRV07]. El cálculo $CCLNC(\mathcal{C})$ contiene reglas para crear puentes, invocar resolutores de restricciones y proyectar restricciones, así como el uso del estrechamiento perezoso (combinación de evaluación perezosa y unificación) que evalúa llamadas a funciones solo en la medida exigida por la resolución de las restricciones que demanda el objetivo a evaluar.

En esta tesis se tratan dos instancias concretas del dominio de coordinación: $\mathcal{C}_{\mathcal{FD}.\mathcal{R}} = \mathcal{M} \oplus \mathcal{H} \oplus \mathcal{FD} \oplus \mathcal{R}$ y $\mathcal{C}_{\mathcal{FD}.\mathcal{FS}} = \mathcal{M} \oplus \mathcal{H} \oplus \mathcal{FD} \oplus \mathcal{FS}$. Para cada uno de estos casos se ha realizado una implementación, disponibles en [TOY12] y `http://gpd.sip.ucm.es/sonia/`

`Prototype.html`, respectivamente. La cooperación entre $\mathcal{FD}$ y $\mathcal{R}$ se ha implementado utilizando los resolutores de SICStus Prolog (versión 3.12) [SIC11], pues este es el sistema sobre el que está desarrollado $\mathcal{TOY}$. Este prototipo extiende la anterior implementación de $\mathcal{TOY}$ con un nuevo almacén para los puentes, incluyendo mecanismos para establecer puentes y proyecciones así como otras operaciones que infieren restricciones de igualdad y desigualdad utilizando diversos almacenes de restricciones. Todo ello de acuerdo con el modelo computacional $CCLNC(\mathcal{C}_{\mathcal{FD}.\mathcal{R}})$.

Posteriormente se decidió desarrollar la cooperación entre los dominios $\mathcal{FD}$ y $\mathcal{FS}$, y aunque SICStus Prolog contiene una biblioteca de operaciones sobre conjuntos ordenados representados como listas, no es propiamente un resolutor de restricciones. Después de una valoración de los sistemas CLP con restricciones de conjuntos, se decidió incorporar los resolutores de los dominios $\mathcal{FD}$ y $\mathcal{FS}$ del sistema ECL^{i}PS^e [ECL]. Por lo tanto, esta implementación necesitaba comunicar dos sistemas Prolog distintos. La comunicación de los sistemas se realizó utilizando los mecanismos de entrada/salida estándar entre procesos. Se han desarrollado dos modos de ejecución o estrategias (*interactivo* y *batch*) que pueden ser aplicables dependiendo de las características del programa y del objetivo que va a ser resuelto.

Como ya se ha comentado, el proceso de estrechamiento de objetivos de $\mathcal{TOY}$ se combina con la resolución de restricciones. Es decir, el objetivo se va tratando por estrechamiento y cuando se obtiene una restricción primitiva, esta se envía al resolutor. En el modo interactivo cada vez que el proceso de estrechamiento obtiene una restricción, la envía a ECL^{i}PS^e y espera que ECL^{i}PS^e devuelva el estado del sistema en ese momento para continuar con el cómputo. Este proceso no es eficiente para problemas complejos debido al coste de la comunicación entre procesos. Una mejora es enviar un conjunto de restricciones en bloque para evitar la comunicación entre procesos en cada restricción. Este funcionamiento solo es posible en determinados programas $\mathcal{TOY}$, pues hay ciertas características de $\mathcal{TOY}$ como el *sharing* o compartición de variables que no permiten este modo. Por lo tanto el modo interactivo se puede llevar a cabo enviando las restricciones de una en una o en bloque.

Además del modo interactivo se ha desarrollado el modo batch, que crea un programa ECL^{i}PS^e que recoge todas las restricciones generadas por la evaluación del objetivo $\mathcal{TOY}$. Cuando termina la evaluación del objetivo $\mathcal{TOY}$ se ejecuta el programa ECL^{i}PS^e Prolog que contiene todas las restricciones computadas por el objetivo $\mathcal{TOY}$. El modo batch es más rápido pero sin embargo está restringido a programas del estilo CP y no cubre todas las características de $\mathcal{TOY}$, como ya se ha indicado.

En la siguiente sección se motiva mediante ejemplos la necesidad y beneficios de utilizar la cooperación de los resolutores sin tener en cuenta las estrategias o modos de ejecución.

1.1 Ejemplos motivadores

En esta sección se muestran mediante ejemplos los distintos mecanismos de cooperación, es decir, cómo se establecen puentes entre distintos dominios y cómo se proyectan las restricciones de un dominio a otro, así como los beneficios derivados de la cooperación propuesta en esta tesis.

Ejemplo motivador 1

El primer ejemplo consiste en determinar los puntos discretos que corresponden a la intersección de dos regiones, una definida con puntos discretos como una cuadrícula y la otra definida como un triángulo continuo. De forma gráfica supongamos que tenemos distintas configuraciones de la cuadrícula o rejilla y del triángulo, como se muestra en la figura 1.1.

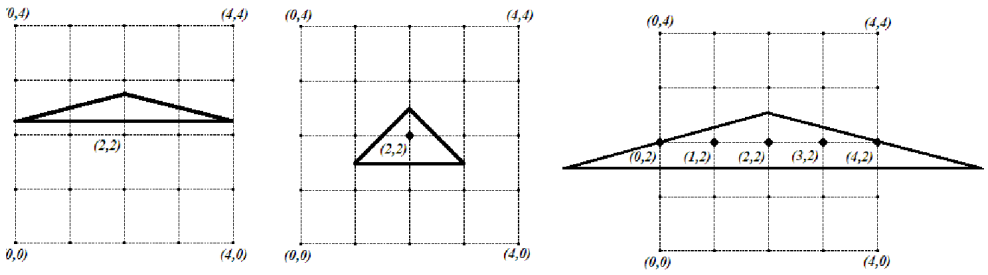


Figura 1.1: Ejemplo 1 de cooperación entre los dominios $\mathcal{FD}$ y $\mathcal{R}$

La cuadrícula se define mediante una restricción del dominio $\mathcal{FD}$ que asigna a las coordenadas `X` e `Y` valores entre `0` y `N` en un programa $\mathcal{TOY}$, mediante la restricción: `domain [X,Y] 0 N`, donde `N` representa el tamaño de la rejilla y debe ser un número entero mayor que 0. Se puede observar que la cuadrícula no se puede definir en el dominio $\mathcal{R}$.

Los triángulos se definen mediante tres inecuaciones que son restricciones de $\mathcal{R}$. Por ejemplo, en $\mathcal{TOY}$ el segundo triángulo se define mediante las inecuaciones `RY >= N/2-0.5`, `RY-RX <= 0.5` y `RY+RX <= N+0.5` donde `N` es el tamaño de la rejilla. Estos triángulos no se pueden definir en $\mathcal{FD}$ pues las variables toman valores reales. Para calcular las soluciones se comprueban los valores de la cuadrícula que cumplen las tres inecuaciones dadas como restricciones. Este proceso de comprobación se hace de forma sistemática con todos los puntos de la cuadrícula.

Para modelar este problema utilizando cooperación se necesita que las restricciones de los dominios $\mathcal{FD}$ y $\mathcal{R}$ trabajen sobre valores equivalentes. Para ello, se utilizan restricciones puente de la forma `X #== RX` que establecen un valor equivalente para variables de distinto tipo (`X::int` y `RX::real`). Este mecanismo nos permite referirnos a los valores equivalentes desde ambos dominios de restricciones. Aparte de la comunicación, la cooperación entre los resolutores es crucial para la eficiencia del cómputo. Supongamos que se define una cuadrícula de tamaño `N` enorme y un triángulo pequeño centrado en la cuadrícula. Para calcular las soluciones se recorre la cuadrícula de forma sistemática y en este caso se recorrerían muchos puntos antes de llegar al triángulo. Sin embargo, este proceso se puede mejorar utilizando *proyecciones* entre dominios. De forma transparente al programa, las proyecciones envían la información que contiene una restricción a otro dominio si es posible. En nuestro caso, las inecuaciones `RY >= N/2-0.5`, `RY-RX <= 0.5` y `RY+RX <= N+0.5` definidas en $\mathcal{R}$ se proyectan al dominio $\mathcal{FD}$ creando las siguientes nuevas restricciones: `Y #>=` $\lceil$`N/2-0.5`$\rceil$, `Y#-X #<=` $\lfloor$`0.5`$\rfloor$ y `Y#+X #<=` $\lfloor$`N+0.5`$\rfloor$. En el dominio $\mathcal{FD}$ los operadores tienen el prefijo `#` (excepto la igualdad y desigualdad) y todos los valores deben ser enteros, para ello se usa $\lceil$`-`$\rceil$ (respectivamente $\lfloor$`-`$\rfloor$) que calcula el menor entero que es cota superior (respectivamente el mayor

entero cota inferior).

Para proyectar las restricciones se aplican unas reglas que están definidas en el cálculo $CCLNC(\mathcal{C}_{\mathcal{FD}.\mathcal{R}})$ que se mostrará en la sección 5.2. Por medio de las proyecciones, el resolutor $\mathcal{FD}$ poda drásticamente los dominios de `X` e `Y`. Esto produce que el número de valores a comprobar sea mucho menor y el proceso de búsqueda sea más eficiente. De hecho, para los dos primeros casos mostrados en la figura 1.1, el mecanismo de proyección hace que se pase de $\mathcal{O}(\mathtt{N}^2)$ pasos necesarios para comprobar si todos los puntos son soluciones a $\mathcal{O}(1)$ pasos necesarios cuando los dominios de `X` e `Y` están podados. En el tercer caso se pasa de una ejecución de $\mathcal{O}(\mathtt{N}^2)$ a un ejecución de $\mathcal{O}(\mathtt{N})$ que encuentra las `N+1` soluciones en el dominio podado por las proyecciones.

Este ejemplo que se acaba de mostrar expone cómo el resolutor del dominio $\mathcal{FD}$ se beneficia de la cooperación con el resolutor del dominio $\mathcal{R}$. Ahora se verá un ejemplo en el cual el resolutor del dominio $\mathcal{R}$ se beneficia de la cooperación con el resolutor del dominio $\mathcal{FD}$.

Ejemplo motivador 2

Se desea saber cuáles son los puntos del plano que corresponden a la intersección de un segmento diagonal discreto ($x = y$) y una parábola definida por la ecuación $y = (x-2)^2$, como se muestra en la figura 1.2.

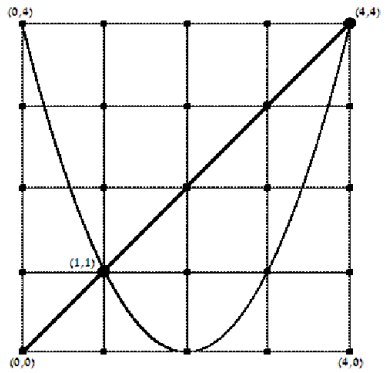


Figura 1.2: Ejemplo 2 de cooperación entre los dominios $\mathcal{FD}$ y $\mathcal{R}$

En general los resolutores de números reales no son capaces de resolver restricciones no lineales, posponiendo su evaluación hasta que tengan suficiente información para resolver estas restricciones (es decir, hasta que las restricciones se conviertan en lineales al asignar valores). En $\mathcal{TOY}$ el segmento diagonal discreto de la figura se define mediante las restricciones de $\mathcal{FD}$: `domain [X,Y] 0 4` y `X == Y`. Nótese que el segmento diagonal discreto no se puede definir en $\mathcal{R}$. La parábola se define mediante la restricción de $\mathcal{R}$: `RY == (RX-2)*(RX-2)`. Al igual que los triángulos, la parábola no se puede definir en $\mathcal{FD}$. Las variables de ambos dominios están comunicadas por puentes de la forma `X #== RX` e `Y #== RY`, donde el operando izquierdo es entero y el derecho es real.

En este ejemplo el resolutor del dominio $\mathcal{R}$ suspende la restricción no lineal `RY == (RX-2)*(RX-2)`, mientras que el resolutor del dominio $\mathcal{FD}$ procesa la restricción `X == Y`

que sustituye la variable `Y` por `X` en todo el objetivo. Esta igualdad en el dominio $\mathcal{FD}$ se comunica al dominio $\mathcal{R}$ y se procede a la sustitución de la variable `RY` por `RX`, aunque la restricción no lineal no se puede resolver todavía. Pero como existe un mecanismo de comunicación entre las variables de ambos dominios, cuando se etiquete (asignen) valores a la variable `X` de $\mathcal{FD}$, sus valores se propagarán mediante el puente `X #== RX` a la variable `RX` del dominio $\mathcal{R}$, despertando la restricción no lineal y comprobando si esta se satisface. De esta forma se puede saber si una restricción no lineal se puede satisfacer para ciertos valores discretos.

En el ejemplo que se acaba de mostrar el resolutor del dominio $\mathcal{R}$ es el que se beneficia de la cooperación con $\mathcal{FD}$, pues utilizando únicamente este resolutor no se pueden calcular los puntos correspondientes a la intersección.

Ejemplo motivador 3

La cooperación natural entre los dominios $\mathcal{FD}$ y $\mathcal{FS}$ viene dada por el cardinal de un conjunto, y son muchos los problemas que utilizan este mecanismo de comunicación. Sin embargo, hay información que se encuentra disponible en el dominio $\mathcal{FS}$ que se puede proyectar al dominio $\mathcal{FD}$ y delegar a los propagadores del dominio $\mathcal{FD}$ la poda del árbol de búsqueda de soluciones, pues los propagadores del dominio $\mathcal{FD}$ son más eficientes. De hecho, como se mostrará en el capítulo 7, se ha comprobado que el ejemplo que se esboza a continuación y se detalla en la sección 7.3 anticipa el fallo de forma más eficiente cuando no existen soluciones.

En este ejemplo se tiene que planificar la realización de una serie de tareas. Cada tarea tiene una duración determinada y se lleva a cabo en días completos. Se pueden establecer precedencias entre las tareas y algunas tareas pueden necesitar de ciertos recursos. Las tareas se representan como tX^{Y}_{mZ}, donde X representa el identificador de una determinada tarea, Y es el tiempo que necesita la tarea en ser completada (duración), y Z es el identificador de la máquina m (recurso) que necesita la tarea para realizarse. Los recursos se utilizan de forma exclusiva. En la figura 1.3 se muestra un grafo de precedencia concreto. Por ejemplo, la tarea $t5$ utiliza el recurso $m1$ y tiene una duración de 5 días, que pueden no ser consecutivos. Por tanto, mientras se está realizando la tarea $t5$ no se pueden realizar las tareas $t4$ ni $t6$, pues las tres tareas utilizan el mismo recurso $m1$. Por otra parte, la tarea $t5$ solo puede realizarse una vez terminada la tarea $t1$.

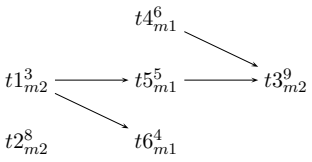


Figura 1.3: Grafo de precedencias para un problema concreto de planificación

Este problema es sencillo de codificar en el dominio $\mathcal{FD}$ si las tareas se realizaran en días consecutivos, representando el inicio de cada tarea con una variable $\mathcal{FD}$ y, como se sabe la

duración, entonces se sabe el día que termina cada tarea. Pero al permitir interrupciones, la representación de las tareas y la imposición de restricciones se complica bastante si se utiliza únicamente el dominio $\mathcal{FD}$.

Un modelado totalmente distinto es representar las tareas como conjuntos donde los elementos de cada conjunto (números enteros) representan los días que se ejecuta cada tarea. En el ejemplo de la figura 1.3, las 6 tareas se representan en el dominio de conjuntos finitos de enteros de $\mathcal{TOY}$ con 6 variables de conjuntos `(T1,...,T6)` cuyos elementos son valores enteros entre 1 y `N` (número de días máximo en los cuales se deben realizar todas las tareas). Las tareas que necesitan un mismo recurso se restringen a ser conjuntos disjuntos dos a dos, y si una tarea precede a otra entonces todos los elementos del conjunto que representa a la tarea que precede deben ser menores que todos los elementos de la tarea precedida. Por ejemplo, todos los elementos del conjunto `T1` deben ser menores que cualquier elemento del conjunto `T5`.

Supongamos que el número de días disponibles para realizar todas las tareas es 15. En este caso es fácil comprobar que no se puede realizar la planificación por las restricciones de precedencia de las tareas 1, 5 y 3, pues la suma de las duraciones de estas tareas (17) sobrepasa el número de días disponibles (15). El resolutor de conjuntos finitos $\mathcal{FS}$ no es capaz de inferir que no existe solución y prueba con todas las posibles combinaciones. Sin embargo, se puede evitar la exploración de todas las combinaciones si se proyecta la información de precedencia de estas tareas al dominio $\mathcal{FD}$ de la siguiente forma:

$$\texttt{1} \leq \texttt{min(T1)},\ \texttt{min(T1)+|T1|} \leq \texttt{max(T1)},\ \texttt{max(T1)} < \texttt{min(T5)},$$
$$\texttt{min(T5)+|T5|} \leq \texttt{max(T5)},\ \texttt{max(T5)} < \texttt{min(T3)},\ \texttt{min(T3)+|T3|} \leq \texttt{15}$$

Los valores correspondientes al mínimo y máximo de los conjuntos no se saben a priori, pero sí se conoce el cardinal porque es la duración de cada tarea (`|T|` representa la cardinalidad del conjunto `T`). Se sustituyen los cardinales por sus valores obteniendo:

$$\texttt{1} \leq \texttt{min(T1)},\ \texttt{min(T1)+3} \leq \texttt{max(T1)},\ \texttt{max(T1)} < \texttt{min(T5)},$$
$$\texttt{min(T5)+5} \leq \texttt{max(T5)},\ \texttt{max(T5)} < \texttt{min(T3)},\ \texttt{min(T3)+9} \leq \texttt{15}$$

Como las tareas están modeladas como conjuntos de enteros positivos, los valores correspondientes a los mínimos y máximos de los conjuntos son también números enteros positivos. De las dos inecuaciones $\texttt{1} \leq \texttt{min(T1)}$ y $\texttt{min(T1)+3} \leq \texttt{max(T1)}$ se deduce que $\texttt{4} \leq \texttt{max(T1)}$. Además, como `max(T1) < min(T5)` se obtiene `4 < min(T5)`. Por el mismo razonamiento, si `4 < min(T5)` y $\texttt{min(T5)+5} \leq \texttt{max(T5)}$ entonces $\texttt{10} \leq \texttt{max(T5)}$. Como $\texttt{max(T5)} < \texttt{min(T3)}$ entonces el valor `min(T3)` es al menos 11 que no satisface la inecuación $\texttt{min(T3)+9} \leq \texttt{15}$ pues `11 + 9` no es menor que `15`. Por lo tanto, el resolutor $\mathcal{FD}$ anticipa el fallo al detectar la inconsistencia.

En este ejemplo el resolutor de conjuntos finitos $\mathcal{FS}$ no es capaz de detectar que las tres tareas no se pueden ejecutar en 15 días y necesita explorar todas las posibles soluciones. Sin embargo, trasladando la información a $\mathcal{FD}$ se detecta rápidamente la inconsistencia pues los propagadores del resolutor del dominio $\mathcal{FD}$ son muy eficientes en este tipo de restricciones. Así el resolutor de conjuntos finitos $\mathcal{FS}$ se beneficia de la cooperación con el resolutor del

dominio $\mathcal{FD}$. Por lo tanto, la cooperación nos proporciona herramientas adicionales para el modelado de problemas y nos permite mejorar la eficiencia en algunos casos, como se muestra a lo largo de la tesis.

1.2 Estructura de la tesis y contribuciones

Una vez realizada una breve introducción, en esta sección se muestra tanto la estructura que sigue la tesis como las contribuciones de la doctoranda a la misma.

Estructura de la tesis

En este primer capítulo se presenta en la siguiente sección una introducción al lenguaje $\mathcal{TOY}$ y sus principales características. En el capítulo 2, se muestra el estado del arte de la cooperación de resolutores de restricciones. En primer lugar se describen los trabajos desarrollados sobre la cooperación en el ámbito de la programación lógica con restricciones, y a continuación los principales lenguajes y sistemas de programación lógico funcional y su extensión a la programación con restricciones. Por último se describen las últimas contribuciones al sistema sobre el que se ha desarrollado esta tesis: $\mathcal{TOY}$.

En el capítulo 3 se establecen los fundamentos teóricos necesarios para establecer la cooperación entre dominios de restricciones que se desarrolla en el resto de la tesis. En particular se introducen las definiciones de dominio de restricciones y sus resolutores asociados. Además, en este capítulo se describen los dominios que intervienen en las cooperaciones que se tratan en esta tesis ($\mathcal{H}$, $\mathcal{R}$, $\mathcal{FD}$ y $\mathcal{FS}$), así como sus correspondientes resultados semánticos.

Una vez definidos los dominios de restricciones y resolutores, se define un dominio de coordinación genérico ($\mathcal{C} = \mathcal{M} \oplus \mathcal{D}_1 \oplus \cdots \oplus \mathcal{D}_n$) en el capítulo 4, junto a la presentación del cálculo $CCLNC(\mathcal{C})$ (*Cooperative Constraint Lazy Narrowing Calculus over* $\mathcal{C}$) para este dominio de coordinación. Como particularizaciones de este dominio de coordinación genérico se tratan dos instancias: $\mathcal{C}_{\mathcal{FD}.\mathcal{R}}$ para la cooperación entre los dominios $\mathcal{FD}$ de enteros y $\mathcal{R}$ de reales, expuesta en el capítulo 5, y $\mathcal{C}_{\mathcal{FD}.\mathcal{FS}}$ para la cooperación entre el dominio $\mathcal{FD}$ y el dominio de conjuntos finitos de enteros $\mathcal{FS}$, expuesta en el capítulo 6 y ampliada en el capítulo 7. En cada una de estas instancias se define con detalle cada dominio de coordinación incluyendo los correspondientes dominios mediadores para poder comunicar los dominios puros, así como las reglas específicas del cálculo $CCLNC$ para cada dominio de coordinación. Se demuestra que estas instancias son correctas y completas con algunas limitaciones con respecto a la semántica declarativa dada por el esquema $CFLP$. Además, se muestra el desarrollo de dos prototipos que utilizan sistemas de restricciones distintos (de SICStus Prolog y ECL^{i}PS^e), así como las correspondientes comparativas respecto a los sistemas relacionados más próximos. Por último, en el capítulo 8 se presenta la sección de conclusiones.

Contribuciones

Como se ha mencionado anteriormente, el capítulo 3 contiene las bases teóricas y las definiciones de dominios de restricciones y resolutores. Este capítulo es fundamental para la comprensión del resto de la tesis pero no se puede considerar íntegramente como contribución de la doctoranda. Las bases teóricas que se incluyen en este capítulo forman parte de la tesis

[dVV08] publicada previamente, excepto los siguientes elementos del capítulo: el lema 2 del apartado 3.1.7, el teorema 1 del apartado 3.3 y el apartado 3.4.

Los trabajos que corresponden a las bases teóricas establecidas en los capítulos 3 y 4 más la instancia concreta de cooperación de los dominios $\mathcal{H}$, $\mathcal{FD}$ y $\mathcal{R}$ que se presenta en el capítulo 5 se detallan a continuación: en el trabajo [EFH$^+$07a] se presentó un cálculo de resolución de objetivos con cooperación entre resolutores para diferentes dominios arbitrarios. La aportación de la doctoranda a este artículo es, junto con los demás coautores, la elaboración del cálculo y las reglas de cooperación. Por otra parte, la primera propuesta de cooperación centrándose en la cooperación entre los dominios $\mathcal{H}$, $\mathcal{FD}$ y $\mathcal{R}$ se realizó en el trabajo [EFH$^+$07b], en el cual la aportación principal de la doctoranda fue su implementación en el sistema $\mathcal{TOY}$. Un año después, en el trabajo [EFH$^+$08] se presentó una primera versión de la descripción formal y una comparación con el sistema más próximo, Meta-S [FHM03a, FHM03b, FHR05]. En este trabajo la doctoranda colaboró en las demostraciones de los resultados teóricos. Adicionalmente se realizaron una serie de trabajos que detallan la implementación de la cooperación entre resolutores en $\mathcal{TOY}$ [EFS07, EFS08]. Finalmente, en el extenso trabajo [EFH$^+$09] se dio una completa descripción de la cooperación entre los dominios $\mathcal{H}$, $\mathcal{FD}$ y $\mathcal{R}$, así como la formalización de los dominios de restricciones, resolutores y del cálculo de resolución de objetivos. En este trabajo la doctoranda colaboró en el diseño de las reglas de cooperación entre dominios y en la sección que detalla la implementación. Con respecto a los trabajos anteriormente mencionados, la implementación de la cooperación en el sistema $\mathcal{TOY}$ fue completamente desarrollada por la autora de esta tesis y está disponible en `http://gpd.sip.ucm.es/sonia/Prototype.html`. Además, como contribución aún no publicada, en el capítulo 3 se plantea inferir el fallo antes que las restricciones sean enviadas al resolutor del dominio $\mathcal{FD}$.

El capítulo 6 trata la cooperación entre los dominios $\mathcal{H}$, $\mathcal{FD}$ y $\mathcal{FS}$. En esta nueva cooperación la publicación fundamental es el artículo [ECS12]. En este trabajo, además de presentar las bases teóricas para esta nueva cooperación, también se presentó la correspondiente adaptación del cálculo, así como los resultados teóricos de corrección y completitud limitada. Una primera aproximación a esta nueva cooperación fue esbozada en los trabajos [EFS09a, EFS09b]. El trabajo [ECS12] desarrolla esa idea inicial con múltiples mejoras y una nueva implementación. En este trabajo, la participación de la doctoranda fue fundamental y cubrió todos los aspectos teóricos y prácticos. En la parte teórica, diseñó las reglas de transformación de restricciones del dominio de conjuntos finitos y de comunicación entre dominios, así como la adaptación del cálculo. Respecto a la parte práctica, continuó extendiendo el sistema $\mathcal{TOY}$ para utilizar los resolutores de bajo nivel disponibles en el sistema CLP ECL^{i}PS^e, incluyendo el mecanismo de comunicación de bajo nivel entre $\mathcal{TOY}$ y ECL^{i}PS^e y el desarrollo del servidor de restricciones en este último sistema. Las pruebas experimentales fueron íntegramente realizadas por la doctoranda.

En el momento de la entrega de esta tesis se está finalizando un artículo de revista, que no contribuye como publicación a la tesis, pero que será enviado próximamente para su evaluación y que incluye la extensión descrita en el capítulo 7. Este artículo estudia una extensión de la instancia de cooperación entre los dominios $\mathcal{H}$, $\mathcal{FD}$ y $\mathcal{FS}$, presentada en [ECS12], con notables diferencias y mejoras.

Las demostraciones A.2, A.3, A.4, A.6, A.8 y A.9 han sido íntegramente realizadas por la doctoranda siguiendo la metodología de las demostraciones publicadas en los artículos en los que se ha trabajado.

A continuación se enumeran todas las publicaciones en orden de relevancia en las cuales ha intervenido la doctoranda y tienen relación con la tesis.

Publicaciones asociadas a la tesis (por orden de relevancia)

- Sonia Estévez-Martín, M. Teresa Hortalá-González, Mario Rodríguez-Artalejo, Rafael del Vado-Vírseda, Fernando Sáenz-Pérez y Antonio J. Fernández. On the Cooperation of the Constraint Domains $\mathcal{H}$, $\mathcal{R}$ and $\mathcal{FD}$ in $CFLP$. *TPLP*, 9:415–527 (113 pág.), 2009.

  Clasificación. Índice de impacto: 1,467. Base: 2009 JCR Science ED. Posición: 27/92, PRIMER TERCIO. Área: Computer Science, Theory & Methods.

- Sonia Estévez Martín, Jesús Correas Fernández y Fernando Sáenz-Pérez. Extending the $\mathcal{TOY}$ System with the ECL^{i}PS^e Solver over Sets of Integers. *FLOPS'12*, volume 7294 of *LNCS*, páginas 120–135, (16 pág.). Springer, 2012.

  Clasificación. CORE (2010): A. Citeseer: posición 468/1221 (top 38 %), impacto 0.69. posición media: top 36 %. Microsoft Academic: 53/167. Área: Programming Languages.

- Sonia Estévez-Martín, Antonio J. Fernández y Fernando Sáenz-Pérez. Playing with TOY: Constraints and domain cooperation. *ESOP*, páginas 112–115, (4 pág.) 2008.

  Clasificación. CORE (2008): A. Citeseer: 87/1221 (7 %), impacto 1.58. CS Conf Rankings: 0.92. posición media: top 16 %. Microsoft Academic: 8/167. Área: Programming Languages.

- Sonia Estévez-Martín, Antonio J. Fernández, M. Teresa Hortalá-González, Mario Rodríguez Artalejo, Fernando Sáenz-Pérez y Rafael del Vado-Vírseda. Cooperation of constraint domains in the TOY system. *PPDP*, páginas 258–268, (11 pág.) ACM Press, 2008.

  Clasificación. CORE (2008): B. Citeseer: 421/1221 (34 %), impacto 0.75. posición media: top 50 %. Microsoft Academic: 29/167. Área: Programming Languages.

- Sonia Estévez-Martín, Antonio J. Fernández, M. Teresa Hortalá-González, Mario Rodríguez Artalejo y Rafael del Vado-Vírseda. A fully sound goal solving calculus for the cooperation of solvers in the $CFLP$ scheme. ENTCS, 177:235–252, (18 pág.) 2007.

  Clasificación. CORE (2008): C. Microsoft Academic: 135/167. Área: Programming Languages.

Contribuciones sin clasificar:

- Ignacio Castiñeiras, Jesús Correas Fernández, Sonia Estévez-Martín y Fernando Sáenz-Pérez (2012). $\mathcal{TOY}$: A CFLP Language and System. ALP Newsletter. `http://www.cs.nmsu.edu/ALP/2012/06/toy-a-cflp-language-and-system/`.

- Sonia Estévez, Antonio J. Fernández y Fernando Sáenz. Cooperation of the Finite Domain and Set Solvers in TOY. Paqui Lucio, Ginés Moreno y Ricardo Peña, editores, *Actas de IX Jornadas sobre Programación y Lenguajes (Prole'09)*, páginas 217–226, (10 pág.) San Sebastián, España, 2009.

- Sonia Estévez-Martín, Antonio J. Fernández-Leiva y Fernando Sáenz-Pérez. $\mathcal{TOY}$: A system for experimenting with cooperation of constraint domains. *ENTCS*, 258(1):79–91, (13 pág.) 2009.

- Sonia Estévez-Martín, Antonio J. Fernández-Leiva y Fernando Sáenz-Pérez. About implementing a constraint functional logic programming system with solver cooperation. *Actas de CICLOPS'07*, páginas 57–71, (15 pág.) 2007.

- Sonia Estévez-Martín, Antonio J. Fernández, M. Teresa Hortalá-González, Mario Rodríguez-Artalejo, Fernando Sáenz-Pérez y Rafael del Vado-Vírseda. A proposal for the cooperation of solvers in constraint functional logic programming. *ENTCS*, 188:37–51, (15 pág.) 2007.

1.3 Introducción al lenguaje $\mathcal{TOY}$

$\mathcal{TOY}$ [ACE+07, LS99b, CCES12] es un lenguaje y sistema lógico funcional con restricciones cuyo mecanismo operacional es el estrechamiento perezoso guiado por la demanda [LLR93, LRV04] combinado con la resolución de restricciones [LRV07]. Con este tipo de estrechamiento se retrasa la evaluación de los argumentos de las funciones hasta que esta evaluación sea imprescindible para el cómputo de dicha función. Los programas $\mathcal{TOY}$ que utilizan restricciones combinan el mecanismo de estrechamiento perezoso con el resolutor de restricciones.

Un programa $\mathcal{TOY}$ es una colección de definiciones de varias entidades, tales como tipos, operadores, funciones perezosas al estilo de Haskell, así como predicados al estilo de Prolog (un predicado es una función Booleana que devuelve el valor `true`). Se denota como DC (Data Constructors) al conjunto de las *constructoras de datos*, TC (Type Constructors) al conjunto de las *constructoras de tipos* y FS (Function Symbols) al conjunto de los *símbolos de función*. Los símbolos de función FS se dividen en dos subconjuntos disjuntos, las *funciones primitivas* PF (Primitive Functions) y las *funciones definidas* DF (Defined Functions). Estos conjuntos determinan la *signatura* del programa. Por convenio, las variables comienzan por mayúscula o por el símbolo `'_'` y los símbolos de función comienzan por minúscula, al igual que las constructoras de datos, los nombres y los alias de los tipos. Se denota como $\mathcal{V}ar$ al conjunto de *variables de datos* y como $\mathcal{TV}ar$ al conjunto *variables de tipo*. Dentro del conjunto de todas las constructoras de datos DC, el subconjunto $DC^k \subset DC$ representa las constructoras de aridad k. La *aridad* de un símbolo de constructora o función es el número de argumentos que espera. Se utilizará la misma notación para el conjunto de símbolos de función. Las constructoras de datos deben estar predefinidas o haber sido introducidas en una declaración de tipos. Además, existe una constructora especial de aridad 0 ($\perp$) que representa

un valor indefinido. Esta constructora no aparece explícitamente en los programas pero es necesaria en la semántica del lenguaje.

Por ejemplo, el *tipo construido* `nat` que representa los números naturales utilizando la aritmética de Peano se define mediante la siguiente declaración de tipos que utiliza la palabra reservada `data`, una constructora de aridad 0 (`cero` $\in DC^0$) y una constructora de aridad 1 (`suc` $\in DC^1$):

```
data nat = cero | suc nat
```

Con esta declaración de tipo se puede definir la función `suma` $\in DF^2$ de la siguiente forma:

```
suma :: nat -> nat -> nat
suma cero X = X
suma (suc X) Y = suc(suma X Y)
```

Utilizando estas declaraciones se pueden evaluar objetivos que se escriben en la línea de comandos del sistema $\mathcal{TOY}$, por ejemplo:

```
Toy()> suma X Y == (suc (suc cero))
```

En este caso, el sistema $\mathcal{TOY}$ responde al usuario vinculando las variables `X` e `Y`. Este objetivo tiene varias respuestas posibles que pueden ser solicitadas de una en una mediante la opción `y` (o `enter`) o se pueden solicitar todas las respuestas mediante la opción `a`. A continuación se muestran las tres respuestas que se obtienen del objetivo planteado:

```
Toy()> suma X Y == (suc (suc cero))
      { X -> cero,
        Y -> (suc (suc cero)) }
      Elapsed time: 0 ms.
sol.1, more solutions (y/n/d/a) [y]?
      { X -> (suc cero),
        Y -> (suc cero) }
      Elapsed time: 0 ms.
sol.2, more solutions (y/n/d/a) [y]?
      { X -> (suc (suc cero)),
        Y -> cero }
      Elapsed time: 0 ms.
sol.3, more solutions (y/n/d/a) [y]?
      no
      Elapsed time: 0 ms.
      Total elapsed time: 0 ms.
```

En esta sección se van a definir brevemente los elementos del lenguaje utilizados en este ejemplo. Otros conceptos como dominio, signatura específica y valores básicos se definirán formalmente en el capítulo 3.

Las expresiones en $\mathcal{TOY}$ se forman combinando símbolos de función, constructoras de datos, variables y valores básicos en notación currificada. Las expresiones dependen del dominio al que pertenecen y pueden tener apariciones de valores básicos específicos del dominio. El conjunto de valores básicos se denota como $\mathcal{B}$ y el conjunto de las expresiones de una signatura Σ sobre el conjunto de valores básicos $\mathcal{B}$ se denota como $Exp_\Sigma(\mathcal{B})$. La sintaxis de una *expresión* $e \in Exp_\Sigma(\mathcal{B})$ es:

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

donde X es una variable ($X \in \mathcal{Var}$), u es un valor básico ($u \in \mathcal{B}$), c es un símbolo de constructora ($c \in DC^n$), f es un símbolo de función ($f \in FS^n$), la expresión $(e\ e_1)$ corresponde a la aplicación de una función e sobre la expresión e_1, y finalmente $(e_1, \cdots, e_n)$ corresponde a una tupla. La constructora $c \in DC^n$ tiene aridad n, lo que significa que espera n argumentos, y de forma similar $f \in FS^n$. Las aplicaciones asocian por la izquierda y los paréntesis más externos se pueden omitir debido a que $\mathcal{TOY}$ utiliza notación currificada.

Las expresiones que solo contienen constructoras de datos pueden ser vistas como representaciones simbólicas de resultados computados. Los *patrones* son una clase de expresiones y pueden ser constructoras o funciones aplicadas parcialmente (con menos argumentos de los que indica su aridad). Es decir, los patrones son expresiones que no contienen llamadas a funciones o aplicaciones totales de funciones y se utilizan para representar valores (datos) que no necesitan ser evaluados. Esto quiere decir que son *expresiones irreducibles* o *formas normales*. La sintaxis de un *patrón* en la signatura Σ sobre $\mathcal{B}$ ($t \in Pat_\Sigma(\mathcal{B})$) es:

$$t ::= X\ |u \mid (t_1, \cdots, t_n) \mid c\ t_1 \cdots t_m \mid f\ t_1 \cdots t_m$$
$$\text{donde } (u \in \mathcal{B}, c \in DC^n, 0 \leq m \leq n) \text{ y } (f \in FS^n, 0 \leq m < n).$$

Los patrones que contienen aplicaciones parciales de constructoras o de funciones se denominan *patrones funcionales* o *patrones de orden superior*. Una expresión o un patrón es *total* cuando está 'totalmente definido', es decir, si no tiene ninguna aparición del valor indefinido $\perp$, y se llama *parcial* en otro caso. Se denominan *expresiones o patrones lineales* a aquellas expresiones o patrones que no tienen variables repetidas. Las *expresiones o patrones básicos* (*ground*) no contienen variables. Los conjuntos de expresiones y patrones básicos se denotan como $GExp_\Sigma(\mathcal{B})$ y $GPat_\Sigma(\mathcal{B})$, respectivamente.

Por ejemplo, a continuación se definen la función `head` que devuelve la cabeza de una lista y la función `filter` que criba los elementos de una lista dependiendo de la propiedad que se establezca a través del parámetro `P`.

```
head :: [A] -> A
head [X|_] = X

filter :: (A -> bool) -> [A] -> [A]
filter L [] = []
filter P [X|Xs] = if P X then [X|filter P Xs]
                         else filter P Xs
```

La expresión `head [1,2,3]` es una aplicación total de la función `head` que se reduce al valor `1`, así un objetivo de la forma `head [1,2,3] == L` vincula la variable `L` a `1`. La expresión `filter (> 1) [1,2,3]` contiene una aplicación parcial de la función `>` pues le falta un argumento

para que pueda evaluarse. Sin embargo la función `filter` toma esta aplicación parcial como argumento de la expresión funcional `P` y de esta forma el objetivo `filter (>1) [1,2,3] == L` vincula la variable `L` a la lista `[2,3]`. Obsérvese que en $\mathcal{TOY}$ las listas pueden ser definidas al estilo Prolog, como en este ejemplo, o al estilo Haskell utilizando la constructora de listas `(:)` de aridad 2. De esta forma, `1:2:3:[]` y `[1,2,3]` representan la misma lista.

Una función $f \in FS^n$ se define con una declaración de tipos opcional $f :: \tau_1 \to \cdots \to \tau_n \to \tau$ y una o más reglas de reescritura condicionales currificadas:

$$f\ t_1\ \ldots\ t_n = r \texttt{ <== } C_1, \ldots, C_m$$

Las funciones deben ser lineales en sus argumentos, es decir, $t_1 \ldots t_n$ no pueden incluir variables repetidas. Los parámetros formales $t_1\ \ldots\ t_n$ deben ser patrones y las condiciones C_i son opcionales. Una regla de esta forma tiene la siguiente lectura: la expresión $f\ t_1\ \ldots\ t_n$ se puede reducir a la expresión r si se satisfacen las restricciones $C_1, \ldots, C_m$. Para simplificar la notación, se utilizará la notación $\overline{t}_n$ para abreviar $t_1 \ldots t_n$. Esta notación se extiende a otros elementos del lenguaje; así por ejemplo, $\overline{\tau}_n$ abrevia $\tau_1 \to \cdots \to \tau_n$. Además, $\overline{Y}$ se utiliza para representar una secuencia de variables.

Con respecto a los objetivos, en $\mathcal{TOY}$ se define un *objetivo* como una conjunción de condiciones $C_1, \ldots, C_m$ donde cada condición C_i se interpreta como una restricción que debe ser resuelta utilizando el estrechamiento perezoso combinado con la resolución de restricciones. Si C_i es únicamente una expresión e entonces el sistema entiende que es una restricción de igualdad $e == true$. Si el objetivo es insatisfactible entonces el sistema responde `no`, y en el caso que sea satisfactible responde `yes` y muestra las sustituciones y restricciones en forma resuelta. Si lo desea el usuario, el cómputo puede continuar hasta encontrar otra solución utilizando *vuelta atrás* (backtracking) o finalizar.

Para completar esta introducción, se mostrarán seguidamente las características más relevantes de $\mathcal{TOY}$ a través de explicaciones introductorias y de ejemplos que en su mayoría utilizan funciones disponibles en [ACE+07].

1.3.1 Tipos

$\mathcal{TOY}$ es un lenguaje fuertemente tipado con todas las ventajas bien conocidas de un proceso de comprobación de tipos. Su sistema de tipos está basado esencialmente en el sistema de tipos polimórficos de Hindley-Milner-Damas [Hin69, Mil78, DM82]. Por lo tanto, el tipo principal de una expresión puede incluir variables de tipo polimórficas. Los tipos de las funciones se infieren y, opcionalmente, se pueden declarar en el programa de forma similar a Haskell. Es posible definir nuevos tipos utilizando constructoras de datos, como por ejemplo el tipo `nat` visto anteriormente. También es posible definir tipos sinónimos para declarar un nuevo identificador como una abreviatura de un tipo complejo.

Una propiedad de los tipos que será utilizada en el desarrollo de la tesis es la *propiedad de transparencia* de los tipos principales. Un tipo $\tau_1 \to \cdots \to \tau_m \to \tau$ se denomina *m-transparente* [GHR01, MR12] si las variables de $\tau_1, \cdots, \tau_m$ están contenidas en las variables de τ y *m-opaco* en caso contrario. Una función f se denomina m-transparente si su tipo principal es m-transparente y m-opaca si su tipo principal es m-opaco. Se usa la misma notación para

constructoras y patrones. Para motivar este requisito se va a mostrar un ejemplo en el cual un objetivo que contiene patrones opacos se descompone en un objetivo mal tipado.

La función `second`, definida a continuación, devuelve su segundo argumento y es una función 1-opaca porque `A` no aparece en el tipo del resultado de la función.

```
second :: A -> B -> B
second X Y = Y
```

El patrón `(second X)` es opaco, lo que conlleva a que su tipo principal `B -> B` [1] no aporte ninguna información sobre el tipo de `X`, y diferentes instancias de `(second X)` conserven el tipo principal `B -> B` independientemente del tipo de la expresión sustituida por `X`. De este modo, el objetivo `second true == second cero` está bien tipado, pues ambos patrones tienen el mismo tipo, `second true :: B -> B` y `second cero :: B -> B`, respectivamente. Sin embargo, en un paso de descomposición obtenemos el subobjetivo `true == cero`, que obviamente está mal tipado.

Las *descomposiciones opacas* se producen cuando se tiene un objetivo de la forma $e\ e_1 \ldots e_m$ `==` $e\ e'_1 \ldots e'_m$ donde $e\ e_1 \ldots e_m$ y $e\ e'_1 \ldots e'_m$ son patrones de la siguiente forma: o bien $e \in DF^n$ y $m < n$ o bien $e \in DC^n$ y $m \leq n$. Supongamos que los patrones $e\ e_1 \ldots e_m$ y $e\ e'_1 \ldots e'_m$ son opacos. El objetivo se resuelve por descomposición produciendo nuevos subobjetivos e_1 `==` e'_1, $\cdots$, e_m `==` e'_m, de tal forma que si alguna variable de tipo τ_i para algún $1 \leq i \leq m$ no aparece en τ_{m+1} `->` $\ldots$ τ_n `->` τ, entonces alguno de los subobjetivos e_i `==` e'_i puede estar mal tipado. Es decir, las restricciones con patrones opacos pueden producir objetivos mal tipados, pues las nuevas restricciones resultantes de la descomposición pueden estar mal tipadas. Por lo tanto, para poder establecer la completitud del cálculo $CCLNC$, que se mostrará en la subsección 4.2, es requisito que los patrones sean transparentes.

1.3.2 Funciones indeterministas o multivaloradas

El indeterminismo es una característica de la programación lógico funcional adoptada de la programación lógica. Se dice que una función es indeterminista si devuelve diferentes resultados para los mismos argumentos ya evaluados. Un ejemplo clásico es la función indeterminista `coin` que modela el lanzamiento de una moneda y devuelve dos posibles valores, el 0 y el 1. El sistema lleva a cabo ambas reducciones por backtracking o vuelta atrás.

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

Si se tiene que evaluar una llamada a una función en la cual alguno de sus argumentos es indeterminista, entonces es necesario establecer una estrategia de evaluación de dichos parámetros. En general hay dos estrategias, *call-time choice* y *run-time choice* [Hus93], aunque recientemente han surgido nuevas propuestas [RR14].

La función `double` que duplica el valor de su parámetro nos sirve como ejemplo para comprender la diferencia entre las dos estrategias.

[1] El tipo que devuelve el sistema $\mathcal{TOY}$ a la consulta `\type(second X)` es `A -> A`, pero se mantiene el tipo `B -> B` en la explicación para que se corresponda con la definición.

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

La evaluación del objetivo `double coin == L` tiene varias opciones. La estrategia *call-time choice* [GHLR99], que implementa $\mathcal{TOY}$, muestra como resultado dos soluciones: `L == 0` y `L == 2`. Esto es debido a que la función `double` recibe como argumento a la llamada a `coin` sin evaluar (por ser un lenguaje perezoso). Al reducir la expresión `coin + coin` primero reduce una de las dos expresiones `coin` y debido a la propiedad de compartición de parámetros (*sharing*), la otra expresión `coin` toma el mismo valor que la primera, ya que las variables que aparecen más de una vez en la parte derecha de la regla son compartidas para evitar evaluar el mismo valor más de una vez. De esta forma se computa `0+0` y `1+1`. La estrategia *run-time choice* no realiza compartición de parámetros y produce cuatro resultados: `L == 0`, `L == 1`, `L == 1` y `L == 2`. En esta estrategia, al reducir de la expresión `coin + coin` se evalúa cada subexpresión `coin` de forma independiente, obteniendo cuatro posibles valores: `0+0`, `0+1`, `1+0` y `1+1`.

Otro objetivo indeterminista es el siguiente:

```
Toy(R)> filter (>0) [1,-1,2,X,3] == L, head L == H
```

Como resultado de su ejecución se devuelven las dos respuestas siguientes:

```
{ L -> [ 1, 2, X, 3 ],
  H -> 1 }
{ X>0.0 }
```

```
{ L -> [ 1, 2, 3 ],
  H -> 1 }
{ X=<0.0 }
```

Obsérvese que `filter` tiene como primer parámetro el patrón `(>0)`, que es un operador aplicado parcialmente. Por lo tanto es una función de orden superior, concepto que se muestra a continuación.

1.3.3 Orden superior

Las funciones de orden superior en $\mathcal{TOY}$ permiten pasar funciones como argumentos y devolver funciones como resultados de forma similar a Haskell y otros lenguajes funcionales, excepto que $\mathcal{TOY}$ no contempla λ-abstraciones. El orden superior se utiliza frecuentemente combinado con aplicaciones parciales (una función o constructora se aplica parcialmente si se invoca con menos argumentos de los que indica su aridad). Esto es posible porque $\mathcal{TOY}$ utiliza notación currificada. Un ejemplo clásico de función de orden superior es la función `map` definida a continuación.

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

La función `map` recibe un parámetro `F` de tipo funcional y una lista, y produce la lista resultante de aplicar `F` a cada elemento de la lista.

El objetivo `map head [[1,2],[3],[4,5]] == L`, que utiliza la función `head` descrita en la sección anterior, instancia la variable `L` a la lista `[1,3,4]`. En este objetivo `head` es una aplicación parcial pues tiene aridad 1 y aparece sin argumentos.

El orden superior se puede utilizar no solo con símbolos de función, sino que también permite constructoras de datos. Un ejemplo de la aplicación parcial de la constructora `(:)` como argumento de la función `map` es el objetivo: `map (1:) [[2,3],[]] == L`. La evaluación de este objetivo vincula la variable `L` a la lista `[[1,2,3], [1]]`.

Una característica de la programación de orden superior que está disponible en $\mathcal{TOY}$ es el uso de las *variables lógicas de orden superior*, que representan valores funcionales. Pueden aparecer en el lado izquierdo de las reglas que definen funciones, así como en los resultados computados. En particular, $\mathcal{TOY}$ permite hacer cómputos que contengan *variables lógicas de orden superior*, es decir, se puede lanzar como objetivo una restricción donde el identificador de la función a ejecutar sea una variable. Por ejemplo, el objetivo `map F [true,X] == [Y,false]` vincula la variable lógica de orden superior `F` a patrones de orden superior. A continuación se muestran algunas soluciones de este objetivo.

```
F -> ((==) true), X -> false, Y -> true
F -> ((==) false), X -> true, Y -> false
F -> ((/=) true), X -> true, Y -> false
F -> ((/=) false), X -> false, Y -> true
F -> not, X -> true, Y -> false
```

El uso de variables lógicas de orden superior en $\mathcal{TOY}$ pueden dar lugar a soluciones mal tipadas como ya se mostró en el trabajo [GHR01]. Esto es debido a que $\mathcal{TOY}$ tiene comprobación de tipos estática y no dinámica, como se mostrará en la subsección 1.3.1. Como consecuencia, la ausencia de errores de tipos en tiempo de ejecución está garantizada para cálculos puramente funcionales, pero no para cálculos que implican variables lógicas de orden superior. Por este motivo, el cálculo $CCLNC(\mathcal{D})$ que se define en la sección 4.2 para la cooperación de dominios de restricciones excluirá el uso de variables lógicas de orden superior para garantizar la completitud.

1.3.4 Evaluación perezosa

$\mathcal{TOY}$ es un lenguaje de evaluación perezosa, es decir, no se evalúan los parámetros de las funciones hasta que estos no sean necesarios para proseguir con el cómputo de la función. Esta característica hace que se puedan tratar estructuras de datos potencialmente infinitas como parámetros de funciones sin crear un cómputo no terminante. Esto es debido a que estas estructuras infinitas se generan según se van demandando.

Un ejemplo clásico es la función `take`, que recibe un número entero `N` y una lista y devuelve una lista con los `N` primeros elementos de la lista que se le pasa como argumento, o la lista entera si el tamaño de la lista es menor o igual que `N`. Se puede utilizar la función `take` combinada con la función `iterate` que devuelve una lista infinita de elementos calculados por las sucesivas aplicaciones de una cierta función F a un elemento inicial, ambos dados como parámetros.

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

iterate :: (A -> A) -> A -> [A]
iterate F X = [X|iterate F (F X)]
```

La evaluación de la expresión `iterate (1 +) 1` crea la lista infinita `[1,2,3,...]`. Sin embargo en el objetivo `take 3 (iterate (1 +) 1) == L` se vincula la variable `L` a la lista `[1,2,3]` pues `take` evalúa su segundo argumento según lo va necesitando.

1.3.5 Restricciones de igualdad estricta

En los paradigmas lógico y funcional existe la restricción de igualdad `(==)`, aunque el significado es distinto al de $\mathcal{TOY}$. Por ejemplo, en Haskell una igualdad entre dos expresiones es cierta si ambas se reducen a formas normales iguales. Así `2 + 2 == 4` se reduce a `True` mientras que `3 + 2 == 4` se reduce a `False`. Pero no puede evaluar restricciones que contengan variables como por ejemplo `x + y == 4`. Prolog trata la igualdad estricta de términos desde el punto de vista puramente sintáctico. De este modo, tanto `2 + 2 == 4` como `X == Y` fallan y `X == X` tiene éxito. Por otra parte, en Prolog la unificación `X = Y` tiene éxito ligando `X` e `Y` pero no evalúa las expresiones como se hace en el estilo funcional. Por ejemplo, `X = 2 + 2` tiene éxito unificando la variable `X` a la expresión `2 + 2`, pero no evalúa las expresiones, solo las unifica.

En $\mathcal{TOY}$ la restricción de igualdad estricta e_1 `==` e_2 es cierta si ambos argumentos, e_1 y e_2, se pueden reducir a formas normales unificables. El calificativo estricta es una cuestión puramente semántica y quiere decir que si alguno de sus argumentos es indefinido ($\perp$) el resultado es indefinido. El concepto de reducción en $\mathcal{TOY}$ es más amplio que en los lenguajes puramente funcionales, pues las expresiones e_1 y e_2 pueden reducirse aun conteniendo variables, y además, al ser un lenguaje indeterminista, una misma expresión se puede reducir a más de una forma normal. De este modo, `2 + 2 == 4` tiene éxito porque se evalúa como en programación funcional, pero además `2 + 2 == X` también tiene éxito y unifica la variable `X` con el valor 4.

De acuerdo a la semántica, una restricción e_1 `==` e_2 podría ser resuelta reduciendo e_1 y e_2 a formas normales (patrones) y después unificarlos. Sin embargo, en la práctica se intercala la reducción de e_1 y e_2 con el objetivo de anticipar el fallo si no fueran unificables. Por ejemplo, si se tiene una restricción de la forma `suc e == cero`, reducir `suc e` a su forma normal puede ser costoso dependiendo de la expresión `e`. Sin embargo, como existen diferentes símbolos de constructora en la misma posición, `suc` y `cero`, el sistema detecta un conflicto de constructoras y falla sin necesidad de evaluar `e`.

1.3.6 Restricciones de desigualdad

Una restricción de desigualdad estricta entre dos expresiones e_1 `/=` e_2 se satisface si e_1 y e_2 pueden reducirse a otras expresiones que contengan una constructora distinta en la misma

posición externa. Es decir, si se detecta conflicto de constructoras. En $\mathcal{TOY}$ las desigualdades son consideradas restricciones y pueden contener variables. Así una restricción de la forma `suc X /= cero` puede resolverse en $\mathcal{TOY}$ pero no en los lenguajes funcionales como por ejemplo Haskell, pues no puede reducir expresiones que contengan variables. Otra característica de $\mathcal{TOY}$ es que las desigualdades pueden formar parte de la respuesta. Por ejemplo, una desigualdad de la forma `(X,2) /= (1,2)` se cumple si `X` se restringe a que sea distinto de `1`. De esta forma, la respuesta `X /= 1` captura las infinitas soluciones.

El indeterminismo nos permite obtener distintas respuestas a una restricción, posiblemente conteniendo desigualdades. Por ejemplo, el siguiente código comprueba si un elemento dado es miembro de una lista:

```
member X [] = false
member X [Y|Ys] = if X == Y then true else member X Ys
```

Consideremos el objetivo `member N [X,Y,Z]`. Al reducir este objetivo con la segunda regla se satisface la restricción `X == N`. Las restricciones de igualdad en las respuestas pueden entenderse como restricciones o como sustituciones. De esta forma, la sustitución `{X -> N}` sería la primera respuesta. Al aplicar vuelta atrás para calcular el resto de soluciones, se obtiene que se deben satisfacer la restricciones `X /= N` y `member N [Y,Z]`. Al igual que el objetivo inicial, `member N [Y,Z]` se reduce mediante su segunda regla obteniendo la restricción `Y == N`. Así, la segunda respuesta al objetivo inicial es: `{Y -> N, X /= N}`. Volviendo a aplicar vuelta atrás, las restricciones que se deben satisfacer son las siguientes: `X /= N`, `Y /= N` y `member N [Z]`. Aplicando de nuevo la segunda regla se obtiene la tercera y última respuesta: `{Z -> N, X /= N, Y /= N}`.

1.3.7 Restricciones aritméticas sobre números reales $\mathcal{R}$

El resolutor de restricciones aritméticas sobre los números reales de $\mathcal{TOY}$ contiene una serie de restricciones predefinidas documentadas en [ACE+07], entre las cuales están los operadores aritméticos (`(+), (-), (*), (/), (**)...`), funciones trigonométricas (`sin, cos, tan...`), relacionales (`(<), (<=), (>), (>=)`), y funciones habituales que trabajan con números reales para construir expresiones (`min, max, div, mod, gcd, round, trunc, floor, ceiling...`). También están disponibles los símbolos de función de igualdad y desigualdad sobre números reales (`(==), (/=)`) que utilizan los mismos símbolos que las restricciones de igualdad y desigualdad estrictas. El resolutor de reales de $\mathcal{TOY}$ se carga mediante el comando `/cflpr`.

El resolutor de restricciones sobre números reales de $\mathcal{TOY}$ está implementado sobre el resolutor de SICStus Prolog que es capaz de resolver restricciones lineales y algunas restricciones no lineales. $\mathcal{TOY}$ procesa las restricciones mediante el estrechamiento perezoso y cuando ya son restricciones primitivas irreducibles entonces se envían al resolutor de SICStus Prolog. El sistema $\mathcal{TOY}$ se encarga de distinguir el tipo de restricción que es y procesarla convenientemente o enviarla a su correspondiente resolutor, siendo este un proceso transparente para el usuario.

Una característica del resolutor de restricciones sobre los números reales de $\mathcal{TOY}$ es que resuelve restricciones que pueden contener llamadas a funciones, pues entrelaza adecuada-

mente la evaluación perezosa de expresiones con la resolución de restricciones. El siguiente ejemplo introduce esta idea.

Ejemplo 1. Definición de regiones en el plano

El siguiente código, incluido en la distribución de $\mathcal{TOY}$ (`regions.toy`) [ACE+07] define dos regiones en el plano, el círculo y el rectángulo, y ciertas operaciones sobre regiones del plano. Toma las regiones como conjuntos de puntos representados por su función característica.

```
type point  = (real,real)
type region = point -> bool

infixr 50 <<-

(<<-) :: point -> region -> bool
P <<- R = R P

rectangle :: point -> point -> region
rectangle (A,B) (C,D) (X,Y) = (X>=A)/\(X<=C)/\(Y>=B)/\(Y<=D)

circle :: point -> real -> region
circle (A,B) R (X,Y) = (X-A)*(X-A)+(Y-B)*(Y-B) <= R*R

outside :: region -> region
intersect, union :: region -> region -> region
outside   R  P = not (P <<- R)
intersect R1 R2 P = P <<- R1  /\ P <<- R2
union     R1 R2 P = P <<- R1  \/ P <<- R2
```

El operador `<<-` decide si un punto pertenece a una región aplicando la función característica de la región a este punto. Un rectángulo se define por su esquina inferior izquierda y su esquina superior derecha, y un círculo por su radio y su centro. Así por ejemplo el objetivo `(1,1) <<- rectangle (0,0) (2,2)` se satisface y el objetivo `(1,3) <<- circle (0,0) 2` falla.

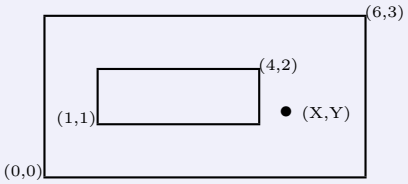
Un objetivo que contiene restricciones como respuesta es `(3,0) <<- circle (0,0) X` que pregunta qué valor ha de tomar el radio de un círculo de centro el origen de coordenadas y que incluye al punto `(3,0)`. La solución a este objetivo es el conjunto de restricciones `{-(X^2.0)+_B==0.0,_B>=9.0}`.

Otro objetivo interesante es `(X,Y) <<- circle (0,0) 2`, que devuelve las restricciones que deben cumplir las variables `X` e `Y` para que el punto `(X,Y)` pertenezca al circulo de centro `(0,0)` y radio `2`.

La restricción `<<-` aplica el segundo argumento sobre el primero obteniendo la expresión `circle (0,0) 2 (X,Y)`. Esta expresión llama a la función `circle` que se reduce a `(X-0)*(X-0)+(Y-0)*(Y-0) <= 2*2`. Esta inecuación se reduce a una serie

de restricciones primitivas mediante una técnica que se denomina *aplanamiento* de una restricción, que se formalizará en la sección 4.2. La idea de aplanar una restricción consiste en ir descomponiendo la restricción utilizando variables nuevas hasta obtener un conjunto de restricciones primitivas que puedan ser enviadas al resolutor. Así, `(X-0)*(X-0)+(Y-0)*(Y-0) <= 2*2` se aplana obteniendo el conjunto de restricciones primitivas: `{V1 <= 2*2, V1 == V2+V3, V2 == V4*V5, V4 == X-0, V5 == X-0, V3 == V6+V7, V6 == Y-0, V7 == Y-0}`. Todas las restricciones primitivas obtenidas se envían al resolutor de SICStus Prolog y, como no se pueden resolver, se quedan suspendidas. Al finalizar la evaluación del objetivo, $\mathcal{TOY}$ muestra el almacén de restricciones.

Otro objetivo más interesante es preguntar qué puntos `(X,Y)` pertenecen a la región que existe entre dos rectángulos, uno interior al otro. Es decir, la intersección entre la región interna del rectángulo mayor y la región externa del rectángulo menor, como muestra el siguiente dibujo:



El siguiente objetivo $\mathcal{TOY}$ resuelve este problema:

```
(X,Y) <<- intersect  (rectangle (0,0) (6,3))
                     (outside (rectangle (1,1) (4,2)))
```

La resolución de este objetivo consiste en esencia en evaluar primero la expresión `rectangle (1,1) (4,2)`, posteriormente `outside` y así sucesivamente hasta terminar con la evaluación de `<<-`. La definición de las funciones `rectangle` y `outside` utilizan el operador de conjunción `/\` y la función `not` predefinidas en el sistema. Durante la evaluación de estas subexpresiones se obtienen llamadas a funciones primitivas que se envían a su correspondiente resolutor. Las soluciones a este objetivo se muestran a continuación:

```
Solución 1      Solución 2      Solución 3      Solución 4
 X >= 1.0        X >= 1.0        X >  4.0        X >= 0.0
 X =< 4.0        X =< 4.0        X =< 6.0        X <  1.0
 Y >  2.0        Y >= 0.0        Y >= 0.0        Y >= 0.0
 Y =< 3.0        Y <  1.0        Y =< 3.0        Y =< 3.0
```

La unión de las cuatros áreas dadas como respuestas constituyen la región deseada. Se puede observar que las soluciones son excluyentes, es decir, no se solapan.

1.3.8 Restricciones de dominios finitos sobre números enteros $\mathcal{FD}$

La particularidad de la programación con restricciones en el dominio $\mathcal{FD}$ es que todas las variables involucradas en restricciones $\mathcal{FD}$ están asociadas a un dominio finito de valores

enteros, tanto si se declaran explícitamente en los programas, como si se producen implícitamente por el resolutor de restricciones. Las restricciones actúan sobre el dominio de las variables reduciéndolas convenientemente. Si una variable no contiene valores en su dominio entonces el conjunto de restricciones es insatisfactible. El resolutor de dominios finitos de $\mathcal{TOY}$ se carga con el comando `/cflpfd`.

La estructura general de un programa que utiliza restricciones de dominios finitos tiene tres partes: asignación de valores a las variables, imposición de restricciones y etiquetado de variables. Para asignar valores a las variables se utilizan restricciones específicas para ello. En $\mathcal{TOY}$ se dispone de la restricción `domain L a b` donde `L` es una lista de enteros que toman valores entre `a` y `b`. Además, $\mathcal{TOY}$ dispone de una serie de restricciones de dominios finitos sobre los números enteros documentadas en [ACE+07, EM04, FHS05b, FHSV07]. Algunas de estas restricciones están formadas mediante los operadores aritméticos (`(#+), (#-), (#*)...`), los operadores relacionales (`(#<), (#<=), (#>), (#>=)`), y funciones habituales de dominios finitos (`sum, scalar_product, all_different, count, exactly...`). Finalmente, si la propagación no es suficiente para calcular valores concretos, entonces se etiquetan o asignan valores a las variables. El etiquetado de $\mathcal{TOY}$ tiene la siguiente sintaxis: `labeling Options L` donde `L` es la lista de variables a etiquetar y `Options` es una lista con las opciones de etiquetado. La enumeración de los valores se realiza dependiendo de ciertas estrategias. Es decir, se puede controlar mediante parámetros el orden en el cual se asignan las variables, el orden en el que se asignan los valores a las variable, o si se desea maximizar o minimizar el valor de una variable. Un ejemplo de objetivo es:

```
1 L == [X,Y,Z], domain L 1 5, sum L (#<) 7, all_different L,
2             labeling [toMaximize Z] L
```

Si se resuelve el objetivo correspondiente a la línea 1 entonces la respuesta no muestra valores concretos para las variables. Pero se puede observar que el dominio de las variables se ha reducido al intervalo `1..4` (valores enteros comprendidos entre `1` y `4`), pues el valor `5` ha sido excluido de todas las variables mediante la propagación de la restricción `X #+ Y #+ Z #< 7`. Si al objetivo anterior se le añade el etiquetado (línea 2), entonces se muestra la única respuesta posible: `L -> [1,2,3]`.

En el dominio $\mathcal{FD}$ también están disponibles las restricciones de igualdad y desigualdad sobre números enteros. Estas restricciones también utilizan los símbolos `(==)` y `(/=)` y es el sistema $\mathcal{TOY}$ quien se encarga de distinguir de qué tipo de restricción se trata y la procesa convenientemente. Aparte de los símbolos (`(==), (/=)`), en $\mathcal{TOY}$ están también disponibles los símbolos (`(#=), (#\=)`) que son sinónimos de restricciones de igualdad y desigualdad de dominios finitos sobre números enteros.

El resolutor $\mathcal{FD}$ de $\mathcal{TOY}$ intercala la evaluación de expresiones con la resolución de restricciones de forma análoga a los resolutores $\mathcal{R}$ y $\mathcal{FS}$. Cuando la evaluación de una expresión deriva en una función primitiva, esta es enviada a un resolutor específico. A continuación se muestra un ejemplo clásico de $\mathcal{FD}$ que incluye además la idea de entrelazar la evaluación de expresiones con la resolución de restricciones.

Ejemplo 2. Send + More = Money

Se trata de resolver el clásico puzzle donde cada letra representa un dígito distinto y se debe cumplir la suma:

```
  S E N D
+ M O R E
---------
M O N E Y
```

El código $\mathcal{TOY}$ que resuelve este acertijo se muestra a continuación:

```
smm :: [int] -> [labelingType] -> bool
smm [S,E,N,D,M,O,R,Y] Label = true <==
      domain [S,E,N,D,M,O,R,Y] 0 9,
      S #> 0,
      M #> 0,
      all_different [S,E,N,D,M,O,R,Y],
      toInt [S,E,N,D] #+ toInt [M,O,R,E] #= toInt [M,O,N,E,Y],
      labeling Label [S,E,N,D,M,O,R,Y]

toInt :: [int] -> int
toInt L = foldr (#+) 0 L' <==
      L' == zipWith (#*) (reverse L) (iterate (#* 10) 1)
```

Siguiendo la estructura habitual de la programación con restricciones de dominios finitos, primero se asignan valores enteros a las variables. Como las variables representan dígitos entonces los correspondientes dominios están definidos entre 0 y 9, excepto `S` y `M` que no pueden tomar el valor 0 por la restricción de la suma. Otra restricción impuesta es que todas las variables deben ser distintas. Para realizar la suma se ha definido la función `toInt` que convierte una lista de dígitos a un entero. De esta forma, la restricción que aplica la suma contiene tres llamadas a la función `toInt`, entrelazando evaluación de expresiones con restricciones. La función `toInt` se ha definido con funciones comunes de la programación funcional. Por una parte `iterate (#* 10) 1` devuelve la lista infinita `[1,10,100..]`. Por otra parte se invierte la lista a tratar. Por ejemplo, a las listas `[D,N,E,S]` y `[1,10,100..]` se les aplica la función `zipWith` junto con el plegado `foldr` obteniendo `D #* 1 #+ N #* 10 #+ E #* 100 #+ S #* 1000`. Una vez convertidas las listas de dígitos a números enteros se puede realizar la suma que impone el problema. Por último la restricción `labeling` enumera todas las posibles soluciones

Una vez que se ha compilado y cargado este código, se puede resolver el acertijo con un objetivo como `smm L [ff]`, cuya solución vincula la variable `L` a la lista `[9,5,6,7,1,0,8,2]`. En este objetivo se ha elegido la estrategia `ff` que sigue el principio `first-fail` y selecciona la variable de menor dominio para iniciar las vinculaciones.

1.3.9 Restricciones de conjuntos finitos de números enteros $\mathcal{FS}$

En $\mathcal{TOY}$ se ha desarrollado un resolutor de conjuntos que resuelve restricciones sobre el dominio de conjuntos finitos de enteros. El dominio de una variable de conjunto se representa mediante un retículo de conjuntos definidos por un ínfimo (*lower bound*), que es el conjunto de elementos que están definitivamente en el conjunto, y un supremo (*upper bound*), que es el conjunto de todos los elementos que pueden estar en el conjunto. La restricción `domainSets [X,Y,Z] (s []) (s [1,2,3])` define el dominio de las variables `X`, `Y` y `Z` como retículos $[\{\}, \{1,2,3\}]$ como se muestra en la figura 1.4. Este resolutor se carga en $\mathcal{TOY}$ mediante el comando `/cflp_ic_sets`.

En $\mathcal{TOY}$ un conjunto básico o *ground* está definido con la constructora `s` $\in DC$ y una lista de números enteros. Por ejemplo, `s [1,2,3]` representa el conjunto `{1,2,3}`. A lo largo de la tesis se mostrarán los conjuntos básicos entre llaves para clarificar la notación cuando no se trate de código $\mathcal{TOY}$ ni de objetivos $\mathcal{TOY}$, pues estos se escriben como lo admite el sistema. [2]

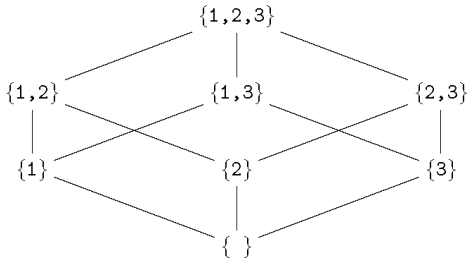


Figura 1.4: Retículo $[\{\}, \{1,2,3\}]$

Además de la restricción de dominio, $\mathcal{TOY}$ dispone de restricciones que se corresponden con las operaciones básicas sobre los conjuntos: `union` ($\cup$), `intersect` ($\cap$), `subset` ($\subset$), `diff` (diferencia: $\setminus$), `symdiff` (diferencia simétrica: $\triangle$) y `disjoint` (disjuntos). También están disponibles las restricciones de igualdad `(==)` y desigualdad `(/=)` sobre los conjuntos finitos.

De forma análoga al dominio $\mathcal{FD}$, los problemas que utilizan restricciones de conjuntos finitos de números enteros siguen la misma estructura básica: primero se asigna el dominio a las variables de conjuntos, después se imponen restricciones y por último se etiquetan las variables para encontrar las posibles soluciones, como ilustra el siguiente objetivo:

```
1  domainSets [X,Y,Z] (s []) (s [1,2,3]), diff X Y (s [1,2]),
2            disjoint Y Z, union Y Z (s [3]),
3                labelingSets [X,Y,Z]
```

La primera restricción define las variables `X`, `Y` y `Z` como conjuntos cuyos elementos pueden ser los números `1`, `2` y `3`. La siguiente restricción `X \ Y == {1,2}` obliga a que los números `1` y `2` pertenezcan a `X` y no pertenezcan a `Y`. Después `Y ∩ Z == { }` permite que `Z` tome los valores `1` y `2` (pero no obliga), pues seguro que no pertenecen a `Y`. Obsérvese que `Z` también

[2] Las llaves no se pueden usar para denotar conjuntos en $\mathcal{TOY}$ porque ya se usaban previamente para delimitar contextos de la regla de indentación.

puede tomar el valor `3`. Por último (`Y` $\cup$ `Z == {3}`) dice que el valor `3` está en `Y` y/o `Z` y que los valores `1` y `2` no están ni en `Y` ni en `Z`. Por lo tanto, el objetivo formado por las líneas 1 y 2 no es capaz de dar una solución pero la propagación de las restricciones poda los retículos que representan a los dominios de los conjuntos transformando los retículos iniciales en los siguientes retículos: `X` $\in [\{1,2\},\{1,2,3\}]$, `Y` $\in [\{\},\{3\}]$ y `Z` $\in [\{\},\{3\}]$. Si al objetivo anterior se le une el etiquetado de la línea 3, entonces $\mathcal{TOY}$ encuentra las tres soluciones donde `X`, `Y` y `Z` son respectivamente los conjuntos `{1,2,3}`, `{3}` y `{}` en la primera solución, `{1,2}, {3}` y `{}` en la segunda solución y `{1,2}, {}` y `{3}` en la tercera solución.

Capítulo 2

Estado del arte

La programación con restricciones tuvo sus inicios en los años 70 con el desarrollo de las técnicas de propagación que fuerzan la consistencia local, en concreto la consistencia de arco y camino [Mon74, Mac77, MH86]. Pero el verdadero hito de la programación con restricciones fue en la década de 1980 con el desarrollo de la programación lógica con restricciones (CLP) [JL87, Bar11]. Al igual que se extendió la programación lógica para tratar problemas de restricciones, se han extendido otros paradigmas de programación. De hecho $\mathcal{TOY}$ [LS99b, ACE$^+$07, TOY12] comenzó siendo un lenguaje FLP y se extendió con restricciones pasando a pertenecer al paradigma $CFLP$. La programación $CFLP$ surgió en torno a 1990 como un intento de combinar dos líneas de investigación de la programación declarativa: la programación lógica con restricciones CLP y programación lógico funcional FLP.

En este capítulo se desarrolla en primer lugar el estado del arte de la cooperación de resolutores en el paradigma CLP y a continuación se estudian las contribuciones más relevantes en el paradigma FLP, así como la programación FLP con restricciones y la cooperación de resolutores en $CFLP$. Finalmente se exponen las contribuciones más recientes sobre $\mathcal{TOY}$.

2.1 Cooperación en la programación lógica con restricciones

La programación lógica con restricciones fue iniciada por J. Jaffar y J.L. Lassez en 1987 en el artículo [JL87]. El objetivo del esquema CLP consiste en definir una familia de lenguajes de programación lógica con restricciones $CLP(\mathcal{D})$ parametrizados por un dominio de restricciones $\mathcal{D}$, de tal manera que los resultados bien establecidos en la semántica declarativa y operacional de los programas lógicos [Llo84] puedan extenderse a los lenguajes $CLP(\mathcal{D})$. En el curso del tiempo, CLP se ha convertido en un paradigma de programación muy exitoso [JM94, MS98, AW07], con una combinación limpia de la programación lógica y de métodos específicos para satisfacción de restricciones dependientes del dominio.

El modelo teórico de Prolog III de Colmerauer [Col90] ya introducía cooperación mediante la integración de *restricciones retardadas* que son válidas sintácticamente pero demasiado complejas para ser procesadas por un resolutor. Por ejemplo, cuando se intentan resolver restricciones no lineales con un resolutor lineal [GMB01]. Obsérvese que Prolog III dispone de un resolutor para procesar restricciones lineales pero no de un resolutor para procesar las

no lineales [Nar99]. Con el paso de Prolog III a Prolog IV se amplía el lenguaje permitiendo el procesamiento de restriciones no lineales sobre números reales y restricciones sobre números enteros mediante la inclusión de resolutores de intervalos y dominios finitos, respectivamente. La disposición de estos nuevos resolutores, más el resolutor lineal que ya estaba disponible en Prolog III, hacen posible la combinación de restricciones de distintos dominios. Esta combinación de restricciones se basa en la idea de que, si se puede extraer la suficiente información de los almacenes de restricciones, las restricciones retrasadas se pueden transformar simbólicamente y así ser procesadas por los resolutores.

En el trabajo [MRS95], en el cual se basó Mircea Marin para realizar su tesis, ya se investigaba la cooperación entre resolutores en el sistema CLP sobre restricciones polinómicas no lineales (igualdades, desigualdades e inecuaciones). Además, se muestra un prototipo llamado *CoSAc* (*Constraint System Architecture*) que soporta la cooperación de resolutores para restricciones no lineales sobre los números reales. Este prototipo se puede extender con nuevos resolutores o reemplazar los existentes. *CoSAc* utiliza tres resolutores: uno para restricciones lineales sobre racionales, otro para restricciones no lineales, y el tercero para simplificar polinomios y computar las raíces de polinomios de una variable.

Por otra parte, y también en el ámbito CLP, el lenguaje `HAL` [Hal03, DGH$^+$99a, DGH$^+$99b, SGD02, DSGH03, GJM$^+$01] está orientado a la construcción y experimentación de resolutores. `HAL` permite extender los resolutores disponibles y construir nuevos resolutores híbridos combinando distintos resolutores.

Con respecto a la combinación de resolutores, `HAL` crea un resolutor nuevo como la combinación de dos resolutores existentes, donde cada variable del resolutor combinado se corresponde con un par de variables, cada una del correspondiente resolutor subyacente. La comunicación entre los resolutores se lleva a cabo mediante objetivos retardados que se crean cuando una variable se inicializa [DGH$^+$99b]. La diferencia fundamental entre `HAL` y la cooperación que se propone en esta tesis es la aportación de un marco formal eficaz de implementar en el que la cooperación define puentes y proyecciones entre dominios para la poda de cómputos.

Dentro del ámbito CLP, y para el desarrollo de esta tesis, es interesante introducir el dominio de los conjuntos finitos y su cooperación con el dominio de los enteros con restricciones de dominio finito. Con respecto a los conjuntos finitos, uno de los primeros trabajos fue desarrollado por Carmen Gervet en los años 90 [Ger94, Ger97]. Gervet definió un marco para un nuevo lenguaje de programación lógica sobre dominios finitos de conjuntos. Este lenguaje, llamado `Conjunto`, combina los aspectos declarativos de los lenguajes lógicos como Prolog con la eficiencia de los resolutores de restricciones. De esta forma se extienden las funcionalidades propias de los lenguajes lógicos con operaciones y relaciones definidas en la teoría de conjuntos, como por ejemplo la intersección y la inclusión. Para establecer este nuevo marco se define el dominio de una variable de conjunto con un intervalo de conjuntos, especificando una cota inferior y una cota superior (*lower* y *upper bounds*), ordenados parcialmente con respecto a la inclusión de conjuntos ($\subseteq$). Por ejemplo: se puede definir la variable X con el intervalo de conjuntos [{},{1,2,3}] donde el dominio de la variable X es el conjunto {{},{1},{2},{3},{1,2},{1,3},{2,3},{1,2,3}}. Esta representación de conjuntos ha sido la base de varios resolutores que contienen conjuntos como por ejemplo: ECL^{i}PS^e [ECL],

`FaCiLe` [FaC], `Mozart-OZ` [Moz], `B-Prolog` [B-P], `CHOCO` [CHO] y `Gecode` [Gec].

Con respecto a la comunicación del dominio de conjuntos con otros dominios hay que decir que `Conjunto` dispone de las llamadas restricciones graduales (*graduated constraints*). Estas restricciones convierten los conjuntos a términos aritméticos para optimizar la resolución de problemas aplicando funciones de coste, como la cardinalidad o el peso (*set weight*). Más concretamente, una función gradual es una función f que hace corresponder cada conjunto a un único natural y satisface: $s_1 \subseteq s_2 \Rightarrow f(s_1) < f(s_2)$.

Otra contribución relevante en el área de los resolutores de conjuntos finitos es [AB00]. En este trabajo se propuso modelar problemas de circuitos mediante conjuntos y reglas de inferencia para podar el espacio de búsqueda usando la cardinalidad. Además, se desarrolló un nuevo resolutor de restricciones de conjuntos implementado en ECL^{i}PS^e: `Cardinal` [Aze07b]. Dicho resolutor está definido con un conjunto de reglas de reescritura sobre un almacén de restricciones de conjuntos y de dominios finitos. En concreto, solo se describieron las reglas de reescritura de conjuntos pues el resolutor de dominios finitos es conocido. Una variable de conjunto en `Cardinal` se define como una variable con atributos, que representan el dominio, la cardinalidad y los objetivos suspendidos.

Siguiendo con esta idea de ampliar la información de las variables de conjuntos, en el trabajo [SG04] se extendió la representación de las variables con la información del menor y el mayor valor lexicográfico para instancias de conjuntos. De esta forma se estableció un dominio híbrido con tres componentes: el dominio basado en la inclusión (*subset-bound domain*), la cardinalidad y una representación lexicográfica del dominio. Esta aproximación fortalecía la propagación de restricciones en el sistema `Conjunto` y también proporcionaba un mecanismo para implementar ciertas roturas de simetrías. Pero tenía dos limitaciones. Por una parte, las restricciones que trabajan entre varios dominios eran complejas y desde el punto de vista computacional eran costosas. Por otra parte, la ordenación lexicográfica estaba parcialmente integrada con la información de la cardinalidad.

Siguiendo en trabajo de Gervet, [GV06] se presenta *length-lex* que aporta un orden total sobre conjuntos, primero por longitud y después lexicográficamente. Esta representación une la información de la cardinalidad y la información lexicográfica en un único dominio y permite la consistencia de límites en restricciones unarias, pero no en restricciones binarias. En los trabajos [VYGD08, SG08] se trata esta cuestión exponiendo un algoritmo genérico para restricciones binarias.

Otra representación totalmente distinta es la que se propone en los trabajos [LS04a, HS05]. En estos trabajos se propone representar lo dominios de las variables de conjuntos finitos usando diagramas binarios de decisión reducidos y ordenados (ROBDD: Reduced Ordered Binary Decision Diagram). Esta representación es totalmente distinta a la representación de variables de conjuntos finitos usada en $\mathcal{TOY}$.

Para terminar con el estudio del dominio de los conjuntos finitos y su cooperación con el dominio de los enteros con restricciones de dominios finitos, el trabajo [DDPR03] estudia la combinación de resolutores correspondientes a los lenguajes de programación CLP(SET) [DPPR00] y CLP(FD) [CD96]. En este trabajo se desarrolla un puente entre la expresividad y la abstracción de alto nivel ofrecida por el lenguaje CLP(SET) y la eficiencia de los resolutores de CLP(FD). Este nuevo lenguaje, llamado CLP($\mathcal{SET}$)$^{\mathtt{int}}$, incluye un resolutor global

$\text{SAT}_{\mathcal{SET}+\mathcal{FD}}$. En esencia, este resolutor es un resolutor de conjuntos en el cual las reglas de reescritura relacionadas con las restricciones de pertenencia e igualdad y desigualdad se han ampliado con nuevas restricciones de $\mathcal{FD}$. Se implementaron diferentes prototipos: uno con el intérprete de Prolog `{log}` [DOPR91] y otro con la biblioteca de Java JSetL [JSe12]. JSetL es una biblioteca Java que combina el paradigma de la programación orientada a objetos de Java con conceptos de lenguajes CLP, como las variables lógicas, listas, unificación, resolución de restricción y no determinismo.

Fuera del ámbito específico de la cooperación de los dominios de conjuntos finitos y enteros con restricciones de dominios finitos, pero dentro del estudio de la cooperación de resolutores en el paradigma CLP, se encuentra el trabajo de Antonio Fernández Leiva y Patricia M. Hill [Fer00, FH99, FH04, FH06] que propone un esquema basado en la teoría de retículos, que es genérico y cooperativo para `CLP(Interval(`X`))` donde X es cualquier dominio de computación con estructura de retículo. Este esquema es un enfoque general para la satisfacción y optimización de restricciones de intervalo, así como para la cooperación entre resolutores de intervalo definidos sobre dominios de computación con estructura de retículo. En el esquema `CLP(Interval(`X`))` la cooperación entre resolutores entre diferentes dominios forma un nuevo dominio con estructura de retículo `CLP(Interval(`$X_1 \times \cdots \times X_n$`))`.

2.2 Programación lógico funcional

Dentro de los lenguajes declarativos de propósito general se encuentran los lenguajes funcionales y los lenguajes lógicos. Los primeros se basan en la noción de función matemática y los segundos en la lógica de predicados. Ambas clases de lenguajes tienen motivaciones similares pero disponen de distintas características. Por ejemplo, los lenguajes funcionales disponen de estrategias de evaluación guiadas por la demanda que soportan estructuras de datos infinitas, mientras que los lenguajes lógicos soportan el indeterminismo. Sin duda, estas características son deseables en el desarrollo del software. Los lenguajes de programación lógico funcionales combinan ambas características, entre otras.

La combinación de los paradigmas lógico y funcional se puede realizar teniendo en cuenta distintos enfoques, por lo que han surgido distintas propuestas de combinación y diferentes lenguajes de programación que las implementan. Una selección de los primeros lenguajes lógico funcionales es LOGLISP [RS82], FUNLOG [SY86], K-LEAF [LPB$^+$87], Babel [Mor89, MR92] y Escher [Llo95]. Algunas propuestas y estudios de lenguajes lógico funcionales se describen en [DL86, Rod02, Han07, AHH$^+$02, Han13].

En general la combinación de estos dos paradigmas toma un lenguaje y lo extiende con características de otro. Entre los lenguajes lógicos que se extienden con características funcionales se encuentra Ciao Prolog [HBC$^+$12], que permite definir funciones que son traducidas a predicados mediante un preprocesador. Un estudio de los principios operacionales y técnicas de implementación utilizadas para la integración de las funciones en la programación lógica puede encontrarse en [Han94]. Por otro lado, las características lógicas se pueden integrar en un paradigma funcional mediante la combinación del mecanismo de resolución con la evaluación perezosa de los lenguajes funcionales. Dentro de esta familia se encuentran los lenguajes lógico funcionales más desarrollados en estos momentos: $\mathcal{TOY}$ [TOY12] y `Curry`

[Cur12]. Ambos lenguajes comparten sus fundamentos y tienen características comunes como la evaluación perezosa, las funciones de orden superior, el sistema de tipos, funciones indeterministas (funciones que pueden devolver distintos valores para los mismos argumentos) y también permiten que las funciones puedan llamarse con variables en los argumentos para que el sistema busque los correspondientes valores de salida, haciendo así un uso reversible de las funciones.

Además de estas características comunes, `Curry` tiene como características propias la residuación [Han92] y la búsqueda encapsulada [HS98]. $\mathcal{TOY}$ tiene como característica propia la incorporación de distintos tipos de restricciones: de igualdad y desigualdad [KLMR92, AGL94, LS99a], sobre los números reales [HLSU97, AHLU96], sobre el dominio finito de los números enteros [EV05, FHS05a, FHS05b] y sobre el dominio de conjuntos finitos [ECS12].

La semántica de un lenguaje declarativo determina el modelo de cómputo desde diversos puntos de vista. De este modo, los lenguajes $\mathcal{TOY}$ y `Curry` adoptan una semántica *no estricta*, donde la evaluación de una función puede llevarse a cabo aunque no todos sus argumentos estén completamente evaluados. Además, la combinación del indeterminismo con la evaluación perezosa puede producir situaciones en las cuales se necesita establecer el comportamiento del sistema. Por ejemplo, cuando se pasan argumentos indeterministas a funciones que contienen varias apariciones de dicho argumento indeterminista. Se puede determinar que todas las copias del argumento compartan el mismo valor o que cada copia obtenga su valor de forma independiente. Con respecto a esta situación se encuentran las semánticas *call-time choice* y *run-time choice* [Hus93] (aunque recientemente han surgido nuevas propuestas [RR14]). $\mathcal{TOY}$ y `Curry` tienen una semántica *call-time choice*. Una explicación de esta situación en $\mathcal{TOY}$, acompañada de un ejemplo, se mostró en la subsección 1.3.2.

Con respecto al mecanismo operacional, los lenguajes funcionales se basan en la reescritura [BN98] y los lenguajes lógicos en la unificación y resolución [CM03]. El resultado de combinar la reescritura y la unificación es el estrechamiento o *narrowing*. Existen varias estrategias estrechamiento [Han94, HK96, Fer92]. $\mathcal{TOY}$ y `Curry` tienen como mecanismo operacional el estrechamiento perezoso guiado por la demanda o *needed narrowing* [LLR93, AEH94]. La idea de este estrechamiento es retrasar o suspender la evaluación de los argumentos de las funciones hasta que sean necesarios para calcular el resultado de la evaluación de la función. En $\mathcal{TOY}$ el estrechamiento se combina con la evaluación de restricciones y en `Curry` con la residuación. En los trabajos [GHLR96, GHLR99] se desarrolla una semántica operacional que se presenta como cálculo de estrechamiento para la resolución de objetivos. En estos trabajos se trata la igualdad, aunque no la desigualdad, que se incorporó posteriormente en los trabajos [SH98, LS99a].

Un marco teórico adecuado para FLP con evaluación no estricta e indeterminismo es la lógica de reescritura basada en constructoras *CRWL Constructor-based ReWriting Logic* [GHLR96, GHLR99, Rod02]. Esta lógica se define en forma de cálculo de pruebas que permite derivar valores a los que se puede reducir una expresión. El marco *CRWL* ha mostrado ser muy adecuado para el estudio formal de los lenguajes lógico funcionales perezosos soportando indeterminismo y funciones no estrictas, como muestra la variedad de extensiones que se han llevado a cabo: orden superior [GHR97], tipos polimórficos [GHR01], constructoras de datos algebraicas [AR97a, AR97b, AR01], tipos ordenados [AGG96, AG97], sistemas de res-

tricciones genéricas y sobre multiconjuntos [ALR98, ALR99], objetos [MR01], desigualdades sintácticas [LS99a, SH98], el fallo finito [LS00, LS02, LS03, LS04b, SH06], depuración declarativa [Cab04, CLR01, CR02, CR04] y una extensión genérica de *CRWL* con razonamiento sobre restricciones [LRV07]. La estrategia de estrechamiento de $\mathcal{TOY}$ está basada en [LLR93] y la adecuación a *CRWL* se establece formalmente en [LS99a, SH98, Vad03]. En los trabajos [Pal02, Pal07] se compara la lógica de reescritura desarrollada por Meseguer [Mes92] y la lógica *CRWL*.

2.3 Programación lógico funcional con restricciones

El primer intento de combinar la programación lógica con restricciones y la programación funcional fue el esquema $CFLP(\mathcal{D})$, propuesto por J. Darlington, Y.K. Guo y H. Pull [DGP91, DGP92]. La idea de este enfoque se puede describir más o menos por la ecuación $CFLP(\mathcal{D}) = CLP(FP(\mathcal{D}))$, que pretende significar que un lenguaje $CFLP$ definido sobre el dominio de restricciones $\mathcal{D}$ se entiende como un lenguaje CLP sobre un dominio de restricciones extendido $FP(\mathcal{D})$ cuyas restricciones incluyen ecuaciones entre expresiones que contienen funciones definidas por el usuario, con objeto de ser resueltas por el mecanismo de estrechamiento.

El esquema $CFLP$ propuesto por F.J. López-Fraguas en [LF92, LF94] proporcionó resultados sobre semánticas declarativas de programas $CFLP$, y fue la base de los trabajos [LRV05, LRV07] que proponen un nuevo esquema $CFLP(\mathcal{D})$ parametrizado por un dominio. Las principales novedades de este nuevo esquema son una nueva formalización de los dominios de restricciones para $CFLP$, una nueva noción de interpretación para los programas $CFLP$, y una nueva lógica de reescritura parametrizada por un dominio de restricciones $CRWL(\mathcal{D})$. Esta nueva lógica de reescritura proporciona una caracterización lógica de la semántica del programa y es una extensión de la lógica de reescritura *CRWL* [GHLR96, GHLR99].

Existen varios métodos de resolución de objetivos basados en estrechamiento que son correctos y completos con respecto a distintas formalizaciones de las semánticas declarativas de programas. Estos métodos de resolución de objetivos se presentan como un cálculo que contiene reglas de transformación que deducen formas resueltas a partir de objetivos iniciales. Así el cálculo de estrechamiento perezoso $CLNC$ (*Constraint Lazy Narrowing Calculus*) presentado en [LRV04] es correcto y completo con respecto a la semántica de la lógica de reescritura *CRWL*. En [EFH$^+$09] se presentó el cálculo perezoso $CCLNC(\mathcal{D})$ para la resolución de objetivos teniendo en cuenta la cooperación de distintos dominios de restricciones. Este cálculo es correcto y completo, con ciertas limitaciones, con respecto a la instancia $CFLP(\mathcal{D})$ del esquema genérico $CFLP$.

Dentro del contexto de la programación lógico funcional con restricciones se encuentran dos tesis doctorales que son referentes de la cooperación entre resolutores en este ámbito: la tesis de Mircea Marin [Mar00] y la tesis de Petra Hofstedt [Hof01].

En la tesis de Mircea Marin [Mar00] se desarrolló un cálculo que servía como semántica operacional de lenguajes lógico funcionales de orden superior con resolución de restricciones. Marin definió un esquema $CFLP$ que combinaba la aproximación que hizo Monfroy a la cooperación entre resolutores en CLP [Mon96] con un cálculo de estrechamiento perezo-

so similar al cálculo desarrollado en [LRV04]. Para diseñar este cálculo definió un esquema `CFLP(X,S,C)` que describía un sistema compuesto por un intérprete lógico funcional, cuya semántica operacional era el cálculo de estrechamiento perezoso `C` y un resolutor de restricciones distribuido para resolver restricciones sobre un dominio de restricciones `X`. Este resolutor estaba definido por una colección de resolutores que cooperaban de acuerdo con la estrategia `S`.

Una instancia de este esquema fue el sistema distribuido `CFLP` [Mar00, MI00, MIS01], que estaba compuesto por: un intérprete lógico funcional que se ejecutaba en una máquina, una serie de resolutores de restricciones ejecutándose posiblemente en distintas máquinas, y un componente especial llamado planificador de restricciones (*constraint scheduler*), que implementaba la estrategia S para coordinar la cooperación entre resolutores. El sistema `CFLP` estaba escrito en *Mathematica* y resolvía ecuaciones con la colaboración de varios resolutores de ecuaciones, tales como un resolutor polinómico, resolutores de ecuaciones diferenciales y resolutores de ecuaciones lineales basados en el algoritmo Simplex. Una nueva versión de `CFLP` fue el sistema distribuido llamado `Open CFLP` [KMI01, KMIC02, KMI03]. En `Open CFLP` los resolutores no se fijaban *a priori* y se podían obtener dinámicamente de un entorno abierto como Internet. Los trabajos anteriormente mencionados y esta tesis se diferencian principalmente en la naturaleza de los resolutores. Su enfoque utiliza resolutores especializados en ecuaciones sobre dominios específicos, como resolutores de ecuaciones polinómicos o resolutores de ecuaciones diferenciales, mientras que en nuestra aproximación los resolutores que cooperan son el resolutor de Herbrand, los resolutores de dominios finitos y los conjuntos finitos. Por esta razón, tanto la naturaleza de la cooperación como las estrategias de cooperación son distintas.

Continuando en el ámbito $CFLP$, la tesis [Hof01] junto a los trabajos [Hof00a, Hof00b] propone un esquema general para la cooperación entre resolutores de restricciones. La base de este sistema es una interfaz uniforme para resolutores de restricciones que permite una especificación formal y minuciosa de intercambio de información entre los resolutores de restricciones. Es decir, la cooperación entre los resolutores se hace a través de una interfaz que permite intercambiar información entre resolutores de distintos dominios. Esta interfaz consiste en, por una parte, una función usada para propagar restricciones a sus respectivos almacenes y, por otra, un conjunto de funciones que describen la proyección de un almacén a otros sistemas de restricciones. Concretamente, propagar una restricción (primitiva) consiste en enviarla a su almacén y seguidamente invocar al correspondiente resolutor. Proyectar un almacén consiste en consultar el contenido de un almacén y deducir restricciones para otros dominios. Sobre esta interfaz se desarrolló un mecanismo abierto y flexible de combinación de resolutores. La combinación es abierta en el sentido de que cada vez que surge un sistema de restricciones con un resolutor asociado, que satisface ciertas propiedades, puede ser fácilmente incorporado en el sistema general, independientemente de su dominio y del lenguaje en el cual está implementado. Es flexible porque es posible definir distintas estrategias para la cooperación de resolutores independientes. Una extensión a este marco y la implementación `Meta-S` se describió en [FHM03b, FHM03a, FHR05]. `Meta-S` permite la integración de resolutores externos a través de un meta-resolutor que tiene vinculados resolutores de diferentes dominios. En particular, un resolutor para ecuaciones lineales aritméticas, resolutores

de dominios finitos de distintos tipos y un resolutor aritmético de intervalos.

En el trabajo [HP07] se presenta una aproximación general para la integración de lenguajes declarativos y sistemas de restricciones. Esta integración requiere los siguientes pasos: en primer lugar identificar el lenguaje de restricciones. Seguidamente extender el lenguaje con restricciones de otros dominios de tal forma que sea posible generar restricciones durante la evaluación del programa. Es decir, el mecanismo de evaluación del lenguaje original (la resolución en la programación lógica o estrechamiento para lenguajes lógico funcionales) se amplía para recopilar las restricciones de otros resolutores. Por último, es necesario definir las funciones del interfaz de acuerdo a los requerimientos del marco general del resolutor de cooperación. En esta aproximación el sistema general para la cooperación entre resolutores permite tratar restricciones híbridas sobre diferentes dominios.

El lenguaje de programación `Curry` [Cur12, Han13] combina características de la programación funcional, lógica y concurrente. Pero `Curry` y su implementación PAKCS no han considerado la cooperación entre resolutores.

La cooperación en $\mathcal{TOY}$ utiliza la idea de proyecciones de `Meta-S` pero, como novedad en $\mathcal{TOY}$, se usa un mecanismo de comunicación denominado *puente* que permite computar proyecciones. En `Meta-S` la proyección de una restricción de un almacén a otro depende de las variables en común de ambos almacenes. Como $\mathcal{TOY}$ es un lenguaje fuertemente tipado, no permite que una variable esté contenida en restricciones de distintos dominios. Esta y otras diferencias entre $\mathcal{TOY}$ y `Meta-S` serán discutidas en detalle en el capítulo 5.

2.4 $\mathcal{TOY}$: un lenguaje y sistema $CFLP$

Como ya se ha anticipado, $\mathcal{TOY}$ es un lenguaje y sistema lógico funcional con restricciones $CFLP$ que toma a Prolog como representante lógico y a Haskell como representante funcional. $\mathcal{TOY}$ ha contribuido notablemente al desarrollo de investigaciones en el grupo de programación declarativa (GPD) de la facultad de informática de la UCM.

$\mathcal{TOY}$ lleva a cabo una traducción a Prolog de los programas fuente de acuerdo con la estrategia guiada por la demanda que se propone en [LLR93]. Además, la mayoría de las características de la lógica *CRWL* están incorporadas en el sistema $\mathcal{TOY}$. Las restricciones también se transforman de acuerdo con la estrategia guiada por la demanda en otras más simples que procesa un resolutor de restricciones.

Como se ha detallado en la introducción, $\mathcal{TOY}$ permite utilizar funciones indeterministas y variables lógicas de orden superior, e incorpora muchas optimizaciones en la traducción de funciones y en la resolución de igualdades y desigualdades. También incorpora un inferidor de tipos similar al de Haskell y permite declarar tipos para las funciones y predicados.

Los últimos trabajos realizados en el grupo GPD que han utilizado $\mathcal{TOY}$ como sistema son los siguientes.

- La tesis [dVV08] propone un marco teórico que permite caracterizar la semántica declarativa y operacional de los lenguajes de programación lógico funcionales perezosos con restricciones. Para ello se desarrolló un esquema genérico $CFLP(\mathcal{D})$ sobre un dominio arbitrario de restricciones $\mathcal{D}$. El desarrollo de la lógica de reescritura con restricciones

$CRWL(\mathcal{D})$, también definida de forma paramétrica sobre un dominio de $\mathcal{D}$, proporciona un nuevo marco lógico que caracteriza la semántica de los programas en el esquema $CFLP(\mathcal{D})$. Con respecto a la semántica operacional, en dicha tesis se proponen dos métodos de resolución de objetivos. El primero es el cálculo de transformación de objetivos $CLNC(\mathcal{D})$ basado en el estrechamiento perezoso con restricciones, y el segundo integra árboles definicionales, lo que asegura una óptima selección de los pasos de estrechamiento necesarios. Además describe el sistema $\mathcal{TOY}$ (FD) como una instancia de $CFLP(FD)$.

- La tesis [Mar12] propone tres sistemas de tipos adecuados para la programación lógico funcional pues el sistema de tipos de Damas y Milner no maneja adecuadamente algunas de las principales características de los lenguajes lógico-funcionales, como son los patrones de orden superior o las variables libres. Los sistemas de tipos propuestos, que tratan diferentes mecanismos de cómputos lógico funcionales, dan solución a los mencionados problemas, proporcionando resultados técnicos de corrección. Además, en esta tesis también se han desarrollado implementaciones de los sistemas de tipos, integrándolos como fase de comprobación de tipos en distintas ramas de desarrollo del sistema $\mathcal{TOY}$.

- La tesis [Rom11] desarrolla dos esquemas paramétricos de programación declarativa con incertidumbre: una extensión con cualificación y proximidad del marco CLP; y una extensión con cualificación de programas $CFLP$ de primer orden. En esta tesis se optó por la utilización del sistema $\mathcal{TOY}$ como parte de la implementación. De hecho el intérprete de $\mathcal{TOY}$ se extiende mediante una serie de comandos que facilitan la carga de programas cuantitativos y la resolución de objetivos cuantitativos.

- La tesis [Cas14] tiene como principal objetivo fomentar el uso de $CFLP(FD)$. Para conseguir este objetivo la tesis está dividida en tres partes: una mejora del rendimiento de resolución de $\mathcal{TOY}(\mathcal{FD})$, una descripción de dos aplicaciones reales que se resuelven utilizando $\mathcal{TOY}(\mathcal{FD})$ y una comparativa en profundidad sobre el modelado y resolución de varios problemas de optimización utilizando sistemas $CP(FD)$ algebraicos, C++ $CP(FD)$, $CLP(FD)$ y $CFLP(FD)$, donde el sistema $CFLP(\mathcal{FD})$ es $\mathcal{TOY}(\mathcal{FD})$.

- En la tesis [Gar14] se estudia el diseño de técnicas para la detección y diagnosis de errores en el campo de las bases de datos y en particular, en consultas a bases de datos. Dentro del ámbito de las bases de datos, se centra en las bases de datos deductivas, relacionales y semiestructuradas. En particular, utiliza patrones de orden superior y las capacidades de generación y prueba propias de la programación lógico funcional de $\mathcal{TOY}$ para localizar errores en las consultas y obtener casos de prueba en forma de documentos XML.

Capítulo 3

Dominios de restricciones y resolutores

En este capítulo se presentan las definiciones formales de dominio de restricciones y resolutor en el contexto $CFLP$ del lenguaje $\mathcal{TOY}$. Para ello, se parte del esquema $CFLP(\mathcal{D})$ elaborado en el trabajo [EFH$^+$09], que es una extensión del esquema presentado en [LRV07] y que sirve como marco lógico y semántico para programas perezosos posiblemente indeterministas y de orden superior sobre un dominio de restricciones paramétrico $\mathcal{D}$. A continuación se definen los dominios que se van a tratar en esta tesis ($\mathcal{R}$, $\mathcal{FD}$, $\mathcal{FS}$ y $\mathcal{H}$) y sus resolutores asociados. En concreto, el resolutor $\mathcal{R}$ se define como un resolutor de caja negra donde los cómputos no están definidos en la semántica de esta tesis. Por lo tanto, se postula que cumple las condiciones que se requieren para los resolutores en la definición genérica. El resolutor $\mathcal{H}$ se define como un resolutor de caja transparente mediante una técnica denominada *sistema de transformación de almacenes*. Los resolutores $\mathcal{FD}$ y $\mathcal{FS}$ se implementan combinando ambas técnicas. Es decir, ambos resolutores están divididos en dos capas: la primera capa es una caja transparente cuya finalidad es anticipar el fallo; la segunda capa es un resolutor de caja negra que resuelve restricciones utilizando resolutores proporcionados por sistemas CLP. Queremos indicar que este capítulo es fundamental para la comprensión de la tesis pero no es una contribución propia de la misma excepto en los elementos detallados en la sección 1.2.

3.1 Definiciones

Para poder definir los dominios de restricciones y sus resolutores asociados en nuestro modelo $CFLP$ se necesitan algunas definiciones previas que se exponen a continuación.

3.1.1 Signaturas y tipos

Se asume una *signatura universal* de la forma $\Omega = \langle TC,\ BT,\ DC,\ DF,\ PF\rangle$, donde:

- $TC = \bigcup_{n\in\mathbb{N}} TC^n$ es una familia de conjuntos enumerables y mutuamente disjuntos de *constructoras de tipos* (*Type Constructors*), indexados por aridades.
- BT es un conjunto de *tipos base* (*Base Types*).

- $DC = \bigcup_{n\in\mathbb{N}} DC^n$ es una familia de conjuntos numerables y mutuamente disjuntos de *constructoras de datos* (*Data Constructors*), indexados por aridades.

- $DF = \bigcup_{n\in\mathbb{N}} DF^n$ es una familia de conjuntos numerables y mutuamente disjuntos de *símbolos de funciones definidas* (*Defined Functions*), indexados por aridades.

- $PF = \bigcup_{n\in\mathbb{N}} PF^n$ es una familia de conjuntos numerables y mutuamente disjuntos de *símbolos de funciones primitivas* (*Primitive Functions*), indexados por aridades.

Los tipos base y los símbolos de funciones primitivas están relacionados con los dominios de restricciones específicos. Por cada familia específica de tipos base $SBT \subseteq BT$ y de símbolos de funciones primitivas $SPF \subseteq PF$ se obtiene una *signatura específica* $\Sigma = \langle TC,\ SBT,\ DC,\ DF,\ SPF\rangle$. Esta signatura específica contiene las constructoras de tipo y de datos y los símbolos de función definida de la signatura universal Ω, pues son comunes a todos los dominios y se pueden utilizar en cualquier programa. Cada dominio de restricciones utiliza una signatura específica.

Por cada dominio de signatura específica Σ se utilizan expresiones que pueden contener valores de tipos básicos. Utilizaremos $\mathcal{B} = \{\mathcal{B}_d\}_{d\in SBT}$ para referirnos a una familia indexada donde cada $\mathcal{B}_d$ es un conjunto no vacío de *valores básicos* de tipo $d \in SBT$. En lo sucesivo, usaremos letras $u, v, ...$ para referenciar a los valores básicos y escribiremos $u \in \mathcal{B}$ en vez de $u \in \bigcup_{d\in SBT} \mathcal{B}_d$ para simplificar la notación. Como se anticipó en la introducción, los conjuntos de expresiones y patrones definidos sobre $\mathcal{B}$ se denotan como $Exp_\Sigma(\mathcal{B})$ y $Pat_\Sigma(\mathcal{B})$, respectivamente. El conjunto de todas la expresiones y patrones básicos sobre $\mathcal{B}$ se denotan respectivamente como $GExp_\Sigma(\mathcal{B})$ y $GPat_\Sigma(\mathcal{B})$. El conjunto de todos los patrones básicos es el *universo de valores* sobre $\mathcal{B}$ y se denota como $\mathcal{U}_\Sigma(\mathcal{B})$.

Aunque el dominio $\mathcal{FD}$ se definirá formalmente en la subsección 3.3, se puede anticipar como ejemplo que su tipo base específico es SBT `= {int}`, y contiene símbolos de funciones primitivas específicas tales como la relación de orden o la suma (`{#<,#+}` $\subset SPF$). Además, el conjunto de los valores básicos de $\mathcal{FD}$ es el conjunto de los números enteros $\mathbb{Z}$. Ejemplos de DC y TC de la signatura universal Ω son las constructoras de datos `true` y `false` y la constructora de tipo `bool`, todas de aridad 0. Los símbolos `smm` y `toInt` mostrados en el ejemplo 2 de la sección 1.3 son ejemplos de funciones definidas (por el usuario) del dominio $\mathcal{FD}$.

Entre expresiones, y por lo tanto entre patrones, se puede definir un orden de la siguiente manera: $\sqsubseteq$ es el menor orden parcial tal que $\bot \sqsubseteq e$ para toda expresión e y además $(e\, e_1) \sqsubseteq (e'\, e'_1)$ cuando $e \sqsubseteq e'$ y $e_1 \sqsubseteq e'_1$. Intuitivamente $e \sqsubseteq e'$ significa que la información proporcionada por e' es mayor o igual que la información proporcionada por e. Por ejemplo, la cadena $\bot \sqsubseteq (0 : \bot) \sqsubseteq (0 : (1 : \bot)) \sqsubseteq (0 : (1 : (2 : \bot))) \sqsubseteq \cdots$ representa un orden entre listas definidas mediante la constructora de listas `(:)`.

Las expresiones se pueden clasificar teniendo en cuenta su estructura. Así, se dice que una expresión es *flexible* si tiene la forma $(X\, \overline{e}_m)$ donde X es una variable y $m \geq 0$. Una expresión es *rígida* si es de la forma $(h\ \overline{e}_m)$ con $h \in DC \cup DF \cup SPF$. Una expresión rígida es *activa* si es de la forma $(h\, \overline{e}_m)$ con $h \in DF^n \cup SPF^n$ y $m \geq n$ y *pasiva* en caso contrario. La idea es que cualquier expresión pasiva tiene la apariencia más externa de un patrón, pero

internamente puede tener subexpresiones activas. Por ejemplo, una tupla $(e_1, \ldots, e_n)$ es una expresión pasiva, aunque alguna e_i sea activa.

Con respecto a los tipos se sigue la disciplina de tipos estática basada en el sistema de tipos de Hindley, Milner y Damas [Hin69, Mil78, DM82]. El trabajo [GHR01] es un detallado estudio de la disciplina de tipos en el contexto FLP.

Un *tipo* $\tau \in Type_\Sigma$ tiene la sintaxis $\tau ::= A \mid d \mid (c_t\ \overline{\tau}_n) \mid (\tau_1, \ldots, \tau_n) \mid (\tau_1 \to \tau_0)$, donde $A \in \mathcal{TV}ar$ ($\mathcal{TV}ar$ es un conjunto infinito de *variables de tipo*), $d \in SBT$ y $c_t \in TC^n$. La constructora de tipo "$\to$" asocia por la derecha y por convenio se omiten los paréntesis cuando no hay ninguna ambigüedad. Los tipos $c_t\ \overline{\tau}_n$, $(\tau_1, \ldots, \tau_n)$ y $\tau_1 \to \tau_0$ representan valores construidos, tuplas y funciones, respectivamente. Un tipo sin ninguna aparición de $\to$ es un *tipo construido* (en inglés *datatype*).

En cualquier signatura Σ los símbolos de función tienen una declaración de tipo principal que representa al tipo más general. Con mayor precisión:

- Cada $c \in DC^n$ debe tener asociada una declaración de *tipo principal* de la forma $c :: \overline{\tau}_n \to c_t\, \overline{A}_k$, $(n, k \geq 0)$, donde $A_1, \ldots, A_k$ son variables de tipo distintas, $c_t \in TC^k$, $\tau_1, \ldots, \tau_n$ son tipos construidos, $\bigcup_{i=1}^{n} tvar(\tau_i) \subseteq \{A_1, \ldots, A_k\}$ y $tvar(\tau)$ es el conjunto de variables de tipo contenidas en τ. Esta propiedad se denomina *propiedad de transparencia*.

- Cada $f \in DF^n$ debe tener asociada una declaración de tipo principal de la forma $f :: \overline{\tau}_n \to \tau$, donde $\tau_i\, (1 \leq i \leq n)$ y τ son tipos arbitrarios.

- Cada $p \in SPF^n$ debe tener asociada una declaración de tipo principal de la forma $p :: \overline{\tau}_n \to \tau$, donde τ_1, ..., τ_n, τ son tipos construidos y τ no es una variable de tipo.

Los lenguajes de programación que adoptan una disciplina de tipos estática comprueban que todas las expresiones incluidas en los programas estén bien tipadas. La comprobación de los tipos se basa en dos clases de información: en primer lugar, los tipos principales de los símbolos que pertenecen a la signatura y, en segundo lugar, los tipos de las variables que aparecen en las expresiones. Para representar esta segunda clase de información se usa un *entorno de tipos* $\Gamma = \{X_1 :: \tau_1, \ldots, X_n :: \tau_n\}$ donde la variable X_i tiene tipo τ_i para todo $1 \leq i \leq n$. Siguiendo las ideas derivadas de los trabajos de Hindley, Milner y Damas [Hin69, Mil78, DM82], es posible definir reglas de inferencia de tipos para derivar *juicios de tipos* de la forma $\Sigma, \Gamma \vdash_{WT} e :: \tau$ que indican que la afirmación $e :: \tau$ ("e tiene tipo τ") se puede deducir de los tipos para los símbolos y variables que figuran en Σ y Γ respectivamente. Una expresión e está *bien tipada (en inglés well-typed)* si y solamente si $\Sigma, \Gamma \vdash_{WT} e :: \tau$ puede ser derivada para algún tipo τ usando la signatura Σ y algún entorno de tipos adecuado Γ. Aunque τ no es único en general, se puede demostrar que existe *el tipo más general* o *tipo principal de* e. Este tipo es único salvo renombramiento de variables. En la práctica, los tipos principales de expresiones bien tipadas se pueden inferir automáticamente por los compiladores. La notación $\Sigma, \Gamma \vdash_{WT} e :: \tau$ se simplifica a $e :: \tau$ cuando Σ y Γ se pueden deducir por el contexto. Además, el valor indefinido $\bot \in DC^0$ pertenece a cualquier tipo $(\bot :: A)$.

Como se ha comentado en la introducción (subsección 1.3.1), $\mathcal{TOY}$ es un lenguaje fuertemente tipado, en el cual los tipos principales de las constructoras de datos se introducen mediante declaraciones de constructoras de tipos, los tipos principales de las funciones primitivas están predefinidos y los tipos principales de las funciones definidas se pueden declarar o inferir. En el siguiente ejemplo, en la línea 1, se muestra la definición del tipo construido correspondiente a las listas, que pertenece a la signatura universal Ω. Esta declaración está predefinida en el sistema. Las declaraciones de las líneas 2 y 3 no están predefinidas y se deben incorporar en el programa para su posterior uso.

```
1 data [A] = [] | (A:[A])
2 data pairOf A = pair A A
3 data either A B = left A | right B
```

El tipo construido de las listas `[A]` se define en la línea 1 mediante las constructoras de datos `[] :: [A]` de lista vacía, y la constructora `(:) :: A -> [A] -> [A]`. Ambas cumplen la propiedad de transparencia. Igualmente, las constructoras de datos `pair`, `left` y `right` cumplen la propiedad de transparencia pues sus correspondientes tipos inferidos a partir de su definición en la líneas 2 y 3 son los siguientes: `pair :: A -> A -> (pairOf A)`, `left :: A -> (either A B)` y `right :: A -> (either B A)`. En este caso `pairOf` y `either` son constructoras de tipo.

Otra utilidad de $\mathcal{TOY}$ son los *alias* de tipos que permiten declarar un nuevo identificador como abreviatura de un tipo más complejo. Las siguientes líneas corresponden a alias de tipos utilizados en el ejemplo 1 de la introducción a $\mathcal{TOY}$ vista en el capítulo 1:

```
1 type point  = (real,real)
2 type region = point -> bool
```

Los alias de tipos no pueden ser recursivos y no se consideran parte de la signatura. Siguiendo el mismo ejemplo de la introducción a $\mathcal{TOY}$, la declaración de la función definida `rectangle :: point -> point -> region` representa a la declaración `rectangle :: (real,real) -> (real,real) -> (real,real) -> bool`.

3.1.2 Sustituciones

Una *sustitución* $\sigma \in Sub_\Sigma(\mathcal{B})$ sobre $\mathcal{B}$ es un conjunto de asignaciones de variables ($\mathcal{V}ar$) a patrones ($Pat_\Sigma(\mathcal{B})$). La notación $e\sigma$ representa la sustitución $\sigma(e)$ para cualquier expresión $e \in Exp_\Sigma(\mathcal{B})$. El símbolo ε representa la sustitución identidad y $\sigma\theta$ representa la *composición* de σ con θ, tal que $e(\sigma\theta) = (e\sigma)\theta$ para cualquier expresión e. Una sustitución σ tal que $\sigma\sigma = \sigma$ se dice que es *idempotente*. El conjunto de variables que componen el *dominio* de la sustitución σ y el conjunto de variables que forman el *rango* de la sustitución σ se definen como: $vdom(\sigma) = \{X \in \mathcal{V}ar \mid X\sigma \neq X\}$ y $vran(\sigma) = \bigcup_{X \in vdom(\sigma)} var(X\sigma)$, donde $var(X\sigma)$ es el conjunto de variables que están contenidas en $(X\sigma)$. La notación usual para sustituciones es $\sigma = \{X_1 \mapsto t_1, \ldots, X_n \mapsto t_n\}$, con $vdom(\sigma) = \{X_1, \ldots, X_n\}$ y $X_i\sigma = t_i$ para todo $1 \leq i \leq n$.

Dadas dos expresiones e y e', si $e' = e\sigma$ para alguna sustitución σ, entonces se dice e' es una *instancia* de e o también que e es más general que e' y se denota $e \preceq e'$.

Las sustituciones se pueden restringir a un conjunto de variables: $\sigma\restriction_{\mathcal{X}}$ es la sustitución σ' tal que para el conjunto de variables $\mathcal{X}$ se cumple $vdom(\sigma') = \mathcal{X}$ y $X\sigma' = X\sigma$ para todo $X \in \mathcal{X}$. Dadas dos sustituciones σ y θ, la notación $\sigma =_{\mathcal{X}} \theta$ indica que $\sigma \restriction_{\mathcal{X}} = \theta \restriction_{\mathcal{X}}$ y significa que las dos sustituciones son iguales con respecto al conjunto de variables $\mathcal{X}$. En general, abreviaremos $\mathcal{V}ar \setminus \mathcal{X}$ como $\setminus\mathcal{X}$. Así, de este modo, $\sigma =_{\mathcal{V}ar\setminus\mathcal{X}} \theta$ se abrevia como $\sigma =_{\setminus\mathcal{X}} \theta$ y significa que las dos sustituciones son iguales con respecto a todas las variables excepto las variables contenidas en el conjunto $\mathcal{X}$.

Dadas dos sustituciones σ y θ, la sustitución $\sigma\star\theta$ es la *aplicación* de θ sobre σ restringida a las variables de σ, es decir, $\sigma\theta \restriction_{vdom(\sigma)}$. Por lo tanto, para cualquier variable X se cumple $X(\sigma \star \theta) = X\sigma\theta$ si $X \in vdom(\sigma)$ y $X(\sigma \star \theta) = X$ en otro caso.

La sustituciones se pueden comparar de dos formas distintas. Sean σ y σ' dos sustituciones, entonces se dice que:

- σ es más general que σ' sobre un conjunto de variables $\mathcal{X}$ ($\sigma \preceq_{\mathcal{X}} \sigma'$) si y solamente si $\sigma\theta =_{\mathcal{X}} \sigma'$ para alguna sustitución θ. La notación $\sigma \preceq \sigma'$ significa que σ *es más general* que σ' o que σ' *es una instancia* de σ para todas las variables.

- σ tiene menos información que σ' sobre un conjunto de variables $\mathcal{X}$ ($\sigma \sqsubseteq_{\mathcal{X}} \sigma'$) si y solamente si $X\sigma \sqsubseteq X\sigma'$ para todo $X \in \mathcal{X}$ donde $\sqsubseteq$ es el orden entre expresiones. La notación $\sigma \sqsubseteq \sigma'$ abrevia $\sigma \sqsubseteq_{\mathcal{V}ar} \sigma'$ y $\sigma \sqsubseteq_{\setminus\mathcal{X}} \sigma'$ abrevia $\sigma \sqsubseteq_{\mathcal{V}ar\setminus\mathcal{X}} \sigma'$.

Las sustituciones se pueden definir análogamente para tipos de la siguiente manera: una *sustitución de tipos* es el conjunto de asignaciones de variables de tipos ($\mathcal{TV}ar$) a tipos ($Type_\Sigma$). La notación $\tau\sigma_t$ representa la sustitución $\sigma_t(\tau)$ para cualquier tipo τ. Además, $\tau \preceq \tau'$ significa que τ' es una instancia de τ o que τ es más general que τ'. Por ejemplo `[A]` $\preceq$ `[int->int]`.

3.1.3 Dominios de restricciones

Intuitivamente un dominio de restricciones proporciona valores y restricciones específicos. Existen diferentes propuestas para formalizar la noción de dominio de restricciones, entre las que se encuentra la definición del trabajo [EFH+09] que fue elaborada a partir del trabajo [LRV07] y es la que se utiliza en esta tesis.

Definición 1. (Dominio de restricciones)

Un *dominio de restricciones* $\mathcal{D}$ con signatura específica Σ se define como una estructura $\mathcal{D} = \langle \mathcal{B}^{\mathcal{D}}, \{p^{\mathcal{D}}\}_{p\in SPF}\rangle$ donde $\mathcal{B}^{\mathcal{D}} = \{\mathcal{B}^{\mathcal{D}}_d\}_{d\in SBT}$ y $p^{\mathcal{D}}$ es la *interpretación* de cada símbolo de función primitiva $p :: \overline{\tau}_n \to \tau$ de SPF^n. La interpretación $p^{\mathcal{D}}$ es un conjunto de tuplas de $(n+1)$ elementos del universo de valores del dominio $\mathcal{D}$ (donde $\mathcal{U}_{\mathcal{D}} = GPat_\Sigma(\mathcal{B}^{\mathcal{D}})$). La notación $p^{\mathcal{D}}\overline{t}_n \to t$ indica $(\overline{t}_n, t) \in p^{\mathcal{D}}$. El significado de $p^{\mathcal{D}}\overline{t}_n \to t$ es que la función primitiva p con argumentos $\overline{t}_n$ devuelve el resultado t en el dominio $\mathcal{D}$. Las interpretaciones de los símbolos de función primitiva deben satisfacer las siguientes condiciones:

1. **Polaridad**: para todo $p \in SPF$, $p^{\mathcal{D}}\overline{t}_n \to t$ se comporta de forma monótona con respecto a los argumentos $\overline{t}_n$ y antimonótona con respecto al resultado t.

   Formalmente: para todo $\overline{t}_n, \overline{t'}_n, t, t' \in \mathcal{U}_{\mathcal{D}}$ tal que $p^{\mathcal{D}}\overline{t}_n \to t$, $\overline{t}_n \sqsubseteq \overline{t'}_n$ y $t \sqsupseteq t'$, se cumple $p^{\mathcal{D}}\overline{t'}_n \to t'$.

2. **Radicalidad**: para todo $p \in SPF$, si los argumentos dados a $p^{\mathcal{D}}$ tienen suficiente información para devolver un resultado que no sea $\bot$, los mismos argumentos son suficientes para devolver un resultado total.

   Formalmente: Para todo $\overline{t}_n, t \in \mathcal{U}_{\mathcal{D}}$, si $p^{\mathcal{D}}\overline{t}_n \to t$ entonces o bien $t = \bot$ o bien existe algún $t' \in \mathcal{U}_{\mathcal{D}}$ total tal que $p^{\mathcal{D}}\overline{t}_n \to t'$ y $t' \sqsupseteq t$.

3. **Buen tipado**: para todo $p \in SPF$, el comportamiento de $p^{\mathcal{D}}$ está bien tipado con respecto a cualquier instancia monomórfica del tipo principal de p.

   Formalmente: para cualquier instancia de tipo monomórfica $(\overline{\tau'}_n \to \tau') \succeq (\overline{\tau}_n \to \tau)$ y para todo $\overline{t}_n, t \in \mathcal{U}_{\mathcal{D}}$ tal que $\Sigma \vdash_{WT} \overline{t}_n :: \overline{\tau'}_n$ y $p^{\mathcal{D}}\overline{t}_n \to t$, entonces se cumple $\Sigma \vdash_{WT} t :: \tau'$.

4. **Igualdad estricta**: la primitiva `==` (en caso de que pertenezca a SPF) se interpreta como la *igualdad estricta* sobre $\mathcal{U}_{\mathcal{D}}$, de modo que para todo $t_1, t_2, t \in \mathcal{U}_{\mathcal{D}}$, se tiene que $t_1 ==^{\mathcal{D}} t_2 \to t$ si y sólo si se cumple alguno de los tres casos siguientes:

   (a) t_1 y t_2 son uno y el mismo patrón total, y $t = true$.

   (b) t_1 y t_2 no tienen ningún límite superior común en $\mathcal{U}_{\mathcal{D}}$ con respecto al orden de información $\sqsubseteq$, y $t = false$.

   (c) $t = \bot$.

   La interpretación $==^{\mathcal{D}}$ satisface las condiciones de polaridad, radicalidad y buen tipado.

Los dominios $\mathcal{R}$, $\mathcal{FD}$, $\mathcal{FS}$ y $\mathcal{H}$ mencionados en la introducción se formalizan de acuerdo a la definición anterior en las secciones 3.2, 3.3, 3.4 y 3.5. El último apartado de la definición anterior establece una interpretación fija del símbolo `==` como la operación de igualdad estricta para cada dominio $\mathcal{D}$ cuya signatura específica incluya esta primitiva.

Para cualquier dominio $\mathcal{D}$ de signatura Σ, las siguientes notaciones $Exp_{\Sigma}(\mathcal{B}^{\mathcal{D}})$, $Pat_{\Sigma}(\mathcal{B}^{\mathcal{D}})$ y $Sub_{\Sigma}(\mathcal{B}^{\mathcal{D}})$ se abrevian como $Exp_{\mathcal{D}}$, $Pat_{\mathcal{D}}$ y $Sub_{\mathcal{D}}$ respectivamente.

En las siguientes secciones utilizaremos la noción de *extensión conservativa* de un dominio $\mathcal{D}$ que se define formalmente como: dados dos dominios $\mathcal{D}$ y $\mathcal{D}$' con respectivas signaturas Σ y Σ', se dice que $\mathcal{D}$' es una *extensión conservativa* de $\mathcal{D}$ si y solamente si se cumple:

1. $\Sigma \subseteq \Sigma'$, es decir $SBT \subseteq SBT'$ y $SPF \subseteq SPF'$.

2. Para todo $d \in SBT$ se cumple $\mathcal{B}^{\mathcal{D}'}_d = \mathcal{B}^{\mathcal{D}}_d$.

3. Para todo $p \in SPF^n$ que no sea == y para todo $\overline{t}_n, t \in \mathcal{U}_\mathcal{D}$, se tiene $p^{\mathcal{D}'}\,\overline{t}_n \to t$ si y solamente si $p^{\mathcal{D}}\,\overline{t}_n \to t$.

3.1.4 Restricciones

Como es habitual en la programación con restricciones, se definen las *restricciones* sobre un dominio $\mathcal{D}$ como fórmulas lógicas construidas a partir de restricciones atómicas usando conjunciones $\wedge$ y cuantificadores existenciales $\exists$. De manera más precisa, una restricción δ sobre el dominio $\mathcal{D}$ tiene la sintaxis $\delta ::= \alpha \mid (\delta_1 \wedge \delta_2) \mid \exists X\delta$, donde α es cualquier restricción atómica sobre $\mathcal{D}$ y $X \in \mathcal{V}ar$ es cualquier variable. Se permiten dos tipos de *restricciones atómicas* α sobre $\mathcal{D}$:

a) $\Diamond$ y $\blacklozenge$, que representan la verdad (éxito) y la falsedad (fallo) respectivamente; o

b) $p\ \overline{e}_n \to!\ t$ con $p \in SPF^n$, donde cada e_i es una expresión y t es un patrón total (es decir, sin apariciones de $\bot$). Intuitivamente, una restricción $p\,\overline{e}_n \to! t$ restringe el valor devuelto por la llamada $p\,\overline{e}_n$ a un patrón total que se ajusta a la forma de t.

A continuación se introduce notación para referirnos a una serie de tipos de restricciones que se utilizarán a lo largo de la tesis. $ACon_\mathcal{D}$ es el conjunto de restricciones atómicas y $Con_\mathcal{D}$ es el conjunto de todas las restricciones sobre un dominio $\mathcal{D}$. Por convenio, las restricciones de la forma $p\ \overline{e}_n \to! \mathit{true}$ se abrevian como $p\ \overline{e}_n$. En particular, las restricciones de igualdad estricta e_1 `==` e_2 y las restricciones de desigualdad estricta e_1 `/=` e_2 se entienden como abreviaturas de `(==)` $e_1\ e_2 \to!$ `true` y `(==)` $e_1\ e_2 \to!$ `false` respectivamente. Una *restricción primitiva atómica* sobre un dominio $\mathcal{D}$ es una restricción atómica de la forma $p\,\overline{t}_n \to! t$ donde $\overline{t}_n$ son patrones. $APCon_\mathcal{D}$ es el conjunto de restricciones primitivas atómicas. Una restricción $\pi \in Con_\mathcal{D}$ es *primitiva* si y solamente si todas las partes atómicas de π son primitivas. Por último, $PCon_\mathcal{D} \subseteq Con_\mathcal{D}$ es el conjunto de todas las restricciones primitivas sobre un dominio $\mathcal{D}$. Obsérvese que $APCon_\mathcal{D} = ACon_\mathcal{D} \cap PCon_\mathcal{D}$. En las secciones 3.2, 3.3 y 3.4 se muestra la sintaxis de las restricciones atómicas y primitivas atómicas de los dominios $\mathcal{R}$, $\mathcal{FD}$ y $\mathcal{FS}$ respectivamente.

Sean $\mathcal{D}$ y $\mathcal{D}$' dominios con signaturas Σ y Σ', respectivamente, y SPF y SPF' son los conjuntos de funciones primitivas de cada dominio. Si $\mathcal{D}$' es una extensión conservativa de $\mathcal{D}$, entonces se dice que una restricción $\delta \in Con_{\mathcal{D}'}$ es *SPF-restringida* si y solamente si todas sus partes atómicas tienen la forma $\Diamond$, $\blacklozenge$ o bien $p\,\overline{e}_n \to! t$, donde $p \in SPF^n$. Se denota como $Con_{\mathcal{D}'}\restriction_{SPF}$ al conjunto de restricciones *SPF-restringidas* sobre $\mathcal{D}$'. Los subconjuntos $APCon_{\mathcal{D}'}\restriction_{SPF} \subseteq ACon_{\mathcal{D}'}\restriction_{SPF} \subseteq Con_{\mathcal{D}'}\restriction_{SPF}$ se definen análogamente. En particular, $APCon_{\mathcal{D}'}\restriction_{SPF}$ es el conjunto de todas las restricciones primitivas atómicas SPF-restringidas sobre $\mathcal{D}$', y son o bien $\Diamond$ o bien $\blacklozenge$ o bien $p\,\overline{t}_n \to! t$, con $p \in SPF^n$, $\overline{t}_n$, $t \in Pat_{\mathcal{D}'}$ y t total.

3.1.5 Valoraciones y soluciones

Una *valoración* $\eta \in Val_\mathcal{D}$ sobre un dominio $\mathcal{D}$ es una sustitución de variables por valores básicos, $vran(\eta) \subseteq \mathcal{U}_\mathcal{D}$. Aquellas valoraciones que satisfacen una restricción π se denominan

soluciones de π. El conjunto de soluciones de la restricción π sobre un dominio $\mathcal{D}$ se denota como $Sol_{\mathcal{D}}(\pi) \subseteq Val_{\mathcal{D}}$. Las soluciones de las restricciones no primitivas que contienen llamadas a funciones definidas $f \in DF^n$ dependen del comportamiento de f que no está incluido en el dominio $\mathcal{D}$, pero se deduce del programa. Por el contrario, las soluciones de las restricciones primitivas dependen únicamente del dominio $\mathcal{D}$ y se definen a continuación.

Definición 2. (Soluciones de restricciones primitivas)

1. El *conjunto de soluciones* de una restricción primitiva $\pi \in PCon_{\mathcal{D}}$ es un subconjunto $Sol_{\mathcal{D}}(\pi) \subseteq Val_{\mathcal{D}}$ definido sobre la estructura sintáctica de π de la siguiente forma:
   - $Sol_{\mathcal{D}}(\Diamond) = Val_{\mathcal{D}}$; $Sol_{\mathcal{D}}(\blacklozenge) = \emptyset$.
   - $Sol_{\mathcal{D}}(p\,\bar{t}_n \to\!!\, t) = \{\eta \in Val_{\mathcal{D}} \mid (p\,\bar{t}_n \to\!!\, t)\eta$ es básica, $p^{\mathcal{D}}\bar{t}_n\eta \to t\eta$, $t\eta$ es total$\}$.
   - $Sol_{\mathcal{D}}(\pi_1 \wedge \pi_2) = Sol_{\mathcal{D}}(\pi_1) \cap Sol_{\mathcal{D}}(\pi_2)$.
   - $Sol_{\mathcal{D}}(\exists X \pi) = \{\eta \in Val_{\mathcal{D}} \mid$ existe $\eta' \in Sol_{\mathcal{D}}(\pi)$ tal que $\eta' =_{\setminus\{X\}} \eta\}$.
2. Cualquier conjunto $\Pi \subseteq PCon_{\mathcal{D}}$ se interpreta como una conjunción y por lo tanto $Sol_{\mathcal{D}}(\Pi) = \bigcap_{\pi\in\Pi} Sol_{\mathcal{D}}(\pi)$.
3. El *conjunto de soluciones bien tipadas* de una restricción primitiva $\pi \in PCon_{\mathcal{D}}$ es el conjunto $WTSol_{\mathcal{D}}(\pi) \subseteq Sol_{\mathcal{D}}(\pi)$ formado por todas las soluciones $\eta \in Sol_{\mathcal{D}}(\pi)$ tales que $\pi\eta$ está bien tipado.
4. Finalmente, para cualquier subconjunto $\Pi \subseteq PCon_{\mathcal{D}}$ se define $WTSol_{\mathcal{D}}(\Pi) = \bigcap_{\pi\in\Pi} WTSol_{\mathcal{D}}(\pi)$.

Las soluciones de una restricción primitiva atómica $p\,\bar{t}_n \to\!!\, t$ son aquellas valoraciones para las cuales $p\,\bar{t}_n$ devuelve un valor que se ajusta a un patrón total t. Por ejemplo, $Sol_{\mathcal{FD}}$`(X #+ Y→!10)` es el conjunto de soluciones de la restricción primitiva atómica `(X #+ Y →! 10)` en el dominio $\mathcal{FD}$ y se compone de todas las valoraciones $\eta \in Val_{\mathcal{FD}}$ tales que `(X #+ Y)`η devuelve el valor 10. Por consiguiente, η_1 = `{X↦-10,Y↦20}` es una posible solución y η_2 = `{X↦20,Y↦-10}` es otra posible solución. Ambas soluciones cumplen que la interpretación de dichas soluciones en el dominio $\mathcal{FD}$ es correcta. Es decir, `(X #+`$^{\mathcal{FD}}$ `Y)`$\eta_1 \to 10\eta_1$ y `(X #+`$^{\mathcal{FD}}$ `Y)`$\eta_2 \to 10\eta_2$ donde $10\eta_1$ y $10\eta_2$ claramente son totales. Con respecto al tipo, ambas valoraciones son soluciones bien tipadas. Así, de esta forma η_1 y η_2 forman parte del conjunto de soluciones $WTSol_{\mathcal{FD}}$`(X #+ Y →! 10)`.

3.1.6 Almacenes de restricciones

El modelo de cooperación que se va a presentar mantiene distintos almacenes de restricciones que corresponden a los distintos dominios. Así, se define un *almacén de restricciones* para

un dominio $\mathcal{D}$ como un par $\Pi \,\square\, \sigma$, donde $\Pi \subset APCon_{\mathcal{D}}$ es un conjunto de restricciones primitivas atómicas, y σ es una sustitución idempotente tal que las variables del dominio de σ y las variables de Π son disjuntas, pues la sustitución σ se ha aplicado sobre Π. El símbolo $\square$ se interpreta como una conjunción y las *soluciones de un almacén de restricciones* se definen de la siguiente forma:

Definición 3. (Soluciones de almacenes de restricciones)

1. $Sol_{\mathcal{D}}(\exists\overline{Y}(\Pi \,\square\, \sigma)) = \{\eta \in Val_{\mathcal{D}} \mid \text{existe } \eta' \in Sol_{\mathcal{D}}(\Pi \,\square\, \sigma) \text{ tal que } \eta' =_{\setminus \overline{Y}} \eta\}$.
2. $Sol_{\mathcal{D}}(\Pi \,\square\, \sigma) = Sol_{\mathcal{D}}(\Pi) \cap Sol(\sigma)$.
3. $Sol(\sigma) = \{\eta \in Val_{\mathcal{D}} \mid \eta = \sigma\eta\}$.
4. $WTSol_{\mathcal{D}}(\exists\overline{Y}(\Pi \,\square\, \sigma)) = \{\eta \in Val_{\mathcal{D}} \mid \text{existe } \eta' \in WTSol_{\mathcal{D}}(\Pi \,\square\, \sigma) \text{ tal que } \eta' =_{\setminus \overline{Y}} \eta\}$.
5. $WTSol_{\mathcal{D}}(\Pi \,\square\, \sigma) = \{\eta \in Sol_{\mathcal{D}}(\Pi \,\square\, \sigma) \mid (\Pi \,\square\, \sigma)\star\eta \text{ está bien tipado, donde } (\Pi \,\square\, \sigma)\star \eta =_{def} \Pi\eta \,\square\, (\sigma \star \eta)$.

Por el punto 3 de la definición anterior se ha de tener en cuenta que para que se cumpla $\eta = \sigma\eta$ se ha de cumplir $X\eta = X\sigma\eta$ para todo $X \in vdom(\sigma)$. Con respecto al punto 5, primero recordemos que dadas dos sustituciones σ y η, la sustitución $\sigma\star\eta$ es la *aplicación* de η sobre σ restringida a las variables de σ, es decir, $\sigma\eta \restriction_{vdom(\sigma)}$. Por lo tanto, para cualquier variable X se cumple $X(\sigma \star \eta) = X\sigma\eta$ si $X \in vdom(\sigma)$ y $X(\sigma \star \eta) = X$ en otro caso.

Los almacenes de restricciones van a servir para definir resolutores transparentes pues permiten indicar paso a paso su comportamiento. Un resolutor transparente detalla cómo se puede transformar un almacén $(\Pi \,\square\, \sigma)$ en otro almacén $(\Pi' \,\square\, \sigma')$ en un paso de cómputo. De esta forma se pueden definir todos los pasos que puede dar un resolutor. Si la sustitución σ es la sustitución identidad ε, entonces el almacén $\Pi \,\square\varepsilon$ se simplifica a Π.

Cada dominio de restricciones tiene un resolutor de restricciones asociado cuya formalización se muestra a continuación.

3.1.7 Resolutores de restricciones

En el transcurso del cómputo de un objetivo se produce una serie de restricciones primitivas atómicas que son enviadas al resolutor correspondiente. El resolutor, si es capaz de encontrar una solución, muestra las sustituciones de las variables como respuesta y, si no lo es, entonces reduce las restricciones primitivas que se le han enviado a formas resueltas y estas se muestran como respuesta.

Algunos de los resolutores que utiliza $\mathcal{TOY}$ son de caja transparente, es decir, su semántica está completamente definida y su implementación no utiliza bibliotecas externas para la resolución de restricciones. Otros resolutores utilizan bibliotecas externas, obligando a postular su "buen comportamiento" para poder obtener resultados semánticos de corrección y completitud.

En esta tesis se definen los resolutores para los dominios $\mathcal{R}$, $\mathcal{FD}$, $\mathcal{FS}$, $\mathcal{H}$ y los dominios mediadores. Como el resolutor más complejo es el resolutor de $\mathcal{H}$, se ha decidido introducir primero la definición de un resolutor para cualquier dominio de los indicados anteriormente excepto $\mathcal{H}$, para ampliar posteriormente esta definición para el caso concreto de $\mathcal{H}$. Así, si el dominio $\mathcal{D}$ es el dominio $\mathcal{H}$, entonces se extiende la definición 4, que se muestra a continuación, con las condiciones de vinculación segura y discriminación de la definición 7, incluida en la sección 3.5.

Definición 4. (Definición y requisitos formales de un resolutor de dominio distinto de $\mathcal{H}$)

Un resolutor de restricciones para el dominio $\mathcal{D}$ (donde $\mathcal{D}$ es un dominio distinto de $\mathcal{H}$) se modela como una función $solve^{\mathcal{D}}$ que se aplica a un conjunto finito de restricciones primitivas atómicas $\Pi \subseteq APCon_{\mathcal{D}}$.

Cualquier invocación al resolutor $solve^{\mathcal{D}}(\Pi)$ debe devolver una disyunción finita de almacenes de restricciones existencialmente cuantificados $\bigvee_{j=1}^{k} \exists \overline{Y}_j (\Pi_j \,\square\, \sigma_j)$, cumpliendo las siguientes condiciones:

1. **Variables locales nuevas:** para todo $1 \leq j \leq k$: $\Pi_j \,\square\, \sigma_j$ es un almacén de restricciones tal que $\overline{Y}_j = var(\Pi_j \,\square\, \sigma_j) \setminus var(\Pi)$ son variables locales nuevas y $vdom(\sigma_j) \cup vran(\sigma_j) \subseteq var(\Pi) \cup \overline{Y}_j$.
2. **Formas resueltas:** para todo $1 \leq j \leq k$: $\Pi_j \,\square\, \sigma_j$ está en forma resuelta. Por definición esto significa que $solve^{\mathcal{D}}(\Pi_j) = \Pi_j \,\square\, \varepsilon$.
3. **Corrección:** $Sol_{\mathcal{D}}(\Pi) \supseteq \bigcup_{j=1}^{k} Sol_{\mathcal{D}}(\exists \overline{Y}_j (\Pi_j \,\square\, \sigma_j))$.
4. **Completitud:** $WTSol_{\mathcal{D}}(\Pi) \subseteq \bigcup_{j=1}^{k} WTSol_{\mathcal{D}}(\exists \overline{Y}_j (\Pi_j \,\square\, \sigma_j))$.

Un ejemplo sencillo de cómo se comporta el resolutor $\mathcal{R}$ implementado en $\mathcal{TOY}$ es el siguiente. Supongamos un objetivo formado por las siguientes restricciones Π = `{X>=1, X<=10, 10==X*Y/(X+Y)}`, el resultado que devuelve $\mathcal{TOY}$ a este objetivo es el siguiente: Π_j = `{X+Y==C, X*Y==D, D==10.0*C, X>=1.0, X=<10.0}`. Como se puede observar se han introducido dos variables locales nuevas `C` y `D`. Las restricciones que forman la respuesta están en forma resuelta. Con respecto a la corrección se puede comprobar que $Sol_{\mathcal{D}}(\Pi) \supseteq Sol_{\mathcal{D}}(\exists \texttt{C,D}(\Pi_j))$. Por la definición 2, para cualquier valoración $\eta \in Sol_{\mathcal{D}}(\exists \texttt{C,D}(\Pi_j))$ existe un $\eta' \in Sol_{\mathcal{D}}(\Pi)$ tal que $\eta' =_{\setminus \overline{Y}} \eta$ y la misma η' cumple $\eta' \in Sol_{\mathcal{D}}(\Pi)$. Por ejemplo, la sustitución η' = `{X↦5,Y↦-10}` y η = `{X↦5,Y↦-10,C↦-5,D↦-50}`. Razonando de forma similar se puede comprobar que la completitud también se cumple.

Como se acaba de definir, una invocación al resolutor $solve^{\mathcal{D}}(\Pi)$ devuelve una disyunción finita de almacenes de restricciones, $\bigvee_{j=1}^{k} \exists \overline{Y}_j (\Pi_j \,\square\, \sigma_j)$. En concreto la disyunción vacía se identifica con la restricción trivialmente insatisfactible ⧫. Operacionalmente, las distintas alternativas son exploradas por medio de *backtracking* (vuelta atrás). Se utilizará la siguiente

notación:

- $\Pi \Vdash_{solve^{\mathcal{D}}} \exists\overline{Y'}(\Pi' \square \sigma')$ indica que $\exists\overline{Y'}(\Pi' \square \sigma')$ es algún $\exists\overline{Y}_j(\Pi_j \square \sigma_j)$ con $1 \leq j \leq k$. En este caso se dice que es una *invocación exitosa al resolutor*.

- $\Pi \Vdash_{solve^{\mathcal{D}}} \blacksquare$ indica que $k = 0$, donde $\blacksquare = \blacklozenge \square \varepsilon$ es un almacén insatisfactible. En este caso se dice que es una *invocación fallida al resolutor*.

Un almacén de restricciones $\Pi \square \sigma$ está en *forma resuelta* si y solamente si $solve^{\mathcal{D}}(\Pi \square \sigma) = \Pi \square \sigma$. En la práctica, las formas resueltas pueden ser reconocidas por criterios sintácticos. Una invocación al resolutor $solve^{\mathcal{D}}(\Pi \square \sigma)$ se lleva a cabo solo en el caso de que $\Pi \square \sigma$ aún no esté en forma resuelta. Cada vez que se invoca un resolutor, la condición de *corrección* exige que no se introduzca ninguna solución nueva falsa, mientras que la condición de *completitud* requiere que ninguna solución bien tipada se pierda. En la práctica se puede esperar que cualquier resolutor sea correcto, pero completo sólo puede ser para algunas elecciones del conjunto de restricciones que van a ser resueltas. De hecho, en la completitud se pide que las soluciones estén bien tipadas para restringir los casos de incompletitud. Las limitaciones con respecto a la completitud de los resolutores de los dominios $\mathcal{FD}$, $\mathcal{R}$ y $\mathcal{FS}$ se verán más adelante.

La especificación semántica de un resolutor se puede establecer por medio de una caja transparente o de una caja negra. En el caso de ser una caja transparente se va a utilizar una técnica de especificación de resolutores llamada *sistema de transformación de almacenes*. Un sistema de transformación de almacenes sobre un dominio de restricciones $\mathcal{D}$ se especifica con un conjunto de *reglas de transformación del almacenes* que describen las diferentes opciones mediante las cuales se puede transformar un almacén dado. Un almacén es *irreducible* si no se puede transformar aplicando alguna regla. Una regla no es aplicable si el almacén no se modifica al aplicarla. Se denota como $\Vdash_{\mathcal{D}}$ a un *paso de transformación de almacenes* sobre un dominio de restricciones $\mathcal{D}$ usando alguna de las reglas de transformación de almacenes disponibles. En particular se usará la siguiente notación:

- $\pi, \Pi \square \sigma \Vdash_{\mathcal{D}} \Pi' \square \sigma'$ indica que el almacén $\pi, \Pi \square \sigma$, que incluye la restricción atómica π más un conjunto de restricciones Π, se transforma en un paso en el almacén $\Pi' \square \sigma'$ usando alguna de las reglas de transformación de almacenes disponibles.

- $\Pi \square \sigma \Vdash_{\mathcal{D}} \blacksquare$, indica un paso de transformación que falla y deja un almacén inconsistente.

- $\Pi \square \sigma \Vdash^*_{\mathcal{D}} \Pi' \square \sigma'$ indica que $\Pi \square \sigma$ puede ser transformado en $\Pi' \square \sigma'$ en un número finito de pasos.

En un sistema de transformación de almacenes sobre $\mathcal{D}$ se define el conjunto de *formas resueltas* de un almacén (*SF Solved Forms*) como las formas más simples posibles de dicho almacén que son las que se muestran al usuario como respuestas. Formalmente:

$$SF_{\mathcal{D}}(\Pi\square\ \sigma) = \{\Pi' \square \sigma' \mid \Pi \square \sigma \Vdash^*_{\mathcal{D}} \Pi' \square \sigma' \ \text{ y } \ \Pi' \square \sigma' \text{ es irreducible}\}$$

Entonces el resolutor definido por un sistema de transformación de almacenes se puede especificar de la siguiente forma:

Definición 5. (Resolutor definido por un sistema de transformación de almacenes)
Dado un sistema de transformación de almacenes definido para un dominio $\mathcal{D}$, el resolutor $solve^{\mathcal{D}}$ se define como:

$$solve^{\mathcal{D}}(\Pi) = \bigvee\{\exists\overline{Y'}(\Pi'\Box\sigma') \mid \Pi'\Box\sigma' \in SF_{\mathcal{D}}(\Pi), \overline{Y'} = var(\Pi'\Box\sigma') \setminus var(\Pi)\}$$

Con esta definición de resolutor para un sistema de transformación de almacenes, la notación $\Pi \Vdash_{solve^{\mathcal{D}}} \exists\overline{Y'}(\Pi' \Box \sigma')$ significa $\Pi \Box \varepsilon \Vdash^*_{\mathcal{D}} \Pi' \Box \sigma'$ con $\Pi' \Box \sigma'$ irreducible e $\overline{Y'} = var(\Pi' \Box \sigma') \setminus var(\Pi)$.

La siguiente definición especifica diferentes propiedades de los sistemas de transformación de almacenes que permite comprobar que los resolutores definidos de esta forma cumplen las condiciones establecidas en la Definición 4. Al igual que se hizo con la definición de resolutor, la siguiente definición será completada con más propiedades para el resolutor $\mathcal{H}$.

Definición 6. (Propiedades de los sistemas de transformación de almacenes, excepto para $\mathcal{H}$)
Un sistema de transformación de almacenes sobre un dominio $\mathcal{D}$, excepto $\mathcal{H}$, cuya relación de transición es $\Vdash_{\mathcal{D}}$ debe satisfacer las siguientes propiedades:

1. **Variables locales nuevas:** si $\Pi \Box \sigma \Vdash_{\mathcal{D}} \Pi' \Box \sigma'$ entonces $\Pi' \Box \sigma'$ es un almacén, $\overline{Y'} = var(\Pi' \Box \sigma') \setminus var(\Pi \Box \sigma)$ son variables locales nuevas y existe una sustitución σ_1 (responsable de las vinculaciones de variables creadas en este paso) tal que $\sigma' = \sigma\sigma_1$ y se cumple $vdom(\sigma_1) \cup vran(\sigma_1) \subseteq var(\Pi) \cup \overline{Y'}$.

2. **Ramificación finita:** para cualquier $\Pi \Box \sigma$ fijo hay un número finito de $\Pi' \Box \sigma'$ tales que $\Pi \Box \sigma \Vdash_{\mathcal{D}} \Pi' \Box \sigma'$.

3. **Terminación:** no existe una secuencia infinita $\{\Pi_i \Box \sigma_i \mid i \in \mathbb{N}\}$ tal que $\Pi_i \Box \sigma_i \Vdash_{\mathcal{D}} \Pi_{i+1} \Box \sigma_{i+1}$ para todo $i \in \mathbb{N}$.

4. **Corrección local:** para cualquier almacén $\Pi \Box \sigma$ del dominio $\mathcal{D}$, el conjunto
   $\bigcup\{Sol_{\mathcal{D}}(\exists\overline{Y'}(\Pi' \Box \sigma')) \mid \Pi \Box \sigma \Vdash_{\mathcal{D}} \Pi' \Box \sigma', \overline{Y'} = var(\Pi' \Box \sigma') \setminus var(\Pi \Box \sigma)\}$
   es un subconjunto de $Sol_{\mathcal{D}}(\Pi \Box \sigma)$.

5. **Completitud local:** para cualquier almacén $\Pi \Box \sigma$ del dominio $\mathcal{D}$, el conjunto $WTSol_{\mathcal{D}}(\Pi \Box \sigma)$ es un subconjunto de
   $\bigcup\{WTSol_{\mathcal{D}}(\exists\overline{Y'}(\Pi' \Box \sigma')) \mid \Pi\Box\sigma \Vdash_{\mathcal{D}} \Pi'\Box\sigma', \overline{Y'} = var(\Pi'\Box\sigma') \setminus var(\Pi\Box\sigma)\}$

Como ya se ha comentado, los sistemas de transformación de almacenes sirven para definir resolutores transparentes. Por lo tanto, las propiedades exigidas en la definición anterior son necesarias para asegurar que se cumplen las propiedades pedidas a los resolutores en la definición 4. Así, las propiedades de *variables locales nuevas*, *corrección local* y *completitud local* son necesarias para las correspondientes propiedades globales de la definición 4 y las propiedades de *ramificación finita* y *terminación* son necesarias porque la invocación al resolutor $solve^{\mathcal{D}}(\Pi)$ debe devolver una disyunción finita de almacenes de restricciones. Formalmente, el siguiente lema asegura que un resolutor definido mediante un sistema de transformación de almacenes que satisface las propiedades de la definición 6, satisface los requisitos para los resolutores dados en la definición 4.

Lema 1. (Resolutores definidos por medio de sistemas de transformación de almacenes, excepto $\mathcal{H}$)
Cualquier sistema de transformación de almacenes de un dominio $\mathcal{D}$ con ramificación finita y terminante verifica:

1. $SF_{\mathcal{D}}(\Pi)$ es siempre finito, y por lo tanto $solve^{\mathcal{D}}$ está bien definido y satisface la propiedad de formas resueltas.
2. $solve^{\mathcal{D}}$ tiene la propiedad de variables locales nuevas si el sistema de transformación de almacenes tiene la propiedad correspondiente.
3. $solve^{\mathcal{D}}$ es correcto si el sistema de transformación de almacenes es localmente correcto.
4. $solve^{\mathcal{D}}$ es completo si el sistema de transformación de almacenes es localmente completo.

En los dominios $\mathcal{FD}$ y $\mathcal{FS}$ es interesante aplicar dos resolutores de forma secuencial: primero un resolutor simbólico de caja transparente denominado $solve1^{\mathcal{D}}$ que trata de anticipar el fallo y después un resolutor de caja negra denominado $solve2^{\mathcal{D}}$ que utiliza alguna biblioteca de resolución de restricciones.

Como se ha visto en la definición 4 una invocación a un resolutor debe devolver una disyunción finita de almacenes de restricciones existencialmente cuantificados, pero en nuestro caso, los resolutores simbólicos de caja transparente de los dominios $\mathcal{FD}$ y $\mathcal{FS}$ no producen variables nuevas. Por este motivo, se relaja la notación y se pide que una invocación al resolutor $solve1^{\mathcal{D}}(\Pi)$, donde $\mathcal{D}$ es el dominio $\mathcal{FD}$ o $\mathcal{FS}$, devuelva una disyunción finita de almacenes de restricciones de la forma $\bigvee_{j=1}^{k}(\Pi_j \,\square\, \sigma_j)$. Es decir, se exige que $solve1^{\mathcal{D}}$ no produzca variables nuevas, por lo tanto los almacenes pertenecientes a la disyunción no están existencialmente cuantificados. A cada uno de estos almacenes se les aplica una invocación de la forma $solve2^{\mathcal{D}}(\Pi_j \,\square\, \sigma_j)$ que devuelve una disyunción finita de almacenes de restricciones existencialmente cuantificados $\bigvee_{l=1}^{n_j} \exists \overline{Y}_{jl}(\Pi_{jl} \,\square\, \sigma_{jl})$ como se muestra en la figura 3.1.

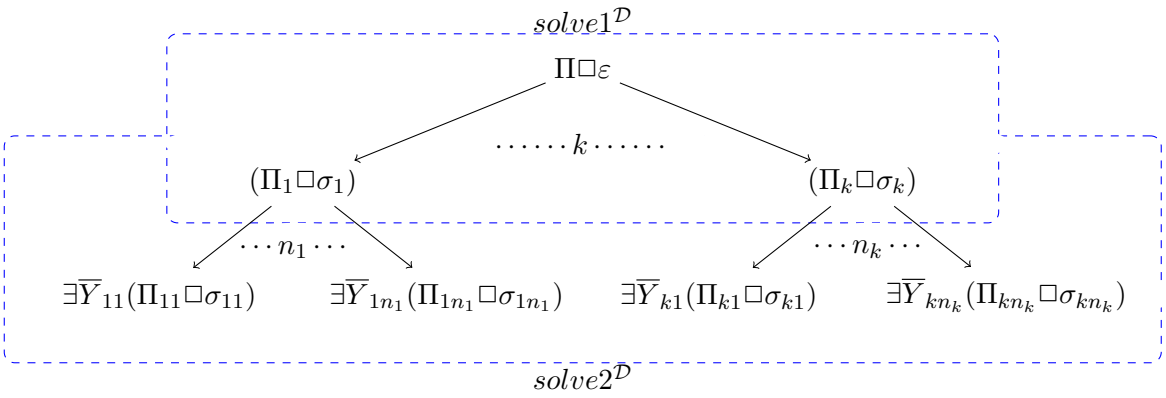


Figura 3.1: Secuenciación de resolutores $solve1^{\mathcal{D}} \diamond solve2^{\mathcal{D}}$

Aunque en esta tesis se aplica la secuenciación de resolutores únicamente en los dominios $\mathcal{FD}$ y $\mathcal{FS}$, en general se puede aplicar a cualquier dominio cuyo resolutor cumpla la definición y requisitos formales dados en la definición 4. Asi, la notación $solve1^{\mathcal{D}} \diamond solve2^{\mathcal{D}}$ indica que los resolutores $solve1^{\mathcal{D}}$ y $solve2^{\mathcal{D}}$, definidos en el mismo dominio $\mathcal{D}$, se aplican de forma secuencial.

Lema 2. (Secuenciación de resolutores)
Dados dos resolutores $solve1^{\mathcal{D}}$ y $solve2^{\mathcal{D}}$ de un mismo dominio $\mathcal{D}$ (distinto de $\mathcal{H}$), que cumplen las propiedades de la definición 4 y $solve1^{\mathcal{D}}$ no produce variables nuevas, entonces la secuenciación de estos dos resolutores $solve1^{\mathcal{D}} \diamond solve2^{\mathcal{D}}$ es también un resolutor $solve^{\mathcal{D}}$ que cumple las mismas propiedades.

Las demostraciones de los lemas 1 y 2 se encuentran disponibles en los apéndices A.1 y A.2. Una vez que se han definido los conceptos de dominio y resolutor pasamos a mostrar los casos concretos de $\mathcal{R}$, $\mathcal{FD}$, $\mathcal{FS}$ y $\mathcal{H}$.

3.2 El dominio $\mathcal{R}$

El dominio de restricciones $\mathcal{R}$ con restricciones aritméticas sobre los números reales fue inicialmente desarrollado como una instancia $CLP(\mathcal{R})$ del esquema CLP desarrollado por [JMSY92]. En el contexto de nuestro marco $CFLP$ se define el dominio de restricciones $\mathcal{R}$ con la signatura específica $\Sigma_{\mathcal{R}} = \langle TC,\ SBT_{\mathcal{R}},\ DC,\ DF,\ SPF_{\mathcal{R}} \rangle$, donde el tipo base específico es $SBT_{\mathcal{R}}$ `= {real}`, el conjunto de valores básicos es el conjunto de los números reales ($\mathcal{B}^{\mathcal{R}}_{\texttt{real}} = \mathbb{R}$), y las funciones primitivas específicas SPF son los siguientes operadores:

- `== :: A -> A -> bool`
- `+, -, *, / :: real -> real -> real`

- `<= :: real -> real -> bool`

Las funciones primitivas se interpretan de la siguiente forma:

- $==^{\mathcal{R}}$ se define del mismo modo que se define en cualquier dominio cuya signatura específica incluya `==`.
- $+^{\mathcal{R}}$: para todo t_1, t_2, $t \in \mathcal{U}_{\mathcal{R}}$ la interpretación $t_1 +^{\mathcal{R}} t_2 \rightarrow t$ debe cumplir alguno de los siguientes casos: o bien t_1, t_2 y t son números reales y t es igual a la suma de t_1 y t_2; o bien $t = \bot$. Las interpretaciones de `-`, `*` y `/` se definen análogamente.
- $<=^{\mathcal{R}}$: para todo t_1, t_2, $t \in \mathcal{U}_{\mathcal{R}}$ la interpretación $t_1 <=^{\mathcal{R}} t_2 \rightarrow t$ debe cumplir alguno de los siguientes casos: o bien t_1 y t_2 son números reales tales que t_1 es menor o igual que t_2 y $t =$ `true`; o bien t_1 y t_2 son números reales tales que t_1 es mayor que t_2 y $t =$ `false`; o bien $t = \bot$.

Además de estas primitivas existen otras en $\mathcal{TOY}$ [ACE+07] pero no se especifican pues no son relevantes para la cooperación.

Las restricciones atómicas de $\mathcal{R}$ se definen de la siguiente forma: $e_1 \odot e_2 \rightarrow\!! t$, donde $\odot$ es el operador de igualdad estricta o bien un operador de orden o bien un operador aritmético. Una restricción atómica de $\mathcal{R}$ se denomina *propia* si y solamente si $\odot$ no es el operador de igualdad estricta y en otro caso se denomina *restricción de Herbrand extendida*. Así las restricciones de igualdad estricta (e_1 `==` e_2) y las restricciones de desigualdad estricta (e_1 `/=` e_2) son abreviaturas de las restricciones de Herbrand extendidas. Otras restricciones de orden se definen como se muestra a continuación:

- e_1 `<` $e_2 =_{def} e_2$ `<=` $e_1 \rightarrow\!!$ `false`
- e_1 `>` $e_2 =_{def} e_1$ `<=` $e_2 \rightarrow\!!$ `false`
- e_1 `>=` $e_2 =_{def} e_2$ `<=` $e_1 \rightarrow\!!$ `true`

Con respecto al resolutor $solve^{\mathcal{R}}$, se espera que sea capaz de tratar con un conjunto de restricciones $\mathcal{R}$-específicas $\Pi \subseteq APCon_{\mathcal{R}}$ que consisten en restricciones primitivas atómicas π de los siguientes tipos:

- Restricciones *propias de* $\mathcal{R}$ que tienen la forma $t_1 \odot t_2 \rightarrow\!! t$, donde $\odot$ es o bien el operador de orden o bien el operador aritmético.
- Restricciones $\mathcal{R}$*-específicas de Herbrand*, t_1 `==` t_2 o bien t_1 `/=` t_2, donde los patrones t_1 y t_2 son o bien un valor constante real o bien una variable de tipo `real`.

El resolutor $solve^{\mathcal{R}}$ está implementado en $\mathcal{TOY}$ como un resolutor de caja negra sobre SICStus Prolog y de su comportamiento solo podemos hacer las siguientes suposiciones.

Postulado 1. Supuestos sobre el resolutor $\mathcal{R}$

El resolutor $solve^{\mathcal{R}}$ satisface las propiedades de *variables locales nuevas*, *formas resueltas* y *corrección* requeridas para un resolutor en la definición 4. Además, siempre que $\Pi \subseteq APCon_{\mathcal{R}}$ es $\mathcal{R}$-específico y $\Pi \Vdash_{solve^{\mathcal{R}}} \exists\overline{Y'}(\Pi' \square \sigma')$, el conjunto de restricciones Π' es también $\mathcal{R}$-específico, y para todo $X \in vdom(\sigma')$: o bien $X\sigma'$ es un valor real, o bien X y $X\sigma'$ pertenecen a $var(\Pi)$.

□

Obsérvese que no se postula la propiedad de *completitud*, esto es debido a que la completitud puede fallar para algunas elecciones de restricciones $\mathcal{R}$-específicas pues en general los resolutores de $\mathcal{R}$ no son completos. Por ejemplo, el resolutor de $\mathcal{TOY}$ no es capaz de resolver restricciones no lineales pues está construido sobre el resolutor de SICStus, que tiene la misma limitación. En concreto, $\mathcal{TOY}$ puede resolver el objetivo `X + X == 4.0`, donde la variable `X` se instancia al valor 2. Sin embargo, $\mathcal{TOY}$ no es capaz de resolver el objetivo `X * X == 4.0` y muestra como solución la restricción `4.0-X^2.0==0.0`. A pesar de esta limitación se pueden resolver problemas interesantes como se mostró en la subsección 1.3.7 y la cooperación con el resolutor $\mathcal{FD}$ hace que se puedan mostrar soluciones enteras a problemas reales, como se mostró en dos primeros ejemplos motivadores de la sección 1.1.

La última parte del postulado anterior exige que la sustitución σ' o bien transforme las variables a números reales o bien a otra variable. Este punto del postulado se utiliza en la demostración de corrección de las reglas que invocan a los resolutores que se definen en la tabla 4.2.

3.3 El dominio $\mathcal{FD}$

La idea de un dominio $\mathcal{FD}$ con restricciones aritméticas y de dominio finito sobre los números enteros es bien conocida en la comunidad CLP [vHSD94, vHSD98]. En nuestro marco $CFLP$ el dominio $\mathcal{FD}$ se define con la signatura $\Sigma_{\mathcal{FD}} = \langle TC,\ SBT_{\mathcal{FD}},\ DC,\ DF,\ SPF_{\mathcal{FD}}\rangle$ donde el tipo base específico $SBT_{\mathcal{FD}}$ = `{int}` y el conjunto de los valores básicos es el conjunto de los números enteros $\mathcal{B}^{\mathcal{FD}}_{\texttt{int}} = \mathbb{Z}$. Las funciones primitivas $SPF_{\mathcal{FD}}$ son las siguientes:

- `== :: A -> A -> bool`
- `#<= :: int -> int -> bool`
- `#+, #-, #*, #/ :: int -> int -> int`
- `domain :: [int] -> int -> int -> bool`
- `all_different :: [int] -> bool`
- `exactly :: int -> [int] -> int -> bool`
- `belongs :: int -> [int] -> bool`
- `labeling :: [labelingType] -> [int] -> bool`

Las interpretaciones de las funciones primitivas son:

- $\texttt{==}^{\mathcal{FD}}$ se define como en cualquier dominio cuya signatura específica incluya `==`.
- $\texttt{\#<=}^{\mathcal{FD}}$: para todo t_1, t_2, $t \in \mathcal{U}_{\mathcal{FD}}$ la interpretación t_1 $\texttt{\#<=}^{\mathcal{FD}} t_2 \rightarrow t$ debe cumplir alguno de los siguientes casos: o bien t_1 y t_2 son números enteros tales que t_1 es menor o igual que t_2 y t = `true`; o bien t_1 y t_2 son números enteros tales que t_1 es mayor que t_2 y t = `false`; o bien $t = \bot$.
- $\texttt{\#+}^{\mathcal{FD}}$: para todo t_1, t_2, $t \in \mathcal{U}_{\mathcal{FD}}$ la interpretación t_1 $\texttt{\#+}^{\mathcal{FD}} t_2 \rightarrow t$ debe cumplir alguno de los siguientes casos: o bien t_1, t_2 y t son números enteros y t es igual a la suma de t_1 y t_2; o bien $t = \bot$. Las interpretaciones de `#-`, `#*` y `#/` se definen análogamente.
- $\texttt{domain}^{\mathcal{FD}}$: para todo t_1, t_2, t_3, $t \in \mathcal{U}_{\mathcal{FD}}$ la interpretación $\texttt{domain}^{\mathcal{FD}}\ t_1\, t_2\, t_3 \rightarrow t$ debe cumplir alguno de los siguientes casos: o bien t_2 y t_3 son números enteros de la forma a y b tal que $a \leq b$, t_1 es una lista finita no vacía de números enteros pertenecientes al intervalo $a..b$ y t = `true`; o bien t_2 y t_3 son números enteros de la forma a y b tal que $a \leq b$, t_1 es una lista finita no vacía de números enteros alguno de los cuales no pertenece al intervalo $a..b$ y t = `false`; o bien t_2 y t_3 son números enteros de la forma a y b tal que $a > b$ y t = `false`; o bien $t = \bot$.
- $\texttt{all\_different}^{\mathcal{FD}}$: para todo t_1, $t \in \mathcal{U}_{\mathcal{FD}}$ la interpretación $\texttt{all\_different}^{\mathcal{FD}}\ t_1 \rightarrow t$ debe cumplir alguno de los siguientes casos: o bien t_1 es una lista finita de números enteros distintos y t = `true`; o bien t_1 es una lista finita de números enteros con algún elemento repetido y t = `false`; o bien $t = \bot$.
- $\texttt{exactly}^{\mathcal{FD}}$: para todo t_1, t_2, t_3, $t \in \mathcal{U}_{\mathcal{FD}}$ la interpretación $\texttt{exactly}^{\mathcal{FD}}\ t_1\, t_2\, t_3 \rightarrow t$ debe cumplir alguno de los siguientes casos: o bien t_1 y t_3 son números enteros, t_2 es una lista finita de números enteros que contiene a t_1 exactamente t_3 veces y t = `true`; o bien t_1 y t_3 son números enteros, t_2 es una lista finita de números enteros que no contiene a t_1 exactamente t_3 veces y t = `false`; o bien $t = \bot$.
- $\texttt{belongs}^{\mathcal{FD}}$: para todo t_1, t_2, $t \in \mathcal{U}_{\mathcal{FD}}$ la interpretación $\texttt{belongs}^{\mathcal{FD}}\ t_1\, t_2 \rightarrow t$ debe cumplir alguno de los siguientes casos: o bien t_1 es un entero, t_2 es una lista finita de números enteros que incluye como elemento a t_1, y t = `true`; o bien t_1 es un entero, t_2 es una lista finita de números enteros que no incluye como elemento a t_1 y t = `false`; o bien $t = \bot$.
- $\texttt{labeling}^{\mathcal{FD}}$: para todo t_1, t_2, $t \in \mathcal{U}_{\mathcal{FD}}$ la interpretación $\texttt{labeling}^{\mathcal{FD}}\ t_1\, t_2 \rightarrow t$ debe cumplir alguno de los siguientes casos: o t_1 es un valor definido del tipo `labelType`, t_2 es una lista finita de números enteros y t = `true`; o bien $t = \bot$.

Además de estas primitivas existen más primitivas en $\mathcal{TOY}$ [ACE+07] que aquí no se especifican pues no son relevantes para la cooperación.

Las restricciones atómicas de $\mathcal{FD}$ son de la forma $p\ \overline{e}_n \rightarrow! t$ donde p es uno de los símbolos de función primitiva descritos, cada e_i es una expresión que puede contener símbolos no primitivos, y t es un patrón total. De forma similar al dominio $\mathcal{R}$, las restricciones atómicas

de $\mathcal{FD}$ se denominan *restricciones de Herbrand extendidas* si tienen la forma e_1 `==` $e_2 \rightarrow! t$ y *propias* en otro caso. Otras restricciones de orden se definen como se muestra a continuación:

- e_1 `#<` $e_2 =_{def} e_2$ `#<=` $e_1 \rightarrow!$ `false`
- e_1 `#>` $e_2 =_{def} e_1$ `#<=` $e_2 \rightarrow!$ `false`
- e_1 `#>=` $e_2 =_{def} e_2$ `#<=` $e_1 \rightarrow!$ `true`

Con respecto al resolutor $solve^{\mathcal{FD}}$, se espera que sea capaz de tratar con restricciones $\mathcal{FD}$-específicas $\Pi \subseteq APCon_{\mathcal{FD}}$ que consisten en restricciones primitivas atómicas π de los siguientes tipos:

- Restricciones *propias de* $\mathcal{FD}$ que tienen la forma $t_1 \odot t_2 \rightarrow! t$, donde $\odot$ es o bien un operador de orden sobre los enteros o bien un operador aritmético de enteros o bien una de las primitivas `domain`, `all_different`, `exactly`, `belongs` o `labeling`.
- Restricciones $\mathcal{FD}$*-específicas de Herbrand*, t_1 `==` t_2 o bien t_1 `/=` t_2, donde los patrones t_1 y t_2 son o bien un valor constante entero o bien una variable de tipo `int`.

El resolutor $solve^{\mathcal{FD}}$ se ha implementado en dos sistemas distintos, SICStus Prolog y ECL^{i}PS^e. El primero interviene en la cooperación de $\mathcal{R}$ con $\mathcal{FD}$ y el segundo en la cooperación de $\mathcal{FD}$ y $\mathcal{FS}$. Como $\mathcal{TOY}$ está implementado en SICStus Prolog, en el caso de la cooperación de $\mathcal{FD}$ y $\mathcal{FS}$, es necesario comunicar estos dos sistemas. Esta comunicación hace que se pierda eficiencia, por ello para los resolutores $\mathcal{FD}$ y $\mathcal{FS}$ que utilizan ECL^{i}PS^e se ha creado un mecanismo para intentar anticipar el fallo del conjunto de restricciones no satisfactibles. Veamos un ejemplo.

Con el siguiente objetivo $\mathcal{TOY}$ ninguno de los resolutores de caja negra utilizados (SICStus y ECL^{i}PS^e) detectan la insatisfactibilidad de las restricciones, y se limitan a mostrar las restricciones suspendidas como respuesta.

```
domain [X,Y] 1 100000, X #<= Y, Y #<= X, X /= Y
```

En general los resolutores de $\mathcal{FD}$ propagan las restricciones hasta un cierto nivel cuyo efecto es podar los dominios, delegando la comprobación de la satisfacción de las restricciones en última instancia al etiquetado. El anterior objetivo con etiquetado es:

```
domain [X,Y] 1 100000, X #<= Y, Y #<= X, X /= Y, labeling [] [X,Y]
```

El proceso de etiquetar las variables está en el orden $\mathcal{O}(\mathtt{N} * \mathtt{M})$ donde `N` y `M` son los tamaños de los dominios de las variables `X` e `Y` respectivamente. Sin embargo, se puede anticipar el fallo detectando sintácticamente la inconsistencia de las restricciones `X #<= Y, Y #<= X, X /= Y`, reduciendo el coste del orden $\mathcal{O}(\mathtt{N} * \mathtt{M})$ al orden $\mathcal{O}(1)$. Este mecanismo de anticipación del fallo para los resolutores $\mathcal{FD}$ y $\mathcal{FS}$ que utilizan ECL^{i}PS^e se activa mediante el comando `/fs`.

Por ello, el resolutor $solve^{\mathcal{FD}}$ se ha definido como la secuenciación de dos resolutores: $solve^{\mathcal{FD}^{\mathcal{T}}} \diamond solve^{\mathcal{FD}^{N}}$. Es decir, primero se aplica un resolutor de caja transparente $solve^{\mathcal{FD}^{\mathcal{T}}}$

que implementa el mecanismo de anticipación del fallo y si no falla, entonces se aplica el resolutor de caja negra $solve^{\mathcal{FD}^N}$.

El resolutor de caja transparente $solve^{\mathcal{FD}^\mathcal{T}}$ se formaliza mediante un sistema de transformación de almacenes con solo dos reglas que se muestran en la tabla 3.1, donde se puede detectar la inconsistencia de un almacén en un paso de reescritura. En esta tabla los patrones F_1 y F_2 son o bien variables o bien constantes enteras. Este resolutor puede ser ampliado con más reglas que muestren situaciones concretas de inconsistencia. Las reglas mostradas ilustran la utilidad de anticipar el fallo antes de tratar la cooperación de los resolutores.

FD1	F_1 == F_2, F_1 /= F_2, $\Pi \Box \sigma \vdash_{\mathcal{FD}^\mathcal{T}} \blacksquare$
FD2	F_1 #<= F_2, F_2 #<= F_1, F_1 /= F_2, $\Pi \Box \sigma \vdash_{\mathcal{FD}^\mathcal{T}} \blacksquare$

Tabla 3.1: Reglas de transformación de almacenes que definen $\vdash_{\mathcal{FD}^\mathcal{T}}$

El siguiente teorema constata que se cumplen los requisitos enunciados en la definición 6 y por lo tanto puede ser considerado como una especificación correcta de un resolutor de caja transparente del dominio $\mathcal{FD}$. También cumple las condiciones de la definición 4 haciendo que $solve^{\mathcal{FD}^\mathcal{T}}$ sea un resolutor correcto. Su demostración se encuentra en el apéndice A.3 y es bastante sencilla pues amabas reglas devuelven derivan en inconsistencia.

Teorema 1. (Propiedades formales de $solve^{\mathcal{FD}^\mathcal{T}}$)

El sistema de transformación de almacenes con relación de transición $\vdash_{\mathcal{FD}^\mathcal{T}}$ cumple las propiedades de la definición 6. Además, como se mostró en la definición 5:

$solve^{\mathcal{FD}^\mathcal{T}}(\Pi) = \bigvee_{j=1}^{k}\{(\Pi_j \Box \sigma_j) \mid \Pi_j \sqsupset \sigma_j \in SF_{\mathcal{FD}^\mathcal{T}}(\Pi), \overline{Y}_j = var(\Pi_j \Box \sigma_j) \setminus var(\Pi)\}$

está bien definido para cualquier conjunto finito $\Pi \subseteq APCon_{\mathcal{FD}^\mathcal{T}}$ de restricciones $\mathcal{FD}$-específicas. El resolutor $solve^{\mathcal{FD}^\mathcal{T}}$ satisface todos los requisitos de los resolutores de la definición 4.

El resolutor $solve^{\mathcal{FD}^N}$ está definido como un resolutor de caja negra y las dos implementaciones de este resolutor están realizadas sobre las correspondientes bibliotecas de SICStus y ECL^{i}PS^e Prolog. El buen comportamiento del resolutor $solve^{\mathcal{FD}^N}$ se asume en el siguiente postulado.

Postulado 2. (Supuestos sobre el resolutor $\mathcal{FD}^N$)

El resolutor $solve^{\mathcal{FD}^N}$ satisface las propiedades de *variables locales nuevas*, *formas resueltas* y *corrección* requeridas para un resolutor en la definición 4. La propiedad de *completitud* puede fallar para algunas elecciones de restricciones $\mathcal{FD}$-específicas elegidas para ser resueltas. Además, siempre que $\Pi \subseteq APCon_{\mathcal{FD}}$ es $\mathcal{FD}$-específico y $\Pi \vdash_{solve^{\mathcal{FD}^N}} \exists\overline{Y'}(\Pi' \sqsupset \sigma')$, el conjunto de restricciones Π' es también $\mathcal{FD}$-específico, y para todo $X \in vdom(\sigma')$: o bien $X\sigma'$ es un valor entero, o si no X y $X\sigma'$ pertenecen a $var(\Pi)$.

□

Para mostrar el comportamiento de $solve^{\mathcal{FD}}$ podemos considerar por ejemplo el conjunto de restricciones primitivas atómicas $\Pi = \{$`domain [X,Y] 0 n, X #<Y, labeling [] [X,Y]`$\}$. La invocación del resolutor $solve^{\mathcal{FD}^{\mathcal{T}}}$ no tiene efecto, pues no se puede aplicar ninguna regla de la tabla 1. Por lo tanto las restricciones `domain [X,Y] 0 n`, `X #<Y` y `labeling [] [X,Y]` están en forma resuelta con respecto al resolutor $solve^{\mathcal{FD}^{\mathcal{T}}}$. Una vez finalizado $solve^{\mathcal{FD}^{\mathcal{T}}}$ se invoca al resolutor $solve^{\mathcal{FD}^{N}}$, que asigna valores a las variables `X` e `Y` y los enumera teniendo en cuenta la desigualdad. Por lo tanto $solve^{\mathcal{FD}}$ devuelve una disyunción de $\sum_{i=1}^{n} i$ alternativas, cada una de las cuales describe una solución única:

$$(\Diamond \Box\{\texttt{X} \mapsto 0, \texttt{Y} \mapsto 1\}) \vee \cdots \vee (\Diamond \Box\{\texttt{X} \mapsto n-1, \texttt{Y} \mapsto n\})$$

3.4 El dominio $\mathcal{FS}$

El dominio $\mathcal{FS}$ se define con la signatura específica $\Sigma_{\mathcal{FS}} = \langle TC, SBT_{\mathcal{FS}}, DC, DF, SPF_{\mathcal{FS}}\rangle$ donde $SBT_{\mathcal{FS}} = \{$`elem`, `set`$\}$. El conjunto de los valores básicos del tipo base `elem` es un conjunto enumerable $\mathcal{B}^{\mathcal{FS}}_{\texttt{elem}}$ con un orden total estricto $\prec$, y el conjunto de los valores básicos para `set` contiene todos los subconjuntos finitos de $\mathcal{B}^{\mathcal{FS}}_{\texttt{elem}}$, es decir, $\mathcal{B}^{\mathcal{FS}}_{\texttt{set}} = \mathcal{P}_f(\mathcal{B}^{\mathcal{FS}}_{\texttt{elem}})$. Un conjunto de elementos de tipo `elem` es un conjunto finito y totalmente definido y se representa como $\{e_1, \cdots, e_n\}$ donde $e_1 \prec \cdots \prec e_n$. Aunque el desarrollo formal puede realizarse utilizando un tipo genérico `elem`, la implementación realizada en ECL^{i}PS^e solo permite conjuntos de enteros. Por ello, a partir de este punto se considera `elem = int`. De esta forma $SBT_{\mathcal{FS}} =$ {`int`, `set`} y $\mathcal{B}^{\mathcal{FS}}_{\texttt{set}} = \mathcal{P}_f(\mathbb{Z})$. Las funciones primitivas $SPF_{\mathcal{FS}}$ se muestran a continuación:

- `== :: A -> A -> bool`
- `domainSets :: [set] -> set -> set -> bool`
- `intSets L A B :: [set] -> int -> int -> bool`
- `intSet :: set -> int -> int -> bool`
- `subset, superset :: set -> set -> bool`
- `intersect, union, diff, symdiff :: set -> set -> set -> bool`
- `intersections, unions :: [set] -> set -> bool`
- `disjoints :: [set] -> bool`
- `isIn, isNotIn :: int -> set -> bool`
- `labelingSets :: [set] -> bool`

Nótese que a diferencia del etiquetado del dominio $\mathcal{FD}$, `labelingSets` no admite opciones de etiquetado. De hecho, las primitivas que se acaban de definir tienen la misma estructura que las correspondientes primitivas disponibles en el sistema ECL^{i}PS^e Prolog, pues el resolutor de $\mathcal{TOY}$ del dominio $\mathcal{FS}$ se va a definir como una secuenciación de resolutores, cuyo segundo

resolutor es una caja negra que usa la librería de conjuntos finitos de enteros de ECL^{i}PS^e Prolog.

Además de las primitivas definidas anteriormente para el dominio $\mathcal{FS}$, se define una nueva restricción primitiva `<<` que no tiene correspondencia con las restricciones primitivas del sistema ECL^{i}PS^e Prolog y se implementa de forma distinta como se mostrará en el capítulo 7.

- `<< :: set -> set -> bool`. Este operador está relacionado con el orden parcial estricto entre dos variables de conjuntos finitos. En particular, S_1 `<<` S_2 significa que el mayor elemento perteneciente al conjunto S_1 es menor que el menor elemento del conjunto S_2.

Las correspondientes interpretaciones de las funciones primitivas específicas de $\mathcal{FS}$ se muestran a continuación:

- `==`$^{\mathcal{FS}}$ se define como en cualquier dominio cuya signatura específica incluya `==`.
- `domainSets`$^{\mathcal{FS}}$: para todo t_1, t_2, t_3, $t \in \mathcal{U}_{\mathcal{FS}}$ la interpretación `domainSets`$^{\mathcal{FS}}$ $t_1\, t_2\, t_3 \rightarrow t$ debe cumplir alguno de los siguientes casos: o bien t_2 y t_3 son conjuntos de elementos de tipo `int` de la forma a y b con $a \subseteq b$, t_1 es una lista finita no vacía de conjuntos de elementos de tipo `int` pertenecientes al retículo con ínfimo a y supremo b y $t =$ `true`; o bien t_2 y t_3 son conjuntos de elementos de tipo `int` de la forma a y b con $a \subseteq b$, t_1 es una lista finita no vacía de conjuntos de elementos de tipo `int`, alguno de los cuales no pertenece al retículo con ínfimo a y supremo b y $t =$ `false`; o bien t_2 y t_3 son conjuntos de elementos de tipo `int` de la forma a y b tal que $a \supset b$ y $t =$ `false`; o bien $t = \bot$.
- `intSets`$^{\mathcal{FS}}$ e `intSet`$^{\mathcal{FS}}$: para todo t_1, t_2, t_3, $t \in \mathcal{U}_{\mathcal{FS}}$ la interpretación `intSets`$^{\mathcal{FS}}$ $t_1\, t_2\, t_3 \rightarrow t$ es la misma que la interpretación de `domainSets`$^{\mathcal{FS}}$ $t_1\, t_2'\, t_3' \rightarrow t$ donde t_2' es el conjunto vacío y t_3' es el conjunto definido por todos los enteros comprendidos entre t_2 y t_3. La primitiva `intSet` es un caso particular de la primitiva `intSets` simplificada a un único elemento. Por lo tanto, su interpretación es similar simplificando la lista finita no vacía de conjuntos de elementos de tipo `int` a un único conjunto de elementos de tipo `int`.
- `subset`$^{\mathcal{FS}}$ y `superset`$^{\mathcal{FS}}$: para todo t_1, t_2, $t \in \mathcal{U}_{\mathcal{FS}}$ la interpretación t_1 `subset`$^{\mathcal{FS}} t_2 \rightarrow t$ debe cumplir alguno de los siguientes casos: o t_1 y t_2 son conjuntos de elementos de tipo `int` tales que t_1 es un subconjunto de t_2 y $t =$ `true`; o bien t_1 y t_2 son conjuntos de elementos de tipo `int` tales que t_1 es un superconjunto estricto de t_2 y $t =$ `false`; o bien $t = \bot$. La interpretación `superset`$^{\mathcal{FS}}$ se define análogamente.
- `intersect`$^{\mathcal{FS}}$, `union`$^{\mathcal{FS}}$, `diff`$^{\mathcal{FS}}$ y `symdiff`$^{\mathcal{FS}}$: para todo t_1, t_2, t_3, $t \in \mathcal{U}_{\mathcal{FS}}$ la interpretación `intersect`$^{\mathcal{FS}} t_1\ t_2\ t_3 \rightarrow t$ debe cumplir alguno de los siguientes casos: o t_1, t_2 y t_3 son conjuntos de elementos de tipo `int` tales que la intersección de t_1 y t_2 es el conjunto t_3 y $t =$ `true`; o bien la intersección de t_1 y t_2 es distinta al conjunto t_3 y $t =$ `false`; o bien $t = \bot$. Las interpretaciones `union`$^{\mathcal{FS}}$, `diff`$^{\mathcal{FS}}$ y `symdiff`$^{\mathcal{FS}}$ se definen análogamente.

- `intersections`$^{\mathcal{FS}}$ y `unions`$^{\mathcal{FS}}$: Estas interpretaciones son análogas a las interpretaciones `intersect`$^{\mathcal{FS}}$ y `union`$^{\mathcal{FS}}$ pues estas primitivas extienden la intersección y la unión a una lista de conjuntos.

- `disjoints`$^{\mathcal{FS}}$: para todo t_1, $t \in \mathcal{U}_{\mathcal{FS}}$ la interpretación `disjoints`$^{\mathcal{FS}} t_1 \rightarrow t$ debe cumplir alguno de los siguientes casos: o bien t_1 es una lista de conjuntos de elementos de tipo `int` mutuamente disjuntos y $t =$ `true`; o bien t_1 es una lista de conjuntos de elementos de tipo `int` no mutuamente disjuntos y $t =$ `false`; o bien $t = \bot$.

- `isIn`$^{\mathcal{FS}}$ y `isNotIn`$^{\mathcal{FS}}$: para todo t_1, t_2, $t \in \mathcal{U}_{\mathcal{FS}}$ la interpretación `isIn`$^{\mathcal{FS}}$ $t_1\, t_2 \rightarrow t$ debe cumplir alguno de los siguientes casos: o bien t_1 es un elemento de tipo `int`, t_2 es un conjunto de elementos de tipo `int` que contiene a t_1 y $t =$ `true`; o bien t_1 es un elemento de tipo `int`, t_2 es un conjunto de elementos de tipo `int` que no contiene a t_1 y $t =$ `false`; o bien $t = \bot$. La interpretación `isNotIn`$^{\mathcal{FS}}$ se define análogamente a la interpretación `isIn`$^{\mathcal{FS}}$.

- `labelingsets`$^{\mathcal{FS}}$: para todo t_1, $t \in \mathcal{U}_{\mathcal{FS}}$ la interpretación `labelingSets`$^{\mathcal{FD}}$ $t_1 \rightarrow t$ debe cumplir alguno de los siguientes casos: o bien t_1 es una lista finita de conjuntos de elementos de tipo `int` y $t =$ `true`; o bien $t = \bot$.

- `<<`$^{\mathcal{FS}}$ para todo t_1, t_2, $t \in \mathcal{U}_{\mathcal{FS}}$ la interpretación `<<`$^{\mathcal{FS}}$ $t_1\, t_2 \rightarrow t$ debe cumplir alguno de los siguientes casos: o bien t_1 y t_2 son conjuntos de elementos de tipos `int` tal que el supremo de t_1 es menor que el ínfimo de t_2 y t es `true`; o bien t_1 y t_2 son conjuntos de elementos de tipos `int` tal que el supremo de t_1 es mayor o igual que el ínfimo de t_2 y t es `false`; o bien $t = \bot$.

Las restricciones atómicas de $\mathcal{FS}$ son de la forma $p\, \overline{e}_n \rightarrow! \, t$ donde p es uno de los símbolos de función primitiva descritos, cada e_i es una expresión y t es un patrón total. Como en los dominios anteriores, las restricciones atómicas de $\mathcal{FS}$ se denominan *restricciones de Herbrand extendidas* si tienen la forma e_1 `==` $e_2 \rightarrow! \, t$ y *propias* en otro caso.

Con respecto al resolutor *solve*$^{\mathcal{FS}}$, se espera que sea capaz de tratar con restricciones $\mathcal{FS}$-específicas $\Pi \subseteq APCon_{\mathcal{FS}}$ que consisten en restricciones primitivas atómicas π divididas en las siguientes dos clases:

- Restricciones *propias de* $\mathcal{FS}$ tienen la forma $p\, \overline{t}_n \rightarrow! \, t$ donde p es uno de los símbolos de función primitiva descritos excepto `==`, cada t_i es un patrón y t es un patrón total.

- Restricciones $\mathcal{FS}$*-específicas de Herbrand*, t_1 `==` t_2 o bien t_1 `/=` t_2, donde cada patrón t_1 y t_2 es o bien un conjunto de elementos enteros o bien una variable de tipo `set`.

El resolutor *solve*$^{\mathcal{FS}}$ ha sido desarrollado utilizando la biblioteca de los conjuntos de enteros `ic_sets` de ECL^{i}PS^e. Como se motivó para el resolutor *solve*$^{\mathcal{FD}}$, hay cierto tipo de objetivos que no se pueden satisfacer y sin embargo el resolutor de ECL^{i}PS^e no es capaz de inferirlo. Para ilustrar esta idea se muestra a continuación el siguiente ejemplo similar al planteado para $\mathcal{FD}$ por el mismo motivo:

(1) `intSets [X,Y] 1 10000`, `subset X Y`, `subset Y X`, `X /= Y`

La mayoría de los sistemas en general no detectan la inconsistencia de objetivos semejantes al objetivo (1) dado que se aplican algoritmos de propagación incompletos. Solo al añadir el etiquetado de variables se puede asegurar la satisfacción del objetivo. Si al objetivo (1) se le añade la restricción de etiquetado entonces forma el objetivo:

(2) `intSets [X,Y] 1 10000`, `subset X Y`, `subset Y X`, `X /= Y`, `labelingSets [X,Y]`

Con el etiquetado todos los sistemas son capaces de detectar el fallo de los objetivos correspondientes a (2), pero recorrer los retículos definidos por `X` e `Y` puede ser computacionalmente costoso. De forma similar a $\mathcal{FD}$, se ha implementado un mecanismo para anticipar el fallo e inferir la inconsistencia del objetivo (1) sin necesidad de recorrer los retículos definidos por `X` e `Y`. Este mecanismo se puede activar opcionalmente mediante el comando `/fs` como ya se ha indicado en el caso de $\mathcal{FD}$.

Por lo expuesto anteriormente, se define $solve^{\mathcal{FS}}$ como la secuenciación de dos resolutores, $solve^{\mathcal{FS}^{\mathcal{T}}} \diamond solve^{\mathcal{FS}^{N}}$. El primero es un resolutor de caja transparente que anticipa el fallo para ciertos casos y el segundo es un resolutor de caja negra que hace uso de la biblioteca `ic_sets` de ECL^{i}PS^e.

El resolutor de caja transparente $solve^{\mathcal{FS}^{\mathcal{T}}}$ se ha formalizado usando un sistema de transformación de almacenes cuyas reglas de transformación están descritas en la tabla 3.2, donde todos los valores S_i representan variables de tipo `set` o conjuntos básicos, si no se especifica lo contrario. La inconsistencia del sistema se detecta de forma simbólica, por ello las reglas comprendidas entre **S1** y **S10** infieren nuevas restricciones de igualdad, desigualdad o subconjunto para que se pueda anticipar el fallo de forma simbólica junto con la regla **S11**. Por ejemplo, aplicando la regla **S5** en el objetivo anterior (1) se tiene que las restricciones `subset X Y`, `subset Y X` producen la restricción `X == Y` que es inconsistente con `X /= Y` por la regla **S11**. En particular, la regla **S1** genera las restricciones de desigualdad S_1 `/= {}`,..., S_n `/= {}` cuando se procesa la restricción `domainSets` $[S_1, \ldots, S_n]$ `lb ub` y `lb` no es el conjunto vacío. Las reglas **S2**, **S3** y **S10** también infieren algunos conjuntos distintos del conjunto vacío. Las reglas **S4**, **S5** y **S8** infieren igualdades y finalmente las reglas **S6**, **S7** y **S9** infieren restricciones `subset`. Es importante mencionar que las reglas **S2**, **S3**, **S6**, **S7**, **S9** y **S10** no producen un cómputo no terminante, pues cada una de estas reglas solo es aplicable una vez sobre los mismos argumentos.

De forma similar al resolutor $solve^{\mathcal{FD}^{\mathcal{T}}}$, este resolutor se puede ampliar con otras reglas con el objetivo de anticipar el fallo antes de resolver las restricciones y de tratar la cooperación entre dominios.

$solve^{\mathcal{FS}^{\mathcal{T}}}$ está implementado como un resolutor de caja transparente y por lo tanto tiene que cumplir los requisitos de los sistemas de transformación de almacenes detallados en la definición 6. Estos requisitos se enuncian como los resultados semánticos que se muestran en el teorema 2, y su demostración se encuentra en el apéndice A.4.

Teorema 2. (Propiedades formales de $solve^{\mathcal{FS}^{\mathcal{T}}}$)

El sistema de transformación de almacenes con relación de transición $\vdash_{\mathcal{FS}^{\mathcal{T}}}$ cumple las propiedades de la definición 6. Además, como se mostró en la definición 5:

S1	`domainSets` $[S_1, \ldots, S_n]$ `lb ub`, `lb/={}`, $\Pi \square \sigma \Vdash_{\mathcal{FS}^\mathcal{T}}$
	`domainSets` $[S_1, \ldots, S_n]$ `lb ub`, `lb/={}`, S_1`/={}`,$\ldots$, S_n `/= {}`, $\Pi \square \sigma$
	si $S_1, \ldots, S_n \in \mathcal{V}ar$
S2	`isIn` X S, $\Pi \square \sigma \Vdash_{\mathcal{FS}^\mathcal{T}}$ `isIn` X S, S `/= {}`, $\Pi \square \sigma$
S3	`subset` S_1 S_2, S_1 `/= {}`, $\Pi \square \sigma \Vdash_{\mathcal{FS}^\mathcal{T}}$
	`subset` S_1 S_2, S_1 `/= {}`, S_2 `/= {}`, $\Pi \square \sigma$
S4	`subset` S_1 S_2, S_2 `== {}`, $\Pi \square \sigma \Vdash_{\mathcal{FS}^\mathcal{T}}$ S_2 `== {}`, S_1 `== {}`, $\Pi \square \sigma$
S5	`subset` S_1 S_2, `subset` S_2 S_1, $\Pi \square \sigma \Vdash_{\mathcal{FS}^\mathcal{T}}$ S_1 `==` S_2, $\Pi \square \sigma$
S6	`union` S_1 S_2 S_1, $\Pi \square \sigma \Vdash_{\mathcal{FS}^\mathcal{T}}$ `union` S_1 S_2 S_1, `subset` S_2 S_1, $\Pi \square \sigma$
S7	`union` S_1 S_2 S_2, $\Pi \square \sigma \Vdash_{\mathcal{FS}^\mathcal{T}}$ `union` S_1 S_2 S_2, `subset` S_1 S_2, $\Pi \square \sigma$
S8	`union` S_1 S_2 S_3, S_3 `== {}`, $\Pi \square \sigma \Vdash_{\mathcal{FS}^\mathcal{T}}$
	S_1 `== {}`, S_2 `== {}`, S_3 `== {}` $\Pi \square \sigma$
S9	`union` S_1 S_2 S_3 $\Pi \square \sigma \Vdash_{\mathcal{FS}^\mathcal{T}}$
	`union` S_1 S_2 S_3, `subset` S_1 S_3, `subset` S_2 S_3, $\Pi \square \sigma$
S10	`intersect` S_1 S_2 S_3, S_3`/={}`, $\Pi \square \sigma \Vdash_{\mathcal{FS}^\mathcal{T}}$
	`intersect` S_1 S_2 S_3, S_3`/={}`, S_1 `/= {}`, S_2 `/= {}`, $\Pi \square \sigma$
S11	S_1 `==` S_2, S_1 `/=` S_2, $\Pi \square \sigma \Vdash_{\mathcal{FS}^\mathcal{T}}$ ■

Tabla 3.2: Reglas de transformación de almacenes que definen $\Vdash_{\mathcal{FS}^\mathcal{T}}$

$solve^{\mathcal{FS}^\mathcal{T}}(\Pi) = \bigvee_{j=1}^{k}\{(\Pi_j \square \sigma_j) \mid \Pi_j \square \sigma_j \in SF_{\mathcal{FS}^\mathcal{T}}(\Pi), \overline{Y}_j = var(\Pi_j \square \sigma_j) \setminus var(\Pi)\}$ está bien definido para cualquier conjunto finito $\Pi \subseteq APCon_{\mathcal{FS}^\mathcal{T}}$ de restricciones $\mathcal{FS}$-específicas. El resolutor $solve^{\mathcal{FS}^\mathcal{T}}$ satisface todos los requisitos de los resolutores de la definición 4.

$solve^{\mathcal{FS}^N}$ está implementado como un resolutor de caja negra sobre ECL^{i}PS^e y reduce restricciones primitivas a una forma resuelta. Postulamos que se cumplen las condiciones que se requieren para los resolutores declaradas en la definición 4, aunque la propiedad de completitud puede fallar en alguna elección de $\Pi \subseteq APCon_{\mathcal{FS}}$. El comportamiento que se espera de $solve^{\mathcal{FS}^N}$ se postula a continuación.

Postulado 3. Supuestos sobre el resolutor $\mathcal{FS}^N$

El resolutor $solve^{\mathcal{FS}^N}$ satisface las propiedades de *variables locales nuevas*, *formas resueltas* y *corrección* requeridas para un resolutor en la definición 4. La propiedad de *completitud* puede fallar para algunas elecciones de restricciones $\mathcal{FS}$-específicas elegidas para ser resueltas. Además, siempre que $\Pi \subseteq APCon_{\mathcal{FS}}$ es $\mathcal{FS}$-específico y $\Pi \Vdash_{solve^{\mathcal{FS}^N}} \exists \overline{Y'}(\Pi' \square \sigma')$, el conjunto de restricciones Π' es también $\mathcal{FS}$-específico, y para todo $X \in vdom(\sigma')$: o bien $X\sigma'$ es un conjunto de números enteros, o si no X y $X\sigma'$ pertenecen a $var(\Pi)$.

□

3.5 El dominio $\mathcal{H}$

La idea de utilizar restricciones de igualdad y desigualdad en programación lógica proviene de Colmerauer [Col84, Col90]. El problema de resolver estas restricciones, así como los problemas de decisión relacionados sobre teorías que involucran ecuaciones e inecuaciones, ha sido ampliamente investigado en trabajos como los siguientes [LMM88, Mah88, CL89, Com91, Fer92, BB94]. Estos trabajos asumen la semántica algebraica clásica para la relación de igualdad y proponen métodos para resolver lo que denominan problemas de unificación y desunificación. Estos métodos tienen bastantes analogías con el sistema de transformación de almacenes que se va a definir en esta sección para el resolutor $solve^{\mathcal{H}}$ y también algunas diferencias, ya que la igualdad estricta en $CFLP$ ha sido diseñada para tratar con funciones perezosas y posiblemente no deterministas, cuyo comportamiento no se corresponde con la semántica de la igualdad en el álgebra clásica y la lógica ecuacional, como se argumenta en [Rod02]. Las restricciones de igualdad y desigualdad se trataron en el trabajo [AGL94] pero no desarrolla ninguna formalización de un resolutor de Herbrand.

En el esquema $CFLP$ el dominio de Herbrand $\mathcal{H}$ trata restricciones de igualdad y desigualdad simbólicas sobre valores de cualquier tipo y se define con la signatura específica $\Sigma_{\mathcal{H}} = \langle TC,\ SBT_{\mathcal{H}},\ DC,\ DF,\ SPF_{\mathcal{H}}\rangle$, donde $SBT_{\mathcal{H}}$ es el conjunto vacío y $SPF_{\mathcal{H}}$ incluye únicamente el operador de igualdad estricta `== :: A -> A -> bool`. Recuérdese que la interpretación de la restricción de igualdad está definida para cualquier dominio cuya signatura específica incluya el símbolo `==`.

Una extensión conservativa de $\mathcal{H}$ es cualquier dominio $\mathcal{D}$ cuya signatura específica incluya la primitiva `==`. A partir de ahora a este dominio $\mathcal{D}$ se le llamará *dominio con igualdad*. Las restricciones {`==`}-restringidas sobre un dominio con igualdad son llamadas *restricciones de Herbrand extendidas*.

El resolutor de Herbrand que se va a definir a continuación reduce un conjunto finito de restricciones primitivas atómicas Π a forma resuelta teniendo en cuenta ciertas variables de Π que se denominan *variables críticas* y que no existen en los demás resolutores que se tratan en esta tesis. La idea es que las variables críticas pueden ser necesarias para algunas soluciones de Π e irrelevantes para otras. Por ejemplo, supongamos la restricción `(A,2) == (1,B)`, si `A` es distinto de `1` entonces la igualdad falla y ya no es necesario el valor de `B`. Por lo tanto, dependiendo del valor que tome la variable `A`, será necesario o irrelevante el cómputo de `B`. En este caso se dice que `A` y `B` son variables críticas. Una solución $\eta \in Sol_{\mathcal{D}}(\Pi)$ que vincula una variable X al valor indefinido $\bot$ quiere decir que el valor de la variable X no se necesita para comprobar la satisfacción de las restricciones contenidas en Π. Si el valor de la variable es necesario entonces se dice que la variable es demandada.

Formalmente, una variable X es *demandada* por un conjunto de restricciones $\Pi \subseteq PCon_{\mathcal{D}}$ si y solamente si $\eta(X) \neq \bot$ para todo $\eta \in Sol_{\mathcal{D}}(\Pi)$. La notación $dvar_{\mathcal{D}}(\Pi)$ representa el conjunto de todas las variables que son demandadas por Π.

En la práctica se pueden reconocer apariciones 'obvias' de variables demandadas en restricciones primitivas atómicas. Por lo tanto, se asume que para cualquier dominio de restric-

ciones $\mathcal{D}$ y cualquier $\pi \in APCon_{\mathcal{D}}$ existe una manera de reconocer el conjunto $odvar_{\mathcal{D}}(\pi)$ de variables obviamente demandadas por la restricción π. La notación $X \in odvar_{\mathcal{D}}(\pi)$ establece que X es *obviamente demandada* por π y $odvar_{\mathcal{D}}(\Pi)$ es el conjunto de todas las variables *obviamente demandadas* contenidas en el conjunto de restricciones Π. Por lo tanto, se cumple la inclusión $odvar_{\mathcal{D}}(\Pi) \subseteq dvar_{\mathcal{D}}(\Pi)$ para cualquier $\Pi \subseteq APCon_{\mathcal{D}}$.

En particular, para cualquier dominio de restricciones $\mathcal{D}$ cuya signatura específica incluya la primitiva de igualdad estricta `==` y para cualquier restricción primitiva atómica $\pi = (t_1 \texttt{==} t_2 \rightarrow\!! \, t)$, se define $odvar_{\mathcal{D}}(\pi)$ por distinción de casos como se muestra a continuación:

- $odvar_{\mathcal{D}}(t_1 \texttt{==} t_2 \rightarrow\!! \, R) = \{R\}$, si $R \in \mathcal{V}ar$.
- $odvar_{\mathcal{D}}(X \texttt{==} Y) = \{X, Y\}$, si $X, Y \in \mathcal{V}ar$.
- $odvar_{\mathcal{D}}(X \texttt{==} t) = odvar_{\mathcal{D}}(t \texttt{==} X) = \{X\}$, si $X \in \mathcal{V}ar$ y $t \notin \mathcal{V}ar$.
- $odvar_{\mathcal{D}}(t_1 \texttt{==} t_2) = \emptyset$, en otro caso.
- $odvar_{\mathcal{D}}(X \texttt{/=} Y) = \{X, Y\}$, si $X, Y \in \mathcal{V}ar$ y X e Y no son idénticas.
- $odvar_{\mathcal{D}}(X \texttt{/=} t) = odvar_{\mathcal{D}}(t \texttt{/=} X) = \{X\}$, si $X \in \mathcal{V}ar$ y $t \notin \mathcal{V}ar$.
- $odvar_{\mathcal{D}}(t_1 \texttt{/=} t_2) = \emptyset$, en otro caso.

Dado un conjunto de restricciones primitivas atómicas Π de un dominio $\mathcal{D}$, se define como el conjunto de variables críticas de Π ($cvar_{\mathcal{D}}(\Pi)$) a aquellas variables que no son obviamente demandadas: $cvar_{\mathcal{D}}(\Pi) = var(\Pi) \setminus odvar_{\mathcal{D}}(\Pi)$.

Para clarificar los conceptos de variables críticas, demandadas y obviamente demandadas vamos a estudiar las diferentes variables contenidas en la restricción $\pi \equiv$ `L /= X:Xs`. La variable `L` es obviamente demandada por π por la definición que acabamos de ver, mientras que las variables `X` y `Xs` no son obviamente demandadas. Por otra parte, se puede argumentar que ni `X` ni `Xs` son demandadas por π. La variable `X` no es demandada porque existe una solución $\eta \in sol_{\mathcal{D}}(\pi)$ tal que $\eta(\texttt{X}) = \bot$ (por ejemplo, si $\eta(\texttt{L}) = [\,]$, o bien $\eta(\texttt{L}) = \texttt{t} : \texttt{ts}$ tal que $\eta(\texttt{Xs})$ es distinto de `ts`), la variable `Xs` no es demandada por razones similares. Por lo tanto, las variables `X` y `Xs` son críticas, lo que conlleva a que existan cuatro posibles elecciones para el conjunto $\mathcal{X}$ de variables críticas $\mathcal{X} \subseteq cvar_{\mathcal{D}}(\pi)$. Estas son: $\emptyset$, {`X`}, {`Xs`} y {`X`, `Xs`}.

Las restricciones $\mathcal{D}$-específicas, donde $\mathcal{D}$ es uno de los dominios puros $\mathcal{FD}$, $\mathcal{R}$ y $\mathcal{FS}$ que se han definido anteriormente, tienen todas sus variables obviamente demandadas y por lo tanto no contienen variables críticas. El único dominio que se trata en esta tesis que contiene variables críticas es el dominio $\mathcal{H}$. Por este motivo se van a extender a continuación las definiciones 4 y 6 para tratar el caso particular de $\mathcal{H}$.

Teniendo en cuenta las variables críticas, el resolutor para el dominio $\mathcal{H}$ se define como una función $solve^{\mathcal{H}}$ que se puede aplicar a pares de la forma $(\Pi, \mathcal{X})$, donde $\Pi \subseteq APCon_{\mathcal{D}}$ es un conjunto finito de restricciones primitivas atómicas y $\mathcal{X} \subseteq cvar_{\mathcal{D}}(\Pi)$ es un conjunto finito que incluye algunas de las variables *críticas* de Π. La notación $solve^{\mathcal{H}}(\Pi, \emptyset)$ se abrevia como $solve^{\mathcal{H}}(\Pi)$. Antes de definir formalmente el resolutor $solve^{\mathcal{H}}$ es necesario ampliar los requisitos que se piden en la definición 4 para tratar variables críticas y también las propiedades de

los sistemas de transformación de almacenes establecidas en la definición 6, pues el resolutor $solve^{\mathcal{H}}$ se va a definir utilizando este mecanismo.

Definición 7. (Definición y requisitos formales para el resolutor del dominio $\mathcal{H}$)
El resolutor de restricciones para el dominio $\mathcal{H}$ se modela como una función $solve^{\mathcal{H}}(\Pi, \mathcal{X})$, donde $\Pi \subseteq APCon_{\mathcal{D}}$ es un conjunto finito de restricciones primitivas atómicas y $\mathcal{X} \subseteq cvar_{\mathcal{D}}(\Pi)$ es un conjunto finito que incluye algunas de las variables críticas de Π. Cualquier invocación al resolutor $solve^{\mathcal{H}}(\Pi, \mathcal{X})$ debe devolver una disyunción finita $\bigvee_{j=1}^{k} \exists \overline{Y}_j (\Pi_j \,\square\, \sigma_j)$ de almacenes de restricciones existencialmente cuantificados, cumpliendo las mismas condiciones de *variables locales nuevas*, *corrección* y *completitud* de la definición 4 y además:

1. **Formas resueltas con respecto a $\mathcal{X}$:** para todo $1 \leq j \leq k$, $\Pi_j \,\square\, \sigma_j$ está en forma resuelta con respecto a $\mathcal{X}$. Por definición esto significa que $solve^{\mathcal{H}}(\Pi_j, \mathcal{X}) = \Pi_j \,\square\, \varepsilon$.

2. **Vinculaciones seguras:** para todo $1 \leq j \leq k$ y para todo $X \in \mathcal{X} \cap vdom(\sigma_j)$, $\sigma_j(X)$ es una constante.

3. **Discriminación:** cada *forma resuelta* $\Pi_j \,\square\, \sigma_j$ $(1 \leq j \leq k)$, computada con respecto a $\mathcal{X}$, debe satisfacer: o bien $\mathcal{X} \cap odvar_{\mathcal{D}}(\Pi_j) \neq \emptyset$ o bien $\mathcal{X} \cap var(\Pi_j) = \emptyset$.

   Es decir, o bien alguna variable crítica en $\mathcal{X}$ se convierte en obviamente demandada, o bien todas las variables críticas de $\mathcal{X}$ desaparecen.

Como acabamos de ver, las invocaciones al resolutor $solve^{\mathcal{H}}(\Pi, \mathcal{X})$ dependen del conjunto de variables críticas que se elijan. En el capítulo 4, cuando se desarrolle el cálculo de resolución de objetivos, se motivará el modo más adecuado de elegir ese conjunto $\mathcal{X}$ de variables críticas.

Con respecto a las formas resueltas, recordamos que en la práctica pueden reconocerse por medio de criterios sintácticos, de forma que una invocación al resolutor $solve^{\mathcal{H}}(\Pi, \mathcal{X})$ se ejecuta solo si el almacén $\Pi \square \sigma$ no está en forma resuelta con respecto a $\mathcal{X}$.

A continuación se ilustra la propiedad de discriminación a través de la restricción $\pi \equiv$ `L /= X:Xs` vista anteriormente. En esta restricción la variable `L` es obviamente demandada y las variables `X` y `Xs` son críticas. Como se vio, hay cuatro elecciones posibles para el conjunto de variables críticas $\mathcal{X} \subseteq cvar_{\mathcal{D}}(\pi)$ en las distintas invocaciones del resolutor. El conjunto $\mathcal{X}$ puede ser: $\emptyset$, {`X`}, {`Xs`} y {`X`, `Xs`}. Aunque el resolutor $solve^{\mathcal{H}}(\Pi, \mathcal{X})$ se definirá formalmente más adelante en esta sección, se anticipan los resultados de sus invocaciones para ilustrar la propiedad de discriminación. Veamos estos casos uno por uno.

- Si $\mathcal{X} = \emptyset$ entonces significa que al resolutor $solve^{\mathcal{H}}$ no se le pide ser discriminante con respecto a ninguna variable crítica, de forma semejante a los resolutores $solve^{\mathcal{R}}$, $solve^{\mathcal{FD}}$ y $solve^{\mathcal{FS}}$. Por lo tanto, $solve^{\mathcal{H}}$(`L/=X:Xs`,$\emptyset$) devuelve `L/=X:Xs` $\square\, \varepsilon$, donde `L/=X:Xs` es una forma resuelta con respecto al conjunto vacío de variables críticas ($solve^{\mathcal{H}}$(`L/=X:Xs`,$\emptyset$) devuelve `L/=X:Xs` al aplicar las reglas **H11** y **H12** del sistema

de transformación de almacenes de la tabla 3.3, que se describe más adelante en esta sección).

- Si $\mathcal{X} = \{\texttt{X}\}$ hace que el resolutor sea discriminante con respecto a la variable crítica `X`. $solve^{\mathcal{H}}(\texttt{L/=X:Xs},\{\texttt{X}\})$ devuelve una disyunción de alternativas:

$$(\Diamond \,\square\{\texttt{L} \mapsto [\,]\}) \vee (\texttt{X}'/= \texttt{X} \,\square\{\texttt{L} \mapsto \texttt{X}' : \texttt{Xs}'\}) \vee (\texttt{Xs}'/= \texttt{Xs} \,\square\{\texttt{L} \mapsto \texttt{X}' : \texttt{Xs}'\})$$

  Hay que tener en cuenta que `X` no se encuentra ni en la primera y ni en la tercera forma resuelta con respecto a $\mathcal{X}$, mientras que en la segunda forma resuelta con respecto a $\mathcal{X}$ se ha convertido en obviamente demandada. Por lo tanto cumple la propiedad de discriminación: o bien alguna variable crítica se convierte en obviamente demandada, o bien todas las variables críticas desaparecen.

- Para cada una de las dos elecciones de conjuntos de variables críticas $\mathcal{X} = \{\texttt{Xs}\}$ y $\mathcal{X} = \{\texttt{X}, \texttt{Xs}\}$, la invocación $solve^{\mathcal{H}}(\texttt{L/=X:Xs}, \mathcal{X})$ devuelve la misma disyunción de tres alternativas del punto anterior. Por lo tanto, la propiedad discriminación también se cumple con respecto al conjunto elegido $\mathcal{X}$ en ambos casos.

Se puede observar que la selección del conjunto de variables críticas de una restricción de Herbrand es relevante para la información que aportan las soluciones que devuelve el resolutor.

De forma similar a la extensión de la definición de resolutor y sus requisitos formales cuando se tratan variables críticas, también se necesita ampliar la definición y las propiedades de los sistemas de transformación de almacenes. Pero antes se introducen algunos conceptos necesarios.

Dado un resolutor $solve^{\mathcal{D}}$ definido mediante un sistema de transformación de almacenes con relación de transición $\vdash_{\mathcal{D}}$ y un subconjunto de las reglas de transformación de almacenes $\mathcal{RS}$, si el sistema de transformación de almacenes es terminante entonces se dice que un almacén $\Pi \,\square\, \sigma$ es $\mathcal{RS}$-*irreducible* si no se puede aplicar ninguna regla del conjunto $\mathcal{RS}$ al almacén $\Pi \,\square\, \sigma$. El almacén $\Pi \,\square\, \sigma$ es *hereditariamente* $\mathcal{RS}$-*irreducible* si y solamente si $\Pi \,\square\, \sigma$ es $\mathcal{RS}$-*irreducible* y todos los almacenes $\Pi' \,\square\, \sigma'$ tal que $\Pi \,\square\, \sigma \vdash_{\mathcal{D},\mathcal{X}} \Pi' \,\square\, \sigma'$ (si existe alguno) son también *hereditariamente* $\mathcal{RS}$-*irreducibles*. Una invocación $solve^{\mathcal{D}}(\Pi)$ es $\mathcal{RS}$-*libre* si y solamente si el almacén $\Pi \,\square\, \varepsilon$ es hereditariamente $\mathcal{RS}$-irreducible. Este concepto es necesario para garantizar la propiedad de completitud del resolutor de $\mathcal{H}$, como se verá en la explicación de la tabla 3.3 con respecto a las reglas **H3**, **H7** y **H13**.

Definición 8. (Propiedades del sistema de transformación de almacenes de $\mathcal{H}$)
Sea un sistema de transformación de almacenes sobre $\mathcal{H}$ cuya relación de transición es $\vdash_{\mathcal{H},\mathcal{X}}$. Entonces el sistema de transformación de almacenes debe satisfacer las propiedades de *variables locales nuevas*, *ramificación finita*, *terminación* y *corrección local* de la definición 6, y además:

1. **Vinculaciones seguras:** si $\Pi \,\square\, \sigma \vdash_{\mathcal{H},\mathcal{X}} \Pi' \,\square\, \sigma'$ entonces $\sigma_1(X)$ es una constante para todo $X \in \mathcal{X} \cap vdom(\sigma_1)$, donde $\sigma' = \sigma\sigma_1$ para alguna sustitución σ_1 (responsable de las vinculaciones de variables creadas en este paso) tal que $vdom(\sigma_1) \cup vran(\sigma_1) \subseteq var(\Pi) \cup \overline{Y'}$.

2. **Completitud local para pasos $\mathcal{RS}$-libres:** sea $\mathcal{RS}$ un conjunto de reglas de transformación de almacenes. Si para cualquier almacén $\Pi \,\square\, \sigma$ que es $\mathcal{RS}$-irreducible pero no está en forma resuelta con respecto a $\mathcal{X}$, $WTSol_{\mathcal{H}}(\Pi \,\square\, \sigma)$ es un subconjunto de la unión

$$\bigcup\{WTSol_{\mathcal{H}}(\exists\overline{Y'}(\Pi' \,\square\, \sigma')) \mid \Pi\square\sigma \vdash_{\mathcal{H},\mathcal{X}} \Pi'\square\sigma', \overline{Y'} = var(\Pi'\square\sigma') \setminus var(\Pi\square\sigma)\}$$

El siguiente lema asegura que $solve^{\mathcal{H}}$ satisface los requisitos de la definición 7.

Lema 3. (Resolutores definidos por medio de sistemas de transformación de almacenes)

Cualquier sistema de transformación de almacenes de un dominio $\mathcal{D}$ con ramificación finita y terminante verifica:

1. $SF_{\mathcal{D}}(\Pi, \mathcal{X})$ es siempre finito, y por lo tanto $solve^{\mathcal{D}}$ está bien definido y satisface la propiedad de las formas resueltas.

2. $solve^{\mathcal{D}}$ tiene la propiedad de variables locales nuevas y vinculación segura si el sistema de transformación de almacenes tiene la propiedad correspondiente.

3. $solve^{\mathcal{D}}$ es correcta si el sistema de transformación de almacenes es localmente correcto.

4. $solve^{\mathcal{D}}$ es completo para llamadas $\mathcal{RS}$-libres si el sistema de transformación de almacenes es localmente completo para pasos $\mathcal{RS}$-libres.

   Si $\mathcal{RS}$ es el conjunto vacío, entonces $solve^{\mathcal{D}}$ es completo si el sistema de transformación de almacenes es localmente completo.

□

El lema 3 se puede usar para demostrar las propiedades globales de los resolutores extensibles, siempre que el sistema de transformación de almacenes trate con almacenes $\Pi \,\square\, \sigma$, donde Π es un conjunto finito de restricciones primitivas atómicas SPF-restringidas sobre alguna extensión conservativa $\mathcal{D}$' de $\mathcal{D}$, y las propiedades locales requeridas por el lema 1 se cumplan para $\mathcal{D}$'.

El resolutor $solve^{\mathcal{H}}$ debe ser capaz de resolver cualquier conjunto finito de restricciones extendidas de Herbrand primitivas atómicas ($\Pi \subseteq APCon_{\mathcal{D}} \restriction_{\{==\}}$) con respecto a cualquier

H1 $(t == s) \rightarrow! R,\ \Pi \square \sigma \vdash_{\mathcal{H},\mathcal{X}} (t == s,\ \Pi)\sigma_1 \square \sigma\sigma_1$ donde $\sigma_1 = \{R \mapsto true\}$.

H2 $(t == s) \rightarrow! R,\ \Pi \square \sigma \vdash_{\mathcal{H},\mathcal{X}} (t\ /= s,\ \Pi)\sigma_1 \square \sigma\sigma_1$ donde $\sigma_1 = \{R \mapsto false\}$.

H3 $h\,\overline{t}_m == h\,\overline{s}_m,\ \Pi \square \sigma \vdash_{\mathcal{H},\mathcal{X}} t_1 == s_1, \ldots, t_m == s_m,\ \Pi \square \sigma$

H4 $t == X,\ \Pi \square \sigma \vdash_{\mathcal{H},\mathcal{X}} X == t,\ \Pi \square \sigma$ si t no es una variable.

H5 $X == t,\ \Pi \square \sigma \vdash_{\mathcal{H},\mathcal{X}} tot(t),\ \Pi\sigma_1 \square \sigma\sigma_1$ si $X \notin \mathcal{X}$, $X \notin var(t)$, $X \neq t$, donde $\sigma_1 = \{X \mapsto t\}$, $tot(t)$ abrevia $\bigwedge_{Y \in var(t)}(Y == Y)$.

H6 $X == t,\ \Pi \square \sigma \vdash_{\mathcal{H},\mathcal{X}} \blacksquare$ if $X \in var(t)$, $X \neq t$.

H7 $h\,\overline{t}_m\ /= h\,\overline{s}_m,\ \Pi \square \sigma \vdash_{\mathcal{H},\mathcal{X}} (t_i\ /= s_i,\ \Pi \square \sigma)$ para cada $1 \leq i \leq m$ (es una elección no determinista).

H8 $h\,\overline{t}_n\ /= h'\,\overline{s}_m,\ \Pi \square \sigma \vdash_{\mathcal{H},\mathcal{X}} \Pi \square \sigma$ si $h \neq h'$ o $n \neq m$.

H9 $t\ /= t,\ \Pi \square \sigma \vdash_{\mathcal{H},\mathcal{X}} \blacksquare$ si $t \in \mathcal{V}ar \cup DC \cup DF \cup SPF$.

H10 $t\ /= X,\ \Pi \square \sigma \vdash_{\mathcal{H},\mathcal{X}} X\ /= t,\ \Pi \square \sigma$ si t no es una variable.

H11 $X /= c\,\overline{t}_n, \Pi \square \sigma \vdash_{\mathcal{H},\mathcal{X}} (Z_i /= t_i, \Pi)\sigma_1 \square \sigma\sigma_1$ si $X \notin \mathcal{X}$, $c \in DC^n$ y $\mathcal{X} \cap var(c\,\overline{t}_n) \neq \emptyset$ donde $1 \leq i \leq n$ (es una elección no determinista), $\sigma_1 = \{X \mapsto c\,\overline{Z}_n\}$ y $\overline{Z}_n$ son variables nuevas.

H12 $X\ /= c\,\overline{t}_n,\ \Pi \square \sigma \vdash_{\mathcal{H},\mathcal{X}} \Pi\sigma_1 \square \sigma\sigma_1$ si $X \notin \mathcal{X}$, $c \in DC^n$ y $\mathcal{X} \cap var(c\,\overline{t}_n) \neq \emptyset$ donde $\sigma_1 = \{X \mapsto d\,\overline{Z}_m\}$, $c \in DC^n$, $d \in DC^m$, $d \neq c$, d pertenece al mismo tipo de datos que c y $\overline{Z}_m$ son variables nuevas.

H13 $X\ /= h\,\overline{t}_m,\ \Pi \square \sigma \vdash_{\mathcal{H},\mathcal{X}} \blacksquare$ si $X \notin \mathcal{X}$, $\mathcal{X} \cap var(h\,\overline{t}_m) \neq \emptyset$ y $h \notin DC^m$.

Tabla 3.3: Reglas de transformación de almacenes que definen $\vdash_{\mathcal{H},\mathcal{X}}$

conjunto de variables críticas ($\mathcal{X} \subseteq cvar_{\mathcal{D}}(\Pi)$), y se define mediante un sistema de transformación de almacenes con las reglas de transformación de la tabla 3.3. Cada regla tiene la forma $\pi, \Pi \square \sigma \vdash_{\mathcal{H},\mathcal{X}} \Pi' \square \sigma'$ que muestra el paso de transformación del almacén $\pi, \Pi \square \sigma$, donde π es una restricción atómica seleccionada para el paso de transformación y Π es un conjunto de restricciones. La evaluación de las restricciones no sigue un orden secuencial.

Las reglas **H1** y **H2** tratan la reificación de la restricción $(t == s)$. Las reglas **H3** y **H7** tratan la descomposición que se mostró en la introducción en la subsección 1.3.1. La regla **H4** orienta la igualdad para que sea tratada por las reglas **H5** y **H6**. La regla **H5** genera la sustitución $\sigma_1 = \{X \mapsto t\}$ y añade restricciones de totalidad $Y == Y$, en forma resuelta, para variables $Y \in var(t)$. Una valoración η es una solución de $Y == Y$ si y solamente si $\eta(Y)$ es

un patrón total. Las reglas **H6** y **H9** corresponden a almacenes insatisfactibles y fallan. La regla **H10** orienta la desigualdad para tratarla mediante alguna de las siguientes tres reglas, si se cumplen las condiciones. Las reglas **H11** y **H12** están diseñadas para asegurar la propiedad de discriminación mientras que preservan la completitud con respecto a las soluciones bien tipadas.

Volviendo al estudio de la restricción $\pi \equiv$ `L /= X:Xs` diferentes conjuntos de variables críticas generan distintas soluciones:

- si $\mathcal{X} = \emptyset$ entonces no se puede aplicar ninguna regla pues no se cumple la condición $\mathcal{X} \cap var(\texttt{X:Xs}) \neq \emptyset$. Por lo tanto `L /= X:Xs` está en forma resuelta respecto a $\mathcal{X} = \emptyset$.
- si $\mathcal{X} = \{\texttt{X}\}$ se aplican las reglas **H11** y **H12** de la siguiente forma $\texttt{L/=X:Xs} \,\square\, \varepsilon \vdash_{\mathcal{H},\{\texttt{X}\}} \Pi' \,\square\, \sigma'$ donde $\Pi' \,\square\, \sigma'$ es o bien el almacén $(\Diamond \,\square \{\texttt{L} \mapsto [\,]\})$, por la regla **H12**, o bien aplicando la regla **H11** se obtiene $(\texttt{X}'/ = \texttt{X} \,\square \{\texttt{L} \mapsto \texttt{X}' : \texttt{Xs}'\})$ o $(\texttt{Xs}'/ = \texttt{Xs} \,\square \{\texttt{L} \mapsto \texttt{X}' : \texttt{Xs}'\})$.
- si $\mathcal{X} = \{\texttt{Xs}\}$ o $\mathcal{X} = \{\texttt{X}, \texttt{Xs}\}$ entonces se aplica la regla **H11** obteniendo los mismos resultados que en el punto anterior.

Es importante remarcar que las reglas **H3**, **H7** y **H13** pueden comprometer la completitud. En concreto, las reglas **H3** y **H7** involucran descomposiciones. Como se mostró en la introducción, si la descomposición es opaca, las nuevas restricciones que resultan de la descomposición pueden estar mal tipadas. Los pasos de descomposición no seguros realizados con instancias opacas por las reglas **H3** y **H7** los denominamos **OH3** y **OH7**. Por otra parte, la regla **H13** asegura la discriminación, pero sacrifica la completitud porque esta regla falla sin asegurarse de que no existen soluciones bien tipadas. Esto corresponde a situaciones poco frecuentes y para las que no se dispone de ningún otro modo de preservar la completitud.

Una invocación $solve^{\mathcal{H}}(\Pi, \mathcal{X})$ del resolutor de $\mathcal{H}$ se define como *segura* si y solamente si se ha computado sin ninguna aplicación opaca de las reglas **H3** y **H7** y sin ninguna aplicación de la regla **H13**. Formalmente, $solve^{\mathcal{H}}(\Pi, \mathcal{X})$ es una *invocación segura* del resolutor $\mathcal{H}$ si y solamente si es $\{\mathbf{OH3}, \mathbf{OH7}, \mathbf{H13}\}$-libre, donde el conjunto $\{\mathbf{OH3}, \mathbf{OH7}, \mathbf{H13}\}$ está formado por la regla **H13** e instancias no seguras **OH3** y **OH7** que corresponden a aplicaciones opacas de las reglas **H3** y **H7** respectivamente.

El siguiente teorema asegura que el sistema de transformación de almacenes, definido en la tabla 3.3, puede ser aceptado como una especificación correcta de un resolutor de caja transparente para el dominio $\mathcal{H}$ y es completo para invocaciones seguras.

Teorema 3. (Propiedades formales de $solve^{\mathcal{H}}$)

El sistema de transformación de almacenes con relación de transición $\vdash_{\mathcal{H},\mathcal{X}}$ tiene ramificación finita y es terminante, y por lo tanto:

$$solve^{\mathcal{H}}(\Pi, \mathcal{X}) = \bigvee \{\exists \overline{Y'}(\Pi' \,\square\, \sigma') \mid \Pi' \,\square\, \sigma' \in SF_{\mathcal{H}}(\Pi, \mathcal{X}),\ \overline{Y'} = var(\Pi' \,\square\, \sigma') \setminus var(\Pi)\}$$

está bien definido para cualquier dominio $\mathcal{D}$ con igualdad, cualquier $\Pi \subseteq APCon(\mathcal{D}) \restriction_{\{==\}}$ finito y cualquier $\mathcal{X} \subseteq cvar_{\mathcal{D}}(\Pi)$.

Además, para cualquier elección arbitraria de un dominio $\mathcal{D}$ con igualdad, $solve^{\mathcal{H}}$ satisface todos los requisitos de la definición 7, excepto que la propiedad de *completitud* puede fallar para algunas elecciones del conjunto de restricciones $\Pi \subseteq APCon(\mathcal{D}) \restriction_{\{==\}}$, y está garantizada que se cumple si la invocación al resolutor $solve^{\mathcal{H}}(\Pi, \mathcal{X})$ es segura (es decir, es $\{\mathbf{OH3}, \mathbf{OH7}, \mathbf{H13}\}$-libre).

La propiedad de terminación del sistema de transformación de almacenes de $\mathcal{H}$ implica que determinar cuándo un almacén es hereditariamente $\{\mathbf{OH3}, \mathbf{OH7}, \mathbf{H13}\}$-irreducible sea un problema decidible. De esta forma se asegura que ninguna descomposición opaca se producirá cuando se resuelva dicho almacén. $APCon_{\mathcal{D}} \restriction_{\{==\}}$ es el conjunto de todas las restricciones primitivas atómicas de la forma $\Diamond$, $\blacklozenge$ o $e_1 == e_2 \rightarrow! t$. La demostración del teorema anterior es bastante técnica y se puede encontrar en [EFH$^+$09, dVV08].

El resolutor $solve^{\mathcal{H}}$ definido mediante el sistema de transformación de almacenes (tabla 7) está implementado en el sistema $\mathcal{TOY}$, [AGL94]. Por otra parte el sistema no advierte al usuario cuando se producen descomposiciones opacas.

Capítulo 4

Dominio de coordinación y cálculo $CCLNC(\mathcal{C})$

La cooperación entre dominios de restricciones se puede establecer desde varios enfoques. El enfoque elegido en esta tesis consiste en crear un dominio de coordinación que contenga los dominios de nuestro interés. Este dominio de coordinación debe cumplir ciertas propiedades y sobre él se define un cálculo de resolución de objetivos. En vez de construir el cálculo desde cero, se ha extendido el cálculo de resolución de objetivos $CLNC$ sobre un dominio paramétrico $\mathcal{D}$ [LRV04, LRV07, dVV08]. Esta extensión se denomina $CCLNC(\mathcal{D})$ [EFH$^+$09] y la diferencia fundamental con $CLNC$ corresponde a la cooperación entre dominios de restricciones. El cálculo $CCLNC(\mathcal{D})$ se define mediante reglas que se han separado en dos bloques fundamentales: el estrechamiento perezoso y la cooperación entre dominios de restricciones. En este capítulo se va a mostrar la parte del cálculo $CCLNC(\mathcal{D})$ que trata sobre el estrechamiento perezoso, pues es común a las distintas instancias de los dominios de coordinación. La parte que trata sobre la cooperación entre dominios de restricciones particulares es dependiente de cada dominio de coordinación concreto, y se define en capítulos posteriores correspondientes a esas instancias concretas.

4.1 Dominio de coordinación

Para realizar la cooperación entre dominios es necesario construir un *dominio de coordinación* $\mathcal{C}$, el cual es un dominio híbrido construido como la combinación del dominio de Herbrand $\mathcal{H}$, de varios dominios puros $\mathcal{D}_i$ y un dominio especial creado para la comunicación entre los dominios puros llamado *dominio mediador* $\mathcal{M}$:

$$\mathcal{C} = \mathcal{M} \oplus \mathcal{H} \oplus \mathcal{D}_1 \oplus \cdots \oplus \mathcal{D}_n$$

En concreto se desarrollarán dos instancias del dominio de coordinación:

- $\mathcal{C}_{\mathcal{FD}.\mathcal{R}} = \mathcal{M}_{\mathcal{FD}.\mathcal{R}} \oplus \mathcal{H} \oplus \mathcal{FD} \oplus \mathcal{R}$ que establece la comunicación entre $\mathcal{FD}$ y $\mathcal{R}$.
- $\mathcal{C}_{\mathcal{FD}.\mathcal{FS}} = \mathcal{M}_{\mathcal{FD}.\mathcal{FS}} \oplus \mathcal{H} \oplus \mathcal{FD} \oplus \mathcal{FS}$ que establece la comunicación entre $\mathcal{FD}$ y $\mathcal{FS}$.

La construcción del dominio de coordinación depende de la denominada *condición de unión* que se define de la siguiente manera. Dos dominios $\mathcal{D}_1$ y $\mathcal{D}_2$, con signaturas específicas $\Sigma_1 = \langle TC,\ SBT_1,\ DC,\ DF,\ SPF_1\rangle$ y $\Sigma_2 = \langle TC,\ SBT_2,\ DC,\ DF,\ SPF_2\rangle$, son *unibles* si y solamente si se cumplen las siguientes condiciones:

- $SPF_1 \cap SPF_2 \subseteq \{\texttt{==}\}$. Es decir, el único símbolo de función primitiva permitida que pertenece a ambos dominios es el operador de igualdad estricta.
- Si $d \in SBT_1 \cap SBT_2$ entonces $\mathcal{B}_d^{\mathcal{D}_1} = \mathcal{B}_d^{\mathcal{D}_2}$. Es decir, si ambos dominios tienen un tipo base común, entonces los correspondientes conjuntos de valores básicos deben ser los mismos.

A partir del concepto de dominios unibles se puede establecer la siguiente definición.

Definición 9. (Suma amalgamada)
Se define la *suma amalgamada* de dos dominios unibles $\mathcal{D}_1$ y $\mathcal{D}_2$ como un nuevo dominio $\mathcal{S} = \mathcal{D}_1 \oplus \mathcal{D}_2$ con signatura $\Sigma = \langle TC,\ SBT_1 \cup SBT_2,\ DC,\ DF,\ SPF_1 \cup SPF_2\rangle$, construido de la siguiente forma:

- Para $i = 1, 2$ y para todo $d \in SBT_i$: $\mathcal{B}_d^{\mathcal{S}} = \mathcal{B}_d^{\mathcal{D}_i}$. Es decir, el conjunto de valores básicos de la suma amalgamada con respecto a un tipo específico d debe ser el mismo que cada conjunto de valores básicos de los dominios puros con respecto a dicho tipo específico d.
- Para $i = 1, 2$ y para todo símbolo de función primitiva $p \in SPF_i$, con p distinto de ==, y para todo $\bar{t}_n$, $t \in \mathcal{U}_{\mathcal{S}}$ se cumple $p^{\mathcal{S}}\bar{t}_n \to t$ si y solamente si una de las dos condiciones siguientes se cumple: o $t = \bot$ o bien existen $\overline{t'}_n$, $t' \in \mathcal{U}_{\mathcal{D}_i}$ tal que $\overline{t'}_n \sqsubseteq \bar{t}_n$, $t' \sqsupseteq t$ y $p^{\mathcal{D}_i}\overline{t'}_n \to t'$.

Se puede observar que si $d \in SBT_1 \cap SBT_2$ entonces el primer punto de la definición de suma amalgamada se establece por la condición de unión. El segundo punto de la definición establece la interpretación de las primitivas del dominio $\mathcal{S}$ (distintas de ==) como una extensión de las primitivas específicas de cada dominio. En general, para cualquier primitiva p, la interpretación $p^{\mathcal{S}}$ es una extensión de cada interpretación $p^{\mathcal{D}_i}$ que cumple las condiciones impuestas a las interpretaciones en la definición 1.

La suma amalgamada del dominio de Herbrand con un dominio cualquiera $\mathcal{H} \oplus \mathcal{D}$ siempre es posible, pues $SBT_{\mathcal{H}}$ es el conjunto vacío y el único símbolo de función primitiva que contiene $SPF_{\mathcal{H}}$ es ==. Por lo tanto, las condiciones requeridas se cumplen. La interpretación de == en $\mathcal{S}$ se comporta de igual manera que se comporta en los dominios contenidos en $\mathcal{S}$.

La suma amalgamada $\mathcal{D}_1 \oplus \cdots \oplus \mathcal{D}_n$ de n dominios unibles dos a dos se define análogamente. La siguiente definición y teorema garantizan el comportamiento esperado de las sumas amalgamadas como extensiones conservativas de sus componentes.

Definición 10. (Restricciones de dominio específico y operadores de truncamiento)
Sea $\mathcal{S} = \mathcal{D}_1 \oplus \cdots \oplus \mathcal{D}_n$ construido como la suma amalgamada de n dominios $\mathcal{D}_i$ unibles dos a dos, entonces para cualquier $1 \leq i \leq n$ arbitrariamente fijado se tienen las siguientes definiciones.

1. Un conjunto $\Pi \subseteq APCon_{\mathcal{D}_i}$ se denomina $\mathcal{D}_i$-*específico* si y solamente si toda valoración $\eta \in Val_{\mathcal{S}}$ tal que $\eta \in Sol_{\mathcal{S}}(\Pi)$ satisface $\eta(X) \in \mathcal{U}_{\mathcal{D}_i}$ para todo $X \in var(\Pi)$.
2. Se define $\mathcal{D}_i$-*truncamiento* de un valor $t \in \mathcal{U}_{\mathcal{S}}$ al valor $\mid t \mid_{\mathcal{D}_i} \in \mathcal{U}_{\mathcal{D}_i}$ que cumple la siguiente condición con respecto al orden de información $\sqsubseteq$ de $\mathcal{U}_{\mathcal{S}}$: $\mid t \mid_{\mathcal{D}_i} \sqsubseteq t$ de manera que cualquier otro $\hat{t} \in \mathcal{U}_{\mathcal{D}_i}$ tal que $\hat{t} \sqsubseteq t$ debe satisfacer $\hat{t} \sqsubseteq \mid t \mid_{\mathcal{D}_i}$.
3. Se define $\mathcal{D}_i$-*truncamiento* de una valoración $\eta \in Val_{\mathcal{S}}$ a la valoración $\mid \eta \mid_{\mathcal{D}_i}$ definida por la condición $\mid \eta \mid_{\mathcal{D}_i} (X) = \mid \eta(X) \mid_{\mathcal{D}_i}$, para todo $X \in \mathcal{V}ar$.

Las restricciones $\mathcal{R}$-específicas, $\mathcal{FD}$-específicas y $\mathcal{FS}$-específicas definidas en las subsecciones 3.2, 3.3 y 3.4 son también específicas en el sentido que se acaba de definir.

Con respecto al segundo punto, una forma efectiva de construir $\mid t \mid_{\mathcal{D}_i}$ a partir de t es sustituir $\bot$ en cualquier subpatrón de t que tiene la siguiente forma: o bien es un valor básico u que no pertenece a $\mathcal{U}_{\mathcal{D}_i}$, o bien es una aplicación parcial de una función primitiva que no pertenece a la signatura específica de $\mathcal{D}_i$. Por lo tanto, $\mid t \mid_{\mathcal{D}_i} = t$ si y solamente si $t \in \mathcal{U}_{\mathcal{D}_i}$. Con respecto al tercer punto, $\mid \eta \mid_{\mathcal{D}_i} = \eta$ si y solamente si $\eta \in Val_{\mathcal{D}_i}$.

Teorema 4. (Propiedades de la suma amalgamada)
Para cualquier suma amalgamada $\mathcal{S} = \mathcal{D}_1 \oplus \cdots \oplus \mathcal{D}_n$ se cumple:

1. $\mathcal{S}$ está bien definido como un dominio de restricciones, es decir, las interpretaciones de los símbolos de las funciones primitivas en $\mathcal{S}$ satisfacen las condiciones de la definición 1.
2. $\mathcal{S}$ es una extensión conservativa de $\mathcal{D}_i$ para todo $1 \leq i \leq n$. Es decir, para todo $1 \leq i \leq n$, para cualquier $p \in SPF_i^m$ (distinto de ==), y para todo $\overline{t}_m, t \in \mathcal{U}_{\mathcal{D}_i}$, se cumple $p^{\mathcal{D}_i}\, \overline{t}_m \to t$ si y solamente si $p^{\mathcal{S}}\, \overline{t}_m \to t$.
3. Para cualquier conjunto de restricciones primitivas $\Pi \subseteq APCon_{\mathcal{D}_i}$ con $1 \leq i \leq n$ y para cada valoración $\eta \in Val_{\mathcal{D}_i}$, se cumple $\eta \in (WT)Sol_{\mathcal{D}_i}(\Pi)$ si y solamente si $\eta \in (WT)Sol_{\mathcal{S}}(\Pi)$.
4. Para cualquier conjunto de restricciones primitivas $\mathcal{D}_i$-específicas $\Pi \subseteq APCon_{\mathcal{D}_i}$ con $1 \leq i \leq n$ y para cada valoración $\eta \in Val_{\mathcal{S}}$, se cumple $\eta \in (WT)Sol_{\mathcal{S}}(\Pi)$ si y solamente si $\mid \eta \mid_{\mathcal{D}_i} \in (WT)Sol_{\mathcal{D}_i}(\Pi)$.

La demostración de este teorema se puede encontrar en el apéndice A.5. Una vez definida la suma amalgamada y dadas sus propiedades, se necesita definir el dominio que proporciona la comunicación entre los dominios $\mathcal{D}_i$ para poder formar el dominio de coordinación.

Se define el *dominio mediador* $\mathcal{M}$ para la comunicación de n dominios $\mathcal{D}_1, \cdots, \mathcal{D}_n$, unibles dos a dos y cada uno de ellos con signatura específica $\Sigma_i = \langle TC,\ SBT_i,\ DC,\ DF,\ SPF_i \rangle$, como un dominio de signatura específica $\Sigma_0 = \langle TC,\ SBT_0,\ DC,\ DF,\ SPF_0 \rangle$ construido de la siguiente forma:

- $SBT_0 \subseteq \bigcup_{i=1}^{n} SBT_i$ y $SPF_0 \cap SPF_i = \emptyset$ para todo $1 \leq i \leq n$. Es decir, el conjunto de tipos del dominio mediador debe contener a todos los tipos de los dominios que va a mediar y los símbolos de las funciones primitivas no deben estar en conflicto.
- Para cada $d \in SBT_0$ y $d \in SBT_i$ con $1 \leq i \leq n$, el conjunto de valores básicos, con respecto a los tipos, no puede variar, es decir, $\mathcal{B}_d^{\mathcal{M}} = \mathcal{B}_d^{\mathcal{D}_i}$.
- Las primitivas $p_{d_i,d_j} \in SPF_0$ se denominan puentes y tienen una declaración de tipos principal $d_i \to d_j \to bool$, para $d_i \in SBT_i$, $d_j \in SBT_j$ con $1 \leq i, j \leq n$.
- Cada puente puede ser interpretado utilizando una función $rel_{d_i,d_j} : \mathcal{B}_{d_i}^{\mathcal{D}_i} \to \mathcal{B}_{d_j}^{\mathcal{D}_j}$ de tal forma que para todo $s, t, r \in \mathcal{U}_{\mathcal{M}}$, se cumple $p_{d_i,d_j}^{\mathcal{M}}\ s\ t \to r$ si y solamente si se cumple alguno de los siguientes casos:
  1. $s \in dom(rel_{d_i,d_j})$, $t \in \mathcal{B}_{d_j}^{\mathcal{D}_j}$, $t = rel_{d_i,d_j}(s)$ y $true \sqsupseteq r$.
  2. $s \in dom(rel_{d_i,d_j})$, $t \in \mathcal{B}_{d_j}^{\mathcal{D}_j}$, $t \neq rel_{d_i,d_j}(s)$ y $false \sqsupseteq r$.
  3. $r = \bot$.

Cada dominio mediador puede contener una o varias relaciones binarias. Cada relación binaria rel_{d_i,d_j} es un puente de comunicación entre los dominios $\mathcal{D}_i$ y $\mathcal{D}_j$. Por otra parte, los $n+1$ dominios $\mathcal{M}$, $\mathcal{D}_1$, ..., $\mathcal{D}_n$ son unibles dos a dos. Por lo tanto, es posible construir la suma amalgamada $\mathcal{C} = \mathcal{M} \oplus \mathcal{D}_1 \oplus \cdots \oplus \mathcal{D}_n$. Este dominio híbrido soporta la comunicación entre los dominios $\mathcal{D}_i$ a través de las restricciones puentes proporcionadas por $\mathcal{M}$.

Por ejemplo, para la cooperación entre los dominios $\mathcal{FD}$ y $\mathcal{R}$ se define el puente como $\texttt{\#==}_{int,real}$ y es interpretado con respecto a la función total inyectiva $inj_{int,real} :: \mathbb{Z} \to \mathbb{R}$, la cual hace corresponder cada valor entero con un valor real equivalente. Para simplificar, se escribirá `#==` en lugar de $\texttt{\#==}_{int,real}$ y se usará en notación infija.

Una vez que se ha definido el dominio de coordinación $\mathcal{C}$, pasamos a definir programas y objetivos para $\mathcal{C} = \mathcal{M} \oplus \mathcal{D}_1 \oplus \cdots \oplus \mathcal{D}_n$.

Un *programa* $\mathcal{P}$ en el contexto $CFLP(\mathcal{C})$ se define como un conjunto de reglas de reescritura posiblemente condicionales que definen el comportamiento de funciones perezosas que pueden ser no deterministas y de orden superior. Estas reglas se denominan reglas de programa. Una regla de programa Rl para un símbolo de función definida $f \in DF^n$, con tipo principal $f :: \overline{\tau}_n \to \tau$ tiene la forma $f\,\overline{t}_n \to r \Leftarrow \Delta$, donde $\overline{t}_n$ es una secuencia lineal de patrones, r es una expresión y Δ es una conjunción finita de restricciones atómicas.

Una regla de programa Rl incluye *apariciones libres de variables lógicas de orden superior* si y solamente si existe alguna variable X que no está contenida en el lado izquierdo de la

regla Rl pero aparece en alguna otra parte de la regla y tiene la forma $X\,\overline{e}_m$ con $m > 0$. Por ejemplo, las siguientes reglas de programa `prueba1 X Y = F X Y` y `prueba2 X Y = true <== F X Y` contienen a `F` como apariciones libres de variables lógicas de orden superior. Un programa incluye apariciones libres de variables lógicas de orden superior si y solamente si alguna de sus reglas las incluyen. Como ya se motivó en la subsección 1.3.3, dedicada al orden superior en $\mathcal{TOY}$, el cálculo $CCLNC(\mathcal{D})$ excluye el uso de variables lógicas de orden superior para garantizar la completitud, pues no se pueden garantizar soluciones bien tipadas al ser la comprobación de tipos estática y no dinámica.

Los programas se utilizan para resolver objetivos. Un *objetivo* sobre el dominio de coordinación $\mathcal{C}$ tiene la siguiente forma general: $G \equiv \exists\overline{U}.P \sqcap C \sqcap M \sqcap H \sqcap D_1 \sqcap \cdots \sqcap D_n$, donde $\sqcap$ representa la conjunción y

- $\overline{U}$ es un conjunto finito de variables existenciales locales creadas durante el cómputo.
- P es un conjunto de *producciones* de la forma $e_1 \to t_1, \ldots, e_m \to t_m$, donde $e_i \in Exp_{\mathcal{D}}$ y $t_i \in Pat_{\mathcal{D}}$ para todo $1 \leq i \leq m$. El conjunto de *variables producidas* de G es el conjunto de variables contenidas en $t_1 \ldots t_m$ y se denota como $pvar(P)$. Durante la resolución del objetivo, las producciones se utilizan para computar valores para las variables producidas en la medida que son demandadas, usando las reglas de estrechamiento perezoso que se presentarán a continuación en la siguiente sección 4.2. Se utilizará la notación $X \gg_P Y$ para representar una relación entre variables producidas que se utilizará más adelante. Esta relación se denomina *relación de producción* y se establece entre dos variables $X, Y \in pvar(P)$ cuando $X \in var(e_i)$ y $Y \in var(t_i)$ para algún $1 \leq i \leq m$.
- C, denominado *pool* de restricciones, es un conjunto finito donde las restricciones están esperando para ser resueltas.
- $M = \Pi_M \sqcap \sigma_M$ es el *almacén del dominio mediador* definido como una conjunción de restricciones primitivas atómicas $\Pi_M \subseteq APCon_{\mathcal{M}}$ denominadas puentes y una sustitución σ_M.
- $H = \Pi_H \sqcap \sigma_H$ es el *almacén del dominio de Herbrand* definido como una conjunción de restricciones primitivas atómicas $\Pi_H \subseteq APCon_{\mathcal{H}}$ y una sustitución σ_H.
- $D_i = \Pi_{D_i} \sqcap \sigma_{D_i}$ con $1 \leq i \leq n$ es un almacén de restricciones del dominio $\mathcal{D}_i$ representado por un conjunto de restricciones primitivas atómicas $\Pi_{D_i} \subseteq APCon_{D_i}$ y una sustitución σ_{D_i}.

En los siguientes capítulos, D_i representa a los almacenes R, F y S de los dominios $\mathcal{R}$, $\mathcal{FD}$ y $\mathcal{FS}$, respectivamente.

La evaluación de un objetivo es el proceso que transforma un *objetivo inicial* en un *objetivo final* mediante la aplicación de una serie de reglas. Un *objetivo inicial* tiene todos sus componentes vacíos excepto el *pool* de restricciones. Un *objetivo final* o *resuelto* no contiene restricciones en el *pool* ni contiene producciones y además todos sus almacenes están en forma resuelta. Si un objetivo es *inconsistente* entonces se denota mediante el símbolo ■.

Un objetivo tiene *apariciones libres de variables lógicas de orden superior* si y solo si existe alguna variable X en el objetivo tal que aparece con la forma $X\,\overline{e}_m$ con $m > 0$. Los pasos que ligan apariciones libres de variables lógicas de orden superior realizan una búsqueda sobre todos los posibles patrones de las reglas que se pueden aplicar dando lugar, en ciertas ocasiones, a objetivos mal tipados.

A continuación se muestra un ejemplo de dos objetivos $\mathcal{TOY}$ escritos con la sintaxis abstracta que se acaba de exponer para los objetivos generales $CFLP(\mathcal{C})$. En el caso concreto de cooperación entre los dominios $\mathcal{R}$ y $\mathcal{FD}$ el objetivo es de la forma:

$$G \equiv \exists \overline{U}.P \;\square\; C \;\square\; M \;\square\; H \;\square\; R \;\square\; F$$

Los objetivos $\mathcal{TOY}$ presentados en los ejemplos 1 y 2 del capítulo 1 se representan mediante los siguientes objetivos iniciales:

(1) $\square$ `(1,1) <<- rectangle (0,0) (2,2)` $\square\,\square\,\square\,\square$
(2) $\square$ `smm L [ff]` $\square\,\square\,\square\,\square$

Los correspondientes objetivos resueltos son:

(1) $\square\,\square\,\square\,\square$ (`X >= 1.0, X =< 4.0, Y > 2.0, Y =< 3.0` $\square\ \varepsilon$) $\square$
$\square\,\square\,\square\,\square$ (`X >= 1.0, X =< 4.0, Y >= 0.0, Y < 1.0` $\square\ \varepsilon$) $\square$
$\square\,\square\,\square\,\square$ (`X > 4.0, X =< 6.0, Y >= 0.0, Y =< 3.0` $\square\ \varepsilon$) $\square$
$\square\,\square\,\square\,\square$ (`X >= 0.0, X < 1.0, Y >= 0.0, Y =< 3.0` $\square\ \varepsilon$) $\square$
(2) $\square\,\square\,\square\,\square\,\square$ ($\Diamond\ \square$ `L` $\mapsto$ `[9,5,6,7,1,0,8,2]`)

En un objetivo de la forma $G \equiv \exists \overline{U}.P \;\square\; C \;\square\; M \;\square\; H \;\square\; D_1 \;\square \cdots \square\; D_n$ se define el conjunto de *variables obviamente demandadas por el objetivo* G, y se denota como $odvar(G)$, al conjunto formado por $odvar_{\mathcal{M}}(\Pi_M) \cup odvar_{\mathcal{H}}(\Pi_H) \cup odvar_{D_1}(\Pi_{D_1}) \cup \cdots \cup odvar_{D_n}(\Pi_{D_n})$. Se denominan *suspensiones* a las producciones de la forma $e \to X$ tal que e es una expresión activa y $X \notin odvar(G)$ es una variable que no es obviamente demandada por el objetivo. Las suspensiones son relevantes en el proceso de resolución de objetivos como se verá en la próxima sección.

Existen ciertas propiedades que se cumplen para objetivos iniciales y se mantienen invariantes a través de las diversas transformaciones de los objetivos. Las propiedades invariantes incluidas en trabajos anteriores como [LRV04] son: cada variable producida se produce solo una vez, todas las variables producidas deben ser existenciales, el cierre transitivo de la relación $\gg_P^+$ entre las variables producidas debe ser irreflexivo y ninguna variable producida aparece en las sustituciones de las respuestas. Además de estos invariantes, existen otros invariantes específicos de la cooperación:

- Los dominios de las sustituciones σ_M, σ_H, σ_{D_i} con $1 \leq i \leq n$ son disjuntos dos a dos.
- Para cualquier almacén A contenido en el objetivo, la aplicación de σ_A no causa ninguna modificación en el objetivo.
- Para cualquier $X \in vdom(\sigma_M)$, $\sigma_M(X)$ es o un valor booleano, o un valor básico contenido en un conjunto $\mathcal{B}^{D_i}$ con $1 \leq i \leq n$.

- En el almacén D_i del dominio $\mathcal{D}_i$, para cualquier $X \in vdom(\sigma_{D_i})$, $\sigma_{D_i}(X)$ es un valor básico del dominio o una variable contenida en Π_{D_i}.

Un *objetivo admisible* es un objetivo que deriva de un objetivo inicial mediante la aplicación iterativa de las reglas de transformación de objetivos y cumple las propiedades invariantes que se acaban de describir.

El primer invariante se garantiza porque, cuando se produce una sustitución y una variable se vincula en uno de los almacenes, entonces la vinculación se propaga al resto del objetivo pero la sitúa o guarda únicamente en la sustitución de ese almacén en particular. Esta idea se formaliza mediante la notación definida en la sección 3.1.2 de la siguiente forma: sea un objetivo $P \,\square\, C \,\square\, M \,\square\, H \,\square\, \cdots D_i \cdots$ donde $D_i = \Pi_{D_i} \,\square\, \sigma_{D_i}$ es el almacén del dominio $\mathcal{D}_i$. La aplicación de la sustitución σ al objetivo $P \,\square\, C \,\square\, M \,\square\, H \,\square\, \cdots D_i \cdots$ se denota como $(P \,\square\, C \,\square\, M \,\square\, H \,\square\, \cdots D_i \cdots)@_{\mathcal{D}_i}\sigma$. La aplicación de σ a cada parte del objetivo es $P\sigma \,\square\, C\sigma \,\square\, M \star \sigma \,\square\, H \star \sigma \,\square\, \cdots D_i@\sigma \cdots$, donde $D_i@\sigma$ se define como $\Pi_{D_i}\sigma \,\square\, \sigma_{D_i}\sigma$. La sustitución $A \star \sigma$ aplica la sustitución σ en el almacén A pero no guarda dicha sustitución. Así, $A \star \sigma$ se define como $\Pi_A\sigma \,\square\, \sigma_A \star \sigma$ [1] con A representando a M, H o al resto de los D_j con $1 \leq j \leq n$ y $i \neq j$. La aplicación de la sustitución σ sobre σ_A restringida a las variables de σ_A es $\sigma_A \star \sigma = \sigma_A\sigma \restriction_{vdom(\sigma_A)}$ [2] que conserva el mismo dominio que σ_A.

Un programa $\mathcal{P}$ o un objetivo G es llamado *seguro* si no tiene apariciones libres de variables de orden superior y no tiene descomposiciones opacas (ver sección 1.3) que puedan producir objetivos mal tipados o invocaciones incompletas a resolutores.

4.2 Cálculo $CCLNC(\mathcal{C})$

Los objetivos que se tratan en esta tesis pueden contener restricciones de varios dominios y se resuelven mediante un cálculo cooperativo de resolución de objetivos que se denomina $CCLNC(\mathcal{C})$ definido sobre un dominio de coordinación. Este cálculo es una extensión del cálculo de resolución de objetivos sobre un dominio paramétrico dado $CLNC(\mathcal{D})$ que se definió en los trabajos [LRV04, LRV07, dVV08] para el esquema $CFLP$. Sin embargo, el cálculo que se presenta en esta tesis es una extensión sustancial con respecto a $CLNC(\mathcal{D})$ con varios mecanismos para la cooperación, proyecciones y reglas que infieren nuevas restricciones o anticipan el fallo. El cálculo $CCLNC(\mathcal{C})$, introducido en [EFH$^+$07a] y desarrollado en [EFH$^+$07b, EFH$^+$08, EFH$^+$09], es correcto y completo con ciertas limitaciones con respecto a la instancia $CFLP(\mathcal{D})$ del esquema genérico $CFLP$ desarrollado en [LRV07]. Es importante mencionar que el cálculo $CCLNC(\mathcal{C})$ está diseñado para no tener apariciones libres de variables lógicas de orden superior, es decir, variables que se utilizan como una función pero no son dadas como parámetros. Como se mostró en [GHR01], las reglas de resolución

[1] Dadas dos sustituciones σ y θ, la sustitución $\sigma\star\theta$ es la *aplicación* de θ sobre σ restringida a las variables de σ, es decir, $\sigma\theta \restriction_{vdom(\sigma)}$. Por lo tanto, para cualquier variable X se cumple $X(\sigma\star\theta) = X\sigma\theta$ si $X \in vdom(\sigma)$ y $X(\sigma \star \theta) = X$ en otro caso (definición dada en la sección 3.1.2).

[2] Una sustitución σ restringida a un conjunto de variables $\mathcal{X}$ se denota como $\sigma\restriction_{\mathcal{X}}$ y es la sustitución σ' tal que para el conjunto de variables $\mathcal{X}$ se cumple $vdom(\sigma') = \mathcal{X}$ y $X\sigma' = X\sigma$ para todo $X \in \mathcal{X}$ (definición dada en la sección 3.1.2).

de objetivos para tratar apariciones libres de variables lógicas de orden superior pueden dar lugar a soluciones mal tipadas.

La tabla 4.1 presenta las reglas del cálculo $CCLNC(\mathcal{C})$ que modelan exclusivamente el comportamiento del estrechamiento perezoso restringido, ignorando la cooperación y las invocaciones a los resolutores. En esencia, las primeras cuatro reglas codifican transformaciones relacionadas con la unificación; la regla **EL** elimina suspensiones innecesarias; la regla **DF** trata las llamadas a funciones definidas en el programa; y la regla **FC**, junto con la regla **PC**, transforma las restricciones atómicas del *pool* en una forma aplanada que consiste en un conjunto de restricciones primitivas atómicas con nuevas variables existenciales. El proceso de aplanar restricciones ya fue estudiado en el trabajo [LRV04] y otros trabajos previos relacionados. Pero en estos trabajos las restricciones primitivas atómicas se enviaban directamente a un almacén de restricciones. Ahora, las restricciones primitivas atómicas se mantienen en el *pool* de restricciones para poder aplicar las reglas de cooperación.

Las suspensiones $e \to X$ pueden ser eliminadas por la regla **EL** en el caso de que X no aparezca en el resto del objetivo. En caso contrario, estas suspensiones deben esperar hasta que X se vincule a un patrón sin variables o bien se convierta en obviamente demandada. En este último caso tiene que ser procesada usando la regla **DF** o la regla **PC**, dependiendo de la forma sintáctica de e.

Ejemplo 3. (Aplanamiento de restricciones)

En este ejemplo se muestra el proceso de aplanamiento de la restricción atómica real `(RX+2*RY)*RZ <= 3.5` resultando una conjunción de cuatro restricciones primitivas atómicas que se ubican en el *pool* de restricciones. En este proceso de aplanamiento se han utilizado tres variables existenciales nuevas. En cada paso se subraya la restricción o producción que se procesa.

$\square\ \underline{(RX + 2 * RY) * RZ \texttt{ <= } 3.5}\ \square\ \square\ \square\ \square \vdash_{\mathbf{FC}}$

$\exists RA.\ \underline{(RX + 2 * RY) * RZ\ \to\ RA}\ \square\ RA \texttt{ <= } 3.5\ \square\ \square\ \square\ \square \vdash_{\mathbf{PC}}$

$\exists RA.\ \square\ \underline{(RX + 2 * RY) * RZ\ \to!\ RA},\ RA \texttt{ <= } 3.5\ \square\ \square\ \square\ \square \vdash_{\mathbf{FC}}$

$\exists RB,\ RA.\ \underline{RX + 2 * RY \to RB}\ \square\ RB * RZ \to!\ RA, RA \texttt{ <= } 3.5\ \square\ \square\ \square\ \square \vdash_{\mathbf{PC}}$

$\exists RB, RA.\ \square\ \underline{RX + 2 * RY\ \to!\ RB},\ RB * RZ \to!\ RA,\ RA \texttt{ <= } 3.5\ \square\ \square\ \square\ \square \vdash_{\mathbf{FC}}$

$\exists RC, RB, RA.\ \underline{2 * RY \to RC}\ \square\ RX + RC \to!\ RB,\ RB * RZ \to!\ RA,\ RA \texttt{ <= } 3.5\ \square\ \square\ \square\ \square \vdash_{\mathbf{PC}}$

$\exists RC, RB, RA.\ \square\ 2 * RY \to! RC,\ RX + RC \to!\ RB,\ RB * RZ \to!\ RA,\ RA \texttt{ <= } 3.5\ \square\ \square\ \square\ \square$

En el cálculo $CCLNC(\mathcal{C})$ son posibles distintas derivaciones debido al indeterminismo. En particular, el indeterminismo *don't know* se manifiesta en la aplicación de funciones no deterministas y el indeterminismo *don't care* está presente la posibilidad de aplicar reglas de transformación diferentes. Por ejemplo, para resolver una desigualdad de la forma X `/=` $c\,\overline{t}_n$ se pueden aplicar las reglas **H11** y **H12** de la tabla 3.3. En la regla **H11**, la variable X se sustituye por una expresión nueva $c\,\overline{Z}_n$ que reproduce con variables frescas la estructura de la expresión $c\,\overline{t}_n$. Las expresiones $c\,\overline{t}_n$ y $c\,\overline{Z}_n$ deben ser distintas, lo que conlleva a varias

DC DeComposition
$\exists\overline{U}.\ h\ \overline{e_m} \to h\ \overline{t_m},\ P \square C \square M \square H \square D_1 \square \cdots \square D_n \vdash_{\mathbf{DC}}$
$\exists\overline{U}.\ \overline{e_m \to t_m},\ P \square C \square M \square H \square D_1 \square \cdots \square D_n$

CF Conflict Failure
$\exists\overline{U}.\ e \to t, P \square C \square M \square H \square D_1 \square \cdots \square D_n \vdash_{\mathbf{CF}} \blacksquare$
Si e es rígida y pasiva, $t \notin \mathcal{V}ar$, e y t tienen sus raíces en conflicto.

SP Simple Production
$\exists\overline{U}.\ s \to t,\ P \square C \square M \square H \square D_1 \square \cdots \square D_n \vdash_{\mathbf{SP}}$
$\exists\overline{U}'.\ (P \square C \square M \square H \square D_1 \square \cdots \square D_n)@_{\mathcal{H}}\sigma'$
Si $s = X \in \mathcal{V}ar$, $t \notin \mathcal{V}ar$, $\sigma' = \{X \mapsto t\}$ y $\overline{U'} = \overline{U}$ o bien si $s \in Pat_{\mathcal{C}}$, $t = X \in \mathcal{V}ar$, $\sigma' = \{X \mapsto s\}$ y $\overline{U}' = \overline{U}\backslash\{X\}$.

IM IMitation
$\exists X, \overline{U}.\ h\ \overline{e_m} \to X,\ P \square C \square M \square H \square D_1 \square \cdots \square D_n \vdash_{\mathbf{IM}}$
$\exists\overline{X_m}, \overline{U}.\ (\overline{e_m \to X_m},\ P \square C \square M \square H \square D_1 \square \cdots \square D_n)\sigma'$
Si $h\ \overline{e_m} \notin Pat_{\mathcal{C}}$ es pasiva, $X \in odvar(G)$ y $\sigma' = \{X \mapsto h\ \overline{X_m}\}$.

EL ELimination
$\exists X, \overline{U}.\ e \to X,\ P \square C \square M \square H \square D_1 \square \cdots \square D_n \vdash_{\mathbf{EL}} \exists\overline{U}.\ P \square C \square M \square H \square D_1 \square \cdots \square D_n$
Si X no aparece en el resto del objetivo.

DF Defined Function
$\exists\overline{U}.\ f\ \overline{e_n} \to t,\ P \square C \square M \square H \square D_1 \square \cdots \square D_n \vdash_{\mathbf{DF}_f}$
$\exists\overline{Y}, \overline{U}.\ \overline{e_n \to t_n}, r \to t,\ P \square C',\ C \square M \square H \square D_1 \square \cdots \square D_n$
Si $f \in DF^n$, $t \notin \mathcal{V}ar$ o $t \in odvar(G)$ y $Rl : f\ \overline{t_n} \to r \Leftarrow C'$ es una variante de una regla en $\mathcal{P}$, con $\overline{Y} = var(Rl)$ variables nuevas.

$\exists\overline{U}.\ f\ \overline{e_n a_k} \to t,\ P \square C \square M \square H \square D_1 \square \cdots \square D_n \vdash_{\mathbf{DF}_f}$
$\exists X, \overline{Y}, \overline{U}.\ \overline{e_n \to t_n}, r \to X,\ X\ \overline{a_k} \to t,\ P \square C',\ C \square M \square H \square D_1 \square \cdots \square D_n$
Si $f \in DF^n$ $(k > 0)$, $t \notin \mathcal{V}ar$ or $t \in odvar(G)$ y $Rl : f\ \overline{t_n} \to r \Leftarrow C'$ es una variante de una regla en $\mathcal{P}$, con $\overline{Y} = var(Rl)$ y X variables nuevas.

PC Place Constraint
$\exists\overline{U}.\ p\ \overline{e_n} \to t,\ P \square C \square M \square H \square D_1 \square \cdots \square D_n \vdash_{\mathbf{PC}} \exists\overline{U}.P \square p\ \overline{e_n} \to! t,\ C \square M \square H \square D_1 \square \cdots \square D_n$
Si $p \in PF^n$ y $t \notin \mathcal{V}ar$ o $t \in odvar(G)$.

FC Flatten Constraint
$\exists\overline{U}.\ P \square p\ \overline{e_n} \to! t,\ C \square M \square H \square D_1 \square \cdots \square D_n \vdash_{\mathbf{FC}}$
$\exists\overline{V}_m, \overline{U}.\ \overline{a_m \to V_m}, P \square p\ \overline{t}_n \to! t,\ C \square M \square H \square D_1 \square \cdots \square D_n$
Si $p \in PF^n$, algún $e_i \notin Pat_{\mathcal{C}}$, $\overline{a_m}$ $(m \leq n)$ son aquellos e_i que no son patrones, $\overline{V_m}$ son variables nuevas, y $p\ \overline{t_n}$ se obtiene a partir de $p\ \overline{e_n}$ reemplazando cada e_i que no sea un patrón por V_i.

Tabla 4.1: Reglas del cálculo $CCLNC(\mathcal{C})$ correspondientes al estrechamiento perezoso

derivaciones debido a la posibilidad de elegir entre las distintas alternativas Z_i /= t_i con $1 \leq i \leq n$. En la regla **H12** la variable X se sustituye por un témino nuevo $d\,\overline{Z}_m$ definido mediante una constructora d distinta a c.

Aunque el cálculo $CCLNC(\mathcal{C})$ permite distintas derivaciones, la implementación la $\mathcal{TOY}$ sigue una estrategia particular. En esencia, la estrategia que sigue $\mathcal{TOY}$ para elegir las reglas de transformación de objetivos se puede esbozar del siguiente modo teniendo en cuenta las reglas de la tabla 4.1:

- Se exploran las producciones que existen en P, de izquierda a derecha, con la única excepción de las suspensiones que son retrasadas hasta que se puedan eliminar de forma segura o hasta que el objetivo las transforme de tal manera que dejen de ser suspensiones. Las producciones no suspendidas pueden conducir a fallo aplicando la regla **CF** o pueden ser evaluadas mediante otras reglas de la tabla 4.1.

- Si P está vacío o únicamente contiene suspensiones que no pueden ser procesadas por las reglas de la tabla 4.1, y además, alguno de los almacenes M, H, D_i no está en forma resuelta y sus restricciones no incluyen variables producidas obviamente demandadas, entonces se invocan los resolutores de tales almacenes.

- Si ninguno de los dos puntos anteriores pueden ser aplicados y C no está vacío, entonces se tratan las restricciones de C de izquierda a derecha de la siguiente forma:

  1. Si $\pi \in C$ es una restricción primitiva atómica, entonces se hace el tratamiento de la cooperación que corresponda y que se explicará en los siguientes capítulos. Después se envía π al almacén que le corresponda.
  2. En el caso contrario, π se aplana y se transforma en un conjunto de restricciones primitivas atómicas mediante las reglas **PC** y **FC**, usando nuevas variables existenciales. Estas nuevas restricciones primitivas atómicas se ubican en C. Algunos cómputos se suspenden en P por medio del estrechamiento perezoso y se tratan como se describe en los primeros puntos.

- Finalmente, el objetivo está resuelto cuando P y C están vacíos y los almacenes están en forma resuelta.

En la explicación anterior se ha indicado que las restricciones se envían a sus correspondientes almacenes. Cuando están en estos almacenes, se invoca al resolutor correspondiente mediante las reglas de la tabla 4.2. La regla **SF** detecta el fallo del objetivo por medio de un resolutor y las otras reglas describen la posible transformación de un objetivo por la invocación del resolutor. Los resolutores distintos de $\mathcal{H}$ se invocan si sus correspondientes almacenes no están en forma resuelta y no contienen ninguna variable producida que pueda hacer que el cómputo prosiga. Con respecto al resolutor de Herbrand, se requiere que ninguna variable producida sea obviamente demandada por el almacén ($pvar(P) \cap odvar_{\mathcal{H}}(\Pi_H) = \emptyset$), pues si existiesen se evaluarían estas producciones antes. Además se requiere que el almacén no esté en forma resuelta con respecto a las variables críticas. Cada invocación $solve^{\mathcal{H}}(\Pi, \mathcal{X})$ depende del conjunto de variables críticas las cuales son elegidas como las variables del almacén que

D_iS D_i-Constraint Solver (*caja negra*)
$\exists\overline{U}.\ P \sqcup C \sqcup M \sqcup H \sqcup \cdots D_i \cdots \vdash_{\mathtt{D_iS}} \exists\overline{Y'}, \overline{U}.\ (P \sqcup C \sqcup M \sqcup H \sqcup \cdots (\Pi' \sqcup \sigma_{D_i}) \cdots)@_{\mathcal{FD}}\sigma'$
Si $pvar(P) \cap var(\Pi_{D_i}) = \emptyset$, $(\Pi_{D_i} \sqcup \sigma_{D_i})$ no está en forma resuelta, $\Pi_{D_i} \vdash_{solve^{D_i}} \exists\overline{Y'}(\Pi' \sqcup \sigma')$.

MS $\mathcal{M}$-Constraint Solver (*caja transparente*)
$\exists\overline{U}.\ P \sqcup C \sqcup M \sqcup H \sqcup \cdots D_i \cdots \vdash_{\mathbf{MS}} \exists\overline{Y'}, \overline{U}.\ (P \sqcup C \sqcup (\Pi' \sqcup \sigma_M) \sqcup H \sqcup \cdots D_i \cdots)@_{\mathcal{M}}\sigma'$
Si $pvar(P) \cap var(\Pi_M) = \emptyset$, $(\Pi_M \sqcup \sigma_M)$ no está en forma resuelta, $\Pi_M \vdash_{solve^{\mathcal{M}}} \exists\overline{Y'}(\Pi' \sqcup \sigma')$.

HS $\mathcal{H}$-Constraint Solver (*caja transparente*)
$\exists\overline{U}.\ P \sqcup C \sqcup M \sqcup H \sqcup \cdots D_i \cdots \vdash_{\mathbf{HS}} \exists\overline{Y'}, \overline{U}.\ (P \sqcup C \sqcup M \sqcup (\Pi' \sqcup \sigma_H) \sqcup \cdots D_i \cdots)@_{\mathcal{H}}\sigma'$
Si $pvar(P) \cap odvar_{\mathcal{H}}(\Pi_H) = \emptyset$, $\mathcal{X} =_{def} pvar(P) \cap var(\Pi_H)$, $(\Pi_H \sqcup \sigma_H)$ no está en forma resuelta con respecto a $\mathcal{X}$, $\Pi_H \vdash_{solve^{\mathcal{H}}_{\mathcal{X}}} \exists\overline{Y'}(\Pi' \sqcup \sigma')$.

SF Solving Failure
$\exists\overline{U}.\ P \sqcup C \sqcup M \sqcup H \sqcup \cdots D_i \cdots \vdash_{\mathbf{SF}} \blacksquare$
Si S es M, H, o D_i con $1 \leq i \leq n$, $pvar(P) \cap odvar_{\mathcal{D}_S}(\Pi_S) = \emptyset$, $\mathcal{X} =_{def} pvar(P) \cap var(\Pi_S)$, $(\Pi_S \sqcup \sigma_S)$ no está en χ-resuelto y $\Pi_S \vdash_{solve^{\mathcal{D}_S}_{\mathcal{X}}} \blacksquare$.
Observación: $\mathcal{X} \neq \emptyset$ en el caso de $S = H$.

Tabla 4.2: Reglas de invocación de resolutores

también son variables producidas ($\mathcal{X} =_{def} pvar(P) \cap var(\Pi_H)$). El estudio del proceso de reconocimiento de las variables críticas se mostró en la sección 3.5.

Nótese que las condiciones requeridas para los resolutores distintos de Herbrand son una simplificación de las condiciones requeridas para $\mathcal{H}$ pues todas sus variables son obviamente demandadas, así $\mathcal{X} = \emptyset$ y $odvar_{\mathcal{D}}(\Pi) = var(\Pi)$. La notación $\Pi \vdash_{solve^{\mathcal{D}}_{\mathcal{X}}} \exists\overline{Y'}(\Pi' \sqcup \sigma')$ introducida en la sección 3.1.7 indica una elección no determinista de una de las alternativas devueltas por la invocación al resolutor correspondiente.

Retomando el estudio de la restricción $\pi \equiv$ `L /= X:Xs` iniciada en la sección 3.5, supongamos un objetivo compuesto únicamente por ella. Inicialmente esta restricción se encuentra en el *pool* de restricciones y, como es una restricción primitiva atómica, se envía a su almacén. Para poder invocar al resolutor hay que elegir un conjunto de variables críticas, pero como no hay variables producidas $\mathcal{X} =_{def} pvar(P) \cap var(\Pi_H) = \emptyset$, por lo tanto la invocación al resolutor $solve^{\mathcal{H}}$(`L/=X:Xs`,$\emptyset$) devuelve `L/=X:Xs` $\sqcup\, \varepsilon$, donde `L/=X:Xs` es una forma resuelta con respecto al conjunto vacío de variables críticas.

Las reglas del cálculo $CCLNC(\mathcal{C})$ que están involucradas en la cooperación no se muestran en este capítulo porque son específicas de cada una de las instancias que se van a tratar en los siguientes capítulos. También se añadirán en esos capítulos ciertas reglas que infieren igualdades y desigualdades a partir de puentes.

Para introducir el concepto de solución de un objetivo se utiliza la lógica de reescritura parametrizada por un dominio de restricciones $CRWL(\mathcal{D})$ (*Constructor-based ReWriting Logic*). Esta lógica proporciona una semántica declarativa para programas $CFLP(\mathcal{D})$ como mues-

tra el trabajo [LRV07] y la tesis [dVV08]. $CRWL(\mathcal{D})$ está basada en la lógica de reescritura $CRWL$ introducida en [GHLR96, GHLR99], que proporciona un cálculo para computar los valores a los que se puede reducir una expresión, soportando *call-time choice*, indeterminismo y funciones perezosas. La lógica $CRWL(\mathcal{D})$ se parametriza con el dominio de coordinación $\mathcal{C}$.

La lógica $CRWL(\mathcal{C})$ permite generar *árboles de prueba*, que se obtienen al aplicar a una expresión las reglas del cálculo presentadas en [LRV07, dVV08]. Una expresión puede ser reducida a distintos patrones por el indeterminismo, lo que dará lugar a distintos árboles de prueba. Un árbol de prueba sirve como testigo de la corrección de una expresión. Formalmente, la notación $\mathcal{P} \vdash_{CRWL(\mathcal{C})} \varphi$ expresa que la expresión φ puede ser deducida del programa $\mathcal{P}$ aplicando las reglas de inferencia que definen la lógica de reescritura $CRWL(\mathcal{C})$. Es decir, φ es demostrable en $CRWL(\mathcal{C})$ con respecto al programa $\mathcal{P}$. Además, la notación $\mathcal{M} : \mathcal{P} \vdash_{CRWL(\mathcal{C})} \varphi$ indica que el *testigo* $\mathcal{M}$ es un multiconjunto formado por árboles de prueba $CRWL(\mathcal{C})$ que demuestran φ para un programa $\mathcal{P}$.

Definición 11. (Soluciones de objetivos)

Sea un objetivo admisible $G \equiv \exists \overline{U}.P \,\square\, C \,\square\, M \,\square\, H \,\square\, D_1 \,\square \cdots \square\, D_n$ para un programa $\mathcal{P}$ en el contexto $CFLP(\mathcal{C})$. El conjunto de soluciones del objetivo G con respecto al programa $\mathcal{P}$ ($Sol_{\mathcal{P}}(G)$) está formado por todas aquellas valoraciones $\mu \in Val_{\mathcal{C}}$ tales que existe alguna $\mu' \in Val_{\mathcal{C}}$ con $\mu' =_{\setminus \overline{U}} \mu$ y $\mu' \in Sol_{\mathcal{P}}(P \,\square\, C \,\square\, M \,\square\, H \,\square\, D_1 \,\square \cdots \square\, D_n)$. Esto se cumple si y solo si se satisfacen las siguientes condiciones:

1. $\mu' \in Sol_{\mathcal{P}}(P \,\square\, C)$. Por definición, esto significa $\mathcal{P} \vdash_{CRWL(\mathcal{C})} (P \,\square\, C)\mu'$, lo cual es equivalente a $\mathcal{P} \vdash_{CRWL(\mathcal{C})} P\mu'$ y $\mathcal{P} \vdash_{CRWL(\mathcal{C})} C\mu'$.
2. $\mu' \in Sol_{\mathcal{C}}(M \,\square\, H \,\square\, D_1 \,\square \cdots \square\, D_n)$, lo que equivale a $\mu' \in Sol_{\mathcal{C}}(M) \cap Sol_{\mathcal{C}}(H) \cap Sol_{\mathcal{C}}(D_1) \cap \cdots \cap Sol_{\mathcal{C}}(D_n)$.

Si un multiconjunto $\mathcal{M}$ contiene árboles de prueba como los mencionados en el punto 1 de la definición 11 entonces se dice que $\mathcal{M}$ es un *testigo* de $\mu \in Sol_{\mathcal{P}}(G)$, y se denota como $\mathcal{M} : \mu \in Sol_{\mathcal{P}}(G)$.

Una solución $\mu \in Sol_{\mathcal{P}}(G)$ está *bien tipada* si y solamente si la valoración $\mu' =_{\setminus \overline{U}} \mu$ de la definición 11 puede ser tomada de tal forma que $(P \,\square\, C \,\square\, M \,\square\, H \,\square\, F \,\square\, R)\mu'$ también está bien tipada y se denota como $\mu' \in WTSol_{\mathcal{P}}(P \,\square\, C \,\square\, M \,\square\, H \,\square\, F \,\square\, R)$. El conjunto de soluciones bien tipadas de G con respecto a $\mathcal{P}$ se escribe como $WTSol_{\mathcal{P}}(G)$. Si $\mathcal{M}$ es un testigo de $\mu \in Sol_{\mathcal{P}}(G)$, entonces $\mathcal{M}$ es un testigo de $\mu \in WTSol_{\mathcal{P}}(G)$ y se denota como $\mathcal{M} : \mu \in WTSol_{\mathcal{P}}(G)$.

Si un objetivo está en forma resuelta S, entonces no contiene ni producciones ni restricciones en el *pool* y, por lo tanto, se puede escribir $Sol_{\mathcal{P}}(S)$ en lugar de $Sol_{\mathcal{C}}(S)$. Análogamente se puede escribir $WTSol_{\mathcal{P}}(S)$ en lugar de $WTSol_{\mathcal{C}}(S)$.

Los próximos capítulos se van a dedicar a distintas instancias de cooperación. Por cada instancia del cálculo $CCLNC(\mathcal{C})$, se definirán reglas que complementarán el cálculo. Ade-

más, se demostrarán las propiedades semánticas de corrección y completitud con algunas limitaciones de cada instancia.

La completitud no se puede garantizar si hay variables lógicas de orden superior. También está limitada por las llamadas a los resolutores establecidas en la tabla 4.2. Por lo tanto se asume que no se hacen llamadas que producen respuestas incompletas de ningún resolutor. Por último, la completitud también está comprometida por las descomposiciones opacas producidas por las reglas **DC** de la tabla 4.1 y las reglas **H3** y **H7** de la tabla 3.3 que define el resolutor de Herbrand en la sección 3.5. Salvando estas limitaciones, se demostrará en los siguientes capítulos que las instancias del cálculo $CCLNC(\mathcal{C})$ son completas para soluciones bien tipadas.

Capítulo 5

Cooperación entre $\mathcal{FD}$ y $\mathcal{R}$

En el capítulo anterior se ha indicado que el procedimiento utilizado para realizar la cooperación entre varios dominios consiste en definir un dominio de coordinación en el cual intervienen los dominios puros que van a cooperar, el dominio de Herbrand y el dominio mediador que contiene los puentes que hacen posible la cooperación. En el caso concreto de la cooperación entre los dominios $\mathcal{FD}$ y $\mathcal{R}$, el dominio de coordinación se denota como:

$$\mathcal{C}_{\mathcal{FD}.\mathcal{R}} = \mathcal{M}_{\mathcal{FD}.\mathcal{R}} \oplus \mathcal{H} \oplus \mathcal{FD} \oplus \mathcal{R}$$

En este capítulo se define en primer lugar el dominio mediador $\mathcal{M}_{\mathcal{FD}.\mathcal{R}}$ y su correspondiente resolutor $solve^{\mathcal{M}_{\mathcal{FD}.\mathcal{R}}}$ mediante un sistema de transformación de almacenes. A continuación se establece el cálculo $CCLNC(\mathcal{C}_{\mathcal{FD}.\mathcal{R}})$ ampliando el cálculo $CCLNC(\mathcal{C})$ definido en la sección 4.2. Esta ampliación consiste en:

- Reglas que generan puentes que comunican los dominios $\mathcal{FD}$ y $\mathcal{R}$.
- Reglas que proyectan restricciones. Las proyecciones consisten en nuevas restricciones creadas en un dominio ($\mathcal{FD}$ o $\mathcal{R}$) a partir de restricciones existentes en el otro dominio en presencia de puentes.
- Reglas que infieren igualdades y desigualdades a partir de los puentes.
- Reglas que invocan a los resolutores de los dominios $\mathcal{FD}$ y $\mathcal{R}$.

Posteriormente se muestran ejemplos donde la cooperación es necesaria y las proyecciones mejoran la eficiencia como se muestra con resultados experimentales. Por último, se dedican dos secciones a la implementación de esta cooperación en el sistema $\mathcal{TOY}$ y su evaluación experimental.

5.1 El dominio mediador $\mathcal{M}_{\mathcal{FD}.\mathcal{R}}$

Para poder definir el dominio de coordinación, es necesario en primer lugar definir el dominio mediador para los dominios $\mathcal{FD}$ y $\mathcal{R}$. Estos dominios son unibles formando una suma amalgamada (definición 9) y produciendo así un nuevo dominio $\mathcal{FD} \oplus \mathcal{R}$ con signatura

$\Sigma_{\mathcal{FD}.\mathcal{R}} = \langle TC,\ SBT_{\mathcal{FD}} \cup SBT_{\mathcal{R}},\ DC,\ DF,\ SPF_{\mathcal{FD}} \cup SPF_{\mathcal{R}} \rangle$. Sin embargo, la suma amalgamada no proporciona mecanismos de comunicación entre estos dominios puros. Esta comunicación se establece mediante un nuevo dominio llamado *dominio mediador* $\mathcal{M}_{\mathcal{FD}.\mathcal{R}}$, cuya signatura es $\Sigma_{\mathcal{M}_{\mathcal{FD}.\mathcal{R}}} = \langle TC,\ SBT_{\mathcal{M}_{\mathcal{FD}.\mathcal{R}}},\ DC,\ DF,\ SPF_{\mathcal{M}_{\mathcal{FD}.\mathcal{R}}} \rangle$ donde:

- $SBT_{\mathcal{M}_{\mathcal{FD}.\mathcal{R}}} = \{\texttt{int}, \texttt{real}\} \subseteq SBT_{\mathcal{FD}} \cup SBT_{\mathcal{R}}$.
- Cada conjunto de valores básicos del dominio mediador corresponde a un conjunto de valores básicos de cada dominio puro: $\mathcal{B}^{\mathcal{M}_{\mathcal{FD}.\mathcal{R}}}_{\texttt{int}} = \mathcal{B}^{\mathcal{FD}}_{\texttt{int}} = \mathbb{Z}$ y $\mathcal{B}^{\mathcal{M}_{\mathcal{FD}.\mathcal{R}}}_{\texttt{real}} = \mathcal{B}^{\mathcal{R}}_{\texttt{real}} = \mathbb{R}$.
- $SPF_{\mathcal{M}_{\mathcal{FD}.\mathcal{R}}} = \{\ \texttt{\#==}\ ::\ \texttt{int} \to \texttt{real} \to \texttt{bool}\}$.

La restricción *puente* `#==` del dominio mediador relaciona cada valor entero con su correspondiente valor real. En otras palabras, `X #== RX` actúa como una *restricción de integralidad* sobre el valor de la variable real `RX`. La interpretación de la restricción puente t `#==`$^{\mathcal{M}_{\mathcal{FD}.\mathcal{R}}}$ $s \to b$ es la siguiente: o bien s es un número real con un valor equivalente al valor entero t y $b =$ `true`; o bien s un número real con un valor no equivalente al valor entero t y $b =$ `false`; o bien $b = \bot$. La interpretación del puente `#==` en $\mathcal{M}_{\mathcal{FD}.\mathcal{R}}$ corresponde a una relación inyectiva $inj : int \to real$ donde a cada valor entero le corresponde un valor real equivalente. Por lo tanto, la interpretación `#==`$^{\mathcal{M}_{\mathcal{FD}.\mathcal{R}}}$ es un subconjunto del producto Cartesiano $\mathbb{Z} \times \mathbb{R}$. Además decimos que una restricción puente de la forma t `#==` s abrevia t `#==` $s \to!$ `true` y una restricción llamada antipuente t `#/==` s abrevia t `#==` $s \to!$ `false`.

El resolutor de este nuevo dominio mediador, $solve^{\mathcal{M}_{\mathcal{FD}.\mathcal{R}}}$, se define mediante un sistema de transformación de almacenes con las reglas definidas en la tabla 5.1. Como en los capítulos anteriores, $\pi, \Pi \,\square\, \sigma \vdash_{\mathcal{M}_{\mathcal{FD}.\mathcal{R}}} \Pi' \,\square\, \sigma'$ indica que el almacén $\pi, \Pi \,\square\, \sigma$, que incluye la restricción primitiva atómica seleccionada π más otras restricciones Π, se transforma en el almacén $\Pi' \,\square\, \sigma'$. Las restricciones de la forma $(t$ `#==` $s) \to! b$ se refieren a puentes donde b es una variable o un valor Booleano y los patrones t y s son o bien una variable o bien un valor numérico del tipo que les corresponda (t `::` `int` y s `::` `real`).

Las reglas **M1** y **M2** de esta tabla corresponden a la reificación del puente. Nótese que cuando b es una variable que se vincula a `false`, el puente se transforma en un antipuente `#/==`. Este símbolo se utiliza para simplificar la notación pero no es especialmente relevante pues no está directamente implicado en la cooperación entre dominios. Las reglas **M3** y **M4** corresponden a puentes con su parte real constante, si es un valor entero entonces tiene éxito (**M3**) y en otro caso falla (**M4**). Cuando la parte $\mathcal{FD}$ del puente es un valor constante y la parte $\mathcal{R}$ es una variable (regla **M5**), entonces siempre se puede asignar a la variable real el valor equivalente a la constante $\mathcal{FD}$. Las reglas que van desde **M6** hasta **M9** consideran los casos con ambas partes constantes, escritos como puentes y antipuentes, que se resuelven dependiendo de los valores de sus argumentos. Por ejemplo si se tiene el puente X `#==` 2.0 entonces se aplica la regla **M3** con u' el valor $2 \in \mathbb{R}$ de forma que existe un valor $u \in \mathbb{Z}$, $u = 2$, que cumple 2 `#==`$^{\mathcal{M}_{\mathcal{FD}.\mathcal{R}}}$ 2.0 $\to$ *true* y $\sigma_1 = \{X \mapsto 2\}$. Si el puente es de la forma X `#==` 2.3 entonces se aplica la regla **M4** que falla.

Este sistema de transformación de almacenes define $solve^{\mathcal{M}_{\mathcal{FD}.\mathcal{R}}}$ y puede ser aceptado como una especificación correcta de un resolutor de caja transparente para el dominio $\mathcal{M}_{\mathcal{FD}.\mathcal{R}}$, como expresa el siguiente teorema que se demuestra en el apéndice A.6.

M1 $(t \texttt{ \#== } s) \rightarrow! B,\ \Pi \square \sigma \vdash_{\mathcal{M}_{\mathcal{FD}.\mathcal{R}}} (t \texttt{ \#== } s,\ \Pi)\sigma_1 \square \sigma\sigma_1$
si $t \in \mathcal{V}ar \cup \mathbb{Z}$, $s \in \mathcal{V}ar \cup \mathbb{R}$, $B \in \mathcal{V}ar$ y $\sigma_1 = \{B \mapsto true\}$.

M2 $(t \texttt{ \#== } s) \rightarrow! B,\ \Pi \square \sigma \vdash_{\mathcal{M}_{\mathcal{FD}.\mathcal{R}}} (t \texttt{ \#/== } s,\ \Pi)\sigma_1 \square \sigma\sigma_1$
si $t \in \mathcal{V}ar \cup \mathbb{Z}$, $s \in \mathcal{V}ar \cup \mathbb{R}$, $B \in \mathcal{V}ar$ y $\sigma_1 = \{B \mapsto false\}$.

M3 $X \texttt{ \#== } u',\ \Pi \square \sigma \vdash_{\mathcal{M}_{\mathcal{FD}.\mathcal{R}}} \Pi\sigma_1 \square \sigma\sigma_1$
si $X \in \mathcal{V}ar$, $u' \in \mathbb{R}$ y $\exists u \in \mathbb{Z}$ tal que $u \texttt{ \#==}^{\mathcal{M}_{\mathcal{FD}.\mathcal{R}}} u' \rightarrow true$ y $\sigma_1 = \{X \mapsto u\}$.

M4 $X \texttt{ \#== } u',\ \Pi \square \sigma \vdash_{\mathcal{M}_{\mathcal{FD}.\mathcal{R}}} \blacksquare$
si $u' \in \mathbb{R}$ y $\nexists u \in \mathbb{Z}$ tal que $u \texttt{ \#==}^{\mathcal{M}_{\mathcal{FD}.\mathcal{R}}} u' \rightarrow true$.

M5 $u \texttt{ \#== } RX,\ \Pi \square \sigma \vdash_{\mathcal{M}_{\mathcal{FD}.\mathcal{R}}} \Pi\sigma_1 \square \sigma\sigma_1$
si $RX \in \mathcal{V}ar$, $u \in \mathbb{Z}$ y $\exists u' \in \mathbb{R}$ tal que $u \texttt{ \#==}^{\mathcal{M}_{\mathcal{FD}.\mathcal{R}}} u' \rightarrow true$, y $\sigma_1 = \{RX \mapsto u'\}$.

M6 $u \texttt{ \#== } u',\ \Pi \square \sigma \vdash_{\mathcal{M}_{\mathcal{FD}.\mathcal{R}}} \Pi \square \sigma$
si $u \in \mathbb{Z}$, $u' \in \mathbb{R}$, y $u \texttt{ \#==}^{\mathcal{M}_{\mathcal{FD}.\mathcal{R}}} u' \rightarrow true$.

M7 $u \texttt{ \#== } u',\ \Pi \square \sigma \vdash_{\mathcal{M}_{\mathcal{FD}.\mathcal{R}}} \blacksquare$
si $u \in \mathbb{Z}$, $u' \in \mathbb{R}$ y $u \texttt{ \#==}^{\mathcal{M}_{\mathcal{FD}.\mathcal{R}}} u' \rightarrow false$.

M8 $u \texttt{ \#/== } u',\ \Pi \square \sigma \vdash_{\mathcal{M}_{\mathcal{FD}.\mathcal{R}}} \Pi \square \sigma$
si $u \in \mathbb{Z}$, $u' \in \mathbb{R}$ y $u \texttt{ \#==}^{\mathcal{M}_{\mathcal{FD}.\mathcal{R}}} u' \rightarrow false$

M9 $u \texttt{ \#/== } u',\ M \square \sigma \vdash_{\mathcal{M}_{\mathcal{FD}.\mathcal{R}}} \blacksquare$
si $u \in \mathbb{Z}$, $u' \in \mathbb{R}$ y $u \texttt{ \#==}^{\mathcal{M}_{\mathcal{FD}.\mathcal{R}}} u' \rightarrow true$.

Tabla 5.1: Reglas de transformación de almacenes que definen $\vdash_{\mathcal{M}_{\mathcal{FD}.\mathcal{R}}}$

Teorema 5. (Propiedades formales del resolutor $solve^{\mathcal{M}_{\mathcal{FD}.\mathcal{R}}}$)
El sistema de transformación de almacenes con relación de transición $\vdash_{\mathcal{M}_{\mathcal{FD}.\mathcal{R}}}$ cumple las propiedades de la definición 6. Además, como se mostró en la definición 5:

$$solve^{\mathcal{M}_{\mathcal{FD}.\mathcal{R}}}(\Pi) = \bigvee\{\Pi' \square \sigma' \mid \Pi' \square \sigma' \in SF_{\mathcal{M}_{\mathcal{FD}.\mathcal{R}}}(\Pi)\}$$

está bien definido para cualquier conjunto $\Pi \in APCon_{\mathcal{M}_{\mathcal{FD}.\mathcal{R}}}$. El resolutor $solve^{\mathcal{M}_{\mathcal{FD}.\mathcal{R}}}$ satisface todos los requisitos de los resolutores de la definición 4.

Además de proporcionar un medio de comunicación, el dominio mediador $\mathcal{M}_{\mathcal{FD}.\mathcal{R}}$ permite inferir que ciertas restricciones atómicas de Herbrand son restricciones $\mathcal{R}$-específicas o $\mathcal{FD}$-específicas.

Sea M el almacén de restricciones del dominio mediador $\mathcal{M}_{\mathcal{FD}.\mathcal{R}}$ y sea π una restricción atómica extendida de Herbrand bien tipada de la forma t_1 `==` t_2 o bien t_1 `/=` t_2, donde los patrones t_1 y t_2 son o una constante numérica v o una variable V. Entonces se dice:

1. $M \vdash \pi$ `in` $\mathcal{FD}$ ('M permite inferir que π es $\mathcal{FD}$-específica') si y solamente si algunas de las tres condiciones siguientes se cumple:

   (a) t_1 o t_2 es una constante entera.
   (b) t_1 o t_2 es una variable que está en el lado izquierdo de algún puente `#==`.
   (c) t_1 o t_2 es una variable que ha sido reconocida de tipo `int` por algún mecanismo dependiente de la implementación.

2. $M \vdash \pi$ `in` $\mathcal{R}$ ('M permite inferir que π es $\mathcal{R}$-específica') si y solamente si algunas de las tres condiciones siguientes se cumple:

   (a) t_1 o t_2 es una constante real.
   (b) t_1 o t_2 es una variable que está en el lado derecho de algún puente `#==`.
   (c) t_1 o t_2 es una variable que ha sido reconocida de tipo `real` por algún mecanismo dependiente de la implementación.

5.2 Cálculo $CCLNC(\mathcal{C}_{\mathcal{FD}.\mathcal{R}})$

Usando el dominio mediador definido en la sección anterior, podemos definir el dominio de coordinación $\mathcal{C}_{\mathcal{FD}.\mathcal{R}}$ de los dominios puros $\mathcal{H}$, $\mathcal{FD}$ y $\mathcal{R}$ como la siguiente suma amalgamada:

$$\mathcal{C}_{\mathcal{FD}.\mathcal{R}} = \mathcal{M}_{\mathcal{FD}.\mathcal{R}} \oplus \mathcal{H} \oplus \mathcal{FD} \oplus \mathcal{R}.$$

Todos los dominios utilizados en $\mathcal{C}_{\mathcal{FD}.\mathcal{R}}$ son unibles dos a dos y la comunicación con $\mathcal{H}$ se realiza automáticamente mediante las sustituciones de variables. Para este dominio de coordinación $\mathcal{C}_{\mathcal{FD}.\mathcal{R}}$, los objetivos tienen la forma: $G \equiv \exists\overline{U}.P \ \square \ C \ \square \ M \ \square \ H \ \square \ F \ \square \ R$, donde las variables locales $\overline{U}$, las producciones P, el *pool* de restricciones C, y el almacén H del dominio de Herbrand ya fueron introducidos en la sección 4.1. De forma similar, se definen:

- $M = \Pi_M \square \sigma_M$ es el *almacén del dominio mediador* $\mathcal{M}_{\mathcal{FD}.\mathcal{R}}$ definido como una conjunción de restricciones primitivas atómicas $\Pi_M \subseteq APCon_{\mathcal{M}_{\mathcal{FD}.\mathcal{R}}}$ y una sustitución σ_M. No se utiliza el subíndice $\mathcal{FD}.\mathcal{R}$ para no saturar la notación.
- $F = \Pi_F \square \sigma_F$ es el *almacén del dominio* $\mathcal{FD}$ donde $\Pi_F \subseteq APCon_{\mathcal{FD}}$ es un conjunto de restricciones primitivas atómicas del dominio $\mathcal{FD}$ y σ_F es una sustitución.
- $R = \Pi_R \square \sigma_R$ es el *almacén del dominio* $\mathcal{R}$ donde $\Pi_R \subseteq APCon_{\mathcal{R}}$ es un conjunto de restricciones primitivas atómicas del dominio $\mathcal{R}$ y σ_R es una sustitución.

Sobre este dominio $\mathcal{C}_{\mathcal{FD}.\mathcal{R}}$ se va a construir el cálculo $CCLNC(\mathcal{C}_{\mathcal{FD}.\mathcal{R}})$ que toma las reglas de estrechamiento perezoso de la sección 4.2. En esta sección se va a definir la parte del cálculo que es particular a la cooperación entre $\mathcal{R}$ y $\mathcal{FD}$ y las invocaciones específicas a los resolutores de $\mathcal{R}$ y $\mathcal{FD}$.

El proceso de cooperación se basa fundamentalmente en la creación de puentes y en la proyección de restricciones que se formaliza mediante las reglas de transformación de objetivos definidas en la tablas 5.2, 5.3 y 5.4. Pero antes se va a mostrar un ejemplo introductorio a la aplicación de las reglas de creación de puentes y a las reglas de proyección que posteriormente serán explicadas. Este ejemplo es extensión del ejemplo 3.

Ejemplo 4. (Generación de puentes y proyecciones)

Sea el siguiente objetivo admisible en el cual se tiene una restricción real atómica en el *pool* de restricciones y tres puentes en el almacén mediador.

$\square\ \underline{(RX + 2 * RY) * RZ \texttt{ <= } 3.5}\ \square\ X \texttt{ \#== } RX, Y \texttt{ \#== } RY, Z \texttt{ \#== } RZ\ \square\square\square \vdash_{\mathbf{FC^3},\mathbf{PC^3}}$

En primer lugar se aplana la restricción real atómica subrayada mediante la aplicación iterativa de las reglas **FC** (Flatten Constraint) y **PC** (Place Constraint) de la tabla 4.1 (ejemplo 3). La notación RL^i significa que la regla RL se ha aplicado i veces. Como resultado se produce una conjunción de cuatro restricciones primitivas atómicas reales implicando tres variables existenciales nuevas:

$\exists RC, RB, RA.\ \square\ \underline{2 * RY \texttt{ }\to!\texttt{ } RC}, \underline{RX + RC \texttt{ }\to!\texttt{ } RB}, \underline{RB * RZ \texttt{ }\to!\texttt{ } RA}, RA \texttt{ <= } 3.5\ \square$
$X \texttt{ \#== } RX, Y \texttt{ \#== } RY, Z \texttt{ \#== } RZ\ \square\square\square \vdash_{\mathbf{SB^3}}$

Las restricciones que se encuentran en el *pool* son restricciones primitivas atómicas. Antes de enviarlas a sus correspondientes almacenes se generan puentes para sus variables, si es posible hacerlo. La regla **SB** (Set Bridges) de la tabla 5.2, que será explicada más adelante, genera puentes para las variables nuevas de los pasos anteriores RA, RB y RC, utilizando las variables nuevas A, B y C.

$\exists C, B, A, RC, RB, RA.\ \square\ \underline{2 * RY \texttt{ }\to!\texttt{ } RC}, \underline{RX + RC \texttt{ }\to!\texttt{ } RB}, \underline{RB * RZ \texttt{ }\to!\texttt{ } RA}, \underline{RA \texttt{ <= } 3.5}\ \square$
$C \texttt{ \#== } RC, B \texttt{ \#== } RB, A \texttt{ \#== } RA, X \texttt{ \#== } RX, Y \texttt{ \#== } RY, Z \texttt{ \#== } RZ\ \square\square\square \vdash_{\mathbf{PP^4}}$

Los puentes generados se utilizan para proyectar las restricciones del *pool* mediante otra regla denominada **PP** (Propagate Projections). En este caso todas las restricciones se proyectan al dominio $\mathcal{FD}$.

$\exists C, B, A, RC, RB, RA.\ \square\ \underline{2 * RY \texttt{ }\to!\texttt{ } RC}, \underline{RX + RC \texttt{ }\to!\texttt{ } RB}, \underline{RB * RZ \texttt{ }\to!\texttt{ } RA}, \underline{RA \texttt{ <= } 3.5}\ \square$
$C \texttt{ \#== } RC, B \texttt{ \#== } RB, A \texttt{ \#== } RA, X \texttt{ \#== } RX, Y \texttt{ \#== } RY, Z \texttt{ \#== } RZ\ \square\square$
$2 \texttt{ \#* } Y \texttt{ }\to!\texttt{ } C, X \texttt{ \#+ } C \texttt{ }\to!\texttt{ } B, B \texttt{ \#* } Z \texttt{ }\to!\texttt{ } A, A \texttt{ \#<= } 3\ \square \vdash_{\mathbf{SC^4}}$

Finalmente, las restricciones que se encuentran en el *pool* de restricciones se envían a su correspondiente almacén R mediante la regla **SC** (Submit Constraints).

$\exists C, B, A, RC, RB, RA.\ \square\square\ C \texttt{ \#== } RC, B \texttt{ \#== } RB, A \texttt{ \#== } RA, X \texttt{ \#== } RX, Y \texttt{ \#== } RY, Z \texttt{ \#== } RZ$
$\square\square\ 2 \texttt{ \#* } Y \texttt{ }\to!\texttt{ } C, X \texttt{ \#+ } C \texttt{ }\to!\texttt{ } B, B \texttt{ \#* } Z \texttt{ }\to!\texttt{ } A, A \texttt{ \#<= } 3\ \square$
$2 * RY \texttt{ }\to!\texttt{ } RC, RX + RC \texttt{ }\to!\texttt{ } RB, RB * RZ \texttt{ }\to!\texttt{ } RA, RA \texttt{ <= } 3.5$

Al no haber más restricciones en el *pool* ya no hay más generación de puentes ni de proyecciones.

El proceso de cooperación se realiza cuando se va a tratar una restricción π ubicada en el *pool* de restricciones C. Las reglas contenidas en la tabla 5.2 son las encargadas de generar puentes, propagar proyecciones y enviar restricciones atómicas a su resolutor. La regla **SB** establece puentes mediante la función *bridges* para las variables contenidas en dicha restricción π, si procede. Después, la regla **PP** proyecta la restricción π mediante la función *proj* creando nuevas restricciones que serán enviadas al correspondiente resolutor, según sea

SB Set Bridges

$$\exists\overline{U}.\ P \sqcap \pi,\ C \sqcap M \sqcap H \sqcap F \sqcap R \vdash_{\mathbf{SB}} \exists\overline{V}', \overline{U}.\ P \sqcap \pi,\ C \sqcap M' \sqcap H \sqcap F \sqcap R$$

Si π es una restricción primitiva atómica y se cumple o bien (i) o bien (ii)

(i) π es una restricción $\mathcal{FD}$ propia o si no una restricción extendida de $\mathcal{H}$ tal que $M \vdash \pi$ `in` $\mathcal{FD}$, y $M' = B', M$, donde $\exists\overline{V'}\, B' = bridges^{\mathcal{FD}\to\mathcal{R}}(\pi, \Pi_M) \neq \emptyset$.

(ii) π es una restricción $\mathcal{R}$ propia o si no una restricción extendida de $\mathcal{H}$ tal que $M \vdash \pi$ `in` $\mathcal{R}$, y $M' = B', M$, donde $\exists\overline{V'}\, B' = bridges^{\mathcal{R}\to\mathcal{FD}}(\pi, \Pi_M) \neq \emptyset$.

PP Propagate Projections

$$\exists\overline{U}.\ P \sqcap \pi,\ C \sqcap M \sqcap H \sqcap F \sqcap R \vdash_{\mathbf{PP}} \exists\overline{V'}, \overline{U}.\ P \sqcap \pi,\ C \sqcap M \sqcap H \sqcap F' \sqcap R'$$

Si π es una restricción primitiva atómica y se cumplen (i) o bien (ii)

(i) π es una restricción $\mathcal{FD}$ propia o si no una restricción extendida de $\mathcal{H}$ tal que $M \vdash \pi$ `in` $\mathcal{FD}$, $\exists\overline{V'}\, \Pi' = proj^{\mathcal{FD}\to\mathcal{R}}(\pi, \Pi_M) \neq \emptyset$, $F' = F$ y $R' = \Pi', R$.

(ii) π es una restricción $\mathcal{R}$ propia o si no una restricción extendida de $\mathcal{H}$ tal que $M \vdash \pi$ `in` $\mathcal{R}$, $\exists\overline{V'}\, \Pi' = proj^{\mathcal{R}\to\mathcal{FD}}(\pi, \Pi_M) \neq \emptyset$, $F' = \Pi', F$ y $R' = R$.

SC Submit Constraints

$$\exists\overline{U}.\ P \sqcap \pi,\ C \sqcap M \sqcap H \sqcap F \sqcap R \vdash_{\mathbf{SC}} \exists\overline{U}.\ P \sqcap C \sqcap M' \sqcap H' \sqcap F' \sqcap R'$$

Si π es una restricción primitiva atómica y se cumple uno de los siguientes casos:

(i) π es una restricción de $\mathcal{M}_{\mathcal{FD}.\mathcal{R}}$, $M' = \pi, M$, $H' = H$, $F' = F$, y $R' = R$.

(ii) π es una restricción extendida de Herbrand tal que ni $M \vdash \pi$ `in` $\mathcal{FD}$ ni $M \vdash \pi$ `in` $\mathcal{R}$, $M' = M$, $H' = \pi, H$, $F' = F$ y $R' = R$.

(iii) π es una restricción propia de $\mathcal{FD}$ o si no una restricción extendida de Herbrand tal que $M \vdash \pi$ `in` $\mathcal{FD}$, $M' = M$, $H' = H$, $F' = \pi, F$ y $R' = R$.

(iv) π es una restricción propia de $\mathcal{R}$ o si no una restricción extendida de Herbrand tal que $M \vdash \pi$ `in` $\mathcal{R}$, $M' = M$, $H' = H$, $F' = F$ y $R' = \pi, R$.

Tabla 5.2: Reglas de generación de puentes y proyecciones para el cálculo $CCLNC(\mathcal{C}_{\mathcal{FD}.\mathcal{R}})$

el caso. Hay que tener en cuenta que no todas las restricciones se pueden proyectar, pues la proyección depende de la semántica de la restricción. Finalmente, la restricción π se envía a su resolutor mediante la regla **SC**. La formulación de las reglas **SB**, **PP** y **SC** de la tabla 5.2 se basa en la identificación de ciertas restricciones primitivas atómicas de Herbrand π como restricciones $\mathcal{FD}$-específicas o $\mathcal{R}$-específicas, como indica la notación $M \vdash \pi$ `in` $\mathcal{FD}$ y $M \vdash \pi$ `in` $\mathcal{R}$.

Las tablas 5.3 y 5.4 definen las funciones *bridges* (nuevos puentes) y *proj* (proyecciones) usadas en la tabla 5.2 para una cierta restricción π del dominio $\mathcal{FD}$ o $\mathcal{R}$, respectivamente. Si la restricción a evaluar (contenida en el *pool* de restricciones) es una restricción primitiva atómica π, entonces se aplica la función $bridges^{\mathcal{FD}\to\mathcal{R}}(\pi, B)$ o la función $bridges^{\mathcal{R}\to\mathcal{FD}}(\pi, B)$,

$\pi \in \mathcal{FD}$	$bridges^{\mathcal{FD}\to\mathcal{R}}(\pi, B)$	$proj^{\mathcal{FD}\to\mathcal{R}}(\pi, B)$
`domain` $[X_1, \ldots, X_n]$ a b	$\{X_i$ `#==` $RX_i \mid 1 \leq i \leq n$, X_i no tiene puentes en B y RX_i es nueva$\}$	$\{a$ `<=` RX_i, RX_i `<=` $b \mid 1 \leq i \leq n$ y $(X_i$ `#==` $RX_i) \in B\}$
`belongs` X $[a_1, \ldots, a_n]$	$\{X$ `#==` $RX \mid X$ no tiene puente en B y RX es nueva$\}$	$\{\min(a_1, \ldots, a_n)$ `<=` RX, RX `<=` $\max(a_1, \ldots, a_n) \mid (X$ `#==` $RX) \in B\}$
t_1 `#<` t_2 (resp. `#<=`, `\#>`, `#=>`)	$\{X_i$ `#==` $RX_i \mid 1 \leq i \leq 2$, t_i es una var. X_i sin puentes en B, y RX_i son nuevas$\}$	$\{t_1^{\mathcal{R}}$ `<` $t_2^{\mathcal{R}} \mid$ Para $1 \leq i \leq 2$: O t_i es una cte. n y $t_i^{\mathcal{R}}$ es el valor real de n, o bien t_i es una variable X_i con $(X_i$ `#==` $RX_i) \in B$, y $t_i^{\mathcal{R}}$ es $RX_i\}$
t_1 `==` t_2	$\{X_i$ `#==` $RX_i \mid 1 \leq i \leq 2$, o t_1 es una cte. entera y t_2 es una variable X sin puentes en B (o viceversa) y RX_i son nuevas$\}$	$\{t_1^{\mathcal{R}}$ `==` $t_2^{\mathcal{R}} \mid$ Para $1 \leq i \leq 2$: $t_i^{\mathcal{R}}$ se determina como en el caso `#<` $\}$
t_1 `/=` t_2	$\{X_i$ `#==` $RX_i \mid 1 \leq i \leq 2$, o t_1 es una cte. entera y t_2 una variable X sin puentes en B (o viceversa) y RX_i son nuevas$\}$	$\{t_1^{\mathcal{R}}$ `/=` $t_2^{\mathcal{R}} \mid$ Para $1 \leq i \leq 2$: $t_i^{\mathcal{R}}$ se determina como en el caso `#<` $\}$
t_1 `#+` $t_2 \to!$ X (resp. `#-`, `#*`)	$\{X_i$ `#==` RX_i, X `#==` $RX \mid 1 \leq i \leq 2$, t_i es una variable X_i sin puentes en B y RX_i son nuevas$\}$	$\{t_1^{\mathcal{R}}$ `+` $t_2^{\mathcal{R}} \to!$ $RX \mid$ Para $1 \leq i \leq 3$: $t_i^{\mathcal{R}}$ se determina como en el caso `#<` $\}$

Tabla 5.3: Generación de puentes y proyecciones desde $\mathcal{FD}$ a $\mathcal{R}$

contenidas en las tablas 5.3 y 5.4 dependiendo de la forma de la restricción π. Se utiliza la notación B para representar al conjunto de restricciones puentes (bridges) del dominio mediador.

La función $bridges^{\mathcal{FD}\to\mathcal{R}}(\pi, B)$ genera nuevos puentes para las variables contenidas en π, si estos no estaban anteriormente en el conjunto B de restricciones puente pertenecientes al almacén del dominio mediador M. Esta generación de restricciones es posible porque cada variable entera tiene asociada una variable real. La función $bridges^{\mathcal{R}\to\mathcal{FD}}(\pi, B)$ es análoga a la función $bridges^{\mathcal{FD}\to\mathcal{R}}(\pi, B)$, pero en este caso la creación de puentes no es siempre posible pues no toda variable real se corresponde con una variable entera.

Una vez establecidos los nuevos puentes se procede con la proyección de la restricción π mediante la aplicación de la función $proj^{\mathcal{FD}\to\mathcal{R}}(\pi, B)$ o la función $proj^{\mathcal{R}\to\mathcal{FD}}(\pi, B)$, según corresponda el caso. La función $proj^{\mathcal{FD}\to\mathcal{R}}(\pi, B)$ construye restricciones análogas a π en el dominio $\mathcal{R}$ teniendo en cuenta los puentes disponibles en B, o envía información que contiene la restricción π como es el caso de la restricción `belongs :: int->[int]->bool` de $\mathcal{FD}$. Esta restricción recibe como primer argumento una constante entera o una variable del dominio $\mathcal{FD}$, como segundo argumento una lista básica o ground de tipo `int` y el tercer argumento reifica la restricción. La proyección de esta restricción establece, mediante primitivas del

$\pi \in \mathcal{R}$	$bridges^{\mathcal{R}\rightarrow\mathcal{FD}}(\pi, B)$	$proj^{\mathcal{R}\rightarrow\mathcal{FD}}(\pi, B)$
RX<RY (resp. >,<=,>=)	$\emptyset$	$\{X$ #< $Y \mid (X$ #== $RX)$,$(Y$ #== $RY) \in B\}$
RX < a	$\emptyset$	$\{X$ #< $\lceil a \rceil \mid a \in \mathbb{R}, (X$ #== $RX) \in B\}$
a < RY	$\emptyset$	$\{\lfloor a \rfloor$ #< $Y \mid a \in \mathbb{R}, (Y$ #== $RY) \in B\}$
RX <= a	$\emptyset$	$\{X$ #<= $\lfloor a \rfloor \mid a \in \mathbb{R}, (X$ #== $RX) \in B\}$
a <= RY	$\emptyset$	$\{\lceil a \rceil$ #<= $Y \mid a \in \mathbb{R}, (Y$ #== $RY) \in B\}$
RX > a	$\emptyset$	$\{X$ #> $\lfloor a \rfloor \mid a \in \mathbb{R}, (X$ #== $RX) \in B\}$
a > RY	$\emptyset$	$\{\lceil a \rceil$ #> $Y \mid a \in \mathbb{R}, (Y$ #== $RY) \in B\}$
RX >= a	$\emptyset$	$\{X$ #>= $\lceil a \rceil \mid a \in \mathbb{R}, (X$ #== $RX) \in B\}$
a >= RY	$\emptyset$	$\{\lfloor a \rfloor$ #>= $Y \mid a \in \mathbb{R}, (Y$ #== $RY) \in B\}$
t_1 == t_2	$\{X$ #== $RX \mid$ o t_1 es una cte. real con parte decimal nula y t_2 es una variable RX sin puentes en B (o viceversa) y X es nueva$\}$	$\{t_1^{\mathcal{FD}}$ == $t_2^{\mathcal{FD}} \mid$ Para $1 \leq i \leq 2$: o t_i es una cte. real n con parte decimal nula y $t_i^{\mathcal{FD}}$ es el entero de n, o si no t_i es una variable RX_i con $(X_i$ #== $RX_i) \in B$ y $t_i^{\mathcal{FD}}$ es $X_i\}$
t_1 /= t_2	$\emptyset$	$\{t_1^{\mathcal{FD}}$ /= $t_2^{\mathcal{FD}} \mid$ Para $1 \leq i \leq 2$: $t_i^{\mathcal{FD}}$ se determina como en el caso anterior$\}$
t_1 + $t_2 \rightarrow!$ t_3 (resp. -, *)	$\{X$ #== $RX \mid$ si t_3 es una variable RX sin puentes en B, t_1 y t_2 son o bien una cte. de parte entera nula o variable contenida en algún puente y X es nueva$\}$	$\{t_1^{\mathcal{FD}}$ #+ $t_2^{\mathcal{FD}} \rightarrow!$ $t_3^{\mathcal{FD}} \mid$ Para $1 \leq i \leq 3$: $t_i^{\mathcal{FD}}$ se determina como en el caso anterior$\}$
t_1 / $t_2 \rightarrow!$ t_3	$\emptyset$	$\{t_2^{\mathcal{FD}}$ #* $t_3^{\mathcal{FD}} \rightarrow!$ $t_1^{\mathcal{FD}} \mid$ Para $1 \leq i \leq 3$: $t_i^{\mathcal{FD}}$ se determina como en el caso anterior$\}$

Tabla 5.4: Generación de puentes y proyecciones desde $\mathcal{R}$ a $\mathcal{FD}$

dominio $\mathcal{R}$, que la correspondiente variable real está acotada por los valores mínimo y máximo de la lista de valores. Por ejemplo, en el sistema $\mathcal{TOY}$ el objetivo `belongs X [1,1000,100], X #== RX` muestra la siguiente solución:

```
Toy(FD+R+p)> belongs X [1,1000,100], X #== RX
    { X #== RX,
      RX=<1000.0,
      RX>=1.0,
      [X]-(X in{1}\/{100}\/{1000}) }
```

En la solución mostrada aparece la restricción puente `X #== RX` que permanece en el almacén, la restricción de $\mathcal{FD}$ que muestra los tres posibles valores de `X in {1}\/{100}\/{1000}` y las dos restricciones del dominio $\mathcal{R}$ que resultan de la proyección: `RX>=1.0` y `RX=<1000.0`. Como se muestra en el ejemplo, la proyección de la restricción `belongs` no trata los posibles agujeros que tenga la lista pues utilizando únicamente las primitivas de $\mathcal{R}$ no se puede tratar la disyunción de valores.

La función $proj^{\mathcal{R}\to\mathcal{FD}}(\pi, B)$ construye restricciones análogas a π en el dominio $\mathcal{FD}$ teniendo en cuenta los puentes disponibles en B. La notación $\lfloor a \rfloor$ (respectivamente $\lceil a \rceil$) redondea el número $a \in \mathbb{R}$ al mayor entero que es menor o igual que a (respectivamente el menor entero que es mayor o igual que a).

Si una restricción $\mathcal{FD}$ o $\mathcal{R}$ no está representada en las tablas 5.3 y 5.4 entonces el resultado de aplicar las funciones *bridge* y *proj* a dichas restricciones es el conjunto vacío.

Las funciones de creación de puentes y proyecciones que se acaban de definir tienen propiedades de corrección y completitud como se muestra a continuación. Su demostración se omite pues no tiene ninguna dificultad técnica.

Proposición 6. (Propiedades de los puentes y proyecciones entre $\mathcal{FD}$ y $\mathcal{R}$)
Sean $\mathcal{D}$ y $\mathcal{D}'$ los dominios $\mathcal{FD}$ y $\mathcal{R}$, o viceversa. Entonces:

1. $bridges^{\mathcal{D}\to\mathcal{D}'}(\pi, B)$ y $proj^{\mathcal{D}\to\mathcal{D}'}(\pi, B)$ están bien definidos para cualquier restricción primitiva atómica π que es o bien propia de $\mathcal{D}$ o bien $\mathcal{D}$-específica de Herbrand para cualquier conjunto finito de puentes B.

2. $bridges^{\mathcal{D}\to\mathcal{D}'}(\pi, B)$ devuelve un conjunto finito de nuevos puentes B' que contienen nuevas variables $\overline{V'}$. En particular, $bridges^{\mathcal{D}\to\mathcal{D}'}(\pi, B) = \emptyset$ cuando la restricción π no está representada en ninguna de las tablas 5.3 y 5.4. Para este nuevo conjunto se cumplen las condiciones de *corrección* $Sol_{\mathcal{C}}(\pi \wedge B) \supseteq Sol_{\mathcal{C}}(\exists\overline{V'}(\pi \wedge B \wedge B'))$ y *completitud* $WTSol_{\mathcal{C}}(\pi \wedge B) \subseteq WTSol_{\mathcal{C}}(\exists\overline{V'}(\pi \wedge B \wedge B'))$.

3. $proj^{\mathcal{D}\to\mathcal{D}'}(\pi, B)$ devuelve un conjunto finito de restricciones atómicas primitivas $\Pi' \subseteq APCon_{\mathcal{D}'}$ que contienen nuevas variables $\overline{V'}$. En particular, $proj^{\mathcal{D}\to\mathcal{D}'}(\pi, B) = \emptyset$ cuando la restricción π no está representada en ninguna de las tablas 5.3 y 5.4. Para este nuevo conjunto se cumplen las condiciones de *corrección* $Sol_{\mathcal{C}}(\pi \wedge B) \supseteq Sol_{\mathcal{C}}(\exists\overline{V'}(\pi \wedge B \wedge \Pi'))$ y *completitud* $WTSol_{\mathcal{C}}(\pi \wedge B) \subseteq WTSol_{\mathcal{C}}(\exists\overline{V'}(\pi \wedge B \wedge \Pi'))$.

Además de las reglas de generación de puentes y proyecciones que intervienen en la cooperación, en la tabla 5.5 se definen una serie de reglas que permiten deducir igualdades y desigualdades de variables a partir de los puentes y antipuentes. En concreto, las reglas que infieren igualdades establecen que si dos puentes coinciden en su parte real entonces deben coincidir en su parte entera y viceversa. Por otra parte los antipuentes con una parte básica establecen restricciones de desigualdad con los valores que se obtienen de puentear los valores básicos correspondientes.

Una vez acabado el proceso de cooperación, la restricción correspondiente que se encuentra en el *pool* C se envía a su resolutor. En la tabla 4.2 del capítulo anterior se definen las reglas que corresponden a las invocaciones de los resolutores de Herbrand, del dominio mediador y de un cierto dominio puro D_i, además de la regla de fallo por la invocación a un resolutor. La adaptación de estas reglas para los cuatro dominios del cálculo $CCLNC(\mathcal{C}_{\mathcal{FD}.\mathcal{R}})$ es trivial y

IE Infer Equalities

$\exists\overline{U}.\ P \sqcup C \sqcup X \texttt{ \#== } RX,\ X' \texttt{ \#== } RX,\ M \sqcup H \sqcup F \sqcup R \vdash_{\mathbf{IE}}$
$\exists\overline{U}.\ P \sqcup C \sqcup X \texttt{ \#== } RX,\ M \sqcup H \sqcup X \texttt{ == } X', F \sqcup R.$

$\exists\overline{U}.\ P \sqcup C \sqcup X \texttt{ \#== } RX,\ X \texttt{ \#== } RX',\ M \sqcup H \sqcup F \sqcup R \vdash_{\mathbf{IE}}$
$\exists\overline{U}.\ P \sqcup C \sqcup X \texttt{ \#== } RX,\ M \sqcup H \sqcup F \sqcup RX\texttt{==}RX', R.$

ID Infer Disequalities

$\exists\overline{U}.\ P \sqcup C \sqcup X\texttt{\#/==}u',\ M \sqcup H \sqcup F \sqcup R \vdash_{\mathbf{ID}} \exists\overline{U}.\ P \sqcup C \sqcup M \sqcup H \sqcup X\texttt{/=}u, F \sqcup R$
si $u \in \mathbb{Z}$, $u' \in \mathbb{R}$ y $u \texttt{ \#==}^{\mathcal{M}_{\mathcal{FD}.\mathcal{R}}} u' \to true$.

$\exists\overline{U}.\ P \sqcup C \sqcup u\texttt{\#/==}RX,\ M \sqcup H \sqcup F \sqcup R \vdash_{\mathbf{ID}} \exists\overline{U}.\ P \sqcup C \sqcup M \sqcup H \sqcup F \sqcup RX\texttt{/=}u', R$
si $u \in \mathbb{Z}$, $u' \in \mathbb{R}$ y $u \texttt{ \#==}^{\mathcal{M}_{\mathcal{FD}.\mathcal{R}}} u' \to true$.

Tabla 5.5: Reglas para inferir restricciones $\mathcal{H}$ utilizando restricciones del dominio $\mathcal{M}_{\mathcal{FD}.\mathcal{R}}$

únicamente se muestran en la tabla 5.6 las reglas correspondientes a las invocaciones de los resolutores $\mathcal{R}$ y $\mathcal{FD}$.

FDS $\mathcal{FD}$-Constraint Solver (*secuenciación de resolutores*)

$\exists\overline{U}.\ P \sqcup C \sqcup M \sqcup H \sqcup F \sqcup R \vdash_{\mathbf{FS}} \exists\overline{Y'}, \overline{U}.\ (P \sqcup C \sqcup M \sqcup H \sqcup (\Pi' \sqcup \sigma_F) \sqcup R)@_{\mathcal{FD}}\sigma'$
Si $pvar(P) \cap var(\Pi_F) = \emptyset$, $(\Pi_F \sqcup \sigma_F)$ no está resuelto, $\Pi_F \vdash_{solve^{\mathcal{FD}}} \exists\overline{Y'}(\Pi' \sqcup \sigma')$.

RS $\mathcal{R}$-Constraint Solver (*caja negra*)

$\exists\overline{U}.\ P \sqcup C \sqcup M \sqcup H \sqcup F \sqcup R \vdash_{\mathbf{RS}} \exists\overline{Y'}, \overline{U}.\ (P \sqcup C \sqcup M \sqcup H \sqcup F \sqcup (\Pi' \sqcup \sigma_R))@_{\mathcal{R}}\sigma'$
Si $pvar(P) \cap var(\Pi_R) = \emptyset$, $(\Pi_R \sqcup \sigma_R)$ no está resuelto, $\Pi_R \vdash_{solve^{\mathcal{R}}} \exists\overline{Y'}(\Pi' \sqcup \sigma')$.

Tabla 5.6: Reglas de resolución de restricciones de los almacenes FD y R

Las reglas para la cooperación presentadas hasta este punto, junto con las reglas para tratar el estrechamiento perezoso del capítulo anterior forman el cálculo $CCLNC(\mathcal{FD}.\mathcal{R})$. Como ya se comentó en el capítulo 4, este cálculo deja un amplio margen para la elección de un determinado paso de transformación, de manera que en principio hay muchos cómputos distintos. Sin embargo, la implementación de $\mathcal{TOY}$ sigue una estrategia en particular que se esbozó en la sección 4.2. A continuación se describe la estrategia utilizada por $\mathcal{TOY}$ con las nuevas reglas definidas en esta sección.

1. Si P contiene alguna producción que pueda ser evaluada por alguna regla del estrechamiento perezoso de la tabla 4.1, entonces de estas producciones, se selecciona la producción que está más a la izquierda y se procesa. Las suspensiones son retrasadas hasta que puedan ser eliminadas de forma segura mediante la regla **EL** o bien dejen de ser suspensiones.

2. Si P es el conjunto vacío o únicamente tiene producciones que no pueden ser procesadas por las reglas de estrechamiento perezoso de la tabla 4.1, y además alguno de los

almacenes M, H, F o R no está en forma resuelta y sus restricciones no contienen variables producidas, entonces se invoca a los resolutores de estos almacenes. En el caso de invocar al resolutor de Herbrand se elige el conjunto de variables críticas $\mathcal{X}$ como se explicó en la sección 4.2.

3. Si ninguno de los dos puntos anteriores pueden ser aplicados y C no está vacío, entonces se procesa la restricción π de C que está más a la izquierda. Si no es una restricción primitiva atómica entonces se aplana y en caso contrario se aplica uno de los siguientes casos:

   (a) Si π es una restricción propia del dominio $\mathcal{FD}$ o una restricción extendida de $\mathcal{H}$ tal que $M \vdash \pi$ `in` $\mathcal{FD}$, entonces π se procesa aplicando las siguientes reglas: **SB** para generar puentes para las variables contenidas en π si no están ya generados, **PP** para crear nuevas restricciones del dominio $\mathcal{R}$ si es posible y ubicarlas en el almacén R y **SC** para enviar la restricción π al almacén F.
   Después se invocan a los resolutores de $\mathcal{FD}$ y $\mathcal{R}$ siempre y cuando sus almacenes no contengan variables producidas.

   (b) Si π es una restricción propia del dominio $\mathcal{R}$ o una restricción extendida de $\mathcal{H}$ tal que $M \vdash \pi$ `in` $\mathcal{R}$, entonces π se procesa aplicando las reglas **SB**, **PP** y **SC**, e invocando a los resolutores de forma análoga al caso anterior.

   (c) Si π es una restricción de Herbrand extendida tal que ni $M \vdash \pi$ `in` $\mathcal{FD}$ ni $M \vdash \pi$ `in` $\mathcal{R}$, entonces π se envía al almacén H aplicando la regla **SC**, y se invoca al resolutor de $\mathcal{H}$ si H no contiene variables producidas obviamente demandadas.

   (d) Si π es un puente del dominio mediador $\mathcal{M}_{\mathcal{FD}.\mathcal{R}}$, entonces se aplica la regla **SC** y π se envía al almacén M. Si es posible, se aplican las reglas de la tabla 5.5 y se invoca al resolutor de $\mathcal{M}_{\mathcal{FD}.\mathcal{R}}$ si las restricciones de M no incluyen variables producidas.

A continuación se ilustra detalladamente lo expuesto hasta el momento. Para ello, se desarrolla el cálculo en el ejemplo motivador 1 mostrado en la sección 1.1 de la introducción.

Ejemplo 5. (Intersección discreta de dos regiones)
El siguiente código $\mathcal{TOY}$ determina los puntos discretos que corresponden a la intersección de dos regiones, una definida con puntos discretos mediante una cuadrícula y la otra definida como un triángulo continuo.

```
1  type dPoint = (int, int)
2  type cPoint = (real, real)

3  type setOf A = A -> bool

4  type grid = setOf dPoint
5  type region = setOf cPoint
```

```
6  triangle :: cPoint -> real  -> real -> region
   triangle (RX0,RY0) B H (RX,RY) :-
      RY >= RY0 - H,
      B * RY - 2 * H * RX <= B * RY0 - 2 * H * RX0,
      B * RY + 2 * H * RX <= B * RY0 + 2 * H * RX0

7  square :: int -> grid
   square N (X,Y) :- domain [X,Y] 0 N

8  isIn :: setOf A -> A -> bool
   isIn Set Element = Set Element

9  bothIn :: region -> grid -> dPoint -> bool
   bothIn Region Grid (X, Y) :- X #== RX, Y #== RY,
      isIn Region (RX, RY), isIn Grid (X,Y), labeling [] [X,Y]
```

En este ejemplo, además de las características de $\mathcal{TOY}$ ya introducidas en la sección 1.3, se dispone de los alias de tipos que utilizan la palabra reservada `type`, de forma análoga al lenguaje Haskell [Pey02]. Con respecto a las funciones, la función `triangle` de tipo `cPoint -> real -> real -> region` define un triángulo de la siguiente manera: una llamada de la forma `triangle (RX0,RY0) B H` devuelve una función Booleana que representa la región de todos los puntos del plano interiores al triángulo isósceles cuyo vértice superior es el punto `(RX0,RY0)`, de base `B` y de altura `H`. Aplicando esta función a la tupla `(RX,RY)` se produce una llamada de función `triangle (RX0,RY0) B H (RX,RY)` que devuelve el valor `true` si el punto `(RX,RY)` cae dentro del triángulo isósceles cuyos vértices son `(RX0,RY0)`, `((RX0-B)/2,RY0-H)`, `((RX0+B)/2,RY0-H)`. Los tres lados del triángulo están determinados por las siguientes ecuaciones:

```
RY = RY0-H
B*RY-2*H*RX = B*RY0-2*H*RX0
B*RY+2*H*RX = B*RY0+2*H*RX0
```

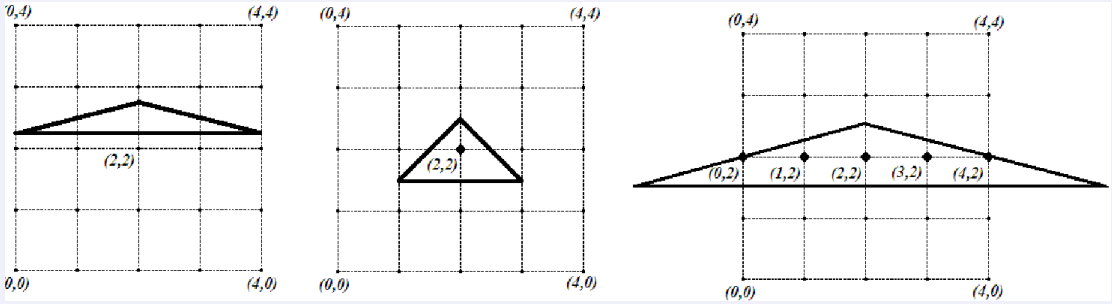
De forma similar se define la rejilla discreta con la función `square`. Una llamada de la forma `square N` devuelve una función Booleana que representa la rejilla de todos los valores discretos del plano con coordenadas definidas entre `0` y `N`. Así, la llamada `square N (X,Y)` devuelve `true` si el punto discreto `(X,Y)` es parte de la rejilla y `false` en caso contrario.

La función `isIn`, definida en la línea 8, aplica la función que recibe como primer argumento sobre su segundo argumento. Por ejemplo `isIn (square 3) (1,0)` aplica `square 3` sobre el punto discreto `(1,0)`.

La función `bothIn` en la línea 9 es la función principal de nuestro ejemplo y es la que determina qué puntos discretos corresponden a la intersección de la cuadrícula discreta y el triángulo continuo. Primero establece los puentes `X #== RX` e `Y #== RY` para asegurar que el punto discreto `(X,Y)` y el punto continuo `(RX,RY)` son equivalentes. Después determina

los puntos del plano que corresponden a la intersección de la región real (definida por la restricción `isIn Region (RX,RY)`) y la rejilla discreta (definida por la restricción `isIn Grid (X,Y)`).

Se pueden plantear diferentes objetivos para el programa que acabamos de detallar en el sistema $\mathcal{TOY}$. Para mostrar cómo procede el cálculo se toman dos valores enteros positivos fijos `d` y `n` tal que `n = 2*d`. Entonces `(d,d)` es el punto medio de la rejilla definida con `(square n)`, la cual incluye $(n+1)^2$ puntos discretos. A continuación se ilustran distintas posibilidades para el caso particular `n = 4` y `d = 2`.



Los tres objetivos siguientes representan los objetivos generales de las tres posibilidades que se muestran en la figura:

- **Objetivo 1**: `bothIn (triangle (d, d+0.75) n 0.5) (square n) (X,Y)`. Este objetivo falla pues no existe ningún punto discreto de la rejilla interior al triángulo.
- **Objetivo 2**: `bothIn (triangle (d, d+0.5) 2 1) (square n) (X,Y)`. Este objetivo calcula una solución para `(X,Y)`: el punto `(d,d)`.
- **Objetivo 3**: `bothIn (triangle (d, d+0.5) (2*n) 1) (square n) (X,Y)`. Este objetivo computa `n+1` soluciones para `(X,Y)`: los puntos `(0,d)`, `(1,d)`, `...`, `(n,d)`.

En los tres casos, la cooperación entre el resolutor $\mathcal{R}$ y el resolutor $\mathcal{FD}$ es crucial para la eficiencia del cómputo. Veámoslo tomando como ejemplo el segundo objetivo. Después de una serie de pasos del cálculo $CCLNC(\mathcal{C}_{\mathcal{FD}.\mathcal{R}})$ se llega a un sistema de restricciones similar al siguiente:

$$\begin{aligned} \mathcal{M}_{\mathcal{FD}.\mathcal{R}}: &\quad \texttt{X \#== RX, Y \#== RY} \\ \mathcal{R}: &\quad \texttt{RY >= d-0.5, RY-RX <= 0.5, RY+RX <= n+0.5} \\ \mathcal{FD}: &\quad \texttt{domain [X,Y] 0 n, labeling [] [X,Y]} \end{aligned}$$

El sistema $\mathcal{TOY}$ tiene la opción de habilitar y deshabilitar el cómputo de las proyecciones mediante el comando `/proj`. Cuando las proyecciones están deshabilitadas, la restricción `labeling [] [X,Y]` fuerza la enumeración de todos los posibles valores para `X` e `Y`. La única solución `X = Y = d` se encuentra después de $\mathcal{O}(n^2)$ pasos. Cuando

las proyecciones están habilitadas, los puentes se utilizan para proyectar las restricciones `RY >= d-0.5`, `RY-RX <= 0.5` y `RY+RX <= n+0.5` del dominio $\mathcal{R}$ al dominio $\mathcal{FD}$ creando nuevas restricciones: `Y #>= d`, `Y#-X #<= 0` y `Y#+X #<= n`. Por lo tanto, el resolutor $\mathcal{FD}$ poda drásticamente los dominios de `X` e `Y`, y así el etiquetado conduce a la única posible solución `X = Y = d`.

Para un valor elevado de `n = 2*d` el rendimiento del cómputo es mucho mayor en comparación con el caso donde las proyecciones están desactivadas, según lo confirmado por los resultados experimentales que se detallarán en la sección 5.4. El rendimiento es mayor porque de los $\mathcal{O}(\mathtt{n}^2)$ pasos necesarios para ejecutar la restricción `labeling [] [X,Y]` cuando los dominios de `X` e `Y` son de tamaño $\mathcal{O}(\mathtt{n})$, se pasa a ejecutar $\mathcal{O}(1)$ pasos para la misma restricción `labeling [] [X,Y]` pero con dominios de `X` e `Y` podados a un tamaño $\mathcal{O}(1)$. El primer objetivo se comporta una forma similar: encuentra el fallo pasando de una ejecución del orden $\mathcal{O}(\mathtt{n}^2)$ a $\mathcal{O}(1)$ cuando las proyecciones están habilitadas. En el tercer objetivo el tiempo de ejecución se reduce de $\mathcal{O}(\mathtt{n}^2)$ a $\mathcal{O}(\mathtt{n})$ cuando se activan las proyecciones.

Ahora se va a mostrar detalladamente el comportamiento del cálculo $CCLNC(\mathcal{C}_{\mathcal{FD}.\mathcal{R}})$ con respecto al segundo objetivo tomando `n=2*d` y `d'=d+0.5`. La estructura de un objetivo es de la forma $\exists\overline{U}.P \;\square\; C \;\square\; M \;\square\; H \;\square\; F \;\square\; R$, donde cada elemento M, H, F y R está formado por un conjunto de restricciones y una sustitución. En el resto del ejemplo no se muestran las sustituciones para simplificar la presentación. Supongamos el siguiente objetivo inicial compuesto por una restricción ubicada en el *pool* de restricciones:

$$\square\ \overbrace{\texttt{bothIn (triangle (d,d') 2 1) (square n) (X,Y) == true}}^{\pi_1}\ \square\square\square\square$$

La restricción π_1 se aplana dando lugar a una producción y a una nueva restricción en el *pool* de restricciones.

$$\vdash_{\mathbf{FC}} \exists\ \texttt{A. bothIn (triangle (d,d') 2 1) (square n) (X,Y)} \rightarrow \texttt{A}\ \square\ \overbrace{\texttt{A == true}}^{\pi_2}\ \square\square\square\square$$

La variable `A` no está obviamente demandada porque la restricción π_2 todavía no está en el almacén de Herbrand. Por lo tanto la producción se queda suspendida.

Se continúa procesando el *pool*, la única restricción que hay es primitiva atómica y no se pueden establecer puentes ni proyectar, así que π_2 se envía a su correspondiente resolutor.

$$\vdash_{\mathbf{SC(ii)}} \exists\ \texttt{A.}\ \overbrace{\texttt{bothIn (triangle (d,d') 2 1) (square n) (X,Y)} \rightarrow \texttt{A}}^{P_1}\ \square\ \square\square\ \texttt{A == true}\ \square\square$$

Ahora la variable `A` se convierte en una variable obviamente demandada y también está producida, por lo tanto no se puede invocar al resolutor de Herbrand. Se procesa la producción P_1 aplicando la regla **DF** de la tabla 4.1 a la función `bothIn`, lo que genera nuevas producciones y nuevas restricciones en el *pool*.

$$\vdash_{\mathbf{DF}} \exists \texttt{A,R,G,X',Y'.}\ \texttt{triangle (d,d') 2 1} \rightarrow \texttt{R},\ \texttt{square n} \rightarrow \texttt{G},\ \texttt{(X,Y)} \rightarrow \texttt{(X',Y')},\ \texttt{true} \rightarrow \texttt{A} \square \texttt{X' \#== RX, Y' \#== RY, isIn R (RX,RY) == true, isIn G (X',Y') == true, labeling [] [X',Y'] } \square\square \texttt{ A == true } \square\square$$

Las cuatro producciones resultantes se procesan aplicando las reglas **DC** y **SP** de la tabla 4.1, propagando vinculaciones y descomposiciones hasta que P se queda vacío. Entonces el resolutor de Herbrand se invoca mediante la regla **HS** de la tabla 4.2. En este punto, el almacén de Herbrand solo contiene una sustitución $\sigma_H = \{\ \texttt{A} \mapsto \texttt{true}\ \}$ resultante de los pasos previos. El correspondiente almacén es $(\emptyset \square \sigma_H)$, que denotaremos como H. El procesamiento de estas cuatro producciones es secuencial pero se agrupa para simplificar la exposición del ejemplo.

$$\vdash^*_{\mathbf{SP^2,DC,SP^3,HS}} \square \overbrace{\texttt{X \#== RX}}^{\pi_3}, \overbrace{\texttt{Y \#== RY}}^{\pi_4}, \texttt{isIn (triangle (d,d') 2 1) (RX,RY) == true, isIn (square n) (X,Y) == true, labeling [] [X,Y] } \square \square H \square \square$$

En este paso no se generan producciones nuevas y las siguientes dos restricciones a procesar del *pool* son dos puentes, π_3 y π_4, que se envían al almacén M y se invoca al resolutor del dominio mediador, que en este paso no tiene ninguna regla aplicable.

$$\vdash^*_{\mathbf{SC(i)^2,MS}} \square$$

$$\overbrace{\texttt{isIn (triangle (d,d') 2 1) (RX,RY) == true}}^{\pi_5}, \overbrace{\texttt{isIn (square n) (X,Y) == true}}^{\pi_6}, \texttt{labeling [] [X,Y] } \square \texttt{ X \#== RX, Y \#== RY } \square H \square \square$$

En este paso tampoco hay nuevas producciones así que se continúa con las siguientes restricciones del *pool*, π_5 y π_6, que son procesadas con pasos similares a los empleados en el procesamiento de la función `bothIn`. Este procesamiento produce nuevas restricciones y sustituciones, resultado de las aplicaciones de las funciones: `isIn`, `triangle` y `square`. Estas nuevas sustituciones, unidas a las anteriores, forman H'.

$$\vdash^* \square \overbrace{\texttt{RY>=d'-1}}^{\pi_7}, \texttt{2*RY-2*1*RX<=2*d'-2*1*d, 2*RY+2*1*RX<=2*d'+2*1*d, domain [X,Y] 0 n, labeling [] [X,Y] } \square \texttt{ X \#== RX, Y \#== RY } \square H' \square \square$$

No hay producciones. La siguiente restricción del *pool*, π_7, se aplana dando lugar a una nueva restricción primitiva `d'-1` $\rightarrow!$ `RA`.

$$\vdash^*_{\mathbf{FC,PC}} \square \overbrace{\texttt{d'-1} \rightarrow! \texttt{RA}}^{\pi_8}, \texttt{RY>=RA, 2*RY-2*1*RX<=2*d'-2*1*d, 2*RY+2*1*RX<=2*d'+2*1*d, domain [X,Y] 0 n, labeling [] [X,Y] } \square \texttt{ X \#== RX, Y \#== RY } \square H' \square \square$$

No hay producciones nuevas. La restricción `d'-1` $\rightarrow!$ `RA` se envía al almacén R y se invoca al resolutor correspondiente, que calcula `d''` como el valor numérico `d'-1` y propaga la sustitución `RA` $\mapsto$ `d''` a todo el objetivo, causando cambios internos en el almacén R.

$$\vdash^*_{\mathbf{SC(iv)},\mathbf{RS}} \square \overbrace{\texttt{RY>=d''}}^{\pi_9}, \texttt{2*RY-2*1*RX<=2*d'-2*1*d}, \texttt{2*RY+2*1*RX<=2*d'+2*1*d}, \texttt{domain [X,Y] 0 n}, \texttt{labeling [] [X,Y]} \square \texttt{X \#== RX}, \texttt{Y \#== RY} \square H' \square \square R$$

No hay producciones. La evaluación de la restricción π_9 no genera ningún puente, pues ya existe el puente `Y #== RY`, pero su proyección sí genera una nueva restricción `Y#>=d` que se envía al almacén F. Después se envía π_9 a su resolutor.

$$\vdash^* \square \overbrace{\texttt{2*RY-2*1*RX<=2*d'-2*1*d}}^{\pi_{10}}, \overbrace{\texttt{2*RY+2*1*RX<=2*d'+2*1*d}}^{\pi_{11}}, \texttt{domain [X,Y] 0 n}, \texttt{labeling [] [X,Y]} \square \texttt{X \#== RX}, \texttt{Y \#== RY} \square H' \square \texttt{Y\#>=d} \square \texttt{RY>=d''}, R$$

No hay producciones. Las restricciones atómicas π_{10} y π_{11} se aplanan convenientemente mediante las reglas **FC** y **PC**. Las restricciones primitivas atómicas resultantes del proceso de aplanamiento se ubican en el *pool* y se procesan aplicando las reglas **SB**, **PP** y **SC** para generar puentes y proyecciones si es el caso y posteriormente se envían a su correspondiente almacén. Obsérvese que en el siguiente estado del objetivo se muestran las restricciones de $\mathcal{FD}$ generadas por la proyección como `2#*Y#-2#*X` $\rightarrow!$ `B` y `2#*Y#+2#*X` $\rightarrow!$ `C` y las restricciones que se envían a $\mathcal{R}$ son `2*RY-2*RX` $\rightarrow!$ `RB` y `2*RY+2*RX` $\rightarrow!$ `RC`. Estas restricciones no se computan como aparecen en el objetivo pues primero se aplanan. Se ha elegido representar de esta forma al conjunto de restricciones primitivas atómicas para facilitar su legibilidad.

$$\vdash^* \exists \texttt{B,C}. \square \texttt{domain [X,Y] 0 n}, \texttt{labeling [] [X,Y]} \square \texttt{X \#== RX}, \texttt{Y \#== RY}, \texttt{B \#== RB}, \texttt{C \#== RC}, M \square H' \square \texttt{Y\#>=d}, \texttt{2\#*Y\#-2\#*X} \rightarrow! \texttt{B}, \texttt{B \#<= 1}, \texttt{2\#*Y\#+2\#*X} \rightarrow! \texttt{C}, \texttt{C \#<= n'}, F \square \texttt{RY>=d''}, \texttt{2*RY-2*RX} \rightarrow! \texttt{RB}, \texttt{RB <= 1}, \texttt{2*RY+2*RX} \rightarrow! \texttt{RC}, \texttt{RC <= n'}, R$$

No hay producciones, los almacenes no están en forma resuelta y sus variables no contienen variables producidas, por lo tanto, se invocan los correspondientes resolutores modificando el estado de sus almacenes:

$$\vdash^* \square \texttt{domain [d,d] 0 n}, \texttt{labeling [] [d,d]} \square M' \square H'' \square F' \square R'$$

Las dos restricciones que quedan en el *pool* se evalúan de forma similar a las restricciones ubicadas anteriormente en el *pool* y, una vez que se han enviado a su almacén, se invoca al correspondiente resolutor. El resolutor de $\mathcal{FD}$ etiqueta las variables `X` e `Y` y satisfacen las restricciones cuando ambas variables toman el valor `d`.

$$\vdash^*_{\mathbf{SC(iii)},\mathbf{FDS},\mathbf{SC(iii)},\mathbf{FDS}} \square\square\square\square (\Diamond \square \{\texttt{X} \mapsto \texttt{d}, \texttt{Y} \mapsto \texttt{d}\}) \square$$

Estas sustituciones se muestran como respuesta del objetivo.

En el ejemplo anterior se ha mostrado el beneficio que se obtiene cuando el resolutor de $\mathcal{FD}$ coopera con el dominio $\mathcal{R}$. Ahora se verá un ejemplo en el cual es el resolutor de $\mathcal{R}$ quien coopera con el resolutor $\mathcal{FD}$ para obtener una solución. El siguiente ejemplo es continuación

del ejemplo 2 de la sección 1.1.

Ejemplo 6. (Intersección de diagonal y parábola)

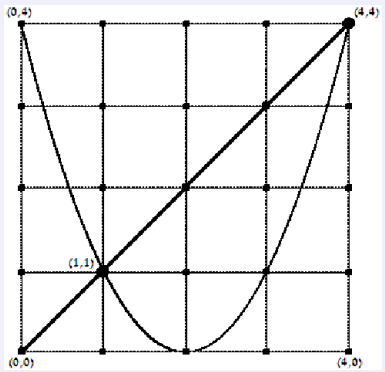
En este ejemplo se estudian los puntos discretos que caen en la intersección de un segmento y una parábola. En este caso no se detallarán las reglas del cálculo, sino que se mostrará el mecanismo de cooperación a un nivel más alto. El código $\mathcal{TOY}$ que se va a utilizar en este ejemplo es el mismo que en el ejemplo anterior añadiendo las funciones `diagonal` y `parabola`.

```
1  diagonal :: int -> grid
   diagonal N (X,Y) :- domain [X,Y] 0 N, X == Y

2  parabola :: cPoint -> region
   parabola (RX0,RY0) (RX,RY) :- RY-RY0 == (RX-RX0) * (RX-RX0)
```

Las declaraciones de tipo y cláusulas para las funciones `diagonal` y `parabola` son semejantes a `triangle` y `square`. Una llamada de la forma `diagonal N` devuelve una función Booleana que representa la diagonal de la cuadrícula de dimensión `N+1`, mientras `parabola (RX0,RY0)` devuelve una función booleana que representa la parábola cuya ecuación es `RY-RY0 = (RX-RX0) * (RX-RX0)`.

El objetivo `bothIn (parabola (2,0)) (diagonal 4) (X,Y)` busca los puntos de intersección del segmento diagonal discreto de tamaño 4 y la parábola de vértice `(2,0)` como se muestra a continuación:



Este objetivo se reduce al siguiente sistema de restricciones simplificado:

$\mathcal{M}_{\mathcal{FD}.\mathcal{R}}$: `X #== RX, Y #== RY`
$\mathcal{R}$: `RY == (RX-2)*(RX-2)`
$\mathcal{FD}$: `domain [X,Y] 0 4, X == Y, labeling [] [X,Y]`

En este ejemplo, los dos primeros puentes se procesan sin causar efecto. La siguiente restricción, `domain [X,Y] 0 4`, proyecta al dominio $\mathcal{R}$ nuevas restricciones primitivas atómicas, `0 <= RX, RX <= 4, 0 <= RY, RY <= 4`. La restricción no lineal `RY ==`

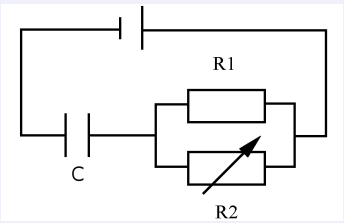
`(RX-2)*(RX-2)` es aplanada y sus correspondientes primitivas atómicas son enviadas al resolutor de $\mathcal{R}$, donde se quedan suspendidas. Estas restricciones primitivas atómicas del dominio $\mathcal{R}$ se proyectan a $\mathcal{FD}$ produciendo nuevas restricciones primitivas atómicas que representan la restricción `Y == (X#-2)#*(X#-2)`. La igualdad `X == Y` sustituye la variable `Y` por `X` en todo el objetivo. Así, los puentes `X #== RX, Y #== RY` se convierten en `X #== RX, X #== RY`, y el resolutor del dominio mediador infiere la igualdad `RX == RY` aplicando la regla **IE** de la tabla 5.5. Esta igualdad se resuelve sustituyendo `RY` por `RX` en todo el objetivo. En este punto, las primitivas atómicas suspendidas en el resolutor de $\mathcal{R}$ representan a la restricción no lineal `RX == (RX-2)*(RX-2)` que todavía no puede ser resuelta.

Finalmente, el etiquetado enumera todos los posibles valores de `X`. Debido al puente `X #== RX`, cada valor entero `v` asignado a `X` causa que la variable `RX` se vincule al número real de valor entero `rv` equivalente a `v`. La vinculación de `RX` a `rv` despierta la restricción `RX == (RX-2)*(RX-2)`, que se convierte en `rv == (rv-2)*(rv-2)` y tiene éxito si `rv` es una solución de valor entero para la ecuación cuadrática. Es decir, el resolutor de $\mathcal{R}$ se aprovecha del etiquetado del resolutor de $\mathcal{FD}$ para comprobar la satisfactibilidad de su almacén.

Por último, se muestra a continuación un ejemplo práctico de cooperación entre $\mathcal{FD}$ y $\mathcal{R}$ tomado del trabajo [Hof00a]. Este problema es interesante porque es necesaria la cooperación entre ambos dominios y no puede ser resuelto utilizando únicamente restricciones de un dominio.

Ejemplo 7. (Circuito eléctrico)

Se dispone de un circuito eléctrico como el mostrado en la figura siguiente, con una resistencia R1 de 0.1 $M\Omega$ conectada en paralelo con una resistencia variable R2 que varía entre 0.1 $M\Omega$ y 0.4 $M\Omega$. Además, se dispone de un condensador C conectado en serie con las dos resistencias. El intervalo de tiempo t de carga el condensador está comprendido entre 0.5 y 1 segundo. Suponiendo que hay condensadores disponibles de $1\mu F$, $2.5\mu F$, $5\mu F$, $10\mu F$, $20\mu F$, y $50\mu F$, se pide seleccionar aquellos condensadores que se ajustan al circuito de tal forma que el voltaje alcance el 99 % del voltaje final en un tiempo determinado.



En este circuito, la evaluación de la diferencia de potencial (voltaje) entre los bornes del condensador viene determinada por la siguiente expresión: $V_c = V_f +$

$(V_o - V_f)\, e^{-t/(R*C)}$ donde V_o, V_c y V_f representan el voltaje inicial, actual y final, respectivamente y C es la capacidad del condensador medida en Faradios. R representa el circuito en paralelo de las dos resistencias, $R = (R1 * R2)/(R1 + R2)$.

Como se desea restringir el tiempo que tarda el voltaje actual V_c en ser el 99 % del voltaje final V_f, se tiene $99/100 * V_f = V_f + (0 - V_f)\, e^{-t/(R*C)}$ pues el voltaje inicial es nulo.

Simplificando se obtiene $99/100 = 1 - e^{-t/(R*C)}$, es decir, $1/100 = e^{-t/(R*C)}$ y despejando t se deduce:

$$t = -R * C * ln(0.01) \qquad [1]$$

En este problema se nos pide que elijamos un condensador entre un conjunto de condensadores disponibles. Además, este condensador debe cumplir una serie de restricciones. Por lo tanto, se puede definir el condensador como una variable $\mathcal{FD}$, que se etiqueta y por cada valor que tome se comprueba si se cumplen las restricciones anteriormente expuestas. Estas restricciones son reales, por lo tanto hay que comunicar el dominio $\mathcal{FD}$ con el dominio $\mathcal{R}$ a través de un puente para el condensador `(KI #== K)`. Pero, entre los posibles valores disponibles para los condensadores se tiene un valor no entero $2.5\mu F$. Para convertirlo a un valor entero se multiplica por 10 a todos los elementos, es decir, las resistencias y el condensador. Como `C` se expresa en Faradios, la ecuación [1] debe multiplicarse por 10^{-7}. A continuación se muestra el correspondiente programa $\mathcal{TOY}$ que resuelve este problema:

```
     ecircuit ::  int
     ecircuit = KI <==
1      R1 == 100000,                          % Restric. real
2      R2 >= 100000, R2 <= 400000,            % Restric. real
3      R == R1*R2/(R1+R2),                    % Restric. real
4      R >= 50000.0, R <= 80000.0,            % Restric. real
5      T == -(ln 0.01)*R*K/10000000.0,        % Restric. real
6      T >= 0.5, T <= 1.0,                    % Restric. real
7      belongs KI [10,25,50,100,200,500],     % Restric. FD
8      KI #== K,                              % Restric. puente
9      labeling [] [KI]                       % Restric. FD
```

Como se ha comentado, para seleccionar distintos condensadores se utiliza una variable entera `KI` que representa la parte entera de `K` (línea 8). El dominio de la variable `KI` son valores enteros asignados mediante la restricción `belongs`. La variable `KI` va tomando cada valor mediante el etiquetado `(labeling)`, que sigue la estrategia que tiene el sistema por defecto para asignar valores a la variable. Por cada valor se verifica si se satisfacen las restricciones reales relacionadas con las resistencias. Aunque `K` tome un valor concreto, el resolutor no es capaz de resolver la restricción 5. Para que el resolutor de reales pueda resolver estas restricciones suspendidas necesita más información, por lo que se introducen las restricciones redundantes de la línea 4 cuyos datos se obtienen de las restricciones anteriores (líneas 1-3).

Finalmente, si se procesa el objetivo `ecircuit == L` se obtiene como respuesta la variable `L` vinculada al valor 25 y no se encuentran más soluciones por el mecanismo de vuelta atrás.

Nota: la implementación de este problema que está contenida en la distribución de $\mathcal{TOY}$ existe un parámetro de entrada (por ejemplo `ecircuit 0.000001 == L`) que corresponde a un intervalo de tolerancia porque los resolutores de números reales por naturaleza acumulan imprecisiones.

5.3 Resultados de corrección y completitud limitada

En esta sección se van a presentar los resultados semánticos de corrección y completitud limitada para el cálculo que se acaba de describir con respecto a la lógica de reescritura $CRWL(\mathcal{D})$ [LRV07] que proporciona la semántica declarativa para los programas $CFLP(\mathcal{C}_{\mathcal{FD}.\mathcal{FS}})$.

A lo largo del cálculo, la condición de corrección exige que no se introduzca ninguna solución que no sea solución del objetivo original, mientras que la condición de completitud requiere que ninguna solución bien tipada del objetivo original se pierda. La corrección asegura que las formas resueltas obtenidas como respuestas computadas para un objetivo inicial, usando las reglas del cálculo de resolución de objetivos cooperativo, son respuestas semánticas válidas de dicho objetivo. Como ya se ha comentado anteriormente, la completitud se puede perder por varias causas: si existen apariciones libres de variables lógicas de orden superior, si se hacen llamadas incompletas a un resolutor o si se producen descomposiciones opacas que dan lugar a subobjetivos que están mal tipados.

En primer lugar se va a establecer la corrección y completitud limitada de forma local, es decir, para un paso de transformación. En el siguiente teorema la notación $G \vdash_{\mathbf{RL},\gamma,\mathcal{P}} G'_j$ indica que el objetivo admisible G de un programa $\mathcal{P}$ se transforma en un nuevo objetivo G' aplicando la regla **RL** a una parte seleccionada γ de G. En el lema de progreso se utiliza la notación $G \vdash^{+}_{\mathbf{RL},\gamma,\mathcal{P}} G'$ que indica la existencia de alguna computación de la forma $G \vdash_{\mathbf{RL},\gamma,\mathcal{P}} G_1 \vdash^{*}_{\mathcal{P}} G'$ que transforma G en G' en n pasos con $n \geq 1$.

Teorema 7. (Corrección local y completitud local limitada del cálculo $CCLNC(\mathcal{C}_{\mathcal{FD}.\mathcal{R}})$)

Sea $\mathcal{P}$ un programa $CFLP(\mathcal{C}_{\mathcal{FD}.\mathcal{R}})$ y G un objetivo admisible para $\mathcal{P}$ que no está en forma resuelta. Se elige una regla **RL** de las tablas 4.1, 4.2, 5.2, 5.5 y 5.6 aplicable a G y se selecciona una parte γ de G sobre la que se aplica la regla **RL**. Entonces existe un número finito de transformaciones $G \vdash_{\mathbf{RL},\gamma,\mathcal{P}} G'_j$ $(1 \leq j \leq k)$, y además:

1. **Corrección local:** $Sol_{\mathcal{P}}(G) \supseteq \bigcup_{j=1}^{k} Sol_{\mathcal{P}}(G'_j)$.
2. **Completitud local limitada:** $WTSol_{\mathcal{P}}(G) \subseteq \bigcup_{j=1}^{k} WTSol_{\mathcal{P}}(G'_j)$, donde la aplicación de la regla **RL** a la parte seleccionada γ de G es *segura* en el siguiente sentido: no es una aplicación opaca de **DC** ni es una aplicación de una regla de las tablas 4.2 o 5.6 que implique una invocación incompleta al resolutor.

En el apéndice A.7.1 se encuentra la demostración del teorema 7.

La corrección global del cálculo $CCLNC(\mathcal{C}_{\mathcal{FD}.\mathcal{R}})$ se establece a partir de la corrección local del teorema 7 y asegura que las formas resueltas obtenidas como respuestas calculadas para un objetivo inicial, usando las reglas del cálculo de resolución de objetivos cooperativo $CCLNC(\mathcal{C}_{\mathcal{FD}.\mathcal{R}})$, son respuestas semánticas válidas para el objetivo G.

Teorema 8. (Corrección del cálculo $CCLNC(\mathcal{C}_{\mathcal{FD}.\mathcal{R}})$)

Sea $\mathcal{P}$ un programa $CFLP(\mathcal{C}_{\mathcal{FD}.\mathcal{R}})$, G un objetivo admisible para $\mathcal{P}$, y S un objetivo resuelto tal que $G \vdash\!\!\!\vdash^*_{\mathcal{P}} S$ utilizando las reglas de las tablas 4.1, 4.2, 5.2, 5.5 y 5.6. Entonces, $Sol_{\mathcal{C}_{\mathcal{FD}.\mathcal{R}}}(S) \subseteq Sol_{\mathcal{P}}(G)$.

Demostración del teorema 8

A partir del primer punto del teorema 7 se obtiene $Sol_{\mathcal{P}}(G') \subseteq Sol_{\mathcal{P}}(G)$ para cualquier G' tal que $G \vdash\!\!\!\vdash_{\mathcal{P}} G'$. Aplicando inducción se obtiene que $Sol_{\mathcal{P}}(S) \subseteq Sol_{\mathcal{P}}(G)$ para cada forma resuelta S tal que $G \vdash\!\!\!\vdash^*_{\mathcal{P}} S$. Además, como $Sol_{\mathcal{P}}(S) = Sol_{\mathcal{C}_{\mathcal{FD}.\mathcal{R}}}(S)$, la corrección queda demostrada. *c.q.d.*

La completitud limitada global para el cálculo $CCLNC(\mathcal{C}_{\mathcal{FD}.\mathcal{R}})$ no se puede deducir inmediatamente del punto 2 del teorema 7, pues necesita asegurar que, dado $\mu \in WTSol_{\mathcal{P}}(G)$, el cómputo $CCLNC(\mathcal{C}_{\mathcal{FD}.\mathcal{R}})$ acaba en una forma resuelta S tal que $\mu \in WTSol_{\mathcal{C}_{\mathcal{FD}.\mathcal{R}}}(S)$, es decir, es terminante. Para demostrarlo se va a utilizar una técnica detallada en el trabajo [BN98] que utiliza un orden de progreso bien fundado. Otros trabajos que utilizan esta técnica son [GHLR99, GHR01, LRV04, EFH$^+$08].

El *orden de progreso bien fundado para el cálculo* $CCLNC(\mathcal{C}_{\mathcal{FD}.\mathcal{FS}})$ se denota $\rhd$ y se establece entre pares $(G, \mathcal{M})$ formados por un objetivo admisible G sin apariciones de variables de orden superior y un testigo $\mathcal{M}$ para $\mu \in Sol_{\mathcal{P}}(G)$ que es un multiconjunto $\{\mathcal{T}_1, \ldots, \mathcal{T}_n\}$. Dado tal par, se define una 7-tupla $||(G, \mathcal{M})|| =_{def} (O_1, O_2, O_3, O_4, O_5, O_6, O_7)$, donde O_1 es un multiconjunto finito de números naturales y O_2, ..., O_7 son números naturales que se obtienen de la siguiente forma:

O$_1$ es el *testigo de tamaño restringido* $\mathcal{M}$, definido como el multiconjunto de números naturales $\{| \mathcal{T}_1 |, \ldots, | \mathcal{T}_n |\}$, donde $| \mathcal{T}_i |$ $(1 \leq i \leq n)$ denota el *tamaño restringido* del árbol de prueba $\mathcal{T}_i$ de la lógica de reescritura $CRWL(\mathcal{C})$, como se define en [LRV07], es decir, como el número de nodos en $\mathcal{T}_i$ correspondientes a los pasos de inferencia $CRWL(\mathcal{C})$ que dependen del significado de las funciones primitivas p (según se interpreta en el dominio de coordinación $\mathcal{C}$) más el número de nodos en $\mathcal{T}_i$ correspondientes a los pasos de inferencia $CRWL(\mathcal{C})$ que dependen del significado de las funciones definidas por el usuario f (de acuerdo con el programa actual $\mathcal{P}$).

$\mathbf{O}_2$ es la suma de $||p\,\overline{e}_n||$, donde $p\,\overline{e}_n$ son aplicaciones totales de funciones primitivas $p \in PF^n$ que aparecen en P y C del objetivo G, y $||p\,\overline{e}_n||$ se define como el número de argumentos de $e_i\,(1 \leq i \leq n)$ que no son patrones.

$\mathbf{O}_3$ es el número de expresiones rígidas y pasivas $h\,\overline{e}_n$ que no son los patrones en las producciones P del objetivo G.

$\mathbf{O}_4$ es la suma del tamaño sintáctico de lados derechos de todas las producciones de P.

$\mathbf{O}_5$ es la suma $sf_M + sf_H + sf_F + sf_R$ donde sf_M es el valor 1 si la regla **MS** que invoca al resolutor de la tabla 4.2 se puede aplicar a G, y 0 en otro caso. Los otros tres indicadores se definen de forma análoga para los demás dominios, utilizando además la tabla 5.6.

$\mathbf{O}_6$ es el número de puentes del almacén mediador M de G.

$\mathbf{O}_7$ es el número de antipuentes del almacén mediador M de G.

Sea $>_{lex}$ el producto lexicográfico de los 7 órdenes $>_i$ $(1 \leq i \leq 7)$, donde $>_1$ es el orden del multiconjunto $>_{mul}$ sobre los multiconjuntos de los números naturales, y $>_i$ el orden ordinario $>$ sobre los números naturales para $2 \leq i \leq 7$. Finalmente, se define el orden de progreso $\rhd$ por la condición $(G, \mathcal{M}) \rhd (G', \mathcal{M}')$ si y solo si $||(G, \mathcal{M})|| >_{lex} ||(G', \mathcal{M}')||$. Como se demostró en [BN98], $>_{mul}$ es un orden bien fundado y el producto lexicográfico de órdenes bien fundados es un orden bien fundado. Por lo tanto, $\rhd$ es un orden bien fundado.

Lema 4. (Lema de progreso del cálculo $CCLNC(\mathcal{C}_{\mathcal{FD}.\mathcal{R}})$)

Sea G un objetivo admisible que no está en forma resuelta para un programa $\mathcal{P}$ y un testigo $\mathcal{M} : \mu \in WTSol_{\mathcal{P}}(G)$. Se asume que ni $\mathcal{P}$ ni G tienen variables libres de orden superior y que G no está en forma resuelta. Entonces:

1. Existe alguna regla **RL** aplicable a G que no es una regla de fallo.
2. Para cualquier elección de una regla **RL** que no sea una regla de fallo y una parte γ de G, tal que **RL** se aplica a γ de una manera segura, existe un número finito de computaciones $G \Vdash^{+}_{\mathbf{RL},\gamma,\mathcal{P}} G'$ tal que:
   - $\mu \in WTSol_{\mathcal{P}}(G')$.
   - Existe un testigo $\mathcal{M}' : \mu \in WTSol_{\mathcal{P}}(G')$ que cumple $(G, \mathcal{M}) \rhd (G', \mathcal{M}')$.

La demostración del lema 4 se encuentra en el apéndice A.7.2. Utilizando el lema de progreso, se puede demostrar el siguiente teorema.

Teorema 9. (Completitud limitada del cálculo $CCLNC(\mathcal{C}_{\mathcal{FD}.\mathcal{R}})$))

Sea G un objetivo para un programa $\mathcal{P}$ y $\mu \in WTSol_{\mathcal{P}}(G)$ una solución bien tipada para dicho objetivo. Asumimos que ni $\mathcal{P}$ ni G contienen apariciones libres de variables de orden superior y las aplicaciones de las reglas de las tablas 4.1, 4.2, 5.2, 5.5 y 5.6 son seguras, es decir, no producen descomposiciones opacas ni invocaciones incompletas de resolutores. Entonces se puede encontrar un cómputo $CCLNC(\mathcal{C}_{\mathcal{FD}.\mathcal{R}})$ de la forma $G \Vdash^* S$, terminando con un objetivo en forma resuelta S tal que $\mu \in WTSol_{\mathcal{C}_{\mathcal{FD}.\mathcal{R}}}(S)$; es decir, $WTSol_{\mathcal{P}}(G) \subseteq WTSol_{\mathcal{C}_{\mathcal{FD}.\mathcal{R}}}(S)$.

Demostración del teorema 9

Como $\mu \in WTSol_{\mathcal{P}}(S)$ es equivalente a $\mu \in WTSol_{\mathcal{C}_{\mathcal{FD}.\mathcal{R}}}(S)$, entonces para demostrar la completitud se puede demostrar $WTSol_{\mathcal{P}}(G) \subseteq WTSol_{\mathcal{P}}(S)$.

Por hipótesis se tiene un objetivo admisible G para un programa $\mathcal{P}$ y una solución $\mu \in WTSol_{\mathcal{P}}(G)$. Se puede tomar un testigo $\mathcal{M} : \mu \in WTSol_{\mathcal{P}}(G)$ y se razona por inducción sobre el orden bien fundado $\rhd$.

Si G está en forma resuelta entonces se toma S como G y se cumple la condición trivialmente. Si G no está en forma resuelta entonces se aplica el lema de progreso 4 a $\mathcal{P}$ y $\mathcal{M} : \mu \in WTSol_{\mathcal{P}}(G)$ y se obtiene una regla **RL** y una parte γ de G tal que **RL** se puede aplicar a γ. Suponiendo que esta aplicación es segura, el lema de progreso 4 también proporciona un cálculo finito $G \Vdash^+_{\mathbf{RL},\gamma,\mathcal{P}} G'$ de tal manera que hay un testigo $\mathcal{M}' : \mu \in WTSol_{\mathcal{P}}(G')$ cumpliendo $(G, \mathcal{M}) \rhd (G', \mathcal{M}')$. Dado que ni $\mathcal{P}$ ni G contienen apariciones libres de variables de orden superior, entonces se cumple lo mismo para G'. Por hipótesis de inducción del orden bien fundado, se concluye que salvo la imposibilidad de poder realizar algún paso de transformación seguro, se puede encontrar un cómputo $G' \Vdash^*_{\mathcal{P}} S$ con S en forma resuelta tal que $\mu \in WTSol_{\mathcal{P}}(S)$. El cómputo deseado es entonces $G \Vdash^+_{\mathbf{RL},\gamma,\mathcal{P}} G' \Vdash^*_{\mathcal{P}} S$. *c.q.d.*

5.4 Implementación

El sistema $\mathcal{TOY}$ [ACE+07] está implementado en SICStus Prolog [SIC11] y se distribuye como software libre de código abierto. Se puede ejecutar sobre varias plataformas, tanto en un terminal de texto (consola) como utilizando un entorno de desarrollo integrado multiplataforma y configurable llamado ACIDE [ACI14].

Para explicar la implementación que se ha realizado para extender el sistema $\mathcal{TOY}$ con la cooperación de los dominios $\mathcal{FD}$ y $\mathcal{R}$ primero se va a describir la arquitectura del sistema $\mathcal{TOY}$ enfocada a la cooperación de estos dominios. Posteriormente se dará una visión general del proceso de compilación de un programa $\mathcal{TOY}$ y de ejecución de objetivos. Finalmente se expone la implementación de los principales mecanismos de cooperación proporcionados por el cálculo $CCLNC(\mathcal{C}_{\mathcal{FD}.\mathcal{R}})$, es decir, los puentes y las proyecciones. Los siguientes trabajos [EFS06, EFS07, EFS08, ACE+07, EFH+09] están relacionados con el desarrollo de la implementación de esta extensión.

5.4.1 Arquitectura del sistema

La figura 5.1 muestra los componentes arquitectónicos del esquema de cooperación $\mathcal{C}_{\mathcal{FD}.\mathcal{R}}$. Está formado por tres resolutores de dominios de restricciones puros $\mathcal{H}$, $\mathcal{R}$ y $\mathcal{FD}$ combinados con el resolutor del dominio mediador $\mathcal{M}_{\mathcal{FD}.\mathcal{R}}$. Los resolutores de caja transparente $\mathcal{H}$ y $\mathcal{M}_{\mathcal{FD}.\mathcal{R}}$ y sus correspondientes almacenes de restricciones están implementados en el código de $\mathcal{TOY}$. Por otra parte y como ya se adelantó en el capítulo 3, el resolutor de $\mathcal{R}$ es una caja negra y parte del resolutor de $\mathcal{FD}$ también. Estas cajas negras corresponden a los módulos `clpr` y `clpfd` de la biblioteca de SICStus Prolog 3.12.

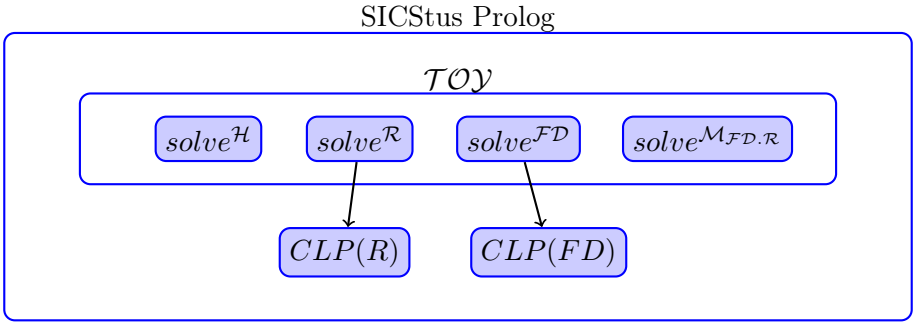


Figura 5.1: Componentes arquitectónicos del esquema de cooperación $\mathcal{C}_{\mathcal{FD}.\mathcal{R}}$ en $\mathcal{TOY}$

5.4.2 Compilación de un programa $\mathcal{TOY}$

El trabajo [SH98] expone una visión general del proceso de compilación de programas y ejecución de objetivos en $\mathcal{TOY}$. En esta sección se explica brevemente este proceso para situar la implementación de las primitivas que forman parte de la cooperación de los dominios $\mathcal{FD}$ y $\mathcal{R}$. Dicha implementación se muestra en las siguientes subsecciones.

En esencia, la compilación de un programa `program.toy` [EFS06] realiza los siguientes pasos: en primer lugar se une el programa definido por el usuario `program.toy` con los elementos primitivos del sistema declarados en `basic.toy` dando lugar al fichero `program.tmp.toy` (ver figura 5.2). Después, se lleva a cabo una primera fase de la compilación que realiza un análisis léxico y sintáctico dando lugar al fichero temporal `program.tmp.out`. Además, en esta fase se construye el *grafo de dependencias entre funciones* que se utiliza posteriormente en la inferencia de tipos. A partir este código intermedio se infieren los tipos no declarados, se verifican los tipos declarados y se genera un código Prolog que implementa la resolución de objetivos por medio del estrechamiento perezoso guiado por árboles definicionales, lo que asegura un comportamiento óptimo de estrechamiento perezoso [LLR93, AEH94, AEH00, Vad03, Vad05, Vad07, dVV08]. Este código objeto se encuentra en el fichero `program.pl`. El proceso que se acaba de describir se realiza desde $\mathcal{TOY}$ mediante el comando `/compile(program)`.

El programa generado se debe cargar en el sistema junto con otros predicados para que se puedan resolver objetivos. En particular, si los resolutores de restricciones de $\mathcal{FD}$ y $\mathcal{R}$ están activados, se deben cargar los correspondientes predicados que tratan las primitivas

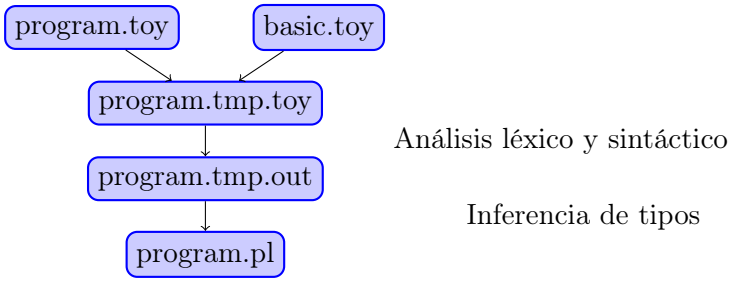


Figura 5.2: Flujo de datos de la compilación del fichero `program.toy`

específicas de estos dominios, como se mostrará en las subsecciones 5.4.5 y 5.4.6. Para poder establecer la comunicación entre los dominios $\mathcal{FD}$ y $\mathcal{R}$ se debe cargar el predicado que trata la restricción puente `#==`. La carga en el sistema del programa fuente compilado y de todos los predicados necesarios según sea la configuración del sistema se realiza mediante el comando `/load(program)`. Sin embargo, es común usar el comando `/run(program)` que realiza las tareas de compilación y carga de forma secuencial.

5.4.3 Implementación de primitivas

La metodología que se ha seguido para realizar la implementación de una función primitiva $f\,\bar{t}_n \to r$ consiste en definir un predicado Prolog $\texttt{f}(t_1, \ldots, t_n, r, Cin, Cout)$ con los siguientes argumentos: los parámetros de la primitiva f, el resultado r y dos argumentos más, uno de entrada y otro de salida, que representan un almacén que contiene información sobre las restricciones de desigualdad de Herbrand, los puentes y las restricciones de totalidad. A este almacén se le denomina a partir de este punto *el almacén mixto de* $\mathcal{TOY}$. En este predicado `f`, en primer lugar se calcula la forma normal de cabeza (hnf, head normal form) de los argumentos de la función f. Una *forma normal de cabeza* es cualquier expresión que no es una llamada a función, es decir, una variable o una expresión que comienza por constructora (aunque sus argumentos pueden contener llamadas a funciones). Así, el predicado Prolog `hnf(E,H,Cin,Cout)` devuelve en su segundo argumento la forma normal de cabeza de su primer argumento teniendo en cuenta las restricciones del almacén mixto. Un estudio detallado de este predicado se encuentra en el trabajo [SH98], aquí solo se va a motivar a continuación el uso de este predicado.

La aplicación del predicado `hnf` se realiza sobre una expresión y de forma anidada sobre los argumentos de dicha expresión hasta que se llega a una forma normal (que no se puede reducir más). Es decir, la forma normal de una expresión `E` se consigue mediante una llamada a `hnf` utilizando como primer argumento la expresión `E`. Posteriormente se evalúan los argumentos de la expresión `E` mediante llamadas al predicado `hnf`, y así sucesivamente se van evaluando a los argumentos más anidados. Estas llamadas se realizan solamente cuando son necesarias por el cómputo perezoso.

Para ilustrar el uso del predicado `hnf` se va a utilizar el ejemplo 3 de la sección 4.2 que muestra el proceso de aplanamiento de la restricción atómica real `(RX+2*RY)*RZ <= 3.5`. La implementación de este proceso de aplanamiento lo realizan las distintas llamadas al

predicado `hnf` que se encuentran en las implementaciones de los predicados `'<='`, `'*'` y `'+'`. Para simplificar la explicación vamos a suponer que la implementación de las primitivas `'<='`, `'*'` y `'+'` únicamente realizan el cálculo de las formas normales de cabeza y seguidamente envían la restricción a su correspondiente resolutor, **sin** tratar la cooperación. De esta forma la implementación simplificada de la primitiva `'<='` sería de la forma:

```
'<='(L,R,Out,Cin,Cout) :-
     hnf(L,HL,Cin,Cout1),
     hnf(R,HR,Cout1,Cout),
     (Out=true,HL=<HR;Out=false,HL>HR).
```

El cómputo del objetivo `(RX+2*RY)*RZ <= 3.5` se realiza mediante una llamada al predicado `'<='((RX+2*RY)*RZ,3.5,true,Cin,Cout)` y sigue la secuencia de llamadas a `hnf` que se muestra a continuación donde aparecen las llamadas más relevantes sin mostrar el tratamiento del almacén mixto.

```
hnf((RX+2*RY)*RZ,RA)
          hnf((RX+2*RY),RB)
                    hnf(RX,RX)
                    hnf(2*RY,RC)
                              hnf(2,2)
                              hnf(RY,RY)
          hnf(RZ,RZ)
hnf(3.5,3.5)
```

Aunque no se muestre, los argumentos que contienen llamadas a funciones se han tratado convenientemente mediante suspensiones. Estas suspensiones corresponden con las producciones suspendidas del ejemplo 3 de variables producidas `RA`, `RB` y `RC`.

El procesamiento de la primitiva `'<='` calcula las formas normales de cabeza de sus dos argumentos `(RX+2*RY)*RZ` y `3.5`. El primer argumento es un producto que todavía no se puede calcular y en el cálculo $CCLNC(\mathcal{C}_{\mathcal{FD}.\mathcal{R}})$ se representa mediante la producción `(RX+2*RY)*RZ`→`RA` donde `RA` es una variable nueva. Como muestra el ejemplo 3, esta producción pasa a ser una restricción a evaluar. Por lo tanto el siguiente paso es evaluar la restricción producto de argumentos `RX+2*RY` y `RZ` y resultado `RA`. Como en el paso anterior, el primer argumento del predicado `'*'(RX+2*RY,RZ,RA,true,Cin,Cout)` es una llamada a una función que todavía no se puede resolver y se representa mediante la producción `(RX+2*RY)`→`RB` donde `RB` es una variable nueva. Igual que antes, esta producción se convierte en una restricción suma de argumentos `RX` y `2*RY` y resultado `RB`, donde su segundo argumento se convierte en la producción `2*RY`→`RC`. Esta producción pasa a ser una restricción producto cuyos argumentos ya están en forma normal de cabeza y se envía al resolutor $\mathcal{R}$. Según van finalizando las llamadas se continúan procesando los predicados que envían las restricciones al resolutor $\mathcal{R}$ y finalmente se obtiene la respuesta: `RA<=3.5, RA==(RB*RZ), RB==RC+RX, RC==2.0*RY`.

Una vez vista la forma más simple que puede tomar la implementación de una primitiva, se va a mostrar la implementación de algunas de las restricciones primitivas involucradas en la cooperación. En concreto, en las siguientes secciones se describe la implementación del puente (`#==`) y de dos primitivas que proyectan restricciones entre dominios, una del dominio

$\mathcal{FD}$ (`#<`) y otra del dominio $\mathcal{R}$ (`>`). En las implementaciones que se van a mostrar se ha simplificado el código para facilitar su presentación. El código completo está disponible en los ficheros `solverCom.pl`, `cooperation.pl`, `cflpfdfile.pl` y `primitivCodClpr.pl` de la distribución de $\mathcal{TOY}$.

5.4.4 La primitiva `#==`

La función primitiva `#==` es la pieza principal de la comunicación entre los dominios $\mathcal{FD}$ y $\mathcal{R}$. La figura 5.3 muestra la implementación de esta primitiva cuyo comportamiento está especificado por las reglas del sistema de transformación de almacenes que definen al resolutor del dominio $\mathcal{M}_{\mathcal{FD}.\mathcal{R}}$ en la tabla 5.1.

```
1  '#=='(L, R, Out, Cin, Cout):-
2    hnf(L, HL, Cin, Cout1),
3    hnf(R, HRaux, Cout1, Cout2),
4    HL in inf..sup,
5    (number(HRaux) ->
         HR is float(HRaux)
     ;
         HR = HRaux
     ),
6    postM(HL, "#==", HR, Out, true, Cout2, Cout).

7  postM(L, Op, R, Out, true, Cin, Cout) :-
8    specializeFDHerbrand(L, true, Cin, Cout1),
9    specializeRHerbrand(R, true, Cout1, Cout2),
10   storeM(L, Op, R, Out, Cout2, Cout).
```

Figura 5.3: Implementación del puente (`#==`)

Siguiendo el esquema general, esta restricción primitiva está definida como un predicado Prolog con cinco argumentos. Los dos primeros argumentos `L` y `R` representan los parámetros izquierdo (entero) y derecho (real) del símbolo `#==`. El argumento `Out` es el resultado Booleano que reifica la restricción, y los argumentos `Cin` y `Cout` representan el almacén mixto antes y después de la ejecución de la primitiva, respectivamente. En primer lugar, las líneas 2 y 3 computan las formas normales de cabeza de `L` y `R` obteniendo `HL` y `HR`, respectivamente. Las líneas 4 y 5 marcan las correspondientes variables con su dominio: `HL` es una variable $\mathcal{FD}$ y `HR` es una variable real. En la línea 6 se llama al predicado `postM` para almacenar el puente en su almacén.

Como se explicó al comienzo de este capítulo, el dominio mediador $\mathcal{M}_{\mathcal{FD}.\mathcal{R}}$ permite inferir que ciertas restricciones de Herbrand son restricciones $\mathcal{R}$-específicas o $\mathcal{FD}$-específicas. Este proceso se lleva a cabo mediante los predicados `specializeFDHerbrand` (línea 8) y `specializeRHerbrand` (línea 9). El siguiente ejemplo ilustra la utilidad de estos predicados. Supongamos que se procesa el objetivo `RX /= RY, X #== RX`. En este caso la primera restricción es una desigualdad que se guarda en el almacén mixto. Al procesar el puente, el predicado

`specializeFDHerbrand` no causa ningún efecto, pero `specializeRHerbrand` determina que la variable `RX` de tipo `real` está involucrada en una restricción de Herbrand, por lo que saca la restricción `RX /= RY` del almacén mixto y la envía al resolutor de reales. De esta forma la restricción `RX /= RY` ha pasado de ser una restricción de Herbrand a ser una restricción $\mathcal{R}$-específica de Herbrand. Nótese que `specializeRHerbrand` realiza este proceso con todas las restricciones de Herbrand (`/=`) del almacén mixto. Por ejemplo, si el objetivo a evaluar es: `RX /= RY, RY /= RZ, X #== RX` al tratar el puente se transforman las restricciones `RX /= RY` y `RY /= RZ` a restricciones $\mathcal{R}$-específicas. Una vez pasado el proceso de especialización de restricciones de Herbrand a restricciones específicas del dominio se continúa con el predicado `storeM` que se muestra en la figura 5.4.

```
1  storeM(L, "#==", R, true, Cin, Cout) :-
2     freeze(L, F is float(L)),
3     tolerance_active(Epsilon),
4     (Epsilon == 0.0 ->
          {R = F}
      ;
          {F - Epsilon =< R, R =< F + Epsilon}
      ),
5     freeze(R, L is integer(round(R))),
6     Cout1 = ['#=='(L,R)|Cin],
7     cleanBridgeStore(Cout1,Cout).

8  storeM(L, "#==", R, false, Cin, Cout) :-
9     freeze(L, (F is float(L),{R =\= F})),
10    freeze(R, (0.0 is float_fractional_part(R) ->
                       (I is integer(R), L #\= I)
                  ;
                       true
                  )
      ),
11    Cout1 = ['#/=='(L,R)|Cin],
12    cleanBridgeStore(Cout1,Cout).
```

Figura 5.4: Continuación de la implementación del puente (`#==`)

Una restricción de la forma `L #== R` $\rightarrow!$ `Out` se comporta como un puente `L #== R` cuando `Out` es `true`, y como un antipuente `L #/== R` cuando `Out` es `false`. El predicado `storeM` trata ambos, aunque en la figura 5.4 solo se muestra el caso del puente pues son casos simétricos. Obsérvese que si `Out` es una variable entonces se vincula primero a `true` y después por medio del mecanismo de vuelta atrás se vincula a `false`, correspondiendo a las reglas **M1** y **M2** de la tabla 5.1, respectivamente.

El predicado `freeze` de Prolog suspende la evaluación de su segundo argumento hasta que el primero sea básico (sin variables). De esta forma, en las líneas 2 y 5 se establece que las variables `L` y `R` corresponden al mismo valor con distintos tipos. Cuando `L` y `R` son básicas, se disparan los correspondientes objetivos congelados y se comprueba si `L` y `R` representan

al mismo valor (regla **M6**) o se produce un fallo cuando `R` es vinculado a un valor real no entero (regla **M7**). Sin embargo, debido a la imprecisión del resolutor de números reales, ocasionalmente los valores de `R` serán una aproximación a un valor entero. Para evitar fallos no deseados producidos por la falta de precisión, en lugar de restringir `R` a un único valor, se toma un intervalo de tolerancia restringiendo la variable real `R` a tomar valores entre `L-Epsilon` y `L+Epsilon`, como se muestra en la línea 4. El intervalo de tolerancia lo define el usuario mediante el comando `/tolerance` y por defecto tiene el valor `0`. En la línea 3 se recupera en la variable `Epsilon` el valor que tiene el sistema en ese momento para la tolerancia. Dependiendo del intervalo de confianza, en la línea 4 se hace corresponder el valor real de `L` con `R` para implementar las reglas **M3** y **M5**. De esta forma, si `R` es un número real fuera del intervalo de confianza de la variable `L` entonces el predicado falla, implementando así la regla **M4**.

En la línea 6 se guarda el puente en el almacén mixto. En la línea 7 la llamada al predicado `cleanBridgeStore` por una parte elimina del almacén los puentes que tienen sus dos argumentos básicos, y por otra implementa la regla **IE** que infiere la igualdad de dos variables que están puenteadas a la misma variable. Por ejemplo, si se tiene en el almacén mixto dos puentes `X #== RX` y `X #== RX'` entonces se unifica `RX` con `RX'` y se elimina un puente del almacén mixto.

En la línea 8 se trata la restricción antipuente `L #== R` $\rightarrow!$ `false`. Esta restricción impone que los argumentos `L` y `R` representen distintos valores. Por lo tanto, tan pronto como `L` o `R` se hagan básicos, se envía una restricción de desigualdad al correspondiente resolutor de SICStus Prolog (líneas 9-10). El código de estas líneas implementa las reglas **M8**, **M9** de la tabla 5.1 y la regla **ID** de la tabla 5.5.

Como se ha comentado anteriormente, la imprecisión del resolutor de reales calcula en ciertas ocasiones aproximaciones a números. La aplicación de un cierto intervalo de tolerancia se ha tenido en cuenta en los puentes pero no en los antipuentes. En los primeros se permite establecer un puente entre un número entero y su aproximación real. Sin embargo, en el segundo caso se decidió que no se permitía establecer antipuentes entre una variable entera y una aproximación real a ella para evitar que casos como el siguiente no sean correctos. Supongamos que el sistema tiene una tolerancia establecida de `0.0001`, entonces el objetivo `2 #== 2.0, 2 #/== 2.0000001` fallaría si se tratase la tolerancia en el antipuente. Además de la estrategia de permitir tolerancia en el puente y no en el antipuente se barajaron otras estrategias, pero se decidió establecer la estrategia que se acaba de ilustrar para abordar problemas prácticos.

5.4.5 Proyección: $\mathcal{FD}$ a $\mathcal{R}$

La proyección de $\mathcal{FD}$ a $\mathcal{R}$ se realiza en la resolución de un objetivo sobre restricciones primitivas atómicas $APCon_{\mathcal{FD}}$ como muestra la tabla 5.3. El sistema puede ejecutarse con el mecanismo de proyección activado o desactivado. Este mecanismo se activa mediante el comando `/proj`. A continuación se muestra como ejemplo de proyección de $\mathcal{FD}$ a $\mathcal{R}$ la implementación de la restricción `#<` (figura 5.5).

Siguiendo la técnica explicada anteriormente, la restricción primitiva atómica `L#<R`$\rightarrow!$`Out`

```
1  '#<'(L, R, Out, Cin, Cout):-
2      hnf(L, HL, Cin, Cout1),
3      hnf(R, HR, Cout1, Cout2),
4      postFDCtrOp(HL, "<", HR, Out, Cout2, Cout).

5  postFDCtrOp(HL, OP, HR, O, Cin, Cout):-
6      specializeFDHerbrand(HL, true, Cin, Cout1),
7      specializeFDHerbrand(HR, true, Cout1, Cout2),
8      storeAndSolveFDCtrOp(HL, OP, HR, O, Cout2, Cout3),
9      (proj_active ->
           projectFDtoRCtrOp(HL,OP,HR,O,Cout3,Cout)
       ;
           Cout = Cout3
       ).

10 projectFDtoRCtrOp(HL, OP, HR, O, Cin, Cout ):-
11     bridgeFDtoR(HL, HLR, true, Cin, Cout1),
12     bridgeFDtoR(HR, HRR, true, Cout1, Cout2),
13     (var(O) ->     % if OP is an arithmetical operator
14         (storeM(O,"#==",OR,true,Cout2,Cout3),
15          storeAndSolveRCtrOp(HLR,OP,HRR,OR,Cout3,Cout)
           )
       ;   % if OP is a relational or equality operator
16         storeAndSolveRCtrOp(HLR,OP,HRR,O,Cout2,Cout)
       ).
```

Figura 5.5: Implementación de (`#<`)

se implementa con un predicado Prolog de cinco argumentos (línea 1): los dos argumentos de la restricción (`L` y `R`), el resultado (`Out`) y el almacén mixto. Primero se reducen los argumentos a forma normal de cabeza (líneas 2 y 3), y se trata la restricción primitiva correspondiente (línea 4) dependiendo del valor del resultado `Out`. A continuación se llama al predicado `postFDCtrOp` que es genérico para todos los operadores de $\mathcal{FD}$ y que realiza el proceso de especialización de restricciones de Herbrand a restricciones $\mathcal{FD}$-específicas (líneas 6 y 7) como se explicó en la sección anterior.

Después del proceso de especialización se continúa con el predicado `storeAndSolveFDCtrOp` (línea 8) que se encarga de tratar convenientemente la restricción y enviarla al resolutor $\mathcal{FD}$. Este predicado trata todas las restricciones aritméticas (`L #+ R` $\rightarrow!$ `O`), relacionales (`L #< R` $\rightarrow!$ `O`) y $\mathcal{FD}$-específicas (`L == R` $\rightarrow!$ `O` y `L /== R` $\rightarrow!$ `O`). En las aritméticas el parámetro `O` se vincula a un número entero mientras que en las relacionales y $\mathcal{FD}$-específicas el parámetro `O` se vincula a un valor Booleano. En el caso particular de la restricción `#<`, se envía al resolutor de $\mathcal{FD}$ la restricción `HL #< HR` si `O` es `true`, o la restricción `HL #>= HR` si `O` es `false`, o bien se trata la reificación convenientemente.

Si la proyección está activada, indicado por el predicado dinámico `proj_active` en la línea 9, entonces continúa con `projectFDtoRCtrOp` para generar los puentes y las proyeccio-

nes. El predicado `bridgeFDtoR` (líneas 11 y 12) busca en el almacén mixto si la variable `HL` está involucrada en algún puente. Si es así, entonces la variable `HLR` se vincula al valor real correspondiente. Si la variable `HL` no está involucrada en ningún puente, este predicado crea un nuevo puente entre las variables `HL` y `HLR`. El tercer argumento de `bridgeFDtoR` indica con `true` que se trata de un puente y no de un antipuente. Los dos últimos argumentos representan el almacén mixto, entrada y salida respectivamente. Este proceso se corresponde con la generación de puentes de la función $bridges^{\mathcal{FD}\to\mathcal{R}}(\pi, B)$ de la tabla 5.3. Se procede igualmente para la variable `HR` en la línea 12. Un caso particular de la generación de puentes de la tabla 5.3 es cuando la restricción que se trata es una operación aritmética. Por la forma en que se realiza el cómputo en el tratamiento de restricciones aritméticas, el predicado `projectFDtoRCtrOp` es llamado con el parámetro `O` sin instanciar, mientras que en los operadores relacionales `O` está instanciado a un valor Booleano. En el primer caso, `O` es una variable entera que necesita ser puenteada con una variable nueva `OR` (línea 14).

Finalmente, la restricción creada mediante la proyección se envía al resolutor $\mathcal{R}$ (líneas 15 o 16) implementando así la función $proj^{\mathcal{FD}\to\mathcal{R}}(\pi, B)$ de la tabla 5.3.

5.4.6 Proyección: $\mathcal{R}$ a $\mathcal{FD}$

La implementación de las proyecciones de $\mathcal{R}$ a $\mathcal{FD}$ es similar a la implementación vista para las proyecciones de $\mathcal{FD}$ a $\mathcal{R}$, pero hay que tener en cuenta que las oportunidades de construir puentes en este caso son menores, pues como muestra la tabla 5.4 únicamente se pueden construir puentes para números reales con parte decimal nula y para aquellas variables que son resultado de las operaciones aritméticas `+`, `-` y `*` entre términos que son o bien números reales con parte decimal nula o bien variables con puentes ya establecidos. Para el resto de casos de la tabla 5.4 no se generan puentes automáticamente para variables reales, pues no se puede asegurar que su valor sea entero. En la figura 5.6 se muestra como ejemplo de proyección el código correspondiente a la restricción `>`.

Como en los casos anteriores, la implementación de la primitiva `>` tiene cinco argumentos (línea 1). Los dos primeros, `L` y `R`, se reducen a sus formas normales de cabeza (líneas 2 y 3) y, se procesa la primitiva mediante el predicado `postRCtrOp` (línea 4). En este punto ya se sabe que los términos `HL` y `HR` son de tipo `real` y por esto se trata la especialización de restricciones de Herbrand a restricciones $\mathcal{R}$-específicas (líneas 6 y 7).

Si la proyección está activada entonces hay que estudiar las distintas posibilidades de `L` y `R` teniendo en cuenta la información del almacén mixto. Para ello se utiliza el predicado `bridgeRtoFD` (líneas 12-13) que recibe dos argumentos de entrada: una constante o variable real (primer parámetro) y el almacén mixto (cuarto parámetro). En concreto, en la llamada de la línea 12 se tienen varias posibilidades:

- Si `L` es un número real de valor entero entonces se devuelve su correspondiente entero en `LFD` y `true` en `Out1`. Este caso implementa las reglas **M3** y **M4** del sistema de transformación de almacenes de la tabla 5.1.

- Si `L` es un número real con valor no entero entonces se devuelve su correspondiente entero en `LFD` y `false` en `Out1`. Nótese que se devuelve el valor entero aun cuando la parte decimal no es cero con el fin de proyectar la restricción si fuese posible.

```
1  '>'(L,R,O,Cin,Cout):-
2      hnf(L,HL,Cin,Cout1),
3      hnf(R,HR,Cout1,Cout2),
4      postRCtrOp(HL, ">", HR, O, Cout2, Cout).

5  postRCtrOp(L, OP, R, O, Cin, Cout):-
6      specializeRHerbrand(L, true, Cin, Cout1),
7      specializeRHerbrand(R, true, Cout1, Cout2),
8      storeAndSolveRCtrOp(L, OP, R, O, Cout2, Cout3),
9      (proj_active ->
           projectRtoFDCtrOp(L,OP,R,O,Cout3,Cout)
       ;
           Cout = Cout3
       ).

11 projectRtoFDCtrOp(L, OP, R, O, Cin, Cout ):-
12     bridgeRtoFD(L, LFD, Out1, Cin, Cout1),
13     bridgeRtoFD(R, RFD, Out2, Cout1, Cout2),
14     ( var(O) ->    % if OP is an arithmetical operator
15        ((Out1 == true, Out2 == true) ->
16           (storeM(OFD,"#==", O, true, Cout2, Cout3),
17            storeAndSolveFDCtrOp(LFD,OP,RFD,OFD,Cout3,Cout))
          ;
18            Cout = Cout2
          )
       ;  % if OP is a relational or equal operator
19       (createFDCtrMate(Out1,Out2,O,LFD,OP,RFD,Cout2,Cout),!)
       ).
```

Figura 5.6: Implementación de (>)

- Si `L` es una variable entonces se busca en el almacén mixto `Cin` su correspondiente puente: en el caso de encontrarlo se devuelve la variable entera del puente en `LFD` y `true` en `Out1`; en caso contrario, no se asigna nada en `Out1`.

Si la restricción es una operación aritmética y existen puentes para los operandos (líneas 14 y 15) entonces se crea un puente para la variable resultado de la operación (línea 16). La correspondiente restricción $\mathcal{FD}$ que es resultado de la proyección se trata en la línea 17. Si la restricción es una operación aritmética pero no existen puentes para los dos operandos entonces simplemente el almacén mixto se deja como estaba (línea 18). Si la restricción es una operación relacional entonces se crea la restricción $\mathcal{FD}$ mediante el predicado `createFDCtrMate` siguiendo la tabla 5.4. Los distintos casos se tratan mediante el predicado `createFDCtrMate`, mostrado en la figura 5.7, para proyectar la restricción `L > R` $\rightarrow!$ `Out` de acuerdo a la tabla 5.4. En cada caso se genera una restricción $\mathcal{FD}$ que se envía a su resolutor. En concreto, se tienen las siguientes posibilidades: o bien `Out1` es `true` y por lo tanto `L` es o bien un número real de valor entero o bien una variable real con su correspondiente puente

en la variable LFD; o bien Out1 es false y L es una constante real u, de forma que LFD es computada como $\lfloor u \rfloor$; o bien Out1 no está instanciado lo que significa que no hay suficiente información y no se procesa. La variable Out2 se trata análogamente.

```
createFDCtrMate(true,true,true,R,">",L,Cin,Cout) :-
     storeAndSolveFDCtrOp(R,">",L,true,true,Cin,Cout).

createFDCtrMate(true,true,false,R,">",L,Cin,Cout) :-
     storeAndSolveFDCtrOp(R,"<=",L,false,true,Cin,Cout).

createFDCtrMate(true,false,true,R,">",L,Cin,Cout) :-
     storeAndSolveFDCtrOp(R,">",L,true,true,Cin,Cout).

createFDCtrMate(true,false,false,R,">",L,Cin,Cout) :-
     storeAndSolveFDCtrOp(R,"<=",L,false,true,Cin,Cout).

createFDCtrMate(false,true,true,R,">",L,Cin,Cout) :-
     storeAndSolveFDCtrOp(R,">=",L,true,true,Cin,Cout).

createFDCtrMate(false,true,false,R,">",L,Cin,Cout) :-
     storeAndSolveFDCtrOp(R,"<",L,false,true,Cin,Cout).
```

Figura 5.7: Continuación de la implementación de (>)

Por ejemplo, supongamos que se tiene el objetivo `X #== RX, RX > 4.3` y la proyección está activada. En primer lugar el puente se almacena en el almacén mixto y después se procesa la restricción >, donde RX se representa por L y 4.3 por R. En este caso O tiene el valor true. Ambos argumentos están en forma normal de cabeza y la restricción se envía al resolutor $\mathcal{R}$. Al explorar el almacén mixto se encuentra el puente `X #== RX`, por lo tanto Out1 se instancia a true y LFD se vincula a X. Como 4.3 es una constante, el valor computado para Out2 es false, y el valor de RFD es $\lfloor 4.3 \rfloor$, es decir 4. Aplicando el caso apropiado de la tabla 5.4 (tercera cláusula de createFDCtrMate), la restricción `X #> 4` será enviada al resolutor $\mathcal{FD}$.

5.5 Resultados experimentales

Con respecto a la implementación realizada para la cooperación entre los dominios $\mathcal{FD}$ y $\mathcal{R}$, es interesante comprobar experimentalmente tres aspectos: por una parte, verificar que la activación del mecanismo de cooperación no penaliza el tiempo de ejecución en problemas que pueden ser resueltos utilizando únicamente un dominio de restricciones. Por otro lado, es interesante estudiar si el mecanismo de cooperación ayuda a mejorar el tiempo de ejecución en problemas donde ambos dominios $\mathcal{FD}$ y $\mathcal{R}$ son necesarios. Por último, comparar los resultados obtenidos en $\mathcal{TOY}$ con respecto a Meta-S, un sistema $CFLP$ que permite cooperación entre los mismos dominios [FHM03a, FHM03b, FHR05, HP07] y que se explicará más adelante.

En el trabajo [EFH$^+$09] se realiza un estudio detallado de estos tres aspectos. En particular, se comprueba empíricamente que la cooperación no afecta negativamente al tiempo de

ejecución en programas que no necesitan cooperación.

En esta sección de la tesis se presenta un resumen de las conclusiones obtenidas en el estudio de los dos últimos aspectos mencionados. Para ello se han elegido los siguientes problemas que utilizan la cooperación entre dominios de una forma natural:

- **Donald (donald)**: Un problema criptoaritmético con 10 variables $\mathcal{FD}$ variables, una ecuación lineal, y una restricción *alldifferent*. Este problema consiste en resolver la ecuación DONALD + GERALD = ROBERT.
- **Send More Money (smm)**: Un problema criptoaritmético con 8 variables $\mathcal{FD}$ una ecuación lineal, 2 desigualdades y una restricción *alldifferent*. Consiste en resolver la ecuación SEND + MORE = MONEY.
- **Problema no lineal (nl-csp)**: Un problema criptoaritmético no lineal con 9 variables $\mathcal{FD}$ y ecuaciones no lineales.
- **Wrong-Wright (wwr)**: Otro problema criptoaritmético con 8 variables $\mathcal{FD}$ una ecuación lineal y una restricción *alldifferent*. Consiste en resolver la ecuación WRONG + WRONG = RIGHT.
- **3 $\times$ 3 Magic Square (mag.sq.)**: Clásico problema del cuadrado mágico que involucra 9 variables $\mathcal{FD}$ y 7 ecuaciones lineales.
- **Equation 10 (eq.10)**: Un sistema de 10 ecuaciones con 7 variables $\mathcal{FD}$.
- **Equation 20 (eq.20)**: Un sistema de 20 ecuaciones con 7 variables $\mathcal{FD}$.
- **Knapsack (knapsack)**: Clásico problema de la mochila con dos versiones, una es un problema de satisfacción de restricciones y la otra versión es con optimización,
- **Electrical Circuit (circuit)**: Ejemplo 7. El objetivo es determinar qué condensador debe ser usado de tal forma que el voltaje alcance el 99 % del voltaje final en un tiempo determinado de duración en un circuito dado.
- **bothIn (goal2)**: es el `Objetivo 2` del ejemplo 5 presentado en este capítulo con distintos valores para `n`.
- **bothIn (goal3)**: es el `Objetivo 3` del ejemplo 5 con distintos valores para `n`.
- **Gas distribution (distrib)**: problema de optimización de una distribución (véase el apéndice 8 de [ACE+07]). Modela una red de comunicación donde existen `NR` proveedores de materia prima continua y `ND` proveedores de materia prima discreta. El objetivo es minimizar el coste total. Las distintas instancias `distrib(ND,NR)` corresponden a distintas elecciones de valores para las variables `ND` y `NR`.

Todas las pruebas han sido realizadas sobre Linux (versión profesional, Suse Linux 9.3) con un procesador Intel Pentium M de 1.70GHz y memoria RAM de 1 GB. Los resultados corresponden al tiempo medio en milisegundos empleado por 10 ejecuciones de cada uno de los experimentos.

La figura 5.8 muestra los resultados de ejecución de los programas anteriores hasta obtener la primera solución y la figura 5.9 muestra todas las soluciones de algunos casos. En cada tabla se incluyen dos columnas que corresponden a dos estrategias de etiquetado: *naïve*, que etiqueta las variables $\mathcal{FD}$ según su orden de aparición (es decir, primero la variable que está más a la izquierda); y *first fail (ff)*, que etiqueta primero la variable con menor dominio. Estas estrategias están combinadas con los distintos modos de activación. Por cada estrategia se ha medido el tiempo de ejecución sin incluir proyección de restricciones entre dominios (columna `noproj`) y con las proyecciones activadas (columna `proj`). La tercera columa de cada estrategia muestra la ganancia de velocidad (*speed-up*) resultado de activar las proyecciones.

	naïve			*first fail (ff)*		
Benchmarks	`noproj`	`proj`	`noproj/proj`	`noproj`	`proj`	`noproj/proj`
circuit	14	14	1.00	13	20	0.65
distrib (2,5.0)	662	144	4.60	506	504	1.00
distrib (3,3.0)	1,486	132	11.26	810	814	1.00
distrib (3,4.0)	2,098	156	13.45	1,290	1,178	1.10
distrib (4,5.0)	20,444	240	85.18	12,670	12,744	0.99
distrib (5,2.0)	29,108	198	147.01	5,162	7,340	0.70
distrib (5,5.0)	141,734	272	521.08	85,856	86,497	0.99
distrib (5,10.0)	568,665	474	1,199.72	464,230	462,980	1.00
goal2 (100)	25	14	1.79	28	14	2.00
goal2 (200)	40	13	3.08	44	15	2.93
goal2 (400)	70	12	5.83	72	13	5.54
goal2 (800)	131	12	10.92	135	15	9.00
goal2 (10000)	704	14	50.29	713	16	44.56
goal2 (20000)	1,271	12	105.92	1,270	16	79.38
goal2 (40000)	2,325	11	211.36	2,333	16	145.81
goal2 (80000)	4,452	13	342.46	4,472	16	279.50
goal2 (200000)	10,725	13	825.00	10,781	15	718.73
goal3 (100)	18	15	1.20	20	16	1.25
goal3 (200)	26	13	2.00	28	13	2.15
goal3 (400)	41	15	2.73	44	16	2.75
goal3 (800)	75	16	4.69	77	17	4.53
goal3 (5000)	354	14	25.29	360	16	22.50

Figura 5.8: Evaluación del mecanismo de proyección en el sistema $\mathcal{TOY}$ de restricciones en programas que demandan la cooperación entre resolutores (primera solución)

Se puede observar que el mecanismo de proyección causa una mejora significativa en la mayoría de los casos. Además, se concluye que o bien la mejora producida por la activación de las proyecciones es muy relevante, o bien no se empeora el rendimiento significativamente. En

	naïve			*first fail (ff)*		
Benchmarks	`noproj`	`proj`	`noproj/proj`	`noproj`	`proj`	`noproj/proj`
goal3 (100)	673	265	2,54	625	242	2,58
goal3 (200)	1.867	329	5,67	1.844	352	5,24
goal3 (400)	6.527	583	11,20	6.573	579	11,35
goal3 (800)	24.460	976	25,06	24.727	994	24,88
goal3 (5000)	911.880	5.365	169,97	920.670	6.135	150,07

Figura 5.9: Evaluación del mecanismo de proyección en el sistema $\mathcal{TOY}$ de restricciones en programas que demandan la cooperación entre resolutores (todas las soluciones)

particular, las pruebas sobre el problema de la distribución (`distribution ND NR`) muestran que la mejora de la eficiencia con la cooperación depende de la estrategia de etiquetado elegida. Los objetivos `goal2(n)` y `goal3(n)` se comportan especialmente bien en ambos etiquetados cuando se busca la primera solución (primera tabla). En la tabla 5.9 se muestran los resultados correspondientes a la búsqueda de todas las soluciones para el objetivo `goal3(n)`, pues es el único con más de una solución. En este objetivo también es significativa la ganancia producida por las proyecciones para ambas estrategias de etiquetado.

Con respecto a otros sistemas $CFLP$, el único que soporta la cooperación entre dominios finitos y reales es el sistema `Meta-S` [Hof01, FHM03a, FHM03b, FHR05, HP07]. `Meta-S` está implementado en *Common Lisp* y permite la integración de resolutores externos a través de un meta-resolutor que tiene vinculados resolutores de diferentes dominios. En particular, un resolutor para ecuaciones lineales aritméticas, un resolutor de dominios finitos y un resolutor aritmético de intervalos. En `Meta-S` se construyen lo que se denominan proyecciones *strong*, que son una disyunción de conjunciones de restricciones. En $\mathcal{TOY}$ las distintas alternativas de soluciones son computadas vía backtraking. En `Meta-S` no se pueden dar estrategias de etiquetado pero dispone de diferentes estrategias para la resolución de restricciones:

- *Standard eager*: propaga la información tan pronto como sea posible.
- *Ordered eager*: como la anterior pero con información del usuario que determina el orden de las proyecciones.
- *Standard heuristic*: como la primera pero con una heurística que da prioridad a las vinculaciones de las variables que están más próximas al fallo.
- *Ordered heuristic*: como la anterior pero con información del usuario que determina el orden de las proyecciones.

En cierta forma, el etiquetado *naïve* y *ff* de $\mathcal{TOY}$ son similares a las estrategias *eager* y *heuristic* de `Meta-S`. Otras diferencias que se encuentran entre ambos sistemas son que `Meta-S` no soporta diferentes modos de activación ($\mathcal{TOY}(\mathcal{FD})$, $\mathcal{TOY}(\mathcal{R})$, $\mathcal{TOY}(\mathcal{FD}+\mathcal{R})$ y $\mathcal{TOY}(\mathcal{FD}+\mathcal{R})_p$), ni facilidades para resolver problemas de optimización, ni diferentes formas de etiquetado, ni restricciones globales como por ejemplo `all_different`. Sin embargo, a

favor de `Meta-S` cabe mencionar que permite integrar y/o redefinir distintas estrategias de resolución de restricciones. Esto hace que su modelo de cooperación sea más flexible que el modelo de cooperación implementado actualmente en $\mathcal{TOY}$ que se basa en una estrategia fija para la resolución de objetivos. También el mecanismo de proyección de $\mathcal{TOY}$ es menos potente que en `Meta-S`, porque las proyecciones no se pueden aplicar a las restricciones dentro de los almacenes. Además, `Meta-S` permite la integración de diferentes lenguajes [FHR05].

Las tablas de las figuras 5.10 y 5.11 muestran una comparación del rendimiento del sistema $\mathcal{TOY}$ con respecto al sistema `Meta-S`. En ambas tablas la columna denominada $\mathcal{TOY}$ se refiere al sistema $\mathcal{TOY}$ con las proyecciones activadas. La columna `Meta-S` corresponde a la ejecución en ese sistema. La última columna contiene la ganancia de velocidad de $\mathcal{TOY}$ respecto a `Meta-S`. El símbolo '-' en los problemas de la mochila y la distribución de materias primas representa que no han podido realizarse los experimentos en `Meta-S` pues son problemas de optimización. Por otra parte, el objetivo `goal3(800)` produce un error en el sistema `Meta-S`.

Benchmarks	$\mathcal{TOY}$	`Meta-S`	`Meta-S`/$\mathcal{TOY}$
donald	188	5,290	28.13
smm	14	580	41.42
nl-csp	15	970	64.66
wwr	18	620	34.44
maq.sq.	87	520	5.97
eq.10	74	60	0.81
eq.20	131	60	0.45
knapsack (csp)	5	60	12.00
knapsack (opt)	11	-	-
distrib(5,10.0)	474	-	-
circuit	13	70	5.38
goal2 (100)	14	330	23.57
goal2 (200)	13	730	56.15
goal2 (400)	12	2,340	195.00
goal2 (800)	12	8,540	711.66
goal3 (100)	15	410	27.33
goal3 (200)	13	900	69.23
goal3 (400)	15	2,870	191.33
goal3 (800)	16	10,630	664.37

Figura 5.10: Comparativa de los sistemas $\mathcal{TOY}$ y `Meta-S` (primera solución)

De las tablas de las figuras 5.10 y 5.11 se extraen las siguientes conclusiones: `Meta-S` parece comportarse muy bien en la resolución de ecuaciones lineales, especialmente cuando el problema no requiere restricciones globales. Esto puede ser debido a dos causas: primero, que el resolutor lineal aritmético de `Meta-S` tenga mejor rendimiento que el correspondiente resolutor de $\mathcal{TOY}$ y, segundo, que aplanar una restricción en $\mathcal{TOY}$ genere tantas restricciones

Benchmarks	$\mathcal{TOY}$	Meta-S	Meta-S/$\mathcal{TOY}$
goal3 (100)	242	6,940	28.67
goal3 (200)	329	46,880	142.49
goal3 (400)	579	346,930	599.18
goal3 (800)	976	error	-

Figura 5.11: Comparativa de los sistemas $\mathcal{TOY}$ y Meta-S (todas las soluciones)

como el número de operadores que incluye. Sin embargo, $\mathcal{TOY}$ tiene en general un 'mejor' rendimiento con respecto a Meta-S, aunque esto debe interpretarse con cuidado. Una de las razones de este comportamiento puede ser que los resolutores numéricos conectados en Meta-S se han implementado solo para experimentar con los conceptos del marco teórico descrito en [Hof01, HP07], sin preocuparse por la optimización del sistema, mientras que $\mathcal{TOY}$ utiliza los resolutores optimizados de SICStus Prolog. Por último, otra ventaja de $\mathcal{TOY}$ es la disponibilidad de restricciones globales tales como `all_different`, que no se encuentran en Meta-S.

Capítulo 6

Cooperación entre $\mathcal{FD}$ y $\mathcal{FS}$

En este capítulo se trata la cooperación entre los dominios $\mathcal{FD}$ y $\mathcal{FS}$ de una forma análoga a la cooperación entre los dominios $\mathcal{FD}$ y $\mathcal{R}$ del capítulo anterior. Una diferencia fundamental entre ambas cooperaciones viene dada por las implementaciones, pues para la cooperación entre $\mathcal{FD}$ y $\mathcal{R}$ se utilizaron los resolutores incluidos en SICStus Prolog, mientra que para la cooperación entre los dominios $\mathcal{FD}$ y $\mathcal{FS}$ se ha necesitado introducir un resolutor de conjuntos. Estudiando las distintas opciones disponibles, como se ha mostrado en el trabajo relacionado, se ha decido extender $\mathcal{TOY}$ con los resolutores de ECL^{i}PS^e para tratar ambos dominios $\mathcal{FD}$ y $\mathcal{FS}$.

De forma análoga al capítulo anterior, para el esquema formal de esta cooperación se define un dominio mediador $\mathcal{M}_{\mathcal{FD}.\mathcal{FS}}$ con su correspondiente resolutor $solve^{\mathcal{M}_{\mathcal{FD}.\mathcal{FS}}}$ y a partir de aquí ya se tienen todos los componentes para formar una instancia del dominio de coordinación $\mathcal{C}_{\mathcal{FD}.\mathcal{FS}} = \mathcal{M}_{\mathcal{FD}.\mathcal{FS}} \oplus \mathcal{H} \oplus \mathcal{FD} \oplus \mathcal{FS}$. Sobre este dominio de coordinación se establece el cálculo $CCLNC(\mathcal{C}_{\mathcal{FD}.\mathcal{FS}})$ que está formado por las reglas de estrechamiento perezoso y las invocaciones a los resolutores que se mostraron en el capítulo 4, más las nuevas reglas específicas para esta cooperación que se mostrarán en este capítulo. En particular, las reglas nuevas de cooperación son

- Reglas que generan puentes que comunican los dominios $\mathcal{FD}$ y $\mathcal{FS}$ basados en la cardinalidad de los conjuntos.
- Reglas que proyectan restricciones entre ambos dominios.
- Reglas que infieren igualdades y fallo.
- Regla que invoca al resolutor $\mathcal{FS}$.

Finalmente, se muestra la implementación y resultados experimentales que se han tomado comparando tres estrategias de nuestra implementación de $\mathcal{TOY}$ con programas sobre conjuntos disponibles en la página web de ECL^{i}PS^e.

6.1 El dominio mediador $\mathcal{M}_{\mathcal{FD}.\mathcal{FS}}$

Los dominios $\mathcal{FD}$ y $\mathcal{FS}$, descritos en las subsecciones 3.3 y 3.4, son unibles formando una suma amalgamada definida como un nuevo dominio $\mathcal{S} = \mathcal{FD} \oplus \mathcal{FS}$ de signatura $\Sigma_{\mathcal{S}} = \langle TC,\ SBT_{\mathcal{FD}} \cup SBT_{\mathcal{FS}},\ DC,\ DF,\ SPF_{\mathcal{FD}} \cup SPF_{\mathcal{FS}} \rangle$. El dominio $\mathcal{S}$ es una extensión conservativa de ambos dominios pero no tiene mecanismos de comunicación entre estos dominios puros. Para permitir esta comunicación se define un nuevo dominio mediador $\mathcal{M}_{\mathcal{FD}.\mathcal{FS}}$ que contiene *restricciones puente de cardinalidad*, y cuya signatura $\Sigma_{\mathcal{M}_{\mathcal{FD}.\mathcal{FS}}} = \langle TC,\ SBT_{\mathcal{M}_{\mathcal{FD}.\mathcal{FS}}},\ DC,\ DF,\ SPF_{\mathcal{M}_{\mathcal{FD}.\mathcal{FS}}} \rangle$ cumple las siguientes condiciones:

- $SBT_{\mathcal{M}_{\mathcal{FD}.\mathcal{FS}}} = \{\texttt{int}, \texttt{set}\} \subseteq SBT_{\mathcal{FD}} \cup SBT_{\mathcal{FS}}$.
- Cada conjunto de valores básicos del dominio mediador corresponde a un conjunto de valores básicos de cada dominio puro: $\mathcal{B}_{\texttt{set}}^{\mathcal{M}_{\mathcal{FD}.\mathcal{FS}}} = \mathcal{B}_{\texttt{set}}^{\mathcal{FS}}$ y $\mathcal{B}_{\texttt{int}}^{\mathcal{M}_{\mathcal{FD}.\mathcal{FS}}} = \mathcal{B}_{\texttt{int}}^{\mathcal{FD}}$.
- $SPF_{\mathcal{M}_{\mathcal{FD}.\mathcal{FS}}} = \{$ `#--` $:: \texttt{int} \to \texttt{set} \to \texttt{bool}\}$. La interpretación de la *restricción puente de cardinalidad* i `#--`$^{\mathcal{M}_{\mathcal{FD}.\mathcal{FS}}}$ $s \to t$ es la siguiente: o s es un conjunto básico de enteros, i es un número entero que coincide con la cardinalidad de s y $t = \texttt{true}$; o bien s es un conjunto básico de enteros, i es un número entero que no coincide con la cardinalidad de s y $t =$`false`; o bien $t = \bot$. Tal y como se ha descrito, `#--`$^{\mathcal{M}_{\mathcal{FD}.\mathcal{FS}}}$ es un subconjunto del producto cartesiano $\mathbb{Z} \times \mathcal{B}_{\texttt{set}}^{\mathcal{FS}}$.

El correspondiente resolutor $solve^{\mathcal{M}_{\mathcal{FD}.\mathcal{FS}}}$ de este nuevo dominio mediador $\mathcal{M}_{\mathcal{FD}.\mathcal{FS}}$ se define mediante un sistema de transformación de almacenes, cuyas reglas se muestran en la tabla 6.1. Como en los capítulos anteriores, $\pi, \Pi \,\square\, \sigma \vdash_{\mathcal{M}_{\mathcal{FD}.\mathcal{FS}}} \Pi' \,\square\, \sigma'$ indica que el almacén $\pi, \Pi \,\square\, \sigma$, que incluye la restricción primitiva atómica seleccionada π más otras restricciones Π, se transforma en el almacén $\Pi' \,\square\, \sigma'$.

M1	X `#--` u', $\Pi \,\square\, \sigma \vdash_{\mathcal{M}_{\mathcal{FD}.\mathcal{FS}}} \Pi\sigma_1 \,\square\, \sigma\sigma_1$
	si $u' \in \mathcal{B}_{\texttt{set}}^{\mathcal{FS}}$, $X \in \mathcal{V}ar$ y $\exists u \in \mathbb{Z}^+$ tal que u `#--`$^{\mathcal{M}_{\mathcal{FD}.\mathcal{FS}}}$ u' y $\sigma_1 = \{X \mapsto u\}$.
M2	u `#--` S, $\Pi \,\square\, \sigma \vdash_{\mathcal{M}_{\mathcal{FD}.\mathcal{FS}}} \Pi\sigma_1 \,\square\, \sigma\sigma_1$
	si $u = 0$, $S \in \mathcal{V}ar$ y $\sigma_1 = \{S \mapsto \{\}\}$
M3	u `#--` u', $\Pi \,\square\, \sigma \vdash_{\mathcal{M}_{\mathcal{FD}.\mathcal{FS}}} \Pi \,\square\, \sigma$
	si $u \in \mathbb{Z}^+$, $u' \in \mathcal{B}_{\texttt{set}}^{\mathcal{FS}}$ tal que u `#--`$^{\mathcal{M}_{\mathcal{FD}.\mathcal{FS}}}$ u'.
M4	u `#--` u', $\Pi \,\square\, \sigma \vdash_{\mathcal{M}_{\mathcal{FD}.\mathcal{FS}}} \blacksquare$
	si $u \in \mathbb{Z}^+$, $u' \in \mathcal{B}_{\texttt{set}}^{\mathcal{FS}}$ y no se cumple u `#--`$^{\mathcal{M}_{\mathcal{FD}.\mathcal{FS}}}$ u'.

Tabla 6.1: Reglas de transformación de almacenes que definen $\vdash_{\mathcal{M}_{\mathcal{FD}.\mathcal{FS}}}$

La regla **M1** representa el caso en el cual el segundo argumento del puente cardinal es un conjunto básico. En este caso la variable X se instancia al cardinal del conjunto. En **M2** el conjunto S tiene cardinal cero y por lo tanto S se instancia al conjunto vacío. En las reglas

M3 y **M4** ambos argumentos son básicos, y se aplicará una u otra regla dependiendo de cómo se satisfagan las condiciones.

El siguiente teorema asegura que el sistema de transformación de almacenes que define $solve^{\mathcal{M}_{\mathcal{FD}.\mathcal{FS}}}$ es una especificación correcta de un resolutor de caja transparente para el dominio $\mathcal{M}_{\mathcal{FD}.\mathcal{FS}}$. Este teorema se demuestra en el apéndice A.8.

Teorema 10. (Propiedades formales del resolutor $solve^{\mathcal{M}_{\mathcal{FD}.\mathcal{FS}}}$)
El sistema de transformación de almacenes con relación de transición $\vdash_{\mathcal{M}_{\mathcal{FD}.\mathcal{FS}}}$ cumple las propiedades de la definición 6 (capítulo 3). Además, como se mostró en la definición 5:

$$solve^{\mathcal{M}_{\mathcal{FD}.\mathcal{FS}}}(\Pi) = \bigvee\{\exists\overline{Y'}(\Pi' \square \sigma') \mid \Pi' \square \sigma' \in SF_{\mathcal{M}_{\mathcal{FD}.\mathcal{FS}}}(\Pi),\ \overline{Y'} = var(\Pi' \square \sigma') \setminus var(\Pi)\}$$

está bien definido para cualquier conjunto $\Pi \in APCon_{\mathcal{M}_{\mathcal{FD}.\mathcal{FS}}}$. El resolutor $solve^{\mathcal{M}_{\mathcal{FD}.\mathcal{FS}}}$ satisface todos los requisitos de los resolutores de la definición 4.

De forma similar al domino mediador $\mathcal{M}_{\mathcal{FD}.\mathcal{R}}$, a partir del dominio mediador $\mathcal{M}_{\mathcal{FD}.\mathcal{FS}}$ se puede inferir que ciertas restricciones atómicas de Herbrand son restricciones $\mathcal{FD}$-específicas o $\mathcal{FS}$-específicas. A continuación se muestra la definición formal.

Sea $\mathcal{M}_{\mathcal{FD}.\mathcal{FS}}$ el dominio mediador y π una restricción atómica extendida de Herbrand bien tipada de la forma t_1 `==` t_2 o bien t_1 `/=` t_2, donde los patrones t_1 y t_2 son o una constante entera o un conjunto de enteros o una variable. Entonces se dice:

1. $M \vdash \pi$ `in` $\mathcal{FD}$ ('M permite inferir que π es $\mathcal{FD}$-específica') si y solamente si algunas de las tres siguientes condiciones se cumple:
   (a) t_1 o t_2 es una constante entera.
   (b) t_1 o t_2 es una variable que está en el lado izquierdo de algún puente `#--` .
   (c) t_1 o t_2 es una variable que ha sido reconocida de tipo `int` por algún mecanismo dependiente de la implementación.

2. $M \vdash \pi$ `in` $\mathcal{FS}$ ('M permite inferir que π es $\mathcal{FS}$-específica') si y solamente si algunas de las tres condiciones siguientes se cumple:
   (a) t_1 o t_2 es un conjunto de números enteros.
   (b) t_1 o t_2 es una variable que está en el lado derecho de algún puente `#--` .
   (c) t_1 o t_2 es una variable que ha sido reconocida de tipo `set` por algún mecanismo dependiente de la implementación.

6.2 Cálculo $CCLNC(\mathcal{C}_{\mathcal{FD}.\mathcal{FS}})$

A partir del dominio mediador definido en la sección anterior, se define el dominio de coordinación de los dominios puros $\mathcal{H}$, $\mathcal{FD}$ y $\mathcal{FS}$ como se muestra a continuación:

$$\mathcal{C}_{\mathcal{FD}.\mathcal{FS}} = \mathcal{M}_{\mathcal{FD}.\mathcal{FS}} \oplus \mathcal{H} \oplus \mathcal{FD} \oplus \mathcal{FS}.$$

Todos los dominios en $\mathcal{C}_{\mathcal{FD}.\mathcal{FS}}$ son unibles dos a dos y la comunicación con $\mathcal{H}$ se realiza por medio de las sustituciones de variables. Como se explicó en los capítulos anteriores, el cálculo $CCLNC$ puede ser instanciado por distintos dominios de restricciones, y en particular por el dominio de coordinación $\mathcal{C}_{\mathcal{FD}.\mathcal{FS}}$, produciendo el cálculo $CCLNC(\mathcal{C}_{\mathcal{FD}.\mathcal{FS}})$ que se define en esta sección.

En $CCLNC(\mathcal{C}_{\mathcal{FD}.\mathcal{FS}})$ se utilizan programas y objetivos definidos como en el cálculo $CCLNC(\mathcal{C}_{\mathcal{FD}.\mathcal{R}})$, pero en este caso, un objetivo se define de la siguiente forma: $G \equiv \exists\overline{U}.P \square C \square M \square H \square F \square S$, donde las variables locales $\overline{U}$, la producciones P, el *pool* de restricciones C y el almacén del dominio de Herbrand H ya fueron definidos en la sección 4.2, y el almacén F fue definido en la sección 5.2. Los almacenes nuevos son:

- $M = \Pi_M \square \sigma_M$ es el *almacén del dominio mediador* $\mathcal{M}_{\mathcal{FD}.\mathcal{FS}}$ definido como una conjunción de restricciones primitivas atómicas $\Pi_M \subseteq APCon_{\mathcal{M}_{\mathcal{FD}.\mathcal{FS}}}$ y una sustitución σ_M. No se utilizan subíndices $\mathcal{FD}.\mathcal{FS}$ para aclarar la notación.

- $S = \Pi_S \square \sigma_S$ es el *almacén del dominio* $\mathcal{FS}$ donde $\Pi_S \subseteq APCon_{\mathcal{FS}}$ es un conjunto de restricciones primitivas atómicas y σ_S es una sustitución.

El cálculo de resolución de objetivos se divide en dos partes fundamentales: la parte correspondiente al estrechamiento perezoso que se mostró en la sección 4.2 y la parte específica de la cooperación entre dominios que se va a mostrar a continuación para el caso concreto de la cooperación entre $\mathcal{FD}$ y $\mathcal{FS}$. Al igual que en la cooperación entre $\mathcal{FD}$ y $\mathcal{R}$, la estrategia que se sigue para la cooperación entre $\mathcal{FD}$ y $\mathcal{FS}$ sigue los siguientes pasos que están representados por las reglas que se muestran en la tabla 6.2. Estos pasos son: en primer lugar, establecer puentes cuando sea posible (regla **SB**), después proyectar restricciones entre ambos dominios (regla **PP**) y por último enviar la restricción a su correspondiente almacén para que sea procesada por el correspondiente resolutor (regla **SC**).

La diferencia fundamental que existe entre las tablas 5.2 y 6.2 se encuentra en la regla **SB** y responde al hecho de que no se puede establecer un puente de cardinalidad para cualquier variable del dominio $\mathcal{FD}$, pues una variable entera en un programa no tiene por qué representar al cardinal de un conjunto. Es decir, la regla **SB** genera restricciones puente de cardinalidad en el almacén mediador M únicamente para variables que están involucradas en restricciones $\mathcal{FS}$. Si en el *pool* de restricciones hay una restricción primitiva atómica $\pi \in APCon_{\mathcal{FS}}$ entonces la función $bridges^{\mathcal{FS}\to\mathcal{FD}}(\pi, B)$ genera nuevos puentes para las variables contenidas en π si estos no estaban anteriormente en el conjunto de restricciones puentes B pertenecientes al almacén de restricciones del dominio mediador M. Esta generación de restricciones es posible porque cada variable conjunto de $\mathcal{FS}$ tiene asociada una variable $\mathcal{FD}$ que representa su cardinalidad. Formalmente:

$$bridges^{\mathcal{FS}\to\mathcal{FD}}(\pi, B) = \{C_i \texttt{ \#-- } S_i \mid S_i \in \pi \text{ y no está incluida en ningún puente de } B, \\ C_i \text{ es una variable nueva}\}.$$

SB Set Bridges

$\exists\overline{U}.P \sqcap \pi, C \sqcap M \sqcap H \sqcap F \sqcap S \Vdash_{\mathbf{SB}} \exists\overline{V}', \overline{U}.P \sqcap \pi, C \sqcap M' \sqcap H \sqcap F \sqcap S$

Si π es una restricción $\mathcal{FS}$ propia o si no una restricción extendida de $\mathcal{H}$ tal que $M \vdash \pi$ `in` $\mathcal{FS}$, y $M' = B', M$ donde $\exists\overline{V'}\, B' = bridges^{\mathcal{FS}\to\mathcal{FD}}(\pi, \Pi_M) \neq \emptyset$.

PP Propagate Projections

$\exists\overline{U}.P \sqcap \pi, C \sqcap M \sqcap H \sqcap F \sqcap S \Vdash_{\mathbf{PP}} \exists\overline{V'}, \overline{U}.P \sqcap \pi, C \sqcap M \sqcap H \sqcap F' \sqcap S'$

Si π es una restricción primitiva atómica y se cumplen o bien (i) o bien (ii)

(i) π es una restricción $\mathcal{FD}$ propia o si no una restricción extendida de $\mathcal{H}$ tal que $M \vdash \pi$ `in` $\mathcal{FD}$, $\exists\overline{V'}\,\Pi' = proj^{\mathcal{FD}\to\mathcal{FS}}(\pi, \Pi_M) \neq \emptyset$, $F' = F$ y $S' = \Pi', S$.

(ii) π es una restricción $\mathcal{FS}$ propia o si no una restricción extendida de $\mathcal{H}$ tal que $M \vdash \pi$ `in` $\mathcal{FS}$, $\exists\overline{V'}\,\Pi' = proj^{\mathcal{FS}\to\mathcal{FD}}(\pi, \Pi_M) \neq \emptyset$, $F' = \Pi', F$ y $S' = S$.

SC Submit Constraints

$\exists\overline{U}.P \sqcap \pi, C \sqcap M \sqcap H \sqcap F \sqcap S \Vdash_{\mathbf{SC}} \exists\overline{U}.P \sqcap C \sqcap M' \sqcap H' \sqcap F' \sqcap S'$

Si π es una restricción primitiva atómica y se cumple uno de los siguientes casos:

(i) π es una restricción de $\mathcal{M}_{\mathcal{FD}.\mathcal{FS}}$, $M' = \pi, M$, $H' = H$, $F' = F$, y $S' = S$.

(ii) π es una restricción extendida de Herbrand tal que ni $M \vdash \pi$ `in` $\mathcal{FD}$ ni $M \vdash \pi$ `in` $\mathcal{FS}$, $M' = M$, $H' = \pi, H$, $F' = F$ y $S' = S$.

(iii) π es una restricción propia de $\mathcal{FD}$ o si no una restricción extendida de Herbrand tal que $M \vdash \pi$ `in` $\mathcal{FD}$, $M' = M$, $H' = H$, $F' = \pi, F$ y $S' = S$.

(iv) π es una restricción propia de $\mathcal{FS}$ o si no una restricción extendida de Herbrand tal que $M \vdash \pi$ `in` $\mathcal{FS}$, $M' = M$, $H' = H$, $F' = F$ y $S' = \pi, S$.

Tabla 6.2: Reglas de generación de puentes y almacenes para el cálculo $CCLNC(\mathcal{C}_{\mathcal{FD}.\mathcal{FS}})$

Las proyecciones se pueden efectuar en ambos sentidos y, como ya se comentó en el capítulo anterior, las restricciones se proyectan dependiendo de la semántica y teniendo en cuenta los puentes disponibles. Las correspondientes funciones $proj^{\mathcal{FD}\to\mathcal{FS}}(\pi, B)$ y $proj^{\mathcal{FS}\to\mathcal{FD}}(\pi, B)$ se definen en las tablas 6.3 y 6.4

π	$proj^{\mathcal{FD}\to\mathcal{FS}}(\pi, B)$
C_1 `#<` C_2 (o bien `#>`, o bien `/=`)	$\{S_1$ `/=` $S_2 \mid (C_1$ `#--` $S_1), (C_2$ `#--` $S_2) \in B\}$

Tabla 6.3: Proyecciones desde $\mathcal{FD}$ a $\mathcal{FS}$

Las proyecciones del dominio $\mathcal{FD}$ al dominio $\mathcal{FS}$ infieren información sobre conjuntos a partir de su cardinal. Concretamente, se infiere que dos conjuntos son distintos si sus cardinales también son distintos. Se puede observar que para poder proyectar una restricción se necesita que todas las variables $\mathcal{FD}$ que representan cardinalidad deben disponer de sus correspondientes puentes a variables de tipo conjunto.

π	$proj^{\mathcal{FS}\to\mathcal{FD}}(\pi, B)$
S_1 == S_2	$\{C_1$ == $C_2 \mid (C_1$ #-- $S_1)$, $(C_2$ #-- $S_2) \in B\}$
domainSets [S_1,..,S_n] l u	$\{$c #<= C_i, C_i #<= c' $\mid$ (c #-- l), (c' #-- u), $(C_i$ #-- $S_i) \in B\}$
intSets [S_1,..,S_n] n m	$\{$0 #<= C_i, C_i #<= m-n+1 $\mid (C_i$ #-- $S_i) \in B\}$
subset S_1 S_2	$\{C_1$ #<= $C_2 \mid (C_1$ #-- $S_1)$,$(C_2$ #-- $S_2) \in B\}$
intersect S_1 S_2 S_3	$\{C_3$#<=C_1,C_3#<=$C_2\} \mid (C_i$ #-- $S_i) \in B\}$
union S_1 S_2 S_3	$\{C_3$ #<= C_1 #+ C_2, C_1 #<= C_3, C_2 #<= $C_3 \mid (C_i$ #-- $S_i) \in B\}$
disjoints [$S_1 \ldots S_n$]	C_1 #+ ... #+ C_n==$C \mid \cup_{k=1}^{n} S_k = S$, $(C$ #-- $S)$, $(C_i$ #-- $S_i) \in B\}$
El rango de i depende de las variables incluidas en π. Además, l, u $\in \mathcal{B}^{\mathcal{FS}}_{\mathtt{set}}$.	

Tabla 6.4: Proyecciones desde $\mathcal{FS}$ a $\mathcal{FD}$

Las proyecciones de $\mathcal{FS}$ a $\mathcal{FD}$ infieren información de los cardinales a partir de restricciones realizadas sobre conjuntos. Para poder proyectar una restricción se necesita que todas las variables de tipo conjunto dispongan de sus correspondientes puentes a variables que representan su cardinalidad. Estos puentes siempre existen pues, como se ha visto anteriormente, se generan automáticamente para todas las variables $\mathcal{FS}$ que aparecen en alguna restricción de conjuntos. Las proyecciones de las restricciones `intersect` y `union` se extienden a las restricciones `intersections, unions :: [set] -> set -> bool`.

Las funciones de creación de puentes y proyecciones definidas mediante las tablas 6.3 y 6.4 tienen propiedades de corrección y completitud similares a las vistas en el teorema 6. Además de las reglas de generación de puentes y restricciones que se acaban de mostrar, el cálculo $CCLNC(\mathcal{C}_{\mathcal{FD}.\mathcal{FS}})$ se extiende con las reglas de la tabla 6.5.

IE Infer Equality

$\exists\overline{U}.P \sqcup C \sqcup (I_1$ #-- S_1, I_2 #-- S_1,$M) \sqcup H \sqcup F \sqcup S \vdash_{\mathbf{IE}}$

$\exists\overline{U}.P \sqcup C \sqcup (I_1$ #-- S_1,$M) \sqcup H \sqcup (I_1$==I_2,$F) \sqcup S$

IF Infer Failure

$\exists\overline{U}.P \sqcup C \sqcup (I_1$ #-- S_1, I_2 #-- S_2,$M) \sqcup H \sqcup (I_1$/=I_2,$F) \sqcup (S_1$==S_2, $S) \vdash_{\mathbf{IF}} \blacksquare$

Tabla 6.5: Reglas de inferencia de igualdad y fallo a partir de restricciones puentes

La regla **IE** de la tabla 6.5 se aplica cuando en el almacén mediador se tienen dos puentes que representan al mismo conjunto. En este caso se deja un único puente y se establece la restricción de igualdad entre cardinales. La regla **IF** deduce el fallo si dos puentes sobre conjuntos restringidos a ser iguales tienen cardinalidad distinta.

Para completar las reglas del cálculo $CCLNC(\mathcal{C}_{\mathcal{FD}.\mathcal{FS}})$, falta la regla que invoca al resolutor del dominio $\mathcal{FS}$, pues la invocación al resolutor del dominio $\mathcal{H}$ se encuentra en la tabla 4.2, al igual que el resolutor del dominio mediador. La invocación al resolutor del dominio $\mathcal{FS}$ se encuentra en la tabla 6.6.

En resumen, el cálculo $CCLNC(\mathcal{C}_{\mathcal{FD}.\mathcal{FS}})$ está formado por las reglas de estrechamiento perezoso de la tabla 4.1, las reglas de generación de puentes y proyecciones (tablas 6.2, 6.3

SetS $\mathcal{FS}^E$-Constraint Solver (caja negra)

$\exists\overline{U}.\ P \sqcup C \sqcup M \sqcup H \sqcup F \sqcup S \vdash_{\mathbf{SetS}} \exists\overline{Y'},\overline{U}.\ (P \sqcup C \sqcup M \sqcup H \sqcup F \sqcup (\Pi' \sqcup \sigma_S))\ @_{\mathcal{FS}}\sigma'$

Si $pvar(P) \cap var(\Pi_S) = \emptyset$, $S = (\Pi_S \sqcup \sigma_S)$ no está resuelto, $\Pi_S \vdash_{solve^{\mathcal{FS}}} \exists\overline{Y'}(\Pi' \sqcup \sigma')$.

Tabla 6.6: Regla de resolución de restricciones del almacén FS

y 6.4), las reglas que infieren restricciones de Herbrand a partir de restricciones puentes (tabla 6.5) y las invocaciones a los correspondientes resolutores definidos en las tablas 4.2, 5.6 (únicamente dominio $\mathcal{FD}$) y 6.6. El cálculo permite que se aplique cualquier regla de transformación que se pueda aplicar. Esto implica distintos cómputos, sin embargo $\mathcal{TOY}$ sigue una estrategia concreta que se esbozó en la sección 5.2 para el caso de cooperación de los dominios $\mathcal{FD}$ y $\mathcal{R}$, y es la misma que sigue la cooperación de los dominios $\mathcal{FD}$ y $\mathcal{FS}$.

En el siguiente ejemplo se muestra en detalle el funcionamiento del cálculo $CCLNC(\mathcal{C}_{\mathcal{FD}.\mathcal{FS}})$ con una función que se utiliza en el problema de las ternas de Steiner, que se muestra más adelante.

Ejemplo 8. Aplicación del cálculo $CCLNC(\mathcal{C}_{\mathcal{FD}.\mathcal{FS}})$ en la resolución de un objetivo

Dada una lista de variables de tipo conjunto definidas sobre un rango, La función `atMostOne` definida a continuación obliga a que todos los conjuntos dados en una lista tengan cardinal 3 y que cualquier par de conjuntos de la lista tenga como mucho un elemento en común.

```
        atMostOne :: [set] -> bool
        atMostOne [] = true
        atMostOne [X|Xs] = atMostOne Xs <==
a             3 #-- X,
b             andL (map (sendCon X) Xs)

        sendCon :: set -> set -> bool
        sendCon SX SY = true <==
c             intersect SX SY SZ,
d             Z #-- SZ, Z #<= 1

        andL :: [bool] -> bool
        andL = foldr (/\) true
```

La función `atMostOne` restringe la cardinalidad de los conjuntos al valor 3 (línea a). Además, usa la función predefinida de orden superior `map` (línea b) que recibe como primer argumento la aplicación parcial `sendCon X` y como segundo argumento la lista `Xs`. Como resultado, `map` devuelve una lista de valores Booleanos. Para que `atMostOne` tenga éxito, la lista devuelta por `map` debe contener solamente valores `true`, lo que se verifica con la función `andL`. En las líneas c y d se impone que dos conjuntos tengan a lo sumo un elemento en común. Un posible objetivo es:

`L==[S1,S2]`, `domainSets L { } {1,2,3,4,5}`, `atMostOne L`, `labelingSets L` [1]

donde `domainSets L { } {1,2,3,4,5}` asigna respectivamente a las variables de conjunto `S1` y `S2` retículos definidos por el ínfimo `{}` y el supremo `{1,2,3,4,5}`. Después se aplica la función `atMostOne` y finalmente `labelingSets` enumera todos los valores básicos para las variables `S1` y `S2`. Algunas respuestas para el objetivo anterior son:

```
S1 -> {1,2,3}, S2 -> {1,4,5}
S1 -> {1,2,3}, S2 -> {2,4,5}
              .....
S1 -> {2,3,5}, S2 -> {1,2,4}
```

A continuación se muestran los pasos que el cálculo de resolución de objetivos aplica en el objetivo anterior. Un objetivo sobre el dominio de coordinación $\mathcal{C}_{\mathcal{FD}.\mathcal{FS}}$ tiene la forma $G \equiv \exists \overline{U}.P \sqcup C \sqcup M \sqcup H \sqcup F \sqcup S$ como se explicó en la página 73. Las sustituciones no se muestran para evitar la sobrecarga de notación. Inicialmente el objetivo [1] está ubicado en el *pool* de restricciones.

$$\emptyset \sqcup \overbrace{\texttt{L==[S1,S2]}}^{\pi_1}, \overbrace{\texttt{domainSets L \{\} \{1,2,3,4,5\}}}^{\pi_2}, \overbrace{\texttt{atMostOne L}}^{\pi_3}, \texttt{labelingSets L} \sqcup \emptyset \sqcup \emptyset \sqcup \emptyset \sqcup \emptyset$$

La restricción π_1 se envía al almacén de $\mathcal{H}$ y el resolutor de $\mathcal{H}$ se invoca produciendo la sustitución $\sigma_1 = \{\texttt{L} \mapsto \texttt{[S1,S2]}\}$. Después, π_2 se procesa utilizando las reglas de la tabla 6.2: primero, la regla **SB** genera las restricciones puente de cardinalidad `I1 #-- S1` e `I2 #-- S2` de acuerdo con la función $bridges^{\mathcal{FS}\rightarrow\mathcal{FD}}$. Las variables `I1` e `I2` son nuevas y las nuevas restricciones puente se añaden al almacén mediador:

$$bridges^{\mathcal{FS}\rightarrow\mathcal{FD}}(\pi_2, \emptyset) = \{\texttt{I1 \#-- S1}, \texttt{I2 \#-- S2}\}$$
$$B = \{\texttt{I1 \#-- S1}, \texttt{I2 \#-- S2}\}$$

Después, la regla **PP** construye nuevas restricciones $\mathcal{FD}$ correspondientes a la proyección de π_2:

$$proj^{\mathcal{FS}\rightarrow\mathcal{FD}}(\pi_2, B) = \texttt{\{0\#<=I1, I1\#<=5, 0\#<=I2, I2\#<=5\}}$$

Una vez realizada la generación de puentes y proyectada la restricción π_2, esta se envía a su correspondiente almacén aplicando la regla **SC**. El resolutor $\mathcal{FS}$ se invoca restringiendo los dominios de las variables `S1` y `S2` a ser retículos definidos por `{}` y `{1,2,3,4,5}`. El objetivo queda transformado en:

$$\exists\ \texttt{I1,I2}\ \emptyset \sqcup \texttt{atMostOne [S1,S2]}, \texttt{labelingSets [S1,S2]} \sqcup \texttt{I1 \#-- S1}, \texttt{I2 \#-- S2} \sqcup \emptyset \sqcup \texttt{0\#<=I1,I1\#<=5,0\#<=I2,I2\#<=5} \sqcup \pi_2$$

Ahora π_3 se aplana aplicando la definición de la función `atMostOne`. Este aplanamiento produce nuevas restricciones primitivas atómicas (se omiten los pasos intermedios de aplanamiento):

$$\exists\ \texttt{I1,I2,S12,I12}.\emptyset \square \overbrace{\texttt{3 \#-- S1}}^{\pi_4}, \overbrace{\texttt{3 \#-- S2}}^{\pi_5}, \overbrace{\texttt{intersect S1 S2 S12}}^{\pi_6}, \overbrace{\texttt{I12 \#-- S12}}^{\pi_7},$$
$$\overbrace{\texttt{I12 \#<= 1}}^{\pi_8}, \overbrace{\texttt{labelingSets [S1,S2]}}^{\pi_9}$$
$$\square\ \texttt{I1 \#-- S1, I2 \#-- S2} \square \emptyset \square \texttt{0\#<=I1, I1\#<=5, 0\#<=I2, I2\#<=5} \square \pi_2$$

Continuando con las restricciones del *pool*, π_4 y π_5 se envían al almacén mediador, y se invoca su correspondiente resolutor $solve^{\mathcal{M}}$. Después, la restricción π_6 se procesa de forma similar a π_2, generando un nuevo puente y nuevas restricciones $\mathcal{FD}$:

$$bridges^{\mathcal{FS}\to\mathcal{FD}}(\pi_6, B) = \{\texttt{I12 \#-- S12}\} = B'$$
$$proj^{\mathcal{FS}\to\mathcal{FD}}(\pi_6, B'') = \{\texttt{I12\#<= I1, I12\#<= I2}\}, \text{ donde } B'' = B \cup B'$$

Entonces, la regla **SC** se aplica a π_7, π_8 y π_9, enviando cada restricción a su correspondiente almacén, $\mathcal{M}_{\mathcal{FD}.\mathcal{FS}}$, $\mathcal{FD}$ y $\mathcal{FS}$, respectivamente. Finalmente, P y C están vacíos.

$$\exists\ \texttt{I1,I2,S12,I12}.\emptyset \square \emptyset \square \texttt{I1 \#-- S1, I2 \#-- S2}, \pi_4, \pi_5, \pi_7 \square \emptyset \square \pi_8, \texttt{0\#<=I1, I1\#<=5,}$$
$$\texttt{0\#<=I2, I2\#<=5, I12\#<=I1, I12\#<=I2}\ \square\ \pi_2, \pi_6, \pi_9.$$

Aplicando dos veces la regla **IE** de la tabla 6.5 a las restricciones `I1 #-- S1`, $\pi_4 \equiv$ `3 #-- S1`, `I2 #-- S2`, y $\pi_5 \equiv$ `3 #-- S2`, es posible deducir las igualdades `I1 == 3` e `I2 == 3`. Estas igualdades se envían al almacén de $\mathcal{H}$, produciendo las sustituciones $\sigma_4 = \{\texttt{I1} \mapsto \texttt{3}, \texttt{I2} \mapsto \texttt{3}\}$. Ahora el objetivo queda reducido a:

$$\exists\ \texttt{I1,I2,S12,I12}.\emptyset \square \emptyset \square \texttt{3 \#-- S1, 3 \#-- S2}, \pi_7 \square \emptyset \square \pi_8, \texttt{I12\#<=3} \square \pi_2, \pi_6, \pi_9$$

Por último, la restricción `labelingSets` enumera todos posibles los valores de `S1` y `S2`. Un primer intento es $\texttt{S1} \mapsto \texttt{\{1,2,3\}}$ y $\texttt{S2} \mapsto \texttt{\{1,2,3\}}$, pero esta sustitución no satisface π_6, π_7 y π_8.

El resto de las valoraciones se asignan por vuelta atrás y se verifican hasta que se obtiene una solución, como por ejemplo $\texttt{S1} \mapsto \{1,2,3\}$, $\texttt{S2} \mapsto \{1,4,5\}$. Si el usuario pide más soluciones, se aplica vuelta atrás para calcularlas.

El ejemplo que se acaba de mostrar describe detalladamente la cooperación entre los dominios $\mathcal{FD}$ y $\mathcal{R}$. La función `atMostOne` forma parte de de un problema clásico en la programación de restricciones de conjuntos: el problema de las ternas de Steiner [SG01, LS04a, HKW04, Aze07a], que se describe brevemente a continuación. El cálculo $CCLNC(\mathcal{C}_{\mathcal{FD}.\mathcal{FS}})$ se aplica de forma similar al resto de funciones del problema de Steiner.

Ejemplo 9. Problema de las ternas Steiner

El problema de las ternas de Steiner de orden `n` consiste en encontrar `n(n-1)/6` conjuntos de cardinal 3, con elementos comprendidos entre `1` y `n` y de tal manera que dos conjuntos cualesquiera tengan a lo sumo un elemento en común.

Por ejemplo, una solución para `n=7` es: `{1,2,3}`, `{1,4,5}`, `{1,6,7}`, `{2,4,6}`, `{2,5,7}`, `{3,4,7}`, `{3,5,6}`. Existen soluciones para `n = 3, 7, 9, 13, 15, 19, ...`

El siguiente código $\mathcal{TOY}$ resuelve este problema con `n` como parámetro de entrada:

```
steiner :: int -> [iset]
steiner N = L <==
    M == round ((N * (N - 1)) / 6),
    hasLength L M,
    intSets L 1 N,
    atMostOne L,
    labelingSets L

hasLength :: [A] -> int -> bool
hasLength [] 0 :- true
hasLength [X|Xs] N :- N > 0, hasLength Xs (N - 1)
```

La función `steiner` devuelve como resultado la lista `L` de longitud `M` cuyos elementos se restringen a ser conjuntos definidos por el retículo con ínfimo el conjunto vacío y supremo el conjunto `{1,...,N}`. La función `hasLength L M` genera en `L` una lista de variables de longitud `M`. La función `atMostOne`, definida en el ejemplo 8, obliga a que todos los conjuntos tengan cardinal 3 y que dos conjuntos cualesquiera tengan a lo sumo un elemento en común.

6.3 Resultados de corrección y completitud limitada

Los siguientes teoremas presentan los resultados semánticos de corrección y completitud limitada para el cálculo $CCLNC(\mathcal{C}_{\mathcal{FD}.\mathcal{FS}})$ que se acaba de describir. Estos resultados son análogos a los resultados semánticos de corrección y completitud limitada del cálculo $CCLNC(\mathcal{C}_{\mathcal{FD}.\mathcal{R}})$ que se ilustraron en la sección 5.3. Aunque son semejantes se ha decidido adjuntarlos para dar una visión completa y lo más independiente posible del cálculo $CCLNC(\mathcal{C}_{\mathcal{FD}.\mathcal{FS}})$. En primer lugar se estudia en el siguiente teorema la corrección y completitud limitada en un paso de transformación.

Teorema 11. (Corrección local y completitud local lim. del cálculo $CCLNC(\mathcal{C}_{\mathcal{FD}.\mathcal{FS}})$)

Sea $\mathcal{P}$ un $CFLP(\mathcal{C}_{\mathcal{FD}.\mathcal{FS}})$ programa y G un objetivo admisible que no está en forma resuelta. Se elige una regla **RL** de las tablas 4.1, 4.2, 6.2, 6.5 y 6.6 aplicable a G y se selecciona una parte γ de G sobre la que se aplica la regla **RL**. Entonces existe un número finito de transformaciones $G \vdash_{\mathbf{RL},\gamma,\mathcal{P}} G'_j$ $(1 \leq j \leq k)$, y además:

1. **Corrección local:** $Sol_{\mathcal{P}}(G) \supseteq \bigcup_{j=1}^{k} Sol_{\mathcal{P}}(G'_j)$.
2. **Completitud local limitada:** $WTSol_{\mathcal{P}}(G) \subseteq \bigcup_{j=1}^{k} WTSol_{\mathcal{P}}(G'_j)$, donde la aplicación de la regla **RL** a la parte seleccionada γ de G es *segura* en el siguiente sentido:

no es una aplicación opaca de **DC** ni es una aplicación de una regla de las tablas 4.2 o 6.6 que implique una invocación incompleta al resolutor.

La demostración de este teorema se encuentra en el apéndice A.9.1.

El resultado global de corrección se obtiene directamente a partir del correspondiente resultado local que se acaba de enunciar. La corrección global se enuncia a continuación y su demostración se omite pues es semejante a la demostración del teorema de corrección del cálculo $CCLNC(\mathcal{C}_{\mathcal{FD}.\mathcal{R}})$, teorema 8 del capítulo 5.

Teorema 12. (Corrección del cálculo $CCLNC(\mathcal{C}_{\mathcal{FD}.\mathcal{FS}})$)

Sea $\mathcal{P}$ un programa $CFLP(\mathcal{C}_{\mathcal{FD}.\mathcal{FS}})$, G un objetivo para $\mathcal{P}$ y S un objetivo resuelto tal que $G \vdash^*_{\mathcal{P}} S$ utilizando las reglas de las tablas 4.1, 4.2, 6.2, 6.5 y 6.6. Entonces, $Sol_{\mathcal{C}_{\mathcal{FD}.\mathcal{FS}}}(S) \subseteq Sol_{\mathcal{P}}(G)$.

La completitud asegura que, bajo ciertas limitaciones, cualquier solución bien tipada puede ser obtenida por el cálculo $CCLNC(\mathcal{C}_{\mathcal{FD}.\mathcal{FS}})$. Para demostrar la completitud se va a seguir la metodología de demostración que se hizo en el cálculo $CCLNC(\mathcal{C}_{\mathcal{FD}.\mathcal{R}})$, donde se demuestra la completitud limitada (ibíd., teorema 9) a partir del resultado de completitud local (ibíd., teorema 8), asegurando la terminación de las soluciones bien tipadas en formas resueltas por medio de un lema de progreso (ibíd., lema 4) con un orden de progreso bien fundado $\blacktriangleright$. A continuación se muestra el orden de progreso bien fundado y el lema de progreso específicos del cálculo $CCLNC(\mathcal{C}_{\mathcal{FD}.\mathcal{FS}})$.

Orden de progreso bien fundado $\blacktriangleright$ para el cálculo $CCLNC(\mathcal{C}_{\mathcal{FD}.\mathcal{FS}})$

El *orden de progreso bien fundado* $\blacktriangleright$ se define entre pares $(G, \mathcal{M})$ formados por un objetivo G y un testigo $\mathcal{M}$ [1]. Para ello, se define una tupla $||(G, \mathcal{M})|| = (O_1, O_2, O_3, O_4, O_5, O_6)$ donde O_1 es un multiconjunto finito de números naturales y $O_2, \ldots, O_6$ son números naturales. Los elementos O_1 a O_4 se definen de la misma forma que en la página 103 de la sección 5.3. Además:

- $\mathbf{O_5}$ es la suma $sf_M + sf_H + sf_{FD} + sf_{FS}$ donde sf_M es el valor 1 si la regla **MS** que invoca al resolutor de la tabla 4.2 se puede ser aplicar a G, y 0 en otro caso. Los otros sumandos se definen análogamente, para los demás dominios. Obsérvese que se ha modificado este orden para incluir el dominio $\mathcal{FS}$.

- $\mathbf{O_6}$ es el número de puentes del almacén mediador M.

Sea el producto lexicográfico $>_{lex}$ definido para los órdenes $>_i$ $(1 \leq i \leq 6)$, donde $>_1$ es el orden del multiconjunto $>_{mul}$ sobre los multiconjuntos de los números naturales, y el

[1] De acuerdo a la lógica de reescritura *CRWL* [LRV07], $\mu \in WTSol_{\mathcal{P}}(G)$ implica la existencia de un testigo $\mathcal{M} : \mu \in WTSol_{\mathcal{P}}(G)$.

resto de los $>_i$ es el orden $>$ sobre los números naturales. Finalmente, se define el orden de progreso $\blacktriangleright$ como $(G, \mathcal{M}) \blacktriangleright (G', \mathcal{M}')$ si y solamente si $||(G, \mathcal{M})|| >_{lex} ||(G', \mathcal{M}')||$. Como se demuestra en [BN98], $>_{mul}$ es un orden bien fundado y el producto lexicográfico es también un orden bien fundado. Por lo tanto, $\blacktriangleright$ es bien fundado.

El siguiente lema de progreso es semejante al lema de progreso 4. Sin embargo se incluye pues su demostración se basa en los órdenes anteriormente definidos.

Lema 5. (Lema de progreso del cálculo $CCLNC(\mathcal{C}_{\mathcal{FD}.\mathcal{FS}})$)

Sea G un objetivo admisible que no está en forma resuelta para un programa $\mathcal{P}$ y un testigo $\mathcal{M} : \mu \in WTSol_{\mathcal{P}}(G)$. Se asume que ni $\mathcal{P}$ ni G tienen variables libres de orden superior y que G no está en forma resuelta. Entonces:

1. Existe alguna regla `RL` aplicable a G y no es una regla de fallo.
2. Para cualquier elección de una regla **RL** que no sea una regla de fallo y una parte γ de G, tal que **RL** se aplica a γ de una manera segura, existe un número finito de computaciones $G \vdash^{+}_{\mathbf{RL},\gamma,\mathcal{P}} G'$ tal que:
   - $\mu \in WTSol_{\mathcal{P}}(G')$.
   - Existe un testigo $\mathcal{M}' : \mu \in WTSol_{\mathcal{P}}(G')$ que verifica $(G, \mathcal{M}) \blacktriangleright (G', \mathcal{M}')$.

La demostración del lema 5 se encuentra en el apéndice A.9.2. Una vez probado que se pueden realizar pasos de cómputo $CCLNC(\mathcal{C}_{\mathcal{FD}.\mathcal{FS}})$, se puede demostrar por inducción sobre $\blacktriangleright$ que dado un objetivo G para un programa $\mathcal{P}$ y asumiendo que $\mathcal{P}$ y G son seguros, así como la aplicación de las reglas, entonces se cumple lo siguiente. Si $\mu \in WTSol_{\mathcal{P}}(G)$, se puede encontrar un cómputo $CCLNC(\mathcal{C}_{\mathcal{FD}.\mathcal{FS}})$ de la forma $G \vdash^{*}_{\mathcal{P}} S$, terminando con un objetivo en forma resuelta S tal que $\mu \in WTSol_{\mathcal{C}_{\mathcal{FD}.\mathcal{FS}}}(S)$. Es decir, a partir del lema de progreso se puede establecer la completitud limitada global del cálculo $CCLNC(\mathcal{C}_{\mathcal{FD}.\mathcal{FS}})$ mediante el siguiente teorema que no se va a demostrar por su semejanza en la demostración con el teorema 9.

Teorema 13. (Completitud limitada del cálculo $CCLNC(\mathcal{C}_{\mathcal{FD}.\mathcal{FS}})$)

Sea G un objetivo para un programa $\mathcal{P}$ y $\mu \in WTSol_{\mathcal{P}}(G)$ una solución bien tipada para dicho objetivo. Asumimos que ni $\mathcal{P}$ ni G contienen apariciones libres de variables de orden superior y las aplicaciones de las reglas de las tablas 4.1, 4.2, 6.2, 6.5 y 6.6 son seguras, es decir, no producen descomposiciones opacas ni invocaciones incompletas de resolutores. Entonces se puede encontrar un cómputo $CCLNC(\mathcal{C}_{\mathcal{FD}.\mathcal{FS}})$ de la forma $G \vdash^{*}_{\mathcal{P}} S$, terminando con un objetivo en forma resuelta S tal que $\mu \in WTSol_{\mathcal{C}_{\mathcal{FD}.\mathcal{FS}}}(S)$.

6.4 Implementación

El sistema $\mathcal{TOY}$ se ha extendido con los resolutores de ECL^{i}PS^e para tratar la cooperación entre $\mathcal{FD}$ y $\mathcal{FS}$. Dado que ECL^{i}PS^e incluye un dominio finito estrechamente integrado con el dominio de conjuntos, se ha reemplazado el dominio finito existente en $\mathcal{TOY}$ (proporcionado por el sistema SICStus) por el dominio finito de ECL^{i}PS^e. Este prototipo, disponible en la página web `http://gpd.sip.ucm.es/sonia/Prototype.html`, tiene tres modos de uso:

- El denominado modo ***interactivo*** que está orientado al razonamiento de modelos, es decir, cuando el modelo puede ser modificado durante la resolución del objetivo. Este modo está basado en una comunicación interactiva entre $\mathcal{TOY}$ y ECL^{i}PS^e, como su propio nombre indica. Cuando $\mathcal{TOY}$ necesita que una restricción sea resuelta, se envía al servidor de ECL^{i}PS^e y espera su respuesta. Ambos sistemas interactúan de la siguiente forma: cuando en $\mathcal{TOY}$ se pide cargar los resolutores de ECL^{i}PS^e mediante el comando `/cflp_ic_sets`, se arranca un proceso separado en el sistema ECL^{i}PS^e ejecutando un programa Prolog específico para la interacción con $\mathcal{TOY}$. Este proceso ECL^{i}PS^e actúa como servidor aceptando y ejecutando las peticiones que le son enviadas desde el proceso $\mathcal{TOY}$. Es decir, durante el cómputo de un objetivo $\mathcal{TOY}$, por cada restricción que necesita ser evaluada se hace una petición al servidor ECL^{i}PS^e, el servidor evalúa la restricción y devuelve el resultado a $\mathcal{TOY}$. Supongamos que estas restricciones primitivas atómicas que se obtienen del proceso de estrechamiento son: π_1, π_2, ..., π_n. Como se muestra en la figura 6.1, cuando el cómputo de $\mathcal{TOY}$ obtiene la restricción primitiva atómica π_1, ésta se envía al proceso ECL^{i}PS^e para que la evalúe y espera su respuesta para continuar con el cómputo. De la misma manera se procede con las demás restricciones primitivas atómicas π_2, ..., π_n.

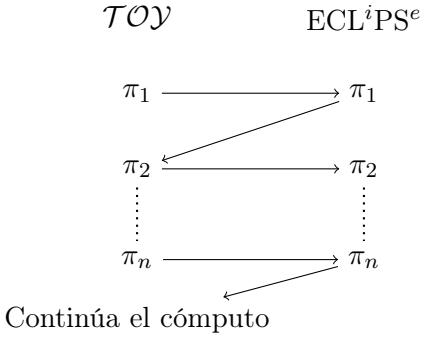


Figura 6.1: Flujo de datos en el modo interactivo

Para comunicar ambos procesos se ha utilizado la biblioteca de entrada/salida estándar a través de un canal (*pipe*) que conecta ambos procesos. Este enfoque aprovecha todas las características del lenguaje $\mathcal{TOY}$ y la nueva cooperación. Sin embargo, esta comunicación entre $\mathcal{TOY}$ y ECL^{i}PS^e por cada restricción tiene un efecto negativo sobre la eficiencia de resolución.

- El modo por lotes (***batch***) es similar al modo interactivo pero, en vez de esperar la respuesta de ECL^{i}PS^e cada vez que se envía una restricción desde $\mathcal{TOY}$, espera a que todas las restricciones o un grupo de ellas estén enviadas para recoger la respuesta. La forma de marcar el bloque de restricciones que se desean evaluar juntas se lleva a cabo mediante las funciones de aridad 0 `batch` y `nobatch`, que siempre devuelven el valor `true`. Si el objetivo se acaba y no se encuentra `nobatch` entonces se asume por defecto. De forma similar al modo anterior, supongamos un objetivo $\mathcal{TOY}$ que en su proceso de estrechamiento obtiene una serie de restricciones primitivas atómicas $\pi_1, \ldots, \pi_n$ y que en medio de esta computación están `batch` y `nobatch` dispuestas de la siguiente forma:

  $$\pi_1, \texttt{batch}, \pi_2, \ldots, \pi_m, \texttt{nobatch}, \pi_{m+1}, \ldots, \pi_n$$

  Como se muestra en el esquema de la izquierda de la figura 6.2, la restricciones π_1, π_{m+1}, $\ldots$, π_n se procesan de forma interactiva mientras que las restricciones π_2, $\ldots$, π_m se almacenan en el fichero `saveConstraints.pl` y se procesan en bloque cuando se encuentra `nobatch` durante el cómputo en $\mathcal{TOY}$.

  Si la resolución del objetivo encuentra la secuencia `batch`, $\pi_1, \ldots, \pi_n$, entonces todas las restricciones son evaluadas en bloque como se muestra en el esquema de la derecha de la figura 6.2.

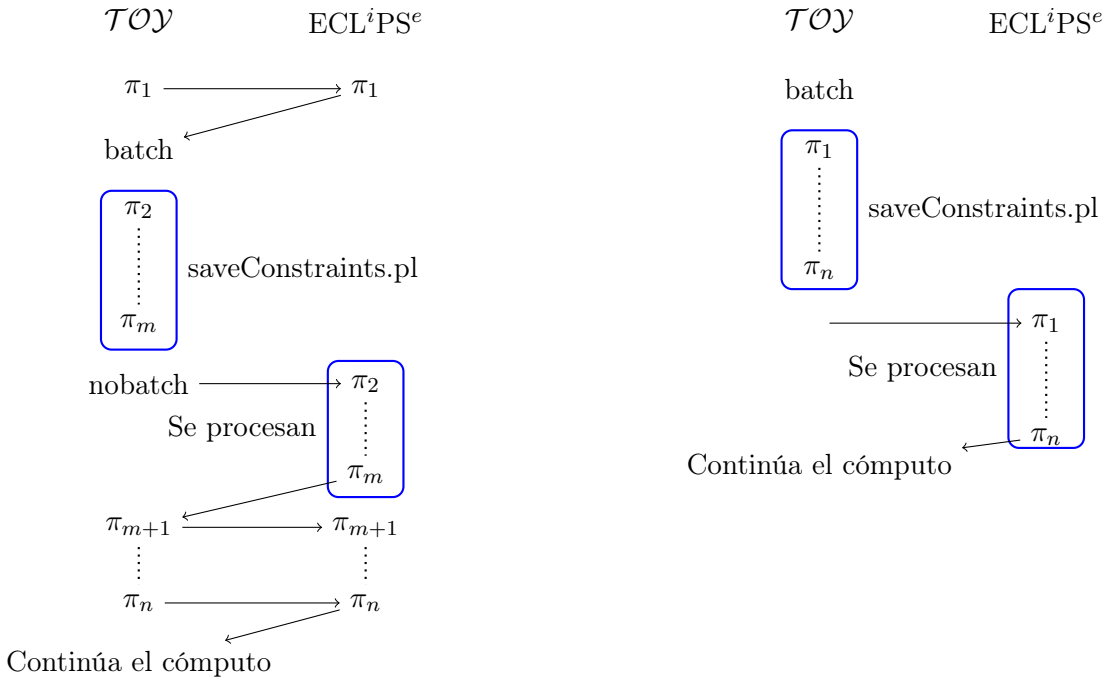


Figura 6.2: Flujo de datos en el modo batch

Este enfoque está destinado a aplicaciones clásicas CP, donde las restricciones primero se especifican y después se envían al resolutor donde se resuelven todas de una vez. Este modo evita la comunicación interactiva entre ambos procesos que puede ralentizar

la ejecución de los objetivos. Sin embargo, la comunicación es inevitable cuando se requieren más respuestas. Así, la vuelta atrás fuerza la comunicación interactiva entre ambos procesos, a pesar de que las restricciones se envían por lotes.

- El modo ECL^{i}PS$^e_{gen}$ mide el tiempo que ECL^{i}PS^e tarda en resolver el conjunto de restricciones enviadas por $\mathcal{TOY}$, eliminando cualquier sobrecarga producida por el estrechamiento y la comunicación entre procesos. Esto se ha hecho generando un programa ECL^{i}PS^e que contiene todas las restricciones primitivas atómicas creadas por el proceso de estrechamiento de $\mathcal{TOY}$, y que se ejecutará aisladamente en ECL^{i}PS^e para la medición. La figura 6.3 muestra este proceso para una serie de restricciones primitivas atómicas $\pi_1, \ldots, \pi_n$ obtenidas del proceso de estrechamiento de $\mathcal{TOY}$.

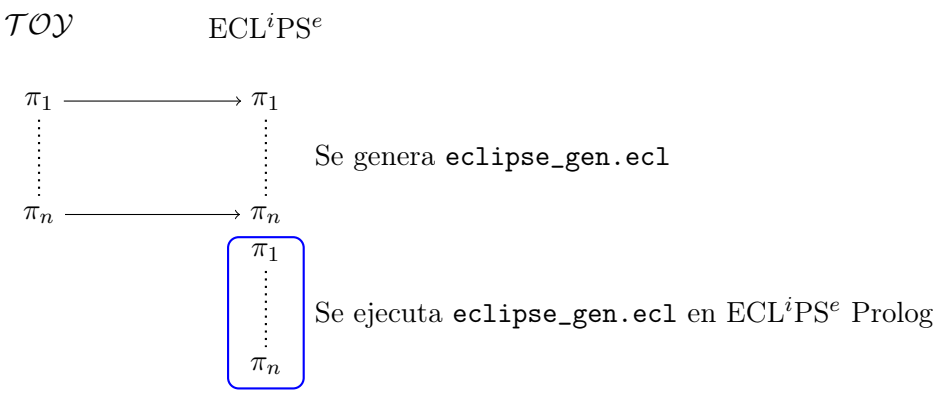


Figura 6.3: Flujo de datos en el modo ECL^{i}PS$^e_{gen}$

Para motivar el uso de los distintos modos, volvemos al clásico ejemplo de la definiciones de `double` y `coin` descrito en la subsección 1.3.2. En el modo ECL^{i}PS$^e_{gen}$ el objetivo `double coin == L` únicamente devuelve un resultado siendo incompleta la ejecución, pues para poder evaluar la función `double` se necesita evaluar primero su argumento `coin`. Una vez que se ha evaluado la función `double` se colecta la restricción de igualdad y se escribe el fichero ECL^{i}PS^e que posteriormente se ejecutará. Este objetivo muestra que el indeterminismo es un claro ejemplo del uso del modo interactivo y para el modo ECL^{i}PS$^e_{gen}$ es inviable. Obsérvese que las operaciones aritméticas entre valores básicos, es decir números, no son restricciones y por lo tanto las realiza $\mathcal{TOY}$ y no se envían al resolutor de ECL^{i}PS^e.

6.4.1 Arquitectura

La figura 6.4 muestra los componentes arquitectónicos del esquema de cooperación $\mathcal{C}_{\mathcal{FD}.\mathcal{FS}}$.

Los resolutores de caja transparente para los dominios $\mathcal{H}$ y $\mathcal{M}_{\mathcal{FD}.\mathcal{R}}$ y sus correspondientes almacenes de restricciones están encapsulados en el código de $\mathcal{TOY}$, que está implementado en SICStus Prolog. Por otra parte los resolutores $\mathcal{FD}$ y $\mathcal{FS}$ están divididos en dos capas: la primera capa corresponde a las cajas correspondientes definidas como $solve^{\mathcal{FD}^{\mathcal{T}}}$ y $solve^{\mathcal{FS}^{\mathcal{T}}}$, respectivamente; la segunda capa es una caja negra y corresponde a las invocaciones de las

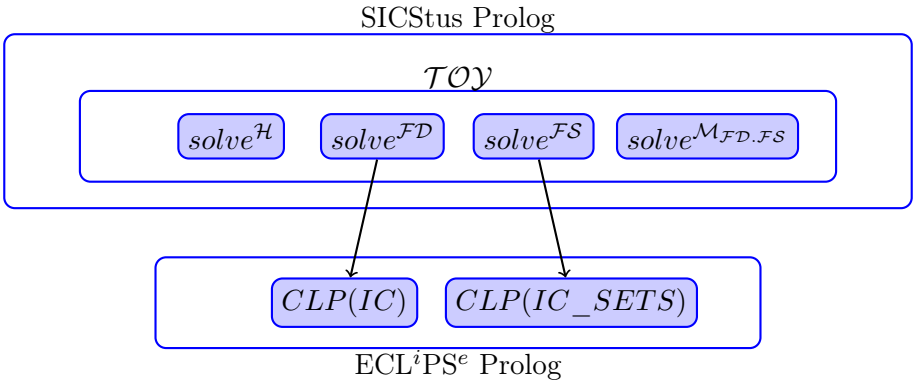


Figura 6.4: Componentes arquitectónicos del esquema de cooperación $\mathcal{C}_{\mathcal{FD}.\mathcal{FS}}$ en $\mathcal{TOY}$

bibliotecas `ic` e `ic_sets` del sistema ECL^{i}PS^e. La implementación que une $\mathcal{TOY}$ a estas bibliotecas se detallará más adelante.

El sistema $\mathcal{TOY}$ está implementado en SICStus Prolog, pero las restricciones primitivas atómicas que se obtienen de la computación del cálculo $CCLNC(\mathcal{C}_{\mathcal{FD}.\mathcal{FS}})$ se envían al sistema ECL^{i}PS^e, por lo tanto hay que enviar estas restricciones del proceso SICStus al proceso ECL^{i}PS^e. Sin embargo, no se pueden enviar directamente las variables del cómputo, pues se perdería la conexión entre las distintas restricciones. Esto se debe a que el sistema ECL^{i}PS^e crea variables nuevas para las variables que se le envían a través de un flujo de entrada. Para evitar este problema, a cada variable $\mathcal{TOY}$ se le asigna un identificador para poder relacionar las variables en ambos procesos. La forma de guardar estas relaciones es la siguiente:

- En la parte de SICStus se han utilizado atributos en las variables, en lo que se almacenan sus identificadores, además de otros parámetros específicos para realizar la cooperación que se explicarán más adelante cuando sea necesario. Por otra parte, cuando ECL^{i}PS^e devuelve el resultado de procesar las restricciones, esta información viene dada mediante los identificadores pues como se ha comentado anteriormente no se envían variables entre procesos. Por lo tanto, el proceso SICStus necesita reemplazar los identificadores que le envía ECL^{i}PS^e por sus correspondientes variables. Para hacer este reemplazamiento se guarda en el almacén mixto la relación de la variable con su identificador.

- En la parte de ECL^{i}PS^e, en los modos interactivo y batch se mantiene una tabla que relaciona cada identificador con una variable. La primera vez que recibe un identificador del proceso de SICStus crea una relación con una variable nueva y posteriormente utiliza siempre esta relación. En el modo ECL^{i}PS$^e_{gen}$ no es necesario mantener esta tabla pues los identificadores hacen las funciones de las propias variables.

Así, por ejemplo, el objetivo `domain [X,Y] 1 10, X #<Y`, que se ejecuta en modo interactivo, asigna a las variables `X` e `Y` los identificadores `'$fdvar'(1)` y `'$fdvar'(2)`. De esta forma el proceso ECL^{i}PS^e recibe las siguientes restricciones:

```
domainFdE('$fdvar'(1),1,10)
domainFdE('$fdvar'(2),1,10)
'$fdvar'(1) #< '$fdvar'(2)
```

De forma análoga, el objetivo `batch, domain [X,Y] 1 10, X #<Y` se ejecuta en modo batch y las restricciones que se guardan en el fichero `saveConstraints.pl` son las mismas. Sin embargo, el objetivo `eclipse_gen, domain [X,Y] 1 10, X #<Y`, que se ejecuta en modo ECL^{i}PS$^e_{gen}$, crea un fichero Prolog (`eclipse_gen.ecl`) con el siguiente predicado:

```
goal(X,Y):-
    X=Fd1,
    Y=Fd2,
    domainFdE(Fd1,1,10),
    domainFdE(Fd2,1,10),
    Fd1#<Fd2.
```

Este predicado se ejecuta en el sistema ECL^{i}PS^e y se ha generado automáticamente. Los identificadores de variables se convierten a nombres de variables del predicado ECL^{i}PS^e. Por ejemplo, el identificador `'$fdvar'(1)` corresponde a la variable `Fd1`. Además, se genera la cabecera del predicado `goal` con los nombres de variables originales del objetivo $\mathcal{TOY}$, y estos nombres de variables se unifican con las variables del cuerpo del predicado. en el ejemplo anterior, se unifican las variables `X` e `Y` del objetivo $\mathcal{TOY}$ con las variables `Fd1` y `Fd2`.

6.4.2 Implementación de las restricciones primitivas atómicas

La implementación de una restricción primitiva atómica de la cooperación de los dominios $\mathcal{FD}$ y $\mathcal{FS}$ está implementada en dos partes, una en SICStus Prolog y la otra en ECL^{i}PS^e Prolog. Además, según sea el modo de ejecución del objetivo se dispone de distintas implementaciones en ECL^{i}PS^e Prolog. Así, una restricción primitiva atómica perteneciente a esta cooperación se define en tres partes como se detalla a continuación.

Por una parte se encuentra un predicado que se ejecuta en el proceso SICStus Prolog y cuyo esquema general se muestra en la figura 6.5. Todos los predicados correspondientes a la parte SICStus de la implementación de una restricción primitiva atómica se encuentran en el fichero `cflp_ic_sets_file.pl`.

En este esquema general una primitiva $f\, t_1, \ldots, t_n \rightarrow r$ se define como un predicado con los siguientes argumentos: los parámetros $t_1, \ldots, t_n$ de la primitiva f, el resultado r y dos argumentos más que representan la entrada y salida del almacén mixto de $\mathcal{TOY}$. Para comunicar con el proceso ECL^{i}PS^e se necesitan los identificadores de conexión (streams) de este proceso. Esta información se obtiene con una llamada al predicado `eclipse_active(Sin, Sout)` que devuelve estos parámetros. La restricción que se envía al proceso ECL^{i}PS^e es el argumento *constraint*`(...)` de la llamada `execute_eclipse(Sin, Sout,` *constraint*`(...), Result)`. Este argumento es una estructura Prolog que representa a la restricción en cuestión donde *constraint* se sustituye por el nombre de la restricción y entre paréntesis se encuentran sus argumentos. Si el argumento `Result` es `batch` entonces no se espera nada del proceso ECL^{i}PS^e

```
f(t1, ..., tn, r, Cin, Cout) :-
      ............
      eclipse_active(Sin, Sout),
      execute_eclipse(Sin, Sout, constraint(...), Result),
      ( Result = batch ->
            true
      ;
            (Result = result(...) ->
                  % se procesa la información
            ;
                  Result = failed, !, fail
            )
      ),
      % Si procede, se realiza el tratamiento de:
      % - las reglas de la tabla 3.2, y
      % - las proyecciones
```

Figura 6.5: Esquema de la implementación de una primitiva en la parte SICStus

y se continúa. Si es distinto de `batch` entonces se trata convenientemente la información que devuelve ECL^{i}PS^e, que pueden ser datos o que el predicado ha fallado. Finalmente, si es el caso, se tratan las reglas del sistema de transformación de almacenes que se dieron en la tabla 3.2 y la proyección de la restricción en cuestión.

Por otra parte se encuentra el código correspondiente a la parte ECL^{i}PS^e. En la figura 6.6 se muestra la estructura que sigue el predicado correspondiente a los modos interactivo y batch pues ambos modos comparten el mismo predicado para todas las primitivas. Este predicado se encuentra en el fichero `eclipsesets.ecl`.

El predicado `process_goal` tiene tres cláusulas por cada una de las primitivas. El primer argumento indica el modo de ejecución, el segundo argumento la primitiva y los dos últimos contienen la tabla de variables. En la línea 2 se hace la llamada `eclipse_libs(Sets,Fd)` que recoge en sus argumentos el nombre de las bibliotecas que serán utilizadas. En ECL^{i}PS^e hay varias bibliotecas disponibles para $\mathcal{FD}$ y $\mathcal{FS}$: `fd` y `fd_sets`; `ic` e `ic_sets`; e `ic` e `ic_hybrid_sets`. En este prototipo se ha implementado únicamente para las bibliotecas `ic` e `ic_sets`, pero se ha dejado preparado para ampliarlo a las demás bibliotecas. Después se trata convenientemente la restricción y se procesa por el sistema ECL^{i}PS^e. Una vez procesada la restricción se envía de vuelta a $\mathcal{TOY}$ la información asociada a las variables (línea 3). A continuación, en la línea 4, se llama de nuevo al servidor de ECL^{i}PS^e para que se mantenga en espera de nuevas peticiones de $\mathcal{TOY}$. Si $\mathcal{TOY}$ pide más soluciones entonces el predicado `server` (línea 4) falla para provocar la vuelta atrás sobre las líneas anteriores y así calcular una nueva solución que será devuelta a $\mathcal{TOY}$ en la línea 3. Si no hay más soluciones, se utiliza la segunda cláusula (línea 5) para comunicar el fallo al proceso $\mathcal{TOY}$ mediante el átomo `failed` y vuelve a llamar al servidor para quedarse en espera. Es posible que antes de esta petición hubiese otra petición y que no se hubiesen devuelto todas las soluciones; en este caso esta petición debe fallar (cláusula 8) para que la vuelta atrás busque otra solución de la

```
1   process_goal(Mode, constraint(...), VarsIn, VarsOut):-
        ............
        % Llamadas a predicados que devuelven las variables
        % de los identificadores y sus atributos
        ............
2       eclipse_libs(Sets,Fd),
        ............
        % se trata la restricción y se procesa
        ............
3       toy_response(Mode,result(...),constraint),
4       server(Mode,NewVars,Vout).

5   process_goal(Mode,constraint(...),Vars,Vout):-
        ............
6       toy_response(Mode,failed,constraint),
7       server(Mode,Vars,Vout).

8   process_goal(_,constraint(...),_,_):-
        !,
        fail.
```

Figura 6.6: Esquema de la implementación de una primitiva en la parte ECL^{i}PS^e. Modos interactivo y batch

petición anterior.

La tercera parte de código que implementa una restricción primitiva atómica corresponde al modo ECL^{i}PS$^e_{gen}$. El esquema general de este código ECL^{i}PS^e para restricciones de conjunto (*constraintSet*) y de dominio finito (*constraintFd*) se muestran en la figura 6.7 y como se puede observar es más sencillo. Sin embargo, las implementaciones de las restricciones `#--` y `<<`, que se verán en el próximo capítulo, no siguen este esquema y son más complejas. Todos los predicados correspondientes a esta tercera parte se encuentran en el fichero `eclipse_commands.ecl`.

```
constraintSet(...) :-                 constraintFd(...):-
1   eclipse_libs(Sets,_),                 eclipse_libs(_,Fd),
    ..............                        ............
2   Sets:(...).                           Fd:(...).
```

Figura 6.7: Esquema de la implentación de una primitiva en la parte ECL^{i}PS^e. Modo ECL^{i}PS$^e_{gen}$

Este código simplemente recoge el nombre de las bibliotecas que se van a utilizar mediante `eclipse_libs(Sets,Fd)` (línea 1) y se trata la restricción (línea 2).

A partir de la visión general que se acaba de dar de las implementaciones de las restricciones primitivas atómicas de los dominios $\mathcal{FD}$ y $\mathcal{FS}$ y su cooperación, se van a presentar en

las siguientes subsecciones las implementaciones concretas del mecanismo de comunicación (subsección 6.4.3) y las proyecciones (subsección 6.4.4). En general, todos los fragmentos de código que se van a mostrar se han simplificado para facilitar su presentación.

6.4.3 Implementación del puente cardinal

En esta sección se muestra la implementación de la primitiva `#-- :: int → set → bool`, cuyo comportamiento se especificó en la tabla 6.1. Como se ha dicho en la subsección anterior, la implementación de esta primitiva se divide en tres piezas de código. Una para la parte SICStus que implementa a $\mathcal{TOY}$ y otras dos para el tratamiento de las restricciones del sistema ECL^{i}PS^e: una corresponde a los modos interactivo y batch y la otra corresponde al modo ECL^{i}PS$^e_{gen}$. Empezamos por el código SICStus.

Al igual que el puente `#==`, descrito en el capítulo anterior, la primitiva `#--` se define en la figura 6.8 como un predicado Prolog de cinco argumentos. Los dos primeros argumentos `N` y `S` representan los parámetros izquierdo (entero) y derecho (conjunto) de la primitiva. El tercer argumento `true` es el resultado Booleano y los dos últimos argumentos, `Cin` y `Cout`, representan el almacén mixto antes y después de la ejecución de la primitiva.

Con respecto al cuerpo del predicado, primero se comprueba que `S` es una variable de conjunto (líneas 2 y 3). Las variables de conjunto tienen asociados atributos mediante la etiqueta `attSet`. Después se calcula la forma normal de cabeza del primer argumento (línea 4). Con el predicado `eclipse_active` de la línea 5 se toman los identificadores de conexión con el proceso ECL^{i}PS^e. A continuación, en las líneas 6 y 7, se obtienen los atributos de ambas variables. Para replicar las variables de tipos $\mathcal{FD}$ y $\mathcal{FS}$ en los dos procesos, se utilizan los predicados `get_id_set` y `get_id_fd` que mantienen información de las variables, además de un identificador (`IdS` e `IdN`, respectivamente) que permite relacionar las variables de ambos procesos que representan la misma variable $\mathcal{TOY}$. Uno de los atributos de `S` es el cardinal del conjunto, este atributo debe ser el mismo que el cardinal impuesto por la restricción `#--`, por lo tanto se aplica la restricción de igualdad de $\mathcal{FD}$ al atributo cardinal del conjunto y el cardinal del puente (línea 8 si `S` es una variable o línea 15 si es una constante). El predicado `equalFd` trata la igualdad en $\mathcal{TOY}$ y luego la envía a ECL^{i}PS^e.

En este punto ya se tiene toda la información necesaria y se procede a enviar la restricción `card(IdS,IdCard)` al servidor de ECL^{i}PS^e (línea 9), que será tratada según el modo de ejecución utilizado, como se muestra en las figuras 6.9 y 6.10. El resultado que devuelve ECL^{i}PS^e a $\mathcal{TOY}$ se trata entre las líneas 10 y 14 siguiendo el esquema de la figura 6.5. Si la información que nos devuelve el proceso ECL^{i}PS^e indica que el conjunto, que era una variable, ha pasado a ser un conjunto básico, es decir que el ínfimo (`Lwb`) y el supremo (`Upb`) del retículo que define al conjunto son los mismos (línea 13), entonces la variable conjunto se unifica con este valor.

En la segunda cláusula se trata el caso en el que `S` es un conjunto básico, es decir, sin variables (línea 15). En este caso el cardinal que viene determinado por el puente, es decir `N`, debe ser el mismo que el cardinal del conjunto básico, es decir, la longitud de la lista que lo representa.

La tercera cláusula (línea 16) representa el caso de que la variable `S` no ha intervenido

```
1   #--( N, S, true, Cin, Cout) :-
2       var(S),
3       get_atts(S, attSet(_,_,_,_,_,_,_)),
        !,
4       hnf(N, N1, Cin, Cout2),
5       eclipse_active(Sin, Sout),
6       get_id_set(S,IdS,Card,IdCard,_,_,Cout2,Cout3),
7       get_id_fd(N1,IdN,Cout3,Cout4),
8       equalFd(N1,Card,Cout4,Cout),
9       execute_eclipse(Sin, Sout,card(IdS,IdCard), Result),
10      (Result = batch ->
11          true
                ;
12         (Result = result(Lwb,Upb,Min,Max) ->
13             ((Lwb=Upb,var(S)) ->
                    S = Lwb
               ;
                    true
               )
           ;
14             Result = failed, !, fail
           )
        ).

15   #--( N, S, true, Cin, Cout) :-
       ground(S),
       !,
       simplify(S,S1),     % Cambia el formato de la lista
       length(S1,M),
       equalFd(N, M, Cin, Cout).

16   #--( N, S, true, Cin, Cout) :-
       hnf(S, S1, Cin, Cout1),
       hnf(N, N1, Cout1, Cout2),
       get_id_set(S1,IdS,N1,IdCard,_,_, Cout2, Cout).
```

Figura 6.8: Implementación del puente cardinal en la parte SICStus

en ninguna restricción de conjuntos y por lo tanto es todavía una variable sin atributos. El predicado `get_id_set` establece los atributos de la variable `S` utilizando la variable `N1` que representa al cardinal establecido por el puente.

La figura 6.9 muestra la segunda pieza de código que es necesaria para implementar el puente cardinal. Este código es utilizado por los modos interactivo y batch y la procesa el servidor ECL^{i}PS^e. Como se ha comentado anteriormente, a cada variable $\mathcal{TOY}$ se le asigna un identificador básico que se comunica de un proceso a otro, y en ECL^{i}PS^e se mantiene una tabla con dicho identificador y una variable ECL^{i}PS^e que reproduce el comportamiento de la

variable $\mathcal{TOY}$. En concreto, la tabla para las variables $\mathcal{FD}$ tiene dos campos: el identificador y la variable ECL^{i}PS^e asociada. Para las variables $\mathcal{FS}$ la tabla tiene cuatro campos, pues junto a la variable $\mathcal{FS}$ se guarda la información del cardinal asociado (variable $\mathcal{FD}$ y su identificador). Los predicados `get_var_set` y `get_var_fd` de las líneas 2 y 3 son los encargados de devolver la variable asociada a un identificador o crear la relación entre el identificador y una variable nueva si no se encontraba en la tabla.

```
1  process_goal(Mode,card(IdS,IdCard),Vars,Vout):-
2        get_var_set(IdS,IdCard,Vars,VarSet,VarCard,NewVars1),
3        get_var_fd(IdCard,NewVars1,VarN,NewVars),
4        VarCard=VarN,
5        eclipse_libs(Sets,Fd),
6        Sets:(#(VarSet,VarCard)),
7        Sets:set_range(VarSet,Lwb,Upb),
8        myfd_min(Fd,VarCard,Min),myfd_max(Fd,VarCard,Max),
9        toy_response(Mode,result(Lwb,Upb,Min,Max),card),
10       server(Mode,NewVars,Vout).
11 process_goal(Mode,card(_,_),Vars,Vout):-
12       toy_response(Mode,failed,card),
13       server(Mode,Vars,Vout).
14 process_goal(_,card(_,_),_,_):-
       !,fail.
```

Figura 6.9: Implementación del puente cardinal en la parte ECL^{i}PS^e. Modos interactivo y batch

De forma análoga a la pieza de código SICStus, se unifican las variables que representan el cardinal del conjunto y la parte entera del puente, pues deben ser el mismo valor aunque a priori pueden ser variables distintas (línea 4). En la línea 5 se recoge el nombre de las bibliotecas que serán utilizadas. En la línea 6 se emplea la restricción de cardinalidad de la biblioteca `ic_sets` de ECL^{i}PS^e con el conjunto y su cardinal. Esto puede parecer redundante pero hay que tener en cuenta que en la línea 4 puede haber cambiado el rango de la variable que representa al cardinal. En las líneas 7 y 8 se toman los valores extremos de las variables para enviarlas al proceso $\mathcal{TOY}$ en la línea 9. Finalmente, se llama recursivamente al servidor ECL^{i}PS^e para que se quede en espera de nuevas peticiones de $\mathcal{TOY}$. El tratamiento de nuevas soluciones y del fallo se hace entre las líneas 11 y 14 como se explicó en la subsección 6.4.2.

La tercera y última pieza de código que implementa el puente cardinal se muestra en la figura 6.10. Este código corresponde al modo ECL^{i}PS$^e_{gen}$ y simplemente toma la biblioteca `ic_sets` y envía la restricción de cardinalidad al resolutor de ECL^{i}PS^e.

6.4.4 Proyecciones y aplicación de las reglas de transformación de almacenes definidas para el resolutor $solve^{\mathcal{FS}^{\mathcal{T}}}$

En esta subsección se va a mostrar el código que implementa a la primitiva `subset` y se explica cómo se han implementado las reglas que corresponden de la tabla 3.2, así como la

```
card(S,F):-
    eclipse_libs(Sets,_),
    Sets:(#(S,F)).
```

Figura 6.10: Implementación del puente cardinal en la parte ECL^{i}PS^e. Modo ECL^{i}PS$^e_{gen}$

proyección de esta restricción sobre el dominio $\mathcal{FD}$. Como se explicó en la subsección 6.4.2, esta implementación consta de varios predicados que se expondrán en distintas figuras de forma simplificada para resaltar las partes más relevantes. La activación de las proyecciones en el sistema a través del puente cardinal se realiza mediante el comando `/projCard`.

En la figura 6.11 se muestra el predicado de la parte SICStus que implementa a la restricción primitiva atómica `subset`. En este código lo primero que se hace es tratar los argumentos de `subset`. En particular, se calculan las formas normales de cabeza en la línea 1. si el conjunto es básico (*ground*) entonces la lista que lo representa se trata para asegurar que tiene el estilo de listas Prolog (línea 2). Como ya se ha explicado anteriormente se toman las referencias de conexión con ECL^{i}PS^e mediante el predicado `eclipse_active` (línea 3), y también se obtienen los identificadores `IdA` e `IdB` asociados a las variables, así como las variables correspondientes a los cardinales y sus identificadores (línea 4). En la línea 5 se añade el predicado dinámico `subset(IdA,IdB)` que indica la relación de subconjunto que existe entre los argumentos de `subset`. Este predicado dinámico se utiliza para intentar anticipar el fallo en la línea 13 siguiendo las reglas de la tabla 3.2 como se verá más adelante. A continuación en la línea 6 se realiza la llamada al correspondiente código de ECL^{i}PS^e para `subset`. El tratamiento de la respuesta depende del modo de ejecución del objetivo como se explicó en la subsección 6.4.2. Así, la línea 7 trata los modos batch y ECL^{i}PS$^e_{gen}$ y las líneas 8, 9 y 10 tratan el modo interactivo. El tratamiento del resultado del modo interactivo (línea 9) es el siguiente:

```
(LwbA=UpbA ->
  SA1 = LwbA
;
  put_atts(SA1,attSet(IdA,LwbA,UpbA,CardA,IdCardA,_,_,_,_,_,_))
),
(MinA=MaxA ->
  CardA = MinA
;
  put_atts(CardA,attFd(IdCardA,MinA,MaxA))
)
```

Este código se replica para el conjunto con identificador `IdB` e indica que si el ínfimo y el supremo del retículo que representa al conjunto son iguales entonces el conjunto es básico y se unifica la variable que representa al conjunto con dicho valor básico. Si no son iguales entonces se actualizan los atributos de la variable que representa al conjunto. Igualmente, si la variable de dominio finito que representa al conjunto tiene sus valores extremos iguales entonces se unifica la variable con este valor entero, en caso contrario se actualizan sus atributos.

```
subset(A, B, true, Cin, Cout):-
1 hnf(A, SA, Cin, Cout1), hnf(B, SB, Cout1, Cout2),
2 simplify(SA,SA1), simplify(SB,SB1),
3 eclipse_active(Sin, Sout),
4 get_id_set(SA1,IdA,CardA,IdCardA,...),
  get_id_set(SB1,IdB,CardB,IdCardB,...),
5 assert_subset(IdA,IdB),
6 execute_eclipse(Sin,Sout,subsetE(IdA,IdB,IdCardA,IdCardB),Result),
7 (Result = batch ->
    true
 ;
8   Result = result(LwbA,UpbA,LwbB,UpbB,MinA,MaxA,MinB,MaxB) ->
9        % Omitido: tratamiento del resultado. Actualización de los
         % atributos de las variables conjunto (extremos y cardinalidad)
         % con el resultado devuelto por ECL^iPS^e.
       ;
10          Result = failed, !, fail
 ),
 (fs_active ->
11   (((ground(MinA),MinA > 0) ->                    % FS S3
         notEqualSet(SB1,s([]),Cout4,Cout5)
     ;
         Cout4=Cout5
     ),
12   ((gound(MaxB),MaxB < 1) ->                      % FS S4
         equalSet(SA1,s([]),Cout5,Cout6)
     ;
         Cout6=Cout5
     ),
13   check_subsets(IdA,IdB)                          % FS S5
    )
    ;
    Cout4 = Cout6
 ),
14 (projCard_active ->
       proj_lessOrEqual(Sin,Sout,IdCardA,IdCardB)
 ;
       true
 ),
15   ...
```

Figura 6.11: Implementación de la restricción `subset` en la parte SICStus

En las líneas 11, 12 y 13 se tratan algunas reglas de transformación de almacenes de la tabla 3.2 y que se reproducen a continuación:

S3 subset S_1 S_2, S_1 /= {}, $\Pi \square \sigma \vdash_{\mathcal{FST}}$ subset S_1 S_2, S_1 /= {}, S_2 /= {}, $\Pi \square \sigma$

S4 `subset` S_1 S_2, S_2 `==` `{}`, $\Pi \Box \sigma \vdash_{\mathcal{FS}^{\mathcal{T}}}$ S_2 `==` `{}`, S_1 `==` `{}`, $\Pi \Box \sigma$
S5 `subset` S_1 S_2, `subset` S_2 S_1, $\Pi \Box \sigma \vdash_{\mathcal{FS}^{\mathcal{T}}}$ S_1 `==` S_2, $\Pi \Box \sigma$

En las líneas 11 y 12 se trata el conjunto vacío mediante la cardinalidad mínima y se aplican las restricciones de igualdad y desigualdad de conjuntos según requieran las reglas **S3** y **S4**. El predicado `check_subsets` comprueba si ambos conjuntos son mutuamente subconjuntos mediante el predicado dinámico de subconjunto `subset/2` (regla **S5**).

```
check_subsets(IdL,IdR) :-
  (subset(IdL,IdR), subset(IdR,IdL)) -> IdR=IdL; true.
```

Por último, en la línea 14 se trata la proyección de la restricción `subset` sobre el dominio $\mathcal{FD}$ descrita en la tabla 6.4 y que se reproduce a continuación:

$$proj^{\mathcal{FS}\to\mathcal{FD}}(\texttt{subset}\ S_1\ S_2,\mathrm{B}) = \{\ C_1\ \texttt{\#<=}\ C_2 \mid (C_1\ \texttt{\#--}\ S_1),(C_2\ \texttt{\#--}\ S_2) \in B\ \}$$

Esta proyección está implementada por el predicado `proj_lessOrEqual` que llama al predicado de desigualdad del dominio $\mathcal{FD}$.

```
proj_lessOrEqual(Sin, Sout,IdCardA,IdCardB) :-
    execute_eclipse(Sin, Sout, #=<(IdCardA,IdCardB), Result),
    (
        Result = batch -> true
    ;
        Result = result(_,_,_,_) -> true
    ;
        Result = failed, !, fail
    ).
```

El código de la restricción `subset` continúa como muestra la línea 15 pero al caer fuera del ámbito de este capítulo se ha omitido.

Una vez acabada la parte SICStus de la implementación de la restricción `subset` se muestra la parte de ECL^{i}PS^e para los modos interactivo y batch como se muestra en la figura 6.12. Esta implementación es análoga a la expuesta en la figura 6.9, por lo que solo se detalla la primera cláusula.

En las líneas 1 y 2 se toman las variables ECL^{i}PS^e correspondientes a las variables $\mathcal{TOY}$ para esta restricción y también se toman las bibliotecas `ic_sets` e `ic` en la línea 3. Una vez que se tiene todo lo necesario ECL^{i}PS^e procesa la restricción `subset` (línea 4). Después de evaluar la restricción `subset` se calculan los cardinales y valores extremos en las líneas 5-8. A continuación se devuelven los valores a $\mathcal{TOY}$, se llama recursivamente al servidor y se hace el tratamiento de la vuelta atrás siguiendo el esquema de la figura 6.6.

Por último, en la figura 6.13 se muestra la implementación que se realiza en el sistema ECL^{i}PS^e para el modo ECL^{i}PS$^e_{gen}$. En este caso simplemente se toman la librería `ic_sets` y se envía la restricción.

```
process_goal(Mode,subsetE(IdA,IdB,IdCardA,IdCardB),Vars,Vout):-
1 get_var_set(IdA,IdCardA,Vars,Var1,Card1,NewVars1),
2 get_var_set(IdB,IdCardB,NewVars1,Var2,Card2,NewVars),
3 eclipse_libs(Sets,Fd),
4 Sets:(Var1 subset Var2),
5 Sets:set_range(Var1,Lwb1,Upb1),
6 Sets:(#(Var1,Card1)),
  myfd_min(Fd,Card1,Min1), myfd_max(Fd,Card1,Max1),
7 Sets:set_range(Var2,Lwb2,Upb2),
8 Sets:(#(Var2,Card2)),
  myfd_min(Fd,Card2,Min2), myfd_max(Fd,Card2,Max2),
9 toy_response(Mode,
      result(Lwb1,Upb1,Lwb2,Upb2,Min1,Max1,Min2,Max2),subsetE),
10 server(Mode,NewVars,Vout).
```

Figura 6.12: Implementación de la restricción `subset` en la parte ECL^{i}PS^e. Modos interactivo y batch

```
subsetE(S1,S2,_,_):-
      eclipse_libs(Sets,_),
      Sets:subset(S1,S2).
```

Figura 6.13: Implementación de la restricción `subset` en la parte ECL^{i}PS^e. Modo ECL^{i}PS$^e_{gen}$

6.5 Resultados experimentales

Este prototipo ha sido probado con ejemplos clásicos utilizados por los resolutores de conjuntos de enteros, tomados de la página web del sistema ECL^{i}PS^e [ECL]. Estos problemas son *Las ternas de Steiner* y *Social Golfers*, disponibles también en `http://www.csplib.org/`, problemas `prob044` y `prob010`. Los programas $\mathcal{TOY}$ están disponibles en `http://gpd.sip.ucm.es/sonia/Prototype.html`.

El problema de las ternas de Steiner de orden `n` ya fue introducido en el ejemplo 9. El problema *Social Golfers* consiste en tratar de planificar a `g*p` golfistas en `g` grupos de `p` jugadores en cada grupo que juegan `w` semanas. Se exige que cada golfista no juegue en el mismo grupo con otro golfista más de una vez. Por ejemplo, supongamos un torneo de dos semanas, para cuatro grupos de dos componentes cada grupo, por lo tanto hay ocho jugadores. A continuación se muestra una planificación de este torneo:

	Grupos			
	1º	2º	3º	4º
1ª semana	[1, 2]	[3, 4]	[5, 6]	[7, 8]
2ª semana	[1, 3]	[2, 4]	[5, 7]	[6, 8]

Se ha comparado la ejecución del programa disponible en la web de ECL^{i}PS^e con su implementación en $\mathcal{TOY}$ en los tres modos de ejecución.

Benchmark	**ECL^{i}PS^e**	**inter**	**SU**	**batch**	**SU**	**ECL^{i}PS$^e_{gen}$**	**SU**
steiner(3)	0	0	n/a	0	n/a	0	n/a
steiner(4)	0	10	n/a	10	n/a	10	n/a
steiner(5)	10	50	0.20	30	0.33	10	1.00
steiner(6)	670	1,280	0.52	810	0.83	680	0.99
steiner(7)	266,210	1,869,850	0.14	1,434,930	0.19	262,480	1.01
golf(4,2,2)	40	360	0.11	90	0.44	50	0.80
golf(5,2,2)	50	410	0.12	100	0.50	50	1.00
golf(6,2,2)	60	540	0.11	130	0.46	60	1.00
golf(7,2,2)	60	700	0.08	140	0.42	70	0.85
golf(8,2,2)	70	790	0.08	190	0.36	70	1.00
golf(9,2,2)	70	1,010	0.06	220	0.31	80	0.87
golf(10,2,2)	80	1,400	0.05	270	0.29	80	1.00
golf(3,3,2)	51,138	724,550	0.07	928,880	0.05	50,630	1.01
golf(4,3,2)	1,043,113	-	n/a	-	n/a	1,046,230	0.99

Tabla 6.7: Experimentos realizados en ECL^{i}PS^e y los tres modos de $\mathcal{TOY}$ (milisegundos)

Estos programas han sido ejecutados con distintos parámetros, como se muestra en la columna **Benchmark** de la tabla 6.7. En esta tabla se muestra el tiempo de cómputo en milisegundos para calcular todas las soluciones o hasta que fracase porque no existan soluciones para los argumentos de entrada. Las simetrías no se han eliminado para tener el mismo comportamiento que los programas originales de ECL^{i}PS^e. Las pruebas han sido realizadas en un sistema con un procesador Intel© Core™ i7-740QM (4 cores) a 1.73 GHz con 3 GB de memoria física y sobre Ubuntu 10.10. Los experimentos marcados con un guión en su tiempo de ejecución han alcanzado un tiempo de 3.600 segundos.

La columna **ECL^{i}PS^e** contiene el tiempo empleado por los programas originales[2] [ECL]. Las columnas **inter** y **batch** contienen el tiempo en milisegundos de los modos interactivo y batch de $\mathcal{TOY}$, respectivamente. Las columnas **SU** contienen la mejora del tiempo en ejecución de cada modo con respecto a la columna ECL^{i}PS^e. n/a representa las ganancias de velocidad que no se han podido calcular para evitar dividir por cero o porque alguno de los valores no está disponible.

Como se observa en la tabla 6.7, el modo batch de $\mathcal{TOY}$ es más rápido que el modo interactivo, donde el cómputo de estrechamiento perezoso en $\mathcal{TOY}$ se ha intercalado con la resolución de restricciones en ECL^{i}PS^e. Es decir, el modo batch no requiere establecer una comunicación desde el servidor de ECL^{i}PS^e a $\mathcal{TOY}$ cada vez que se procesa una restricción, y por lo tanto el cómputo en ECL^{i}PS^e se lleva a cabo sin interrupciones para comunicar resul-

[2]El problema de Social golfers de ECL^{i}PS^e ha sido modificado para hacerlo comparable al programa $\mathcal{TOY}$. La versión de la página web de ECL^{i}PS^e no ejecuta el etiquetado de conjuntos, produciendo resultados muy pobres.

tados intermedios a $\mathcal{TOY}$. En estos ejemplos en particular, el modo batch es muy apropiado pues la estructura de la solución se genera primero en $\mathcal{TOY}$ junto con todas las restricciones que son resueltas en un segundo paso, de forma similar a como se especifican y resuelven los problemas CP. Sin embargo, como se motivó en la página 135, los objetivos que utilizan el indeterminismo se deben ejecutar en modo interactivo.

Es notable que en todos los casos ECL^{i}PS$^e_{gen}$ toma aproximadamente el mismo tiempo que el programa original de ECL^{i}PS^e, en particular para `golf4_3_2` que genera un gran número de soluciones, el tiempo es similar. Estos resultados muestran que se pueden resolver en $\mathcal{TOY}$ problemas del paradigma CP sin introducir una sobrecarga relevante con respecto a las pruebas realizadas directamente en ECL^{i}PS^e. No se ha incluido en la tabla 6.7 el tiempo dedicado al tratamiento del estrechamiento y evaluación perezosa en la generación del programa ECL^{i}PS^e porque en el tipo de problemas que hemos estudiado no es relevante.

Capítulo 7

Cooperación extendida entre $\mathcal{FD}$ y $\mathcal{FS}$

En este capítulo se trata la extensión de la cooperación de los dominios $\mathcal{FD}$ y $\mathcal{FS}$. Esta extensión se fundamenta en dos nuevos puentes que relacionan conjuntos del dominio $\mathcal{FS}$ con elementos del dominio $\mathcal{FD}$ que representan el mínimo y máximo de estos conjuntos. Con respecto al cálculo $CCLNC(\mathcal{C}_{\mathcal{FD}.\mathcal{FS}})$ también se extiende consecuentemente con operaciones de creación de puentes, nuevas proyecciones entre los dominios $\mathcal{FD}$ y $\mathcal{FS}$ y nuevas reglas que infieren igualdades y anticipan el fallo. De forma análoga, se demuestra que el cálculo extendido que captura la semántica operativa de estos nuevos puentes es también correcto y completo con las mismas limitaciones que el cálculo $CCLNC(\mathcal{C}_{\mathcal{FD}.\mathcal{FS}})$.

La motivación de esta extensión consiste en ganar eficiencia en la resolución de ciertos problemas. Se ha separado de la cooperación de los dominios $\mathcal{FD}$ y $\mathcal{FS}$ del capítulo anterior pues el enfoque propuesto en este capítulo es menos general, ya que esta cooperación no se puede establecer sobre conjuntos vacíos.

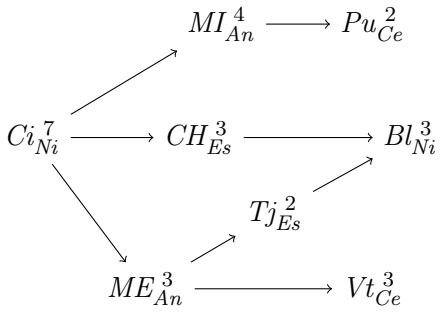
La estructura que se sigue en este capítulo es similar a la estructura de los capítulos anteriores. En una sección introductoria se motiva la extensión de la cooperación de $\mathcal{FD}$ y $\mathcal{FS}$ mediante un ejemplo y se describe informalmente su mecanismo de cooperación. A continuación se especifica cómo se extiende el dominio mediador y se describe la extensión del cálculo. Finalmente, se muestra la implementación y los resultados experimentales.

7.1 Introducción

Esta sección motiva la extensión de la cooperación a través de un ejemplo e introduce informalmente las restricciones puentes `minSet` y `maxSet`.

7.1.1 Ejemplo motivador

Supongamos que se quiere planificar el proyecto de construir una casa de forma similar al problema expuesto en [MS98] (página 17). La construcción de una casa consiste en una serie de tareas como asentar cimientos, construir paredes, tejado y chimenea, colocar puertas, baldosas y ventanas, etc. Cada tarea requiere un cierto número de días y las tareas se realizan en días



Ci:	Cimientos		
MI:	Muros interiores		
CH:	Chimenea	*Ni:*	Nivel
ME:	Muros exteriores	*An:*	Andamio
Vt:	Ventanas	*Es:*	Escalera
Bl:	Baldosas	*Ce:*	Cepillo
Pu:	Puertas		
Tj:	Tejado		

Figura 7.1: Proyecto de construcción de una casa [MS98]

completos. Algunas tareas preceden a otras y también hay tareas que comparten recursos que no se pueden utilizar simultáneamente. Por lo tanto, las tareas que comparten estos recursos tampoco se pueden realizar simultáneamente. Así, el problema original [MS98] se ha ampliado con la compartición de recursos como muestra el grafo de precedencias de la figura 7.1.

Si las tareas se realizan en días consecutivos, codificar este problema con restricciones del dominio $\mathcal{FD}$ es sencillo. Sin embargo, si las tareas se pueden realizar en días alternos entonces la codificación en el dominio $\mathcal{FD}$ se complica.

Un modelo alternativo para representar este problema consiste en utilizar restricciones de conjuntos de números enteros. En este caso cada tarea se representa como una variable de conjunto cuyos elementos son los días que se planifica la tarea, por ejemplo si la tarea de poner los cimientos se planifica en los 7 primeros días entonces la variable *Ci* que la representa se instancia al conjunto {1,2,3,4,5,6,7}. La duración de las tareas corresponde con el cardinal del conjunto. Para evitar el solapamiento de las tareas que requienen el mismo recurso, se restringen los conjuntos correspondientes para que sean disjuntos. Nótese que, las relaciones de precedencia entre tareas no pueden establecerse utilizando las operaciones básicas de conjuntos ($\subseteq, \cup, \cap, \ldots$). Para solventar este problema se ha creado una restricción nueva sobre conjuntos: `<<` , donde Set_1 `<<` Set_2 fuerza a que todos los elementos de Set_1 sean menores que los elementos de Set_2. Con esta nueva restricción la codificación del proyecto se reduce al código mostrado en la figura 7.2.

Para este código, un objetivo de la forma `proyecto 16 Set` vincula la variable `Set` a la lista de conjuntos: `[{1,2,3,4,5,6,7}, {8,12,13,14}, {8,9,10}, {9,10,11}, {15,16}, {12,13}, {14,15,16}, {12,13,14}]`.

Además de modelar este problema de una forma sencilla mediante conjuntos, es posible mejorar el rendimiento del sistema si los resolutores de los dominios $\mathcal{FD}$ y $\mathcal{FS}$ cooperan. Supongamos que se dispone de 14 días para realizar el proyecto anterior. En este caso es fácil ver que no se pueden planificar las tareas: cimientos, muros exteriores, tejado y baldosas (*Ci*, *ME*, *Tj* y *Bl*), pues estas tareas necesitan 15 días para poderse realizar y existen relaciones de precedencia entre ellas. Sin embargo, esta situación, que provocaría un fallo de cómputo, no se puede detectar con la restricción `<<` antes del etiquetado, dado que ninguno de sus argumentos es básico. Esta restricción necesita que uno de los conjuntos sea básico para poder establecer una cota en el otro conjunto y eliminar todos los elementos que no cumplan dicha cota. Por ejemplo, si *Ci* se vincula al conjunto `{1,2,3,4,5,6,7}`, entonces los conjuntos

```
proyecto :: int -> [iset] -> bool
proyecto Dias Tareas  = true <==
    % Las tareas son ctos con dominio en el retículo [{},{1,...,Dias}]
    Tareas == [Ci,MI,CH,ME,Pu,Tj,Bl,Vt],
    intSets Tareas 1 Dias,
    % La duración de una tarea corresponde al cardinal de su conjunto
    andL (zipWith (#--) [7,4,3,3,2,2,3,3] Tareas),
    % Se establecen las precedencias
    Ci << MI, Ci << CH, Ci << ME, MI << Pu,
    CH << Bl, ME << Tj, ME << Vt, Tj << Bl,
    % Se evitan solapamientos
    disjoints([Ci,Bl]), disjoints([MI,ME]),
    disjoints([Pu,Vt]), disjoints([CH,Tj]),
    labelingSets Tareas
```

Figura 7.2: Código $\mathcal{TOY}$ para el proyecto de construir una casa

MI, *CH* y *ME* tienen como menor elemento posible el número 8. Cuando *ME* es básico se establece una cota para *Tj* y cuando *Tj* es básico se establece una cota para *Bl*. Por lo tanto, es en el proceso de etiquetado donde se detecta que no se satisfacen las restricciones. Sin embargo, se puede anticipar el fallo si la información de las restricciones de precedencia se envía al resolutor de dominio $\mathcal{FD}$.

El envío de información entre los dominios $\mathcal{FD}$ y $\mathcal{FS}$ se lleva a cabo a través de unos puentes similares al puente cardinal visto en el capítulo 6. Estos puentes van a permitir relacionar cada conjunto con dos variables $\mathcal{FD}$: una para representar el valor mínimo que puede contener el conjunto, y otra para el máximo. Así, supongamos que *MinCi* denota la variable $\mathcal{FD}$ que representa el menor elemento del conjunto *Ci*, y respectivamente con el resto de variables. Estas variables tienen un dominio inicial comprendido entre 1 y 14, que son los posibles elementos del conjunto. Con estas variables se pueden establecer las siguientes restricciones de dominio finito:

1 $1 \leq MinCi$,
2 $MinCi + 7 \leq MinME$,
3 $MinME + 3 \leq MinTj$,
4 $MinTj + 2 \leq MinBl$,
5 $MinBl + 3 \leq 14 + 1$

En este sistema se puede inferir mediante las desigualdades 1, 2 y 3 que $11 \leq minTj$, que junto con 4 se deduce que el valor de $MinBl$ es al menos 13, lo cual no satisface la restricción $MinBl + 3 \leq 15$. De esta forma, el resolutor de $\mathcal{FD}$ puede detectar la inconsistencia y anticipar el fallo como se muestra en la sección 7.6 de resultados experimentales.

7.1.2 Nuevas restricciones puente

La cooperación entre los dominios $\mathcal{FD}$ y $\mathcal{FS}$ vista en el capítulo 6 se extiende en esta sección con dos nuevas restricciones puente denominadas `minSet` y `maxSet`. De esta forma se pueden comunicar los dominios $\mathcal{FD}$ y $\mathcal{FS}$ mediante el puente cardinal (que ya existía) y estos dos puentes nuevos que en esencia tratan en el dominio $\mathcal{FD}$ la consistencia de los valores extremos (*bound consistency*) de los conjuntos: los valores máximo y mínimo que una variable de conjunto puede tomar. Es importante observar que las restricciones `minSet` y `maxSet` no se pueden aplicar sobre conjuntos vacíos. Así, cuando se establece una de estas restricciones, el rango de la variable $\mathcal{FD}$ está determinado por la forma que tiene el retículo que define al conjunto.

Las figuras 7.3 y 7.4 muestran respectivamente el dominio de las variables `Min` y `Max` en las restricciones `minSet Set Min` y `maxSet Set Max` según sea el retículo que define al conjunto `Set`. Estas restricciones no se especifican como operadores infijos a diferencia de los puentes anteriores. La figura 7.3 muestra los dos escenarios posibles: el primer escenario (izquierda) corresponde a un conjunto definido con un retículo en el que el ínfimo es distinto del conjunto vacío (`Set` $\in [\{l_1, \ldots, l_m\}, \{u_1, \ldots, u_n\}]$). En este caso los elementos $l_1, \ldots, l_m$ deben pertenecer obligatoriamente al conjunto. Por lo tanto, l_1 es cota superior del menor elemento del conjunto. Además, pueden existir en el conjunto elementos menores a l_1, por lo tanto, en este caso el mínimo está delimitado por el menor elemento que puede contener el conjunto (u_1) y el menor elemento que obligatoriamente está en el conjunto (l_1), así `Min` $\in [u_1, l_1]$. Nótese que por la definición del retículo se tiene $u_1 \leq l_1$. El segundo escenario (derecha) corresponde a un conjunto definido con un retículo de ínfimo el conjunto vacío y supremo un conjunto no vacío, es decir, `Set` $\in [\{\},\{u_1, \ldots, u_n\}]$, en este caso el mínimo puede ser cualquier valor del conjunto, así `Min` $\in [u_1, u_n]$.

Figura 7.3: Posibles dominios de la variable `Min` en la restricción `minSet Set Min`

Análogamente a la figura 7.3, la figura 7.4 representa los valores que toma la variable `Max` en la restricción `maxSet Set Max`, dependiendo de la forma que tome el retículo que define al conjunto `Set`.

Figura 7.4: Posibles dominios de la variable `Max` en la restricción `maxSet Set Max`

A continuación se muestran unos ejemplos sencillos de objetivos en el sistema $\mathcal{TOY}$ que muestran cómo se asigna el dominio a las variables $\mathcal{FD}$ dependiendo de la forma de los conjuntos.

```
Toy(ic_sets)> maxSet (s [1,2,3,4]) Max
      { Max -> 4 }

Toy(ic_sets)> domainSets [S] (s []) (s [1,2,3,4]), minSet S Min,
   maxSet S Max
      { S in [{},{1,2,3,4}] and Card is 1..4,
        Max in 1..4,
        Min in 1..4 }

Toy(ic_sets)> domainSets [S] (s [3]) (s [1,2,3,4]),minSet S Min,
   maxSet S Max
      { S in [{3},{1,2,3,4}] and Card is 1..4,
        Max in 3..4,
        Min in 1..3 }
```

El primer ejemplo corresponde a un conjunto básico de retículo [{1,2,3,4},{1,2,3,4}]. En este caso l_m y u_n son el número `4`, por lo tanto el dominio del máximo es [4,4] y la variable `Max` se vincula a `4`. En los otros dos ejemplos las variables `Min` y `Max` toman valores siguiendo la explicación anterior.

Hasta ahora se ha detallado cómo se pueden restringir los rangos de valores que toman las variables $\mathcal{FD}$ con respecto al conjunto. Como `minSet` y `maxSet` son restricciones, el conjunto también se puede modificar dependiendo del dominio de la variable $\mathcal{FD}$. La figura 7.5 muestra distintas configuraciones de las variables $\mathcal{FD}$ y $\mathcal{FS}$ y cómo actúa la restricción `minSet Set Min`. En particular, se supone que `Min` está definida en el intervalo $[m_1, m_k]$ y `Set` está definida mediante el retículo $[\{l_1,\ldots,l_m\},\{u_1, \ldots\ldots, u_n\}]$, con ínfimo distinto del vacío. Los elementos están colocados en el eje horizontal atendiendo al orden $<$ (los menores más a la izquierda y los mayores a la derecha). Se añaden líneas verticales para aclarar las posiciones concretas.

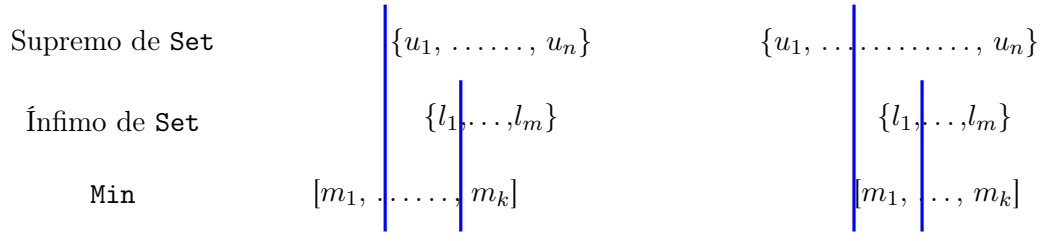


Figura 7.5: Dominios que toman las variables `Min` y `Set` en la restricción `minSet Set Min`

En el primer escenario de la figura 7.5 se considera $m_1 < u_1$. En este caso se eliminan de la variable `Min` todos los elementos menores que u_1. Además, si $l_1 < m_k$ se eliminan de `Min` todos los elementos mayores que l_1. En el segundo escenario, $u_1 < m_1 < l_1$ y se eliminan del

supremo del retículo de `Set` todos los elementos menores que m_1. Además, como en el caso anterior, en el dominio de `Min` se deben quitar los elementos mayores que l_1.

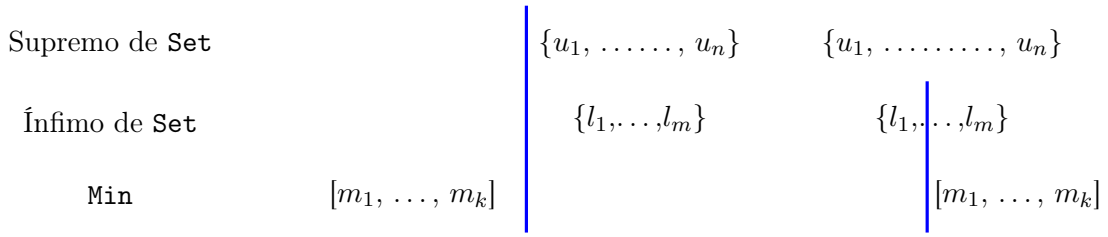


Figura 7.6: Escenarios de fallo de la restricción `minSet Set Min`

En la figura 7.6 se muestran los dos escenarios de fallo de la restricción `minSet Set Min`. Si $m_k < u_1$ se produce fallo porque no existe ningún elemento del conjunto que pertenezca al dominio de `Min`. Si $m_1 > l_1$ se produce fallo porque existe un elemento del conjunto que es menor que el mínimo. El máximo se comporta de forma simétrica al mínimo.

A continuación se muestran unos ejemplos de objetivos sencillos que representan estos escenarios.

```
Toy(ic_sets)> domainSets [Set] (s [2,3]) (s [1,2,3,4]),
  domain [Min] 0 4, minSet Set Min
      { Set in [{2,3},{1,2,3,4}] and Card is 2..4,
        Min in 1..2 }

Toy(ic_sets)> domainSets [Set] (s [3]) (s [1,2,3,4]),
  domain [Min] 2 4, minSet Set Min
      { Set in [{3},{2,3,4}] and Card is 1..3,
        Min in 2..3 }

Toy(ic_sets)> domainSets [Set] (s [3]) (s [1,2,3,4]),
  domain [Min] (-6) (-4), minSet Set Min
      no

Toy(ic_sets)> domainSets [Set] (s [3]) (s [1,2,3,4]),
  domain [Min] 4 6, minSet Set Min
      no
```

La figura 7.7 representa las distintas configuraciones de las variables `Min` y `Set` cuando el ínfimo del retículo es el conjunto vacío. En todos estos casos, si $m_k > u_n$ entonces se deben eliminar de `Min` todos los elementos que son mayores que u_n. Además, en el primer caso se eliminan de `Min` todos los elementos menores que u_1 y en el segundo caso se eliminan del supremo de `Set` todos los elementos menores que m_1. El tercer caso falla, pues el mínimo `Min`

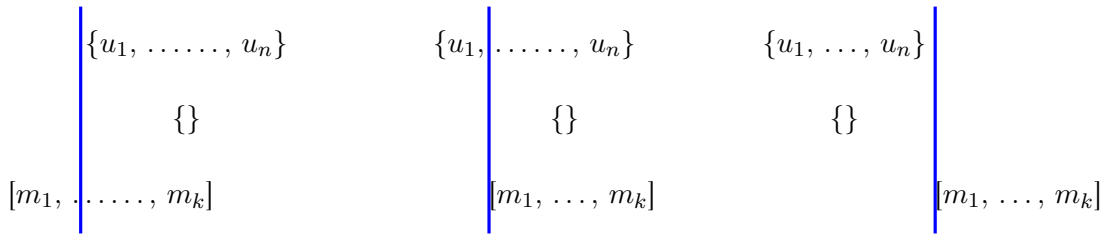


Figura 7.7: Dominios que toman las variables `Min` y `Set` en la restricción `minSet Set Min`

del conjunto `Set` no es un elemento del mismo. A continuación se muestran más ejemplos de objetivos $\mathcal{TOY}$.

```
Toy(ic_sets)> domainSets [Set] (s []) (s [1,2,3,4]),
  domain [Min] 0 5, minSet Set Min
      { Set in [{},{1,2,3,4}] and Card is 1..4,
        Min in 1..4 }

Toy(ic_sets)> domainSets [Set] (s []) (s [1,2,3,4]),
  domain [Min] 2 5, minSet Set Min
      { Set in [{},{2,3,4}] and Card is 1..3,
        Min in 2..4 }

Toy(ic_sets)> domainSets [Set] (s []) (s [1,2,3,4]),
  domain [Min] 5 6, minSet Set Min
      no
```

7.2 Extensión del dominio mediador

El dominio mediador $\mathcal{M}_{\mathcal{FD}.\mathcal{FS}}$ descrito en el capítulo 6 se extiende en esta sección con las nuevas restricciones puente `minSet` y `maxSet`. Este nuevo dominio, que denominamos $\mathcal{M}'_{\mathcal{FD}.\mathcal{FS}}$, se define aplicando la definición de *extensión conservativa de un dominio* dada en la página 42. Así $\mathcal{M}'_{\mathcal{FD}.\mathcal{FS}}$ se define como la extensión conservativa de $\mathcal{M}_{\mathcal{FD}.\mathcal{FS}}$ con signatura $\Sigma_{\mathcal{M}'_{\mathcal{FD}.\mathcal{FS}}} = \langle TC,\ SBT_{\mathcal{M}'_{\mathcal{FD}.\mathcal{FS}}},\ DC,\ DF,\ SPF_{\mathcal{M}'_{\mathcal{FD}.\mathcal{FS}}}\rangle$ donde:

1. $\Sigma_{\mathcal{M}_{\mathcal{FD}.\mathcal{FS}}} \subseteq \Sigma_{\mathcal{M}'_{\mathcal{FD}.\mathcal{FS}}}$, por lo tanto se debe cumplir $SBT_{\mathcal{M}_{\mathcal{FD}.\mathcal{FS}}} \subseteq SBT_{\mathcal{M}'_{\mathcal{FD}.\mathcal{FS}}}$ y $SPF_{\mathcal{M}_{\mathcal{FD}.\mathcal{FS}}} \subseteq SPF_{\mathcal{M}'_{\mathcal{FD}.\mathcal{FS}}}$.

   Estas condiciones se cumplen pues $SBT_{\mathcal{M}_{\mathcal{FD}.\mathcal{FS}}} = SBT_{\mathcal{M}'_{\mathcal{FD}.\mathcal{FS}}} =$ `{int, set}` y además $SPF_{\mathcal{M}_{\mathcal{FD}.\mathcal{FS}}} =$ { `#--` } $\subseteq SPF_{\mathcal{M}'_{\mathcal{FD}.\mathcal{FS}}} =$ { `#--` , `minSet`, `maxSet`}.

2. Con respecto a cada tipo, los valores básicos no se modifican en la extensión, es decir, $\mathcal{B}^{\mathcal{M}_{\mathcal{FD}.\mathcal{FS}}}_{\mathtt{set}} = \mathcal{B}^{\mathcal{M}'_{\mathcal{FD}.\mathcal{FS}}}_{\mathtt{set}}$ y $\mathcal{B}^{\mathcal{M}_{\mathcal{FD}.\mathcal{FS}}}_{\mathtt{int}} = \mathcal{B}^{\mathcal{M}'_{\mathcal{FD}.\mathcal{FS}}}_{\mathtt{int}}$.

3. En la extensión, la interpretación del puente cardinal no varía: para todo $n \in \mathbb{Z}$ y todo $s \in \mathcal{B}^{\mathcal{FS}}_{\texttt{set}}$, se cumple n `#--`$^{\mathcal{M}_{\mathcal{FD}.\mathcal{FS}}}$ s si y solamente si se cumple n `#--`$^{\mathcal{M}'_{\mathcal{FD}.\mathcal{FS}}}$ s.

La interpretación formal de la restricción `minSet`, y respectivamente de `maxSet` en el dominio $\mathcal{M}'_{\mathcal{FD}.\mathcal{FS}}$ es la siguiente: sea `minSet`$^{\mathcal{M}'_{\mathcal{FD}.\mathcal{FS}}}$ s $m \rightarrow t$, donde `minSet`$^{\mathcal{M}'_{\mathcal{FD}.\mathcal{FS}}}$ es un subconjunto del producto cartesiano $\mathcal{B}^{\mathcal{FS}}_{\texttt{set}} \times \mathbb{Z}$ que cumple alguna de las siguientes condiciones: o bien s es un conjunto distinto del conjunto vacío, m es el menor entero del conjunto s y t es `true`; o bien s es un conjunto distinto del conjunto vacío, m no es el menor entero del conjunto s y t es `false`; o bien $t = \bot$.

El sistema de transformación de almacenes que define al resolutor $solve^{\mathcal{M}_{\mathcal{FD}.\mathcal{FS}}}$, dado en la tabla 6.1, se extiende con nuevas reglas de transformación de almacenes definidas en la tabla 7.1. En concreto solo se definen las reglas para el puente `minSet` pues las reglas del puente `maxSet` son totalmente análogas. Se utiliza la notación $dom(X)$ para denotar el conjunto de valores que puede tomar la variable X, es decir, el dominio de la variable. Por lo tanto, la unión de las tablas 6.1 y 7.1 define un nuevo resolutor $solve^{\mathcal{M}'_{\mathcal{FD}.\mathcal{FS}}}$ que es una extensión del resolutor $solve^{\mathcal{M}_{\mathcal{FD}.\mathcal{FS}}}$.

M5	`minSet` $\emptyset$ Min, $\Pi \sqcup \sigma \vdash_{\mathcal{M}'_{\mathcal{FD}.\mathcal{FS}}} \blacksquare$
M6	`minSet` s Min, $\Pi \sqcup \sigma \vdash_{\mathcal{M}'_{\mathcal{FD}.\mathcal{FS}}} \Pi\sigma_1 \sqcup \sigma\sigma_1$ si $s \in \mathcal{B}^{\mathcal{FS}}_{\texttt{set}}$, $Min \in \mathcal{V}ar$ y $\exists m \in \mathbb{Z}$ t.q. `minSet`$^{\mathcal{M}'_{\mathcal{FD}.\mathcal{FS}}}$ s $m \rightarrow$ `true` y $\sigma_1 = \{Min \mapsto m\}$
M7	`minSet` s m, $\Pi \sqcup \sigma \vdash_{\mathcal{M}'_{\mathcal{FD}.\mathcal{FS}}} \Pi \sqcup \sigma$ si $s \in \mathcal{B}^{\mathcal{FS}}_{\texttt{set}}$, $m \in \mathbb{Z}$ y `minSet`$^{\mathcal{M}'_{\mathcal{FD}.\mathcal{FS}}}$ s $m \rightarrow true$
M8	`minSet` s m, $\Pi \sqcup \sigma \vdash_{\mathcal{M}'_{\mathcal{FD}.\mathcal{FS}}} \blacksquare$ si $s \in \mathcal{B}^{\mathcal{FS}}_{\texttt{set}}$, $m \in \mathbb{Z}$ y `minSet`$^{\mathcal{M}'_{\mathcal{FD}.\mathcal{FS}}}$ s $m \rightarrow false$

Tabla 7.1: Reglas de transformación de almacenes que definen $\vdash_{\mathcal{M}'_{\mathcal{FD}.\mathcal{FS}}}$

La aplicación de la regla **M5** produce fallo pues no se puede asignar un valor mínimo al conjunto vacío. La regla **M6** corresponde al caso de un puente donde el conjunto es un valor básico s y el mínimo es una variable Min. En este caso existe un número entero m tal que m se interpreta como el mínimo del conjunto s y la variable Min se sustituye por m.

Las reglas **M7** y **M8** tienen ambos argumentos básicos. En este caso o se comprueba que m es el mínimo de s y se elimina la restricción del almacén o se falla. Obsérvese, que en estas reglas no se especifica que el conjunto sea distinto del vacío, esto es consecuencia del hecho de que el mínimo es un valor básico m. Entonces, si el conjunto es vacío por la regla **M8** el cómputo falla.

Como se ha comentado anteriormente, el resolutor $solve^{\mathcal{M}'_{\mathcal{FD}.\mathcal{FS}}}$ se define mediante la unión de los dos sistemas de transformación de almacenes: $\vdash_{\mathcal{M}_{\mathcal{FD}.\mathcal{FS}}}$ definido en la tabla 6.1 y $\vdash_{\mathcal{M}'_{\mathcal{FD}.\mathcal{FS}}}$ definido en la tabla 7.1. En el siguiente teorema se enuncian las propieda-

des formales del resolutor $solve^{\mathcal{M}'_{\mathcal{FD}.\mathcal{FS}}}$. La demostración de este teorema es semejante a la demostración del teorema 10.

Teorema 14. (Propiedades formales del resolutor $solve^{\mathcal{M}'_{\mathcal{FD}.\mathcal{FS}}}$)

El sistema de transformación de almacenes definido por medio de las relaciones de transiciones $\vdash_{\mathcal{M}_{\mathcal{FD}.\mathcal{FS}}}$ y $\vdash_{\mathcal{M}'_{\mathcal{FD}.\mathcal{FS}}}$ cumple las propiedades de la definición 6. Además, como se mostró en la definición 5:

$$solve^{\mathcal{M}'_{\mathcal{FD}.\mathcal{FS}}}(\Pi) = \bigvee\{\exists\overline{Y'}(\Pi' \sqcap \sigma') \mid \Pi' \sqcap \sigma' \in SF_{\mathcal{M}'_{\mathcal{FD}.\mathcal{FS}}}(\Pi), \overline{Y'} = var(\Pi' \sqcap \sigma') \setminus var(\Pi)\}$$

está bien definido para cualquier conjunto $\Pi \in APCon_{\mathcal{M}'_{\mathcal{FD}.\mathcal{FS}}}$ y satisface los requisitos formales de la definición 4.

7.3 Cálculo $CCLNC(\mathcal{C}'_{\mathcal{FD}.\mathcal{FS}})$

El cálculo $CCLNC(\mathcal{C}_{\mathcal{FD}.\mathcal{FS}})$ descrito en la sección 6.2 se extiende para cubrir la extensión conservativa $\mathcal{M}'_{\mathcal{FD}.\mathcal{FS}}$ del dominio mediador $\mathcal{M}_{\mathcal{FD}.\mathcal{FS}}$. La extensión del cálculo se realiza añadiendo nuevas reglas que amplían la semántica operativa de dicho cálculo. En esencia se incorporan nuevas reglas que infieren información a partir de las restricciones `minSet` y `maxSet` y se amplían con nuevos casos las funciones $proj^{\mathcal{FD}\to\mathcal{FS}}$ y $proj^{\mathcal{FS}\to\mathcal{FD}}$ del cálculo $CCLNC(\mathcal{C}_{\mathcal{FD}.\mathcal{FS}})$. En esta sección se reutiliza la notación M para referirnos al almacén mediador de la extensión conservativa $\mathcal{M}'_{\mathcal{FD}.\mathcal{FS}}$.

Las reglas de la tabla 6.2 que establecen puentes, proyectan restricciones y envían restricciones a sus correspondientes almacenes son prácticamente las mismas. La regla **SB** no puede establecer puentes nuevos de mínimo y máximo para variables que puedan ser vinculadas a conjuntos vacíos. Por lo tanto la definición de $bridges^{\mathcal{FS}\to\mathcal{FD}}(\pi, B)$ no se modifica y $bridges^{\mathcal{FD}\to\mathcal{FS}}(\pi, B)$ no se incluye pues una variable de dominio finito no tiene por qué ser obligatoriamente el mínimo o máximo de un conjunto. Las reglas **PP** y **SC** se mantienen sin cambios. La regla **PP** proyecta nuevas restricciones mediante la función $proj^{\mathcal{FD}\to\mathcal{FS}}$, que se define como la unión de las tablas 6.3 y 7.2, y la función $proj^{\mathcal{FS}\to\mathcal{FD}}$, que se define como la unión de las tablas 6.4 y 7.3.

π	$proj^{\mathcal{FD}\to\mathcal{FS}}(\pi, B)$
Min_1 `/=` Min_2 (resp. `#<`, `#>`, `all_different`)	$\{S_1$ `/=` $S_2 \mid$ `minSet` $S_1\ M_1$, `minSet` $S_2\ M_2 \in B\}$
Max_1 `/=` Max_2 (resp. `#<`, `#>`, `all_different`)	$\{S_1$ `/=` $S_2 \mid$ `maxSet` $S_1\ M_1$, `maxSet` $S_2\ M_2 \in B\}$

Tabla 7.2: Extensión de proyecciones de $\mathcal{FD}$ a $\mathcal{FS}$

En la tabla 7.2 se indica que se proyectar que dos conjuntos son distintos cuando se tiene

la seguridad de que los valores mínimos de los dos conjuntos son distintos (respectivamente los valores máximos). Otras proyecciones más interesantes son las que se establecen en la tabla 7.3 y proyectan restricciones del dominio $\mathcal{FS}$ al dominio $\mathcal{FD}$. Estas proyecciones tienen efecto sobre la eficiencia de ciertos programas como se mostrará en la sección 7.6 de resultados experimentales.

π	$proj^{\mathcal{FS}\to\mathcal{FD}}(\pi, B)$
S_1 == S_2	{ M_1 == M_2 | (minSet S_1 M_1), (minSet S_2 M_2) $\in B$ } (resp. maxSet)
subset S_1 S_2 (resp. superset)	{ Min_1 #>= Min_2, Max_1 #<= Max_2 | (minSet S_i Min_i), (maxSet S_i Max_i) $\in B$, $1 \leq i \leq 2$}
intersect S_1 S_2 S_3 (resp. intersections)	{ Min_3#>=Min_1, Min_3#>=Min_2, Max_3#<=Max_1, Max_3#<=Max_2 | (minSet S_i Min_i), (maxSet S_i Max_i) $\in B$, $1 \leq i \leq 3$}
union S_1 S_2 S_3 (resp. unions)	{ Min_3#<=Min_1, Min_3#<=Min_2, Max_3#>=Max_1, Max_3#>=Max_2 | (minSet S_i Min_i), (maxSet S_i Max_i) $\in B$, $1 \leq i \leq 3$}
S_1 << S_2	{ Max_1#<Min_2, Min_1#+$Card_1$#<=Min_2, Max_1#<=Max_2#-$Card_2$ | (minSet S_i Min_i), (maxSet S_i Max_i), ($Card_i$ #-- S_i) $\in B$, $1 \leq i \leq 2$}

Tabla 7.3: Extensión de proyecciones de $\mathcal{FS}$ a $\mathcal{FD}$

Las proyecciones de la tabla 7.3 calculan los valores extremos de un conjunto en el dominio $\mathcal{FD}$, lo que hace que el resolutor de dominios finitos trabaje conjuntamente con el resolutor de conjuntos. Obsérvese que en la proyección de la restricción << se utilizan los puentes de cardinalidad, de mínimo y de máximo para formar las restricciones que se proyectan al dominio $\mathcal{FD}$.

En este punto se puede observar que una restricción se puede proyectar de un dominio a otro de varias maneras. Por ejemplo, la igualdad de dos conjuntos se proyecta al dominio finito de dos formas distintas: igualando los cardinales (Tabla 6.4), e igualando las variables mínimo y máximo (Tabla 7.3). De forma similar, la restricción subset se proyecta por una parte estableciendo un orden entre los cardinales de los conjuntos (Tabla 6.4), y por otra parte se establecen restricciones entre los valores mínimo y máximo de los conjuntos (tabla 7.3). Es decir, la extensión del dominio mediador que se propone no solo proporciona nuevos puentes sino que aporta la capacidad de trabajar con la combinación de estos. En el ejemplo 10 se verá que pueden establecerse puentes de ambos tipos sobre las mismas variables conjunto.

Las funciones de proyección definidas mediante las tablas 7.2 y 7.3 tienen propiedades de corrección y completitud similares a las vistas en la proposición 6 del capítulo 5 (página 91), por lo que no se van a enunciar de nuevo.

Volviendo a la definición de la ampliación del cálculo, terminamos esta ampliación con dos reglas nuevas que infieren información a partir del dominio mediador. Estas reglas están definidas en la tabla 7.4 y complementan la tabla 6.5. En la tabla 7.4 solo se muestran reglas

relacionadas con el puente `minSet`. Las reglas correspondientes al puente `maxSet` son análogas. La primera regla **IE** expresa que si un conjunto tiene dos puentes `minSet` con sendas variables entonces se establece una restricción de igualdad entre dichas variables. La regla **IF** infiere fallo cuando dos conjuntos son iguales pero sus mínimos son distintos.

IE Infer Equality

$\exists \overline{U}.P \sqcup C \sqcup (\texttt{minSet}\ S_1\ M_1, \texttt{minSet}\ S_1\ M_2, M) \sqcup H \sqcup F \sqcup S \vdash_{\mathbf{IE}}$
$\exists \overline{U}.P \sqcup C \sqcup (\texttt{minSet}\ S_1\ M_1, M) \sqcup H \sqcup (M_1{==}M_2, F) \sqcup S$

IF Infer Failure

$\exists \overline{U}.P \sqcup C \sqcup (\texttt{minSet}\ S_1\ M_1, \texttt{minSet}\ S_2\ M_2, M) \sqcup H \sqcup (M_1{/=}M_2, F) \sqcup (S_1{==}S_2, S) \vdash_{\mathbf{IF}} \blacksquare$

Tabla 7.4: Reglas de transformación de almacenes para la extensión de $CCLNC(\mathcal{C}'_{\mathcal{FD}.\mathcal{FS}})$

Una vez definidas todas las reglas del cálculo $CCLNC(\mathcal{C}'_{\mathcal{FD}.\mathcal{FS}})$ veamos el código $\mathcal{TOY}$ que resuelve el problema de la planificación de tareas de una forma genérica. Este código cubre el ejemplo motivador 3 de la página 6 y el ejemplo introductorio de la sección 7.1.1.

Ejemplo 10. Generalización del problema de la planificación de tareas.

Supongamos que tenemos que planificar una serie de tareas, con una duración determinada, y que se llevan a cabo en días completos. Se pueden establecer precedencias entre las tareas, así como disponer de ciertos recursos. Las tareas se representan como tX^Y_{mZ}, donde X representa el identificador de una determinada tarea, Y el tiempo que necesita la tarea en ser completada (duración), y Z es el identificador del recurso o de la máquina m que necesita la tarea para realizarse.

El programa $\mathcal{TOY}$ que se muestra a continuación modela este problema de planificación para un grafo genérico.

```
    scheduling :: int -> [iset] -> bool
    scheduling  Days Sets = true <==
1      Sets == genList durationsList Days,          % FS
2      precedences precedencesList Sets,
3      share resourcesList Sets,
4      labelingSets Sets                            % FS

    genList :: [ int ] -> int -> [ iset ]
    genList [] _ = []
    genList [D|Ds] Days = [Set| genList Ds Days] <==
5           intSet Set 1 Days,                      % FS
6           D #-- Set                               % FD y FS
```

```
    precedences :: [(int,int)] -> [iset] -> bool
    precedences [] _ = true
    precedences [(N,M)|Rest] Sets = precedences Rest Sets <==
7         (Sets !! (N-1)) << (Sets !! (M-1))          % FS

    share :: [[int]] -> [iset] -> bool
    share [] _ = true
    share [List | Rest] Sets = share Rest Sets <==
            L == transform List Sets,
8           disjoints L                                % FS

    transform :: [int] -> [A] -> [A]
    transform [] _ = []
    transform (X:Xs) Sets = [Sets !! (X-1)|transform Xs Sets]
```

En este modelo los datos se definen con las siguientes funciones de aridad cero:

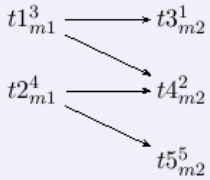
- `durationsList` devuelve las duraciones de las tareas;
- `precedencesList` devuelve una lista de pares `(i,j)` que significa que la `i`-ésima tarea debe ser realizada en su totalidad antes de la tarea `j`-ésima; y
- `resourcesList` devuelve una lista cuyos elementos son listas y cada lista contiene las tareas que comparten el mismo recurso.

Por ejemplo:

```
durationsList :: [int]
durationsList=[3,4,1,2,5]

precedencesList :: [(int,int)]
precedencesList=[(1,3),(1,4),(2,4),(2,5)]

resourcesList :: [[int]]
resourcesList=[[1,2],[3,4,5]]
```



La función `scheduling` encuentra, si es posible, una planificación para una serie de días (`Days`), que se proporciona como argumento de entrada. Lo primero que hace esta función es generar una lista `Sets` que representa las distintas tareas, línea 1. Las variables de esta lista son conjuntos que pueden contener elementos entre `1` y `Days` (línea 5),

pues las tareas no requieren realizarse necesariamente en días consecutivos. Además, la función `genList` establece puentes de cardinalidad `#--` en la línea 6, pues la duración de cada tarea corresponde al cardinal del conjunto que representa dicha tarea. Como se ha dicho anteriormente, las duraciones de las tareas vienen determinadas por la función `durationsList` que se utiliza como argumento de la función `genList` en la línea 1.

Las restricciones de precedencia y de compartición de recursos se establecen en las líneas 2 y 3. La función `precedences` usa la restricción de conjuntos `<<` para imponer las relaciones de precedencia entre tareas (línea 7). La función `share` impone la restricción `disjoints` a las tareas que requieren compartir el mismo recurso (línea 8). Finalmente, todas las soluciones se encuentran etiquetando los conjuntos en la línea 4.

La proyección propaga la información de las restricciones de conjuntos al dominio $\mathcal{FD}$. Los dominios de las variables $\mathcal{FD}$ se reducen por los mecanismos de propagación podando el árbol de búsqueda y mejorando el rendimiento como se mostrará en la subsección 7.6. Se puede observar que las restricciones `minSet` y `maxSet` no se usan explícitamente en el programa pues se proyecta la información sobre las correspondientes variables de una forma transparente. Así, las restricciones `<<` y `disjoints` envían restricciones al dominio $\mathcal{FD}$ cuando las proyecciones están activadas reduciendo en general el espacio de búsqueda.

7.4 Resultados de corrección y completitud limitada

Como se ha mostrado anteriormente, el dominio mediador $\mathcal{M}_{\mathcal{FD}.\mathcal{FS}}$ se ha extendido obteniendo un nuevo dominio mediador $\mathcal{M}'_{\mathcal{FD}.\mathcal{FS}}$, el dominio de coordinación $\mathcal{C}_{\mathcal{FD}.\mathcal{FS}}$ se ha extendido al dominio de coordinación $\mathcal{C}'_{\mathcal{FD}.\mathcal{FS}}$ y el cálculo $CCLNC(\mathcal{C}_{\mathcal{FD}.\mathcal{FS}})$ al cálculo $CCLNC(\mathcal{C}'_{\mathcal{FD}.\mathcal{FS}})$. En la sección 6.3 se presentaron resultados semánticos de corrección y completitud limitada para el cálculo $CCLNC(\mathcal{C}_{\mathcal{FD}.\mathcal{FS}})$ con respecto a dominio de coordinación $\mathcal{C}_{\mathcal{FD}.\mathcal{FS}}$. Los resultados semánticos de corrección y completitud limitada para el cálculo $CCLNC(\mathcal{C}'_{\mathcal{FD}.\mathcal{FS}})$ son una extensión natural de los correspondientes resultados del cálculo anterior. Por este motivo se omite en este capítulo el teorema de corrección local y completitud local limitada del cálculo $CCLNC(\mathcal{C}'_{\mathcal{FD}.\mathcal{FS}})$ análogo al teorema 11 y el lema de progreso análogo al lema 5.

A continuación se muestran los resultados generales de corrección y completitud limitada. La demostración es similar a las demostraciones de los teoremas 12 y 13.

Teorema 15. Corrección

Sea $\mathcal{P}$ un programa $CFLP(\mathcal{C}'_{\mathcal{FD}.\mathcal{FS}})$, G un objetivo admisible para $\mathcal{P}$ y S un objetivo resuelto tal que $G \vdash^*_{\mathcal{P}} S$ utilizando las reglas de las tablas 4.1, 4.2, 6.2, 6.5, 6.6 y 7.4. Entonces, $Sol_{\mathcal{C}'_{\mathcal{FD}.\mathcal{FS}}}(S) \subseteq Sol_{\mathcal{P}}(G)$.

Teorema 16. Completitud limitada

Sea G un objetivo para un programa $\mathcal{P}$ y $\mu \in WTSol_{\mathcal{P}}(G)$ una solución bien tipada para dicho objetivo. Asumimos que ni $\mathcal{P}$ ni G contienen apariciones libres de variables de orden superior y las aplicaciones de las reglas de las tablas 4.1, 4.2, 6.2, 6.5, 6.6 y 7.4 son seguras, es decir, no producen descomposiciones opacas ni invocaciones incompletas de resolutores. Entonces se puede encontrar un cómputo $CCLNC(\mathcal{C}'_{\mathcal{FD}.\mathcal{FS}})$ de la forma $G \mathrel{+\!\!\!\vdash}^* S$, terminando con un objetivo en forma resuelta S tal que $\mu \in WTSol_{\mathcal{C}'_{\mathcal{FD}.\mathcal{FS}}}(S)$.

7.5 Implementación

En esta sección se trata la extensión de la implementación en el sistema $\mathcal{TOY}$ que cubre el cálculo $CCLNC(\mathcal{C}'_{\mathcal{FD}.\mathcal{FS}})$. Al tratarse de una extensión, la arquitectura del sistema no varía así como la estructura general de las restricciones primitivas atómicas, y se mantienen los tres modos de uso que se mostraron en la sección 6.4. En esta sección se describe la implementación correspondiente a los puentes `minSet` y `maxSet` y a las proyecciones de ciertas restricciones a través de estos puentes. A partir de ahora, en las explicaciones solo se mostrará el puente `minSet`, pues `maxSet` se simétrico al anterior.

7.5.1 Implementación del puente `minSet`

El predicado que implementa el puente `minSet` correspondiente a la parte SICStus es semejante al predicado del puente cardinal visto en el capítulo anterior (figura 6.8). Fundamentalmente se diferencian en que en la restricción `minSet` el conjunto debe ser distinto del vacío y que los datos que obtiene SICStus con respecto al cardinal se deben actualizar.

En la parte ECL^{i}PS^e, la implementación del puente `minSet` varía con respecto a la correspondiente implementación del cardinal (figura 6.10), pues en este caso no se dispone en ECL^{i}PS^e de una restricción apropiada como la restricción de cardinalidad (`#(?Set,?Card)`). Así, el puente `minSet` se ha implementado utilizando herramientas de programación disponibles en el sistema, en particular se han utilizado objetivos suspendidos que se activan cuando alguno de los dos argumentos se restringe durante el cómputo. Esta suspensiones corresponden a las líneas 1 y 2 de la figura 7.8 y su explicación se dejará para el final.

En la línea 3 se tiene en cuenta si el valor mínimo es básico. En este caso, línea 4, se llama al predicado `constrainNumberMin` que restringe al mínimo a ser el menor valor del infimo del retículo y elimina del supremo del retículo todos los elementos que son menores que el mínimo. Así, por ejemplo, un objetivo de la forma `domainSets [S] (s [3]) (s [1,2,3]), minSet S 1` restringe la variable `S` al retículo `[{1,3},{1,2,3}]`, un objetivo de la forma `domainSets [S] (s [3]) (s [1,2,3]), minSet S 3` vincula la variable `S` al conjunto `{3}`, mientras que el objetivo `domainSets [S] (s [2,3]) (s [1,2,3]), minSet S 3` falla, pues

```
minSet(Set,Min):-
 #(Set,C), C #> 0,
1 suspend(minSet_demon_Min(Min,Set,SMin),0,[Min->constrained],SMin),
2 suspend(minSet_demon_Set(Min,Set,SSet),0,[Set->constrained],SSet),
3 (number(Min) ->
4     constrainNumberMin(Min,Set)
  ;
5     constrain_min_Min(Min,Set),
6     constrain_min_Set(Min,Set)
  ).

constrainNumberMin(Min,Set):-
        ic_sets:(Min in Set),
        ic_sets:set_range(Set,Lwb,Upb),
        Lwb=[Min|_],
        selectLargerEqValues(Upb,Min,New),
        ic_sets:(Set::Lwb..New).

constrain_min_Min(Min,Set):-
        ic:get_min(Min,MinNew),
        ic_sets:set_range(Set,Lwb,Upb),
        selectLargerEqValues(Upb,MinNew,New),
        ic_sets:(Set :: Lwb..New).

constrain_min_Set(Min,Set) :-
        ic_sets:set_range(Set,Lwb,Upb),
        (Lwb == [] ->
                min_list(Upb,UpbMin),
                max_list(Upb,UpbMax),
                ic:(Min #:: UpbMin .. UpbMax)
        ;
                min_list(Lwb,LwbMin),
                min_list(Upb,UpbMin),
                ic:(Min #:: UpbMin .. LwbMin)
        ).
```

Figura 7.8: Implementación del puente `minSet` en la parte ECL^{i}PS^e

la restricción `domainSets` impone que el mínimo esté comprendido entre `1` y `2` y la restricción `minSet` impone que el mínimo del conjunto sea `3`, llegando a una contradicción. El predicado de librería `set_range` obtiene y el infimo y el supremo de una variable de tipo conjunto.

Si el mínimo es una variable de dominio finito entonces se debe restringir el conjunto `Set` con respecto al mínimo (línea 5) y la variable `Min` con respecto al conjunto (línea 6). En concreto, el predicado `constrain_min_Min` elimina del supremo del retículo todos los elementos que son menores que el mínimo y el predicado `constrain_min_Set` restringe el dominio de la variable `Min` a estar comprendido entre el menor y el mayor valor del supremo del retículo si `Lwb == []`, o bien restringe el dominio de la variable `Min` entre el menor valor del supremo del retículo y el menor valor del ínfimo del retículo. Así, el objetivo `domainSets [S] (s [2,3]) (s [1,2,3,4]), minSet S M` asigna a la variable `M` el intervalo entero `1..2` y el objetivo `domainSets [S] (s []) (s [1,2,3,4]), minSet S M` asigna a `M` el intervalo `1..4`. Pero la variable `M` puede tener un dominio inicial distinto del intervalo que se le asigna, en este caso se actualiza el retículo y el mínimo convenientemente. Por ejemplo, el objetivo `domainSets [S] (s [3]) (s [1,2,3,4,5]), domain [M] 2 4, minSet S M` reduce la variable `M` al intervalo `2..3` y el retículo `S` a `[{3},{2,3,4,5}]`. Los predicados `min_list` y `max_list` obtiene el minimo y el máximo de una lista Prolog de números enteros.

Como se explicó anteriormente, las líneas 1 y 2 suspenden los predicados `minSet_demon_Min` y `minSet_demon_Set` que actúan como se explica a continuación. En ECL^{i}PS^e, los objetivos se suspenden mediante el predicado `suspend(+Objetivo, +Prioridad, +Cond, -Susp)`. La idea es que el objetivo queda esperando hasta que se le despierte, con un cierta prioridad `Prioridad`, tan pronto como la condición `Cond` se haga cierta. El cuarto parámetro `Susp` es un identificador de la suspensión creada. ECL^{i}PS^e utiliza un modelo de ejecución que se basa en las prioridades de los objetivos y que garantiza que el objetivo programado con mayor prioridad se ejecute siempre antes que cualquier objetivo de menor prioridad. La prioridad es un número entero que varía de 1 a 12, siendo 1 la prioridad más alta y 12 la más baja. Además, en una lista de términos o un término simple, como en nuestro caso, que representan las condiciones que se establecen para que el objetivo se despierte. Los casos predefinidos de suspensión son: `'inst'` (para la instanciación de variables), `'bound'` (instanciación o alias a otra variable) y `'constrained'` (cualquier modificación de los atributos de restricción).

Un objetivo *demonio* se diferencia de un objetivo común únicamente en su comportamiento al despertar. Mientras que un objetivo suspendido desaparece del resolutor cuando se despierta, si está marcado como un demonio sigue permaneciendo en el resolutor después de despertarse. En ECL^{i}PS^e, un objetivo que invoca un predicado marcado como demonio se añade a la lista de objetivos suspendidos cada vez que se despierta. Por ello, hay que matarlo explícitamente cuando ya no se necesite. Por este motivo, se pasa el identificador de la suspensión en uno de los argumentos y así se puede matar cuando se requiera mediante el predicado `kill_suspension`.

La figura 7.9 muestra uno de los demonios que se han implementado para el puente `minSet`. Primero se especifica que el predicado `minSet_demon_Min` es un demonio. Las dos cláusulas de este predicado corresponden a los casos de mínimo básico y no básico, respectivamente. En ambos casos se llaman a predicados mostrados en la figura 7.8. Si el mínimo es básico se mata al demonio y en caso contrario se deja activo para que actúe cuando se vuelvan a restringir las variables del puente `minset`.

Por ejemplo, anteriormente se mostró que el objetivo

```
:- demon(minSet_demon_Min/3).
minSet_demon_Min(Min,Set,Susp):-
          number(Min),
          !,
          constrainNumberMin(Min,Set),
          kill_suspension(Susp).
minSet_demon_Min(Min,Set,_):-
          constrain_min_Min(Min,Set).
```

Figura 7.9: Implementación del demonio minSet_demon_Min del puente `minSet`

```
domainSets [S] (s [3]) (s [1,2,3,4,5]), domain [M] 2 4, minSet S M
```

que reduce `M` al intervalo `[2,3]` y `S` al retículo `[{3},{2,3,4,5}]`. Si a este objetivo se le añade la restricción `M #> 2` entonces la variable `M` se vincula a 3 y se despierta el demonio `minSet_demon_Min` reduciendo el retículo a `[{3},{3,4,5}]`.

La figura 7.10 muestra el segundo demonio implementado para el puente `minSet`. Este demonio se ejecuta cuando se restringe el conjunto, si llega a ser básico se actualiza el mínimo y se mata al demonio. En caso contrario se mantiene el demonio y se trata el mínimo convenientemente.

```
:- demon(minSet_demon_Set/3).
minSet_demon_Set(Min,Set,Susp):-
         ground(Set),
         !,
         ic_sets:set_range(Set,_,Upb),
         min_list(Upb,Min),
         kill_suspension(Susp).
minSet_demon_Set(Min,Set,_):-
         constrain_min_Set(Min,Set).
```

Figura 7.10: Implementación del demonio minSet_demon_Set del puente `minSet`

7.5.2 Implementación de la primitiva << y su proyección

En la definición del dominio $\mathcal{FS}$ en la sección 3.4 (página 57) se definió una restricción primitiva que no tenía correspondencia con las restricciones primitivas del sistema ECL^{i}PS^e Prolog. Esta primitiva es << y se va a mostrar en esta sección su implementación. Recordemos que la restricción S_1 << S_2 indica que el mayor elemento perteneciente al conjunto S_1 es menor que el menor elemento del conjunto S_2. Otra peculiaridad de esta primitiva es su proyección,

```
'<<'(A, B, true, Cin, Cout) :-
  % tratamiento inicial de las variables
  ..........
1 ( proj_active -> ProjFlag = 4
  ..........
  ; projCard_active -> ProjFlag = 1
  ; ProjFlag = 0
  ),
2 DataPrj=(ProjFlag,_,IdCardA,_,IdCardB,IdMinA,IdMaxA,IdMinB,IdMaxB),
3 execute_eclipse(Sin, Sout,lesslessE(IdA,IdB,DataPrj), Result),
      % tratamiento del resultado.
```

Figura 7.11: Implementación de la primitiva << en la parte SICStus

pues de todas las restricciones que se proyectan siguiendo las tablas 7.2 y 7.3, la restricción << es la única que utiliza los puentes `minSet` y `maxSet` definidos en este capítulo junto con el puente de cardinalidad definido en el capítulo anterior.

En la implementación de << se ha utilizado la suspensión de objetivos y el uso de demonios. La parte de << implementada en SICStus se muestra en la figura 7.11, mientras que la parte implementada en ECL^{i}PS^e se muestra en la figura 7.12. La implementación de las proyecciones de esta primitiva se muestra en la figura 7.13.

La figura 7.11 muestra esquemáticamente la implementación de la primitiva <<. En esta figura se han omitido los detalles que son comunes al esquema general de las primitivas implementadas en SICStus, descritos en el capítulo anterior. Como se ha comentado anteriormente, esta restricción tiene como peculiaridad que es la única restricción primitiva que requiere de todos los puentes para poder realizar la proyección. Para comunicar a ECL^{i}PS^e el tipo de proyecciones que están activadas, se utiliza la variable `ProjFlag` que toma valores distintos dependiendo de las proyecciones activadas por el usuario. Se tratan todas las proyecciones cuando se han activado mediante el comando `/proj` y se almacena junto con los identificadores de las variables en 2 para ser enviada a ECL^{i}PS^e en la línea 3. Obsérvese que el sistema está preparado para tratar más tipos de proyecciones. Aunque en el prototipo solo se permite la proyección si ambos puentes están disponibles, en 1 se trata de forma detallada qué puentes están habilitados para permitir posteriores extensiones.

La figura 7.12 muestra el tratamiento de la primitiva << en el proceso ECL^{i}PS^e. La primera cláusula de la figura 7.12 trata el caso de que los dos argumentos que representan conjuntos son variables. En este caso lo único que se puede afirmar es que son disjuntos y esperar a que alguno sea básico para poder establecer el dominio del otro conjunto. Esta espera se realiza por medio de suspensiones de predicados definidos como demonios que se detallan más adelante en la figura 7.14.

Las siguientes dos cláusulas corresponden al caso en el que uno de los conjuntos es básico. Si el conjunto básico es el conjunto vacío, la restricción se satisface trivialmente. Si no es

```
'<<'(S1,S2) :-
    var(S1),var(S2), !,
    (proj_mm_active -> projection_mm(S1,S2) ; true),
    ic_sets:disjoint(S1,S2),
    suspend(mm_demon1(S1,S2,Susp1,Susp2),6,S1->inst,Susp1),
    suspend(mm_demon2(S1,S2,Susp2,Susp1),6,S2->inst,Susp2).

'<<'(S1,S2):-
    ground(S1), var(S2), !,
    set_range(S1,_,Upb1),
    (Upb1 == [] ->
      true
    ;
      ic_sets:disjoint(S1,S2),
      (proj_mm_active -> projection_mm(S1,S2); true),
      suspend(mm_demon2(S1,S2,Susp2,none),6,S2->inst,Susp2)
    ).

'<<'(S1,S2):-
    var(S1), ground(S2), !,
    % simétrica a la cláusula anterior

'<<'(S1,S2):-
    ground(S1), ground(S2), !,
    set_range(S1,_,Upb1),
    set_range(S2,_,Upb2),
    ( Upb1 == [] ->
        true
    ; ( Upb2 == [] ->
          true
      ;
          max_list(Upb1,Max1),
          min_list(Upb2,Min2),
          Max1 < Min2
      )
    ).
```

Figura 7.12: Implementación de la primitiva << en ECL^{i}PS^e

el conjunto vacío, además de aplicar la restricción de ser disjuntos, se suspende el objetivo clasificado como demonio en 2.3. Finalmente, la última cláusula trata el caso de que los dos

conjuntos sean básicos. En este caso hay que comprobar que el máximo del primero es menor que el mínimo del segundo si los dos son distintos del vacío.

La implementación de la proyección la restricción << sobre el dominio $\mathcal{FD}$ se muestra en la figura 7.13. Esta proyección está definida en la tabla 7.3 y delimita los extremos de las variables $\mathcal{FD}$ que representan el mínimo y el máximo teniendo en cuenta las cardinalidades de los conjuntos.

```
projection_mm(S1,S2) :-
   #(S1,Card1), #(S2,Card2),
   ic:get_min(Card1,MinCard1), ic:get_min(Card2,MinCard2),
   ((MinCard1>0,MinCard2>0) ->
            minSet(S1,Min1), maxSet(S1,Max1),
            minSet(S2,Min2), maxSet(S2,Max2),
            Min1 #=< Max1, Min2 #=< Max2,
            Max1 #< Min2,
            Min1 + Card1 #=< Max1 + 1,
            Min1 + Card1 #=< Min2,
            Min2 + Card2 #=< Max2 + 1,
            Max1 #=< Max2 - Card2
   ;
            true
   ).
```

Figura 7.13: Implementación de la proyección de << desarrollada en ECL^{i}PS^e

Finalmente, en la figura 7.14 se muestra uno de los predicados clasificado como demonio que corresponde a la situación en que el primer argumento `S1` de la restricción se hace básico. El segundo demonio es simétrico a este. Si `S1` es distinto del conjunto vacío, se utiliza el elemento máximo del conjunto para recalcular el supremo del retículo que define al otro conjunto `S2`. De hecho, se impone al conjunto `S2` que todos sus elementos estén contenidos en la lista obtenida de cribar los elementos mayores que el máximo de `S1`. Por último, después de matar al correspondiente demonio también se mata el demonio que trata la instanciación de la segunda variable si estuviera suspendido.

Nótese que los predicados `mm_demon1` y `mm_demon2` se han definido como demonios aunque solo se ejecutan una vez. Esto es debido a que en el caso de la primera cláusula de << (figura 7.12) se dejan los dos predicados suspendidos y en cuanto un conjunto se hace básico el demonio correspondiente debe matar a los dos demonios.

7.6 Resultados experimentales

El enfoque propuesto en este capítulo también ha sido probado para medir la mejora de la eficiencia de las proyecciones generadas por el mecanismo de cooperación. De los tres

```
:- demon(mm_demon1/4).
mm_demon1(S1,S2,Susp1,Susp2) :-
      ground(S1), !,
      set_range(S1,_,Upb1),
      set_range(S2,Lwb2,Upb2),
      (Upb1 == [] ->
          true
      ;
          max_list(Upb1,Max1),
          selectLargerValues(Upb2,Max1,New),
          ic_sets:(S2 :: Lwb2..New)
      ),
      kill_suspension(Susp1),
      (Susp2 == none ->
          true
      ;
          kill_suspension(Susp2)
      ).
```

Figura 7.14: Implementación del demonio

modos de ejecución de $\mathcal{TOY}$, se ha utilizado el modo ECL^{i}PS$^e_{gen}$ para realizar las pruebas de este capítulo pues es el único modo que no tiene interacción entre procesos. De esta forma se mide únicamente el tiempo de resolución de restricciones. Estos experimentos no han podido compararse con problemas documentados del sitio web de ECL^{i}PS^e, pues con las primitivas de conjuntos básicas es muy dificil representar este problema sin recurrir al mecanismo etiquetado de vuelta atrás.

Los experimentos de las tablas 7.5 y 7.6 han sido realizados en un sistema con un procesador Intel© Core™ i7-740QM (4 cores) a 1.73 GHz con 3 GB de memoria física y sobre Ubuntu 14.04. Por cada experimento se ha calculado la media del tiempo de cinco ejecuciones. Las tablas 7.5 y 7.6 muestran el tiempo de ejecución en milisegundos. Los experimentos marcados con un guión corresponden a valores no calculables.

Las tablas 7.5 y 7.6 muestran el tiempo de ejecución que toman los distintos problemas de planificación descritos en los ejemplos anteriores, en buscar una o todas las soluciones respectivamente. La columnas encabezadas con **Benchmark** contienen objetivos con distintos parámetros. En concreto, las pruebas etiquetadas con el prefijo `scheduling1` corresponden al grafo del ejemplo 10, `scheduling2` corresponde al grafo del ejemplo motivador visto en 1.1 y por último `scheduling_house` corresponde al proyecto de construir una casa que se mostró en la subsección 7.1.1. El número que se encuentra al final del nombre de cada prueba es el total de días disponibles y se usa en los objetivos como argumento de la correspondiente

función. Las columnas **No Proj.** y **Proj.** corresponden a la no activación y activación de las proyecciones utilizando los puentes de cardinal, mínimo y máximo, respectivamente. Las columnas con encabezado **No Proj./Proj.** contienen la ganancia de velocidad de la columna **Proj.** con respecto a la columna **No Proj.** Además, existe otra columna relativa a las proyecciones, **Proj-Disj.** que se introduce más adelante. En la columna **#Sols.** se muestra el número de soluciones encontradas. En la tabla 7.5 se muestra con `1` que se ha encontrado una primera solución para el objetivo, mientras que un `0` indica que el objetivo ha fallado.

Benchmark	**No Proj.**	**Proj.**	**No Proj./Proj.**	**Proj-Disj.**	**#Sols.**
scheduling1_9	26	4	6.50	0	0
scheduling1_10	84	0	-	8	0
scheduling1_11	376	2	188.00	50	0
scheduling1_12	856	26	32.92	240	1
scheduling1_13	718	44	16.32	252	1
scheduling1_14	0	6	0.00	4	1
scheduling1_15	0	6	0.00	2	1
scheduling2_15	204	0	-	0	0
scheduling2_16	1,140	0	-	2	0
scheduling2_17	4,976	0	-	0	0
scheduling2_18	20,802	4	5,200.50	78	0
scheduling2_19	150,104	2	75,052.00	864	0
scheduling2_20	0	10	0.00	4	1
scheduling_house_15	876	8	109.50	10	0
scheduling_house_16	656	10	65.60	8	1
scheduling_house_17	1,130	10	113.00	8	1
scheduling_house_18	1,216	10	121.60	10	1

Tabla 7.5: Resultados para la primera solución

En estos experimentos se puede observar que, en general, cuando las proyecciones están habilitadas, el tiempo de ejecución que se emplea es claramente menor que si no están habilitadas, o bien la diferencia no es relevante. Como se puede comprobar, en los objetivos que no tienen solución, el tiempo de ejecución aumenta progresivamente pues el cómputo tiene que recorrer todo el árbol de búsqueda. En el resto la primera solución se encuentra rápidamente, pues el número de soluciones es tan alto que es más rápido probar una combinación cualquiera que evaluar nuevas restricciones para tratar la consistencia de los valores extremos de los conjuntos en el dominio $\mathcal{FD}$. El número total de soluciones de estos objetivos se muestra en la tabla 7.6.

Por otra parte, en todos estos objetivos, cuando las proyecciones están habilitadas se generan restricciones de acuerdo a las tablas 7.3 y 6.4. Además de evaluar el uso de las proyecciones en general, en estos experimentos se ha medido cómo influye en particular la proyección de `disjoints` (tabla 6.4) que podría introducir una sobrecarga importante. La

columna **Proj-Disj.** muestra la ejecución de los mismos objetivos con proyecciones excepto la de `disjoints`. Si comparamos las columnas **Proj.** y **Proj-Disj.** de la tabla 7.5, se puede observar una diferencia relevante en los objetivos `scheduding1` de `11`, `12` y `13` días y `scheduding2` de `18` y `19` días. Observando estos casos, se puede deducir que las proyecciones de `disjoints` ayudan efectivamente a podar el árbol de búsqueda. En el resto de objetivos las proyecciones de las restricciones `<<` son suficientes para anticipar el fallo o encontrar una solución rápidamente y las proyecciones de las restricciones `disjoints` no son relevantes.

Nótese que en los objetivos correspondientes a la construcción de la casa, la diferencia entre las columnas **Proj.** y **Proj-Disj.** no es significativa para ningún caso. Esto es debido a la propia construcción del grafo. Es decir, un grafo con más relaciones de precedencia saca más provecho de las proyecciones de `<<` . Por otra parte, el grafo de la figura 7.1 tiene 8 nodos y comparten recursos dos a dos. Esto no provoca una gran poda por parte de las proyecciones de las restricciones `disjoints`.

Benchmark	**No Proj.**	**Proj.**	**No Proj./Proj.**	**Proj-Disj.**	**#Sols.**
scheduling1_12	990	70	14.14	346	84
scheduling1_13	3,418	1,086	3.15	2,212	2,520
scheduling1_14	7,504	8,570	0.88	11,326	35,385
scheduling1_15	24,970	64,276	0.39	65,712	316,680
scheduling2_20	1,164,054	4,818	241.61	9,724	20,790
scheduling_house_16	4,242	46	92.22	48	72
scheduling_house_17	22,996	2,794	8.23	2,354	11,592
scheduling_house_18	149,904	112,404	1.33	92,840	600,992

Tabla 7.6: Resultados para todas soluciones

La tabla 7.6 muestra el tiempo que tarda el sistema en buscar todas las soluciones. Nótese que los objetivos que no tienen ninguna solución no se muestran porque el tiempo que se tarda en recorrer el árbol de búsqueda completo es el mismo tiempo que el mostrado en la tabla 7.5. Cuando se buscan todas las soluciones, se puede observar que la ganancia de velocidad disminuye según va aumentando el número de soluciones. En concreto, para los objetivos `scheduding1` de `14` y `15` días, la ganancia de velocidad es menor a 1. Esto es debido a la sobrecarga de la propagación de las restricciones proyectadas

En la tabla 7.6 la diferencia entre las columnas **Proj.** y **Proj-Disj.** es significativa prácticamente en todos los objetivos. Para `scheduding1` se puede observar que la activación de las proyecciones en `disjoints` en general mejora la eficiencia de los objetivos evaluados. Sin embargo, la diferencia entre estas columnas se va reduciendo según aumenta el número de días y también el número de soluciones. Es decir, cuando el árbol de búsqueda es demasiado grande, la poda que hacen las restricciones generadas por las proyecciones no compensa el esfuerzo que se tiene que hacer para calcular la unión de conjuntos y restringir el cardinal de la unión a la suma de los cardinales.

En `scheduling_house` este efecto es aún más relevante. De hecho, `scheduling_house` de `17` y `18` días es más rápido si únicamente se aplica la proyección de la restricción `<<`

(columna **Proj-Disj.**) que si se aplican todas las proyecciones (columna **Proj**). Como ya se ha anticipado, cuando el número de soluciones es alto, no compensa comprobar que el cardinal de la unión de una serie de conjuntos es igual a la suma de los cardinales de dichos conjuntos.

Como conclusión de los resultados experimentales mostrados en las tablas 7.5 y 7.6 se deduce que las proyecciones mejoran los tiempos de ejecución en casi todos los casos. Como se puede observar en la tabla 7.6 según aumenta el número de soluciones de un mismo problema (cuando se amplía el árbol de búsqueda) la proyección es menos efectiva de tal manera que a partir de un cierto punto no compensa la propagación de las restricciones proyectadas, pues debe calcularse la unión de conjuntos y restringir el cardinal de la unión a la suma de los cardinales. Por último, también se deduce que ambas proyecciones intervienen en la mejora de la eficiencia en la mayoría de los casos al obtener la primera solución.

Capítulo 8

Conclusiones y trabajo futuro

Esta tesis contribuye al estudio de la cooperación entre los dominios de restricciones y resolutores en los lenguajes y sistemas declarativos. En particular, se ha estudiado la cooperación entre dominios de restricciones en el lenguaje y sistema lógico funcional con restricciones $\mathcal{TOY}$. Este estudio se ha centrado en dos casos particulares de importancia práctica que involucran los dominios puros $\mathcal{H}$ (Herbrand), $\mathcal{FD}$ (dominio finito), $\mathcal{R}$ (reales) y $\mathcal{FS}$ (conjuntos finitos). Los dos casos particulares de cooperación son:

$$\mathcal{C}_{\mathcal{FD}.\mathcal{R}} = \mathcal{M}_{\mathcal{FD}.\mathcal{R}} \oplus \mathcal{H} \oplus \mathcal{FD} \oplus \mathcal{R} \qquad \text{y} \qquad \mathcal{C}_{\mathcal{FD}.\mathcal{FS}} = \mathcal{M}_{\mathcal{FD}.\mathcal{FS}} \oplus \mathcal{H} \oplus \mathcal{FD} \oplus \mathcal{FS}$$

Como resultado, esta tesis aporta un modelo formal computacional para la resolución de objetivos, el desarrollo de un prototipo que implementa este modelo, y una comparativa con los sistemas relacionados más próximos. Concretamente, hay que destacar las siguientes conclusiones:

- Se ha definido una primitiva completamente nueva en el dominio $\mathcal{FS}$ (<<) y, por lo que sabemos, no disponible en otros sistemas. Al dotar a $\mathcal{TOY}$ de una nueva primitiva, su capacidad expresiva aumenta y permite definir de forma más compacta problemas complejos.

- Los resolutores de los dominios $\mathcal{FD}$ y $\mathcal{FS}$ se han ampliado para detectar el fallo antes de enviar las restricciones a los resolutores. Para ello se han dividido los resolutores en dos capas. La primera capa es una caja transparente especificada mediante un sistema de transformación de almacenes y cuyo objetivo es anticipar el fallo. La segunda capa está basada en un resolutor externo que desde el marco $CFLP$ se ve como una caja negra que debe cumplir ciertos requisitos formales. Se ha demostrado que los resolutores de ambos dominios, $\mathcal{FD}$ y $\mathcal{FS}$, son correctos y completos con ciertas limitaciones. Se puede concluir que este enfoque de dividir los dominios en una parte sintáctica y un resolutor externo es idóneo para anticipar el fallo.

- La cooperación entre dominios se ha definido utilizando dominios mediadores ($\mathcal{M}_{\mathcal{FD}.\mathcal{R}}$ y $\mathcal{M}_{\mathcal{FD}.\mathcal{FS}}$) cuyos resolutores se han definido como cajas transparentes mediante sistemas de transformación de almacenes. Se ha demostrado que los resolutores de ambos

dominios son correctos y completos. Se concluye que la idea de utilizar dominios que soporten la comunicación entre otros dominios puros es especialmente conveniente en lenguajes fuertemente tipados como $\mathcal{TOY}$, pues facilitan la identificación de restricciones con sus respectivos dominios.

- Se han definido varios puentes pertenecientes a los dominios mediadores $\mathcal{M}_{\mathcal{FD}.\mathcal{R}}$ y $\mathcal{M}_{\mathcal{FD}.\mathcal{FS}}$ como herramientas adicionales para escribir programas más declarativos. En concreto, se ha definido un puente para la cooperación entre $\mathcal{FD}$ y $\mathcal{R}$ que equipara variables enteras y reales al mismo valor, y tres puentes para la cooperación entre $\mathcal{FD}$ y $\mathcal{FS}$, uno que relaciona las variables enteras y los conjuntos mediante la cardinalidad, y otros dos que propagan al dominio $\mathcal{FD}$ los valores extremos de los conjuntos. Con el uso de los puentes se tienen más posibilidades de codificar programas, pues se amplía el número de dominios que intervienen. Consecuentemente, se obtiene como resultado de esta tesis un nuevo enfoque en la programación $CFLP$ con múltiples dominios de restricciones.

- Los puentes permiten proyectar restricciones entre dominios, de manera que no sólo se trata de un mero intercambio de datos sino que además se propaga información que modifica el modelo en los dominios proyectados. Las proyecciones mejoran en varios casos la eficiencia de los programas, como se muestra en las comparativas descritas en esta tesis. Por lo tanto, otro resultado de esta tesis es la inclusión de proyecciones y la comprobación experimental de su conveniencia.

- Sobre los dominios de coordinación se han desarrollado sendos cálculos de resolución de objetivos ($CCLNC(\mathcal{C}_{\mathcal{FD}.\mathcal{R}})$ y $CCLNC(\mathcal{C}_{\mathcal{FD}.\mathcal{FS}})$). Estos cálculos se definen mediante reglas de cómputo que infieren igualdades, desigualdades y fallo, y crean nuevos puentes y nuevas restricciones resultado de la proyección entre dominios. Se ha demostrado que $CCLNC(\mathcal{C}_{\mathcal{FD}.\mathcal{R}})$ y $CCLNC(\mathcal{C}_{\mathcal{FD}.\mathcal{FS}})$ son correctos y completos con limitaciones. Forma parte de las publicaciones de las que resulta esta tesis la confirmación de la validez formal de ambos cálculos.

- Con respecto a la implementación, se ha desarrollado un prototipo que permite las dos cooperaciones introducidas anteriormente. Por una parte, la implementación de la cooperación entre los dominios $\mathcal{H}$, $\mathcal{FD}$ y $\mathcal{R}$ utiliza las bibliotecas de los resolutores de SICStus Prolog. Por otra parte, la implementación de la cooperación entre los dominios $\mathcal{H}$, $\mathcal{FD}$ y $\mathcal{FS}$ permite comunicar el sistema $\mathcal{TOY}$, desarrollado en SICStus Prolog, con las bibliotecas de los resolutores del sistema ECL^{i}PS^e. Este prototipo confirma la validez experimental del enfoque propuesto y, además, aporta una forma de comunicar $\mathcal{TOY}$ con el sistema ECL^{i}PS^e.

- En la cooperación entre los dominios $\mathcal{H}$, $\mathcal{FD}$ y $\mathcal{FS}$ se han implementado distintas estrategias o modos de comunicación. En esencia hay dos estrategias: una basada en la comunicación interactiva entre ECL^{i}PS^e y $\mathcal{TOY}$, y otra que genera un programa ECL^{i}PS^e que contiene todas las restricciones primitivas atómicas creadas por el proceso de estrechamiento de $\mathcal{TOY}$. La primera estrategia está enfocada al razonamiento

de modelos, es decir, cuando el modelo que representa a la especificación del programa puede ser modificado durante la resolución del objetivo. En esta estrategia el proceso ECL^{i}PS^e actúa como servidor aceptando y ejecutando las peticiones que le son enviadas desde el proceso $\mathcal{TOY}$ durante el cómputo de un objetivo. Una variante de esta estrategia consiste en almacenar una serie de restricciones que se envían juntas al servidor ECL^{i}PS^e, lo que permite mejorar el tiempo de ejecución. La segunda estrategia está enfocada a problemas CP que primero especifican las restricciones y luego se resuelven. En esta estrategia no hay interacción entre $\mathcal{TOY}$ y ECL^{i}PS^e, evitando así cualquier pérdida de tiempo por la comunicación entre ambos sistemas. En conclusión, aunque la comunicación entre distintos sistemas puede penalizar el tiempo de ejecución, se han estudiado estrategias alternativas para las que se ha verificado experimentalmente que no empeoran la eficiencia del programa.

A partir de estas conclusiones se deriva que las propuestas incluidas en esta tesis son factibles tanto formal como experimentalmente.

Respecto al trabajo futuro, la investigación posterior a la defensa de esta tesis se puede desarrollar en varios aspectos. Por una parte sería interesante integrar los dos dominios de coordinación en uno. Es decir, estudiar la cooperación entre los dominios $\mathcal{H}$, $\mathcal{FD}$, $\mathcal{R}$ y $\mathcal{FS}$ con los puentes que se han establecido en esta tesis y posiblemente añadiendo otros puentes nuevos. Por ejemplo, se podría pensar en un puente que relacionase los dominios $\mathcal{R}$ y $\mathcal{FS}$, pues actualmente no existe ninguna comunicación entre ellos. También se podrían ampliar los cálculos definidos en esta tesis o el correspondiente cálculo a la cooperación integrada de los dominios $\mathcal{H}$, $\mathcal{FD}$, $\mathcal{R}$ y $\mathcal{FS}$ con nuevas reglas. Por ejemplo, se pueden estudiar otras propiedades de la teoría de conjuntos y definir nuevas reglas que ayuden a anticipar el fallo. También se pueden definir nuevas proyecciones, y se puede desarrollar el resolutor $\mathcal{R}$ para dividirlo en dos capas de forma análoga a como se han dividido en esta tesis los resolutores $\mathcal{FD}$ y $\mathcal{FS}$. En definitiva, se puede ampliar el estudio de la cooperación entre los dominios de restricciones considerados en esta tesis o estudiar otros dominios de coordinación estableciendo nuevos mecanismos de comunicación o puentes y nuevas reglas para el cálculo.

Otro aspecto relevante consiste en extender el sistema $\mathcal{TOY}$ con otros resolutores de restricciones y estudiar la cooperación entre ellos. Por una parte se puede extender el sistema $\mathcal{TOY}$ con otras bibliotecas de los resolutores de los sistemas SICStus y ECL^{i}PS^e como, por ejemplo, la biblioteca del primero que resuelve restricciones sobre el dominio Boolean. Esta extensión es directa pues el mecanismo de comunicación ya está establecido.

Otra aportación, en línea con la tendencia actual en la comunidad científica de construcción de un lenguaje genérico de modelado de sistemas de restricciones, consiste en conectar $\mathcal{TOY}$ con el lenguaje de modelado `MiniZinc`, pues este lenguaje de modelado de restricciones tiene un buen número de resolutores disponibles, incluidos los resolutores de SICStus y ECL^{i}PS^e. De hecho, siempre ha estado en la mente del equipo de desarrollo que el sistema $\mathcal{TOY}$ pueda ejecutarse sobre varias plataformas Prolog y aunque conectar $\mathcal{TOY}$ a `MiniZinc` no solventa este deseo sí que da la oportunidad de que $\mathcal{TOY}$ utilice un amplio abanico de resolutores.

Una forma de conectar $\mathcal{TOY}$ con `MiniZinc` puede seguir la idea mostrada en esta tesis para trabajar con dos sistemas distintos y presentada como la estrategia $\text{ECL}^i\text{PS}^e_{gen}$. Con esta estrategia se genera un conjunto de restricciones a partir de un objetivo y un programa $\mathcal{TOY}$. Estas restricciones son el cuerpo de un programa nuevo que, como se ha visto en los capítulos anteriores, es un programa ECL^iPS^e. Este programa está definido en un fichero nuevo para su posterior evaluación en el sistema ECL^iPS^e. Esta forma de conectar $\mathcal{TOY}$ con `MiniZinc` serviría para programas tipo CP y no sería viable para otros tipos de problemas que puede resolver $\mathcal{TOY}$, como por ejemplo problemas que usan el indeterminismo. Por lo tanto, para cubrir todas las posibilidades que ofrece $\mathcal{TOY}$, también habría que conectar $\mathcal{TOY}$ con `MiniZinc` de forma interactiva, aunque de esta forma se pierde eficiencia por la comunicación entre procesos. Para evitar tener dos modos de ejecución, una mejora relevante consiste en estudiar un mecanismo de traducción más sofisticado que permita trasladar al resolutor de `MiniZinc` aspectos de $\mathcal{TOY}$ tales como el indeterminismo.

Por último, también sería deseable aprovechar los mecanismos de concurrencia de los resolutores con los que se extiende $\mathcal{TOY}$. Dichos resolutores reciben un gran número de primitivas y hacen un uso intensivo de la concurrencia para mejorar la eficiencia. De hecho, una primera aproximación de esta línea ha sido la utilización de demonios en la codificación de las primitivas añadidas a los dominios $\mathcal{FD}$ y $\mathcal{FS}$.

Apéndices

Apéndice A

Demostraciones

A.1 Demostración de los lemas 1 (pág. 49) y 3 (pág. 65)

El lema 3 es una generalización del lema 1 para tratar el dominio $\mathcal{H}$. Por lo tanto se va a demostrar únicamente el lema 3.

1. La relación de transición $\vdash_{\mathcal{D},\mathcal{X}}$ genera un árbol de raíz $\Pi \square \varepsilon$, cuyas hojas corresponden a los almacenes de $SF_{\mathcal{D}}(\Pi, \mathcal{X})$. Como $\vdash_{\mathcal{D},\mathcal{X}}$ es finitamente ramificado y terminante entonces este árbol es localmente finito y no tiene ramas infinitas. Por el lema de *König* [BN98] el árbol debe ser finito. Por lo tanto, debe tener una cantidad finita de hojas y $SF_{\mathcal{D}}(\Pi, \mathcal{X})$ es finito. La siguiente notación se utilizará más adelante

   $solve^{\mathcal{D}}(\Pi, \mathcal{X}) = \bigvee\{\exists\overline{Y'}(\Pi' \square \sigma') \mid \Pi \square \varepsilon \vdash^*_{\mathcal{D},\mathcal{X}} \ \Pi' \square \sigma' \text{ con } \Pi' \square \sigma' \text{ irreducible e } \overline{Y'} = var(\Pi' \square \sigma') \setminus var(\Pi)\}$

2. Supongamos que el sistema de transformación de almacenes posee la propiedad de variables locales nuevas y la propiedad de vinculaciones seguras. Por el punto anterior, para cada forma resuelta $\exists\overline{Y'}(\Pi' \square \sigma')$, con respecto a $\mathcal{X}$, calculada por la llamada al resolutor $solve^{\mathcal{D}}(\Pi, \mathcal{X})$ existe alguna secuencia de pasos

   $$\Pi \square \varepsilon = \Pi'_0 \square \mu'_0 \vdash_{\mathcal{D},\mathcal{X}} \Pi'_1 \square \mu'_1 \vdash_{\mathcal{D},\mathcal{X}} \ldots \vdash_{\mathcal{D},\mathcal{X}} \Pi'_n \square \mu'_n$$

   tal que $\Pi'_n \square \mu'_n = \Pi' \square \sigma'$ es irreducible, y se cumplen las siguientes condiciones para todo $1 \leq i \leq n$:

   - $\Pi'_i \square \mu'_i$ es un almacén con $\overline{Y'_i} = var(\Pi'_i \square \mu'_i) \setminus var(\Pi'_{i-1} \square \mu'_{i-1})$ variables locales nuevas
   - $\mu'_i = \mu'_{i-1}\mu_i$ para alguna sustitución μ_i tal que $vdom(\mu_i) \cup vran(\mu_i) \subseteq var(\Pi'_{i-1}) \cup \overline{Y'_i}$
   - $\mu_i(X)$ es una constante para todo $X \in \mathcal{X} \cap vdom(\mu_i)$, por la propiedad de vinculaciones seguras del sistema transformaciones de almacenes.

Entonces, $\overline{Y'} = \overline{Y'_1}, \ldots, \overline{Y'_n}$, aplicando inducción sobre n se puede probar $vdom(\sigma') \cup vran(\sigma') \subseteq var(\Pi) \cup \overline{Y'}$ y que $\sigma'(X)$ es una constante para todo $X \in \mathcal{X} \cap vdom(\sigma')$.

Por lo tanto, $solve^{\mathcal{D}}$ también satisface la propiedad de las variables locales nuevas y la propiedad de vinculaciones seguras.

3. Supongamos que el sistema de transformación de almacenes es localmente correcto. Para demostrar la corrección de $solve^{\mathcal{D}}$ es suficiente probar

   $\bigcup\{Sol_{\mathcal{D}}(\exists\overline{Y'}(\Pi' \sqcap \sigma')) \mid \Pi \sqcap \sigma \Vdash^{*}_{\mathcal{D},\mathcal{X}} \Pi' \sqcap \sigma',$ con $\Pi' \sqcap \sigma'$ irreducible e $\overline{Y'} = var(\Pi' \sqcap \sigma') \setminus var(\Pi \sqcap \sigma)\} \subseteq Sol_{\mathcal{D}}(\Pi \sqcap \sigma)$.

   Para demostrarlo, se asume que $\Pi \sqcap \sigma \Vdash^{n}_{\mathcal{D},\mathcal{X}} \Pi' \sqcap \sigma'$ es irreducible e $\overline{Y'} = var(\Pi' \sqcap \sigma') \setminus var(\Pi \sqcap \sigma)$ y se demuestra que $Sol_{\mathcal{D}}(\exists\overline{Y'}(\Pi' \sqcap \sigma')) \subseteq Sol_{\mathcal{D}}(\Pi \sqcap \sigma)$ por inducción sobre n:

   $\underline{n = 0}$: En este caso $\overline{Y'} = \emptyset$, $\Pi' \sqcap \sigma' = \Pi \sqcap \sigma$. La inclusión es trivial.

   $\underline{n > 0}$: Es este caso $\Pi \sqcap \sigma \Vdash_{\mathcal{D},\mathcal{X}} \Pi'_1 \sqcap \sigma'_1 \Vdash^{n-1}_{\mathcal{D},\mathcal{X}} \Pi' \sqcap \sigma'$ con $\Pi' \sqcap \sigma'$ irreducible para algún almacén $\Pi'_1 \sqcap \sigma'_1$.

   Sea $\overline{Y'_1} = var(\Pi'_1 \sqcap \sigma'_1) \setminus var(\Pi \sqcap \sigma)$ e $\overline{Y''} = var(\Pi' \sqcap \sigma') \setminus var(\Pi'_1 \sqcap \sigma'_1)$.

   Entonces $\overline{Y'} = \overline{Y'_1}, \overline{Y''}$ donde $\overline{Y'_1}, \overline{Y''}$ es $var(\Pi' \sqcap \sigma') \setminus var(\Pi \sqcap \sigma)$.

   Por hipótesis de inducción, asumimos $Sol_{\mathcal{D}}(\exists\overline{Y''}(\Pi' \sqcap \sigma')) \subseteq Sol_{\mathcal{D}}(\Pi'_1 \sqcap \sigma'_1)$.

   Entonces para cualquier $\eta \in Sol_{\mathcal{D}}(\exists\overline{Y'}(\Pi' \sqcap \sigma'))$ se puede demostrar que $\eta \in Sol_{\mathcal{D}}(\Pi \sqcap \sigma)$ por el siguiente razonamiento: por definición de $Sol_{\mathcal{D}}$, existe $\eta' \in Sol_{\mathcal{D}}(\Pi' \sqcap \sigma')$ tal que $\eta' =_{\setminus\overline{Y'}} \eta$ y por lo tanto $\eta' =_{var(\Pi \sqcap \sigma)} \eta$. Trivialmente, se deduce que $\eta' \in Sol_{\mathcal{D}}(\exists\overline{Y''}(\Pi' \sqcap \sigma'))$, lo que implica $\eta' \in Sol_{\mathcal{D}}(\Pi'_1 \sqcap \sigma'_1)$ por hipótesis de inducción. Trivialmente de nuevo, $\eta' \in Sol_{\mathcal{D}}(\exists\overline{Y'_1}(\Pi'_1 \sqcap \sigma'_1))$ lo que implica $\eta' \in Sol_{\mathcal{D}}(\Pi \sqcap \sigma)$ por la propiedad de corrección local del sistema de transformación de almacenes. Como $\eta' =_{var(\Pi \sqcap \sigma)} \eta$, se puede concluir que $\eta \in Sol_{\mathcal{D}}(\Pi \sqcap \sigma)$.

4. Sea $\mathcal{RS}$ un conjunto de reglas de transformación de almacenes tal que el sistema de transformación de almacenes es localmente completo para pasos $\mathcal{RS}$-libres. Por el primer apartado, para demostrar la completitud de $solve^{\mathcal{D}}$ para llamadas $\mathcal{RS}$-libres, es suficiente demostrar que

   $WTSol_{\mathcal{D}}(\Pi \sqcap \sigma) \subseteq \bigcup\{WTSol_{\mathcal{D}}(\exists\overline{Y'}(\Pi' \sqcap \sigma')) \mid \Pi \sqcap \sigma \Vdash^{*}_{\mathcal{D},\mathcal{X}} \Pi' \sqcap \sigma'$ irreducible, $\overline{Y'} = var(\Pi' \sqcap \sigma') \setminus var(\Pi \sqcap \sigma)\}$

   donde $\Pi \sqcap \sigma$ es hereditariamente $\mathcal{RS}$-irreducible. Esto se puede ver como una propiedad del almacén $\Pi \sqcap \sigma$ que puede ser demostrada por *inducción bien fundada* (ver de nuevo [BN98]) sobre la terminación de la relación $\Vdash_{\mathcal{D},\mathcal{X}}$:

   $\underline{\text{Caso base}}$: $\Pi \sqcap \sigma$ es irreducible con respecto a $\Vdash_{\mathcal{D},\mathcal{X}}$. En este caso, la unión se reduce al conjunto $WTSol_{\mathcal{D}}(\Pi \sqcap \sigma)$ y la inclusión es trivial.

   $\underline{\text{Caso de inducción}}$: $\Pi \sqcap \sigma$ es reducible con respecto a $\Vdash_{\mathcal{D},\mathcal{X}}$.

   En este caso, como $\Pi \sqcap \sigma$ es hereditariamente $\mathcal{RS}$-irreducible y el sistema de transformación de almacenes es localmente completo para pasos $\mathcal{RS}$-libres, para cualquier

$\eta \in WTSol_{\mathcal{D}}(\Pi \,\square\, \sigma)$ existe algún $(\Pi'_1 \,\square\, \sigma'_1)$ hereditariamente $\mathcal{RS}$-irreducible tal que $\Pi \,\square\, \sigma \vdash_{\mathcal{D},\mathcal{X}} \Pi'_1 \,\square\, \sigma'_1$ y $\eta \in WTSol_{\mathcal{D}}(\exists\overline{Y'_1}(\Pi'_1 \,\square\, \sigma'_1))$, donde $\overline{Y'_1} = var(\Pi'_1 \square \sigma'_1) \backslash var(\Pi \,\square\, \sigma)$.

Entonces, por la definición de $Sol_{\mathcal{D}}$, existe $\eta'_1 \in WTSol_{\mathcal{D}}(\Pi'_1 \,\square\, \sigma'_1)$ tal que $\eta'_1 =_{\backslash\overline{Y'_1}} \eta$.

La hipótesis de inducción puede aplicarse a $\Pi'_1 \,\square\, \sigma'_1$, y debe haber alguna $\Pi' \,\square\, \sigma'$ tal que $\Pi'_1 \,\square\, \sigma'_1 \vdash^*_{\mathcal{D},\mathcal{X}} \Pi' \,\square\, \sigma'$ irreducible, $\overline{Y''} = var(\Pi' \,\square\, \sigma') \setminus var(\Pi'_1 \square \sigma'_1)$ y $\eta'_1 \in WTSol_{\mathcal{D}}(\exists\overline{Y''}(\Pi' \,\square\, \sigma'))$.

Por definición de $Sol_{\mathcal{D}}$, existe $\eta' \in WTSol_{\mathcal{D}}(\Pi' \,\square\, \sigma')$ tal que $\eta' =_{\backslash\overline{Y''}} \eta'_1$.

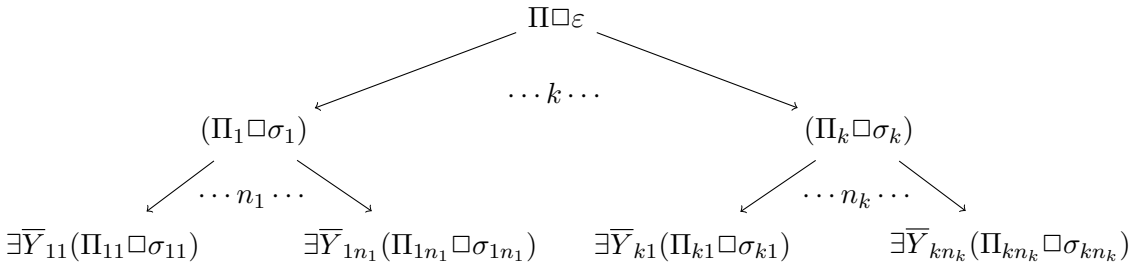
Además, $\Pi \,\square\, \sigma \vdash^*_{\mathcal{D},\mathcal{X}} \Pi' \,\square\, \sigma'$ irreducible e $\overline{Y'} = \overline{Y'_1}, \overline{Y''} = var(\Pi' \,\square\, \sigma') \setminus var(\Pi \,\square\, \sigma)$ tal que $\eta' =_{\backslash\overline{Y'}} \eta$, y por lo tanto $\eta \in WTSol_{\mathcal{D}}(\exists\overline{Y'}(\Pi' \,\square\, \sigma'))$.

c.q.d.

A.2 Demostración del lema 2 (pág. 50)

Este lema nos dice que dos resolutores $solve1^{\mathcal{D}}$ y $solve2^{\mathcal{D}}$ un mismo dominio $\mathcal{D}$ distinto de $\mathcal{H}$, que cumplen las propiedades de la definición 4, si se aplican de forma secuencial $(solve1^{\mathcal{D}} \diamond solve2^{\mathcal{D}})$ donde $solve1^{\mathcal{D}}$ no produce variables nuevas, entonces se forma un nuevo resolutor $solve^{\mathcal{D}}$. Para demostrarlo veamos que $solve^{\mathcal{D}}$ cumple las condiciones dadas en la definición 4.

Una invocación al resolutor $solve1^{\mathcal{D}}(\Pi)$ devuelve una disyunción finita de almacenes de restricciones $\bigvee_{j=1}^{k}(\Pi_j \,\square\, \sigma_j)$, como no produce variables nuevas entonces no está existencialmente cuantificado. A cada uno de estos almacenes se le aplica una invocación al resolutor $solve2^{\mathcal{D}}(\Pi_j \,\square\, \sigma_j)$ que devuelve una disyunción finita de almacenes de restricciones existencialmente cuantificados $\bigvee_{l=1}^{n_j} \exists\overline{Y}_{jl}(\Pi_{jl} \,\square\, \sigma_{jl})$, como se muestra en la siguiente figura:



Veamos que $solve^{\mathcal{D}}$ cumple las condiciones dadas en la definición 4:

1. **Variables locales nuevas:** hay que demostrar que para todo $1 \leq j \leq k$ y $1 \leq l \leq n_j$: $(\Pi^j{}_l \,\square\, \sigma^j{}_l)$ es un almacén de restricciones tal que $\overline{Y}_l = var(\Pi^j{}_l \,\square\, \sigma^j{}_l) \setminus var(\Pi)$ son variables locales nuevas y $vdom(\sigma^j{}_l) \cup vran(\sigma^j{}_l) \subseteq var(\Pi) \cup \overline{Y}_l$.

   La invocación al resolutor $solve1^{\mathcal{D}}(\Pi)$ no produce variables nuevas, por lo tanto se tiene $var(\Pi_j \,\square\, \sigma_j) = var(\Pi)$.

Por la propiedad de variables nuevas del resolutor $solve2^{\mathcal{D}}(\Pi_j \square \sigma_j)$ se sabe que $\overline{Y}_l = var(\Pi^j{}_l \square \sigma^j{}_l) \setminus var(\Pi_j \square \sigma_j)$. Por lo tanto se cumple $\overline{Y}_l = var(\Pi^j{}_l \square \sigma^j{}_l) \setminus var(\Pi)$.

Veamos la demostración de la segunda condición.

Por la invocación al resolutor $solve1^{\mathcal{D}}(\Pi)$ se tiene que $vdom(\sigma_j) \cup vran(\sigma_j) \subseteq var(\Pi)$ y por la invocación al resolutor $solve2^{\mathcal{D}}(\Pi_j \square \sigma_j)$ se tiene $vdom(\sigma^j{}_l) \cup vran(\sigma^j{}_l) \subseteq var(\Pi_j \square \sigma_j) \cup \overline{Y}_l$. Como $solve1^{\mathcal{D}}(\Pi)$ no produce variables nuevas entonces $var(\Pi_j \square \sigma_j) = var(\Pi)$, obteniendo $vdom(\sigma^j{}_l) \cup vran(\sigma^j{}_l) \subseteq var(\Pi) \cup \overline{Y}_l$

2. **Formas resueltas:** hay que probar que para todo $1 \leq j \leq k$ y $1 \leq l \leq n_j$: $\Pi^j{}_l \square \sigma^j{}_l$ está en forma resuelta. Por definición, esto significa que $solve^{\mathcal{D}}(\Pi^j{}_l \square \sigma^j{}_l) = \Pi^j{}_l \square \sigma^j{}_l$.

   Como $solve1^{\mathcal{D}}(\Pi_j \square \sigma_j)$ cumple la propiedad de forma resueltas de los resolutores entonces se tiene se tiene que $solve1^{\mathcal{D}}(\Pi_j \square \sigma_j) = \Pi_j \square \sigma_j$. Igualmente $solve2^{\mathcal{D}}(\Pi_j \square \sigma_j)$ cumple la propiedad de forma resueltas de los resolutores entonces $solve2^{\mathcal{D}}(\Pi^j{}_l \square \sigma^j{}_l) = \Pi^j{}_l \square \sigma^j{}_l$. El resolutor $solve^{\mathcal{D}}$ es la secuenciación de los resolutores $solve1^{\mathcal{D}}$ y $solve2^{\mathcal{D}}$. por lo tanto la aplicación de $solve1^{\mathcal{D}}$ deja en formas resueltas el conjunto de restricciones. El resolutor $solve2^{\mathcal{D}}$ toma esas restricciones y aplica su propia resolución dejándolas en forma resuelta, de esta forma $solve^{\mathcal{D}}$ deja en forma resuelta el conjunto de restricciones inicial.

3. **Corrección:** hay que probar $Sol_{\mathcal{D}}(\Pi) \supseteq \bigcup_{j=1}^{k} \bigcup_{l=1}^{n_j} Sol_{\mathcal{D}}(\exists \overline{Y}_l(\Pi^j{}_l \square \sigma^j{}_l))$.

   Esta condición se cumple porque por la corrección de $solve1^{\mathcal{D}}(\Pi)$ se tiene que $Sol_{\mathcal{D}}(\Pi) \supseteq \bigcup_{j=1}^{k} Sol_{\mathcal{D}}(\Pi_j \square \sigma_j)$ y por la corrección de $solve2^{\mathcal{D}}(\Pi_j \square \sigma_j)$ se tiene que para todo $1 \leq j \leq k$ se cumple $Sol_{\mathcal{D}}(\Pi_j \square \sigma_j) \supseteq \bigcup_{l=1}^{n_j} Sol_{\mathcal{D}}(\exists \overline{Y}_l(\Pi^j{}_l \square \sigma^j{}_l))$.

4. **Completitud:** hay que demostrar $WTSol_{\mathcal{D}}(\Pi) \subseteq \bigcup_{j=1}^{k} WTSol_{\mathcal{D}}(\exists \overline{Y}_j(\Pi_j \square \sigma_j))$. La demostración es similar a la demostración de corrección.

A.3 Demostración del teorema 1 (pág. 55)

Primero se demuestra que el sistema de transformación de almacenes con relación de transición $\vdash_{\mathcal{FD}^{\mathcal{T}}}$ definido mediante la tabla 3.1 cumple las propiedades de la definición 6.

Variables locales nuevas: trivial, pues las dos reglas derivan en fallo y no existen variables locales nuevas.

Ramificación finita y terminación: triviales, pues las dos reglas derivan en fallo.

Corrección local: trivial, pues todas las reglas de transformación de almacenes de la tabla 3.1 producen fallo. Por lo tanto, $Sol_{\mathcal{FD}^{\mathcal{T}}}(\blacksquare) = \emptyset$ y el conjunto vacío es un subconjunto de $Sol_{\mathcal{FD}^{\mathcal{T}}}(\Pi \square \sigma)$.

Completitud local: supongamos cualquier almacén $\Pi \square \sigma$ del dominio $\mathcal{FD}^{\mathcal{T}}$ y, como todas las reglas de la tabla 3.1 producen fallo, entonces se reduce a probar que el conjunto $WTSol_{\mathcal{FD}^{\mathcal{T}}}(\Pi \square \sigma)$ es un subconjunto del conjunto vacío.

Si se puede aplicar la regla **FD1**, entonces {F_1 `#<=` F_2, F_2 `#<=` F_1, F_1 `/=` F_2} $\subset \Pi$. Por otra parte {F_1 `#<=` F_2, F_2 `#<=` F_1, F_1 `/=` F_2} es inconsistente, lo que implica $WTSol_{\mathcal{FD}^{\mathcal{T}}}(\Pi \,\square\, \sigma) = \emptyset$. Del mismo modo, si se puede aplicar la regla **FD2**, entonces $WTSol_{\mathcal{FD}^{\mathcal{T}}}(F_1$ `==` F_2, F_1 `/=` $F_2, \Pi \,\square\, \sigma) = \emptyset$.

Aplicando el lema 1 a las propiedades que se acaban de demostrar se concluye que $solve^{\mathcal{FD}^{\mathcal{T}}}$ satisface todos los requerimientos de los resolutores de la definición 4.

c.q.d.

A.4 Demostración del teorema 2 (pág. 59)

Primero se demuestra que el sistema de transformación de almacenes con relación de transición $\vdash_{\mathcal{FS}^{\mathcal{T}}}$ definido mediante la tabla 3.2 cumple las propiedades de la definición 6.

1) **Variables locales nuevas:** ninguna regla produce variables locales nuevas.

2) **Ramificación finita:** trivial, pues Π es un conjunto finito de restricciones, hay un conjunto finito de reglas para aplicar y no hay elecciones no deterministas.

3) **Terminación:** considérese la posibilidad de un almacén formado por $\Pi\square\sigma$ donde Π es un conjunto finito de restricciones y σ es una sustitución idempotente. Debemos probar que se obtiene un almacén irreducible después de una secuencia finita de pasos $\Pi\square\sigma \vdash^*_{\mathcal{FS}^{\mathcal{T}}} \Pi_n\square\sigma_n$, es decir, debemos probar que cada regla solo se puede aplicar un número finito de veces.

   Sea una secuencia finita de i pasos $\Pi\square\sigma \vdash^*_{\mathcal{FS}^{\mathcal{T}}} \Pi_i\square\sigma_i$. Si $\Pi_i\square\sigma_i$ es reducible (un almacén es reducible si hay una regla de transformación de almacenes que pueda ser aplicada para transformarlo), entonces ocurre uno de los siguientes casos:

   - **S1** puede ser aplicada si Π_i contiene las restricciones `domainSets` $[S_1, \ldots, S_n]$ `lb ub`, y `lb /= {}`. En este caso, se aplica **S1** y se añaden nuevas restricciones S_1 `/= {}`,..., S_n `/= {}` al almacén Π_{i+1}.

     Obsérvese que una regla no se aplica si el almacén no se transforma en modo alguno. Por lo tanto, **S1** no se puede aplicar de nuevo en las mismas restricciones `domainSets` $[S_1, \ldots, S_n]$ `lb ub`, y `lb /= {}` en cualquier almacén Π_j con $j > i$.

     Por otra parte, Π es un conjunto finito y las reglas comprendidas entre **S1** y **S10** no generan nuevas restricciones `domainSets`, entonces la regla **S1** solo puede ser aplicada un número finito de veces.

     Se puede aplicar un razonamiento similar a las reglas **S2**, **S3**, **S6**, **S7**, **S9** y **S10**.

   - **S4** puede ser aplicada si Π_i contiene las restricciones `subset` S_1 S_2 y S_2 `== {}`. En este caso la restricción `subset` es eliminada y las variables se restringen a ser conjuntos vacíos. Como Π_i es finito, entonces **S4** solo puede ser aplicada un número finito de veces sobre Π_i.

Por otra parte, las reglas **S6**, **S7**, y **S9** generan nuevas restricciones `subset` pero, como se ha visto anteriormente, estas reglas puede ser aplicadas un número finito de veces, generando un número finito de nuevas restricciones `subset`. Por lo tanto, el número total de restricciones `subset` sobre las que **S4** se puede aplicar es finito.

Se puede aplicar un razonamiento similar a la regla **S5**.

- **S8** puede ser aplicada si Π_i contiene las restricciones `union` S_1 S_2 S_3 y S_3 `==` `{}`. Esta regla restringe todas las variables a ser conjuntos vacíos y elimina la restricción `union`. Por lo tanto, esta regla puede ser aplicada un número finito de veces, pues Π_i es finito.
- **S11** puede ser aplicada y el cómputo termina con fallo.

Por lo tanto solo se puede aplicar un número finito de pasos demostrando así que el cómputo es terminante.

4) **Corrección local:** para cualquier almacén $\Pi \,\square\, \sigma$ del dominio $\mathcal{FS}^{\mathcal{T}}$, el conjunto
$\bigcup\{Sol_{\mathcal{FS}^{\mathcal{T}}}(\exists\overline{Y'}(\Pi' \,\square\, \sigma')) \mid \Pi \,\square\, \sigma \Vdash_{\mathcal{FS}^{\mathcal{T}}} \Pi' \,\square\, \sigma', \overline{Y'} = var(\Pi' \,\square\, \sigma') \setminus var(\Pi \,\square\, \sigma)\}$
es un subconjunto de $Sol_{\mathcal{FS}^{\mathcal{T}}}(\Pi \,\square\, \sigma)$.

Es decir, hay que demostrar que para todo almacén $\Pi' \,\square\, \sigma'$ tal que $\Pi \,\square\, \sigma \Vdash_{\mathcal{FS}^{\mathcal{T}}} \Pi' \,\square\, \sigma'$
$Sol_{\mathcal{FS}^{\mathcal{T}}}(\Pi \,\square\, \sigma) \supseteq Sol_{\mathcal{FS}^{\mathcal{T}}}(\exists\overline{Y'}(\Pi' \,\square\, \sigma'))$ con $\overline{Y'} = var(\Pi' \,\square\, \sigma') \setminus var(\Pi \,\square\, \sigma)$.

Esto se demuestra por una distinción de casos de las reglas definidas en la tabla 3.2. En cada caso asumimos que los almacenes tienen exactamente la forma mostrada por la correspondiente regla de transformación en la tabla 3.2.

- **S1**: sea $\eta \in Sol_{\mathcal{FS}^{\mathcal{T}}}($`domainSets` $[S_1, \ldots, S_n]$ `lb ub`, `lb /= {}`, S_1 `/= {}`,...,S_n `/= {}`, $\Pi)$; trivialmente, η está también en $Sol_{\mathcal{FS}^{\mathcal{T}}}($`domainSets` $[S_1, \ldots, S_n]$ `lb ub`, `lb /= {}`, $\Pi)$.

  Se puede aplicar un razonamiento similar a las reglas **S2**, **S3**, **S6**, **S7**, **S9** y **S10**.
- **S4**: sea $\eta \in Sol_{\mathcal{FS}^{\mathcal{T}}}(S_1$ `==` `{}`, S_2 `==` `{}`, $\Pi)$; entonces S_1 y S_2 son conjuntos vacíos en η, y por lo tanto la restricción `subset` S_1 S_2 se satisface y $\eta \in Sol_{\mathcal{FS}^{\mathcal{T}}}($`subset` S_1 S_2, S_2 `==` `{}`, $\Pi)$.

  Se puede aplicar un razonamiento similar a la regla **S8**.
- **S5**: sea $\eta \in Sol_{\mathcal{FS}^{\mathcal{T}}}(S_1$ `==` $S_2, \Pi)$; entonces en η, S_1 y S_2 tienen los mismos elementos, trivialmente la misma η pertenece a $Sol_{\mathcal{FS}^{\mathcal{T}}}($`subset` S_1 S_2,`subset` S_2 S_1,$\Pi)$.
- **S11**: trivial, porque $Sol_{\mathcal{FS}^{\mathcal{T}}}(\blacksquare) = \emptyset$ es subconjunto de las soluciones de cualquier almacén.

Con esta distinción de casos se ha probado que toda solución de $\Pi' \square \sigma'$ es solución de $\Pi \square \sigma$ y de aquí se deduce $Sol_{\mathcal{FS}^{\mathcal{T}}}(\Pi \,\square\, \sigma) \supseteq Sol_{\mathcal{FS}^{\mathcal{T}}}(\exists\overline{Y'}(\Pi' \,\square\, \sigma'))$.

5) **Completitud local:** hay que probar que, para cualquier almacén $\Pi \,\square\, \sigma$ del dominio $\mathcal{FS}^{\mathcal{T}}$, el conjunto $WTSol_{\mathcal{FS}^{\mathcal{T}}}(\Pi \,\square\, \sigma)$ es un subconjunto de

$\bigcup\{WTSol_{\mathcal{FS}^\mathcal{T}}(\exists\overline{Y'}(\Pi' \square \sigma')) \mid \Pi\square\sigma \Vdash_{\mathcal{FS}^\mathcal{T}} \Pi'\square\sigma', \overline{Y'} = var(\Pi'\square\sigma') \setminus var(\Pi\square\sigma)\}$

Es decir, hay que demostrar que para todo almacén $\Pi' \square \sigma'$ tal que $\Pi \square \sigma \Vdash_{\mathcal{FS}^\mathcal{T}} \Pi' \square \sigma'$

$Sol_{\mathcal{FS}^\mathcal{T}}(\Pi \square \sigma) \subseteq Sol_{\mathcal{FS}^\mathcal{T}}(\exists\overline{Y'}(\Pi' \square \sigma'))$ con $\overline{Y'} = var(\Pi' \square \sigma') \setminus var(\Pi \square \sigma)$.

Esto se demuestra por distinción de casos de acuerdo a las reglas de transformación de almacenes definidas en la tabla 3.2. En cada caso, asumimos que los almacenes tienen exactamente la forma mostrada por la correspondiente transformación en la tabla 3.2.

- **S1**: sea $\eta \in WTSol_{\mathcal{FS}^\mathcal{T}}(\texttt{domainSets}\ [S_1, \ldots, S_n]\ \texttt{lb ub}, \texttt{lb /= \{\}}, \Pi)$.

  Como `domainSets` restringe el dominio de cada variable de conjunto de la lista con respecto a un ínfimo `lb` y un supremo `ub` formando un retículo, y `lb /= {}`, entonces en η cada variable de conjunto $S_1, \ldots, S_n$ se vincula a un conjunto que al menos contiene los valores de `lb`. Por lo tanto, $S_1, \ldots, S_n$ no están vinculadas a conjuntos vacíos en η, y $\eta \in WTSol_{\mathcal{FS}^\mathcal{T}}(\texttt{domainSets}\ [S_1, \ldots, S_n]\ \texttt{lb ub}, \texttt{lb /= \{\}}, S_1\ \texttt{/= \{\}}, \ldots, S_n\ \texttt{/= \{\}}, \Pi)$.

  Se puede aplicar un razonamiento similar a las reglas **S2**, **S3** y **S10**.

- **S4**: sea $\eta \in Sol_{\mathcal{FS}^\mathcal{T}}(\texttt{subset}\ S_1\ S_2, S_2\ \texttt{== \{\}}, \Pi)$. Por la teoría de conjuntos se sabe que el único subconjunto del conjunto vacío es el conjunto vacío. Por lo tanto, $\eta \in Sol_{\mathcal{FS}^\mathcal{T}}(S_1\ \texttt{== \{\}}, S_2\ \texttt{== \{\}}, \Pi)$.

  Se puede aplicar un razonamiento similar a la regla **S8**.

- **S5**: sea $\eta \in Sol_{\mathcal{FS}^\mathcal{T}}(\texttt{subset}\ S_1\ S_2, \texttt{subset}\ S_2\ S_1, \Pi)$. Por la teoría de conjuntos, sabemos que $S_1 \subseteq S_2$ y $S_2 \subseteq S_1 \iff S_1 = S_2$. Por lo tanto, $\eta \in Sol_{\mathcal{FS}^\mathcal{T}}(S_1\ \texttt{==}\ S_2, \Pi)$.

  Se puede aplicar un razonamiento similar a las reglas **S6**, **S7**, y **S9**.

- **S11**: trivial, porque $Sol_{\mathcal{FS}^\mathcal{T}}(S_1\ \texttt{==}\ S_2, S_1\ \texttt{/=}\ S_2, \Pi) = \emptyset$.

Con esta distinción de casos se ha probado que toda solución de $\Pi\square\sigma$ es solución de $\Pi'\square\sigma'$ y de aquí se deduce $Sol_{\mathcal{FS}^\mathcal{T}}(\Pi \square \sigma) \subseteq Sol_{\mathcal{FS}^\mathcal{T}}(\exists\overline{Y'}(\Pi' \square \sigma'))$.

Aplicando el lema 1 a las propiedades que se acaban de demostrar se concluye que $solve^{\mathcal{FS}^\mathcal{T}}$ satisface todos los requerimientos de los resolutores de la definición 4.

c.q.d.

A.5 Demostración del teorema 4 (pág. 71)

Para la demostración de este teorema son necesarios los siguientes lemas.

El primer lema asegura que el tipo de cualquier expresión es también válido para todas sus aproximaciones semánticas. Este lema es similar al lema que se presenta en el trabajo [GHR01] como *Typing Monotonicity Lemma* y su demostración se omite aquí.

Lema 6. (Lema de preservación de tipos)
Sea Σ, $\Gamma \vdash_{WT} e' :: \tau$ y $e \sqsubseteq e'$. Entonces se cumple Σ, $\Gamma \vdash_{WT} e :: \tau$.

Lema 7. (Lema de monotonicidad) Seas $\Pi \subseteq PCon_{\mathcal{C}}$ y $\eta, \eta' \in Val_{\mathcal{C}}$ tal que $\eta \sqsubseteq \eta'$ y $\eta \in (WT)Sol_{\mathcal{C}}(\Pi)$, entonces $\eta' \in (WT)Sol_{\mathcal{C}}(\Pi)$.

Para demostrar el teorema 4 tomamos el dominio $\mathcal{S}$ de signatura Σ construido como la suma amalgamada $\mathcal{S} = \mathcal{D}_1 \oplus \cdots \oplus \mathcal{D}_n$ de n dominios unibles $\mathcal{D}_i$ de signaturas Σ_i, $1 \leq i \leq n$. Se deben probar los siguientes cuatro puntos:

1. $\mathcal{S}$ está bien definido como un dominio de restricciones, es decir, las interpretaciones de los símbolos de las funciones primitivas en $\mathcal{S}$ satisfacen las condiciones de la definición 1. Veamos estas condiciones una por una asumiendo que p no es la primitiva == excepto para la cuarta condición.

   (a) **Polaridad**: Sea $p \in SPF^m$ y $\overline{t}_m, \overline{t'}_m, t, t' \in \mathcal{U}_{\mathcal{S}}$ tal que $p^{\mathcal{S}}\,\overline{t}_m \to t$, $\overline{t}_m \sqsubseteq \overline{t'}_m$ y $t \sqsupseteq t'$. Si t es $\bot$, entonces $p^{\mathcal{S}}\,\overline{t'}_n \to t'$ porque t' debe ser $\bot$. En otro caso, por la primera asunción y por la definición de $p^{\mathcal{S}}$, debe haber algún $1 \leq i \leq n$ y algún $\overline{t''}_m, t'' \in \mathcal{U}_{\mathcal{D}_i}$ tal que $\overline{t''}_m \sqsubseteq \overline{t}_m$, $t'' \sqsupseteq t$ y $p^{\mathcal{D}_i}\,\overline{t''}_m \to t''$. Como $\overline{t''}_m \sqsubseteq \overline{t}_m \sqsubseteq \overline{t'}_m$ y $t'' \sqsupseteq t \sqsupseteq t'$, $p^{\mathcal{D}_i}\,\overline{t''}_m \to t''$ implica $p^{\mathcal{S}}\,\overline{t'}_m \to t'$ por definición de $p^{\mathcal{S}}$.

   (b) **Radicalidad**: Sea $p \in SPF^m$ y $\overline{t}_m, t \in \mathcal{U}_{\mathcal{S}}$ tal que $p^{\mathcal{S}}\,\overline{t}_m \to t$ y t distinto de $\bot$. Por la definición de $p^{\mathcal{S}}$ debe haber algún $1 \leq i \leq n$ y algún $\overline{t''}_m$, $t'' \in \mathcal{U}_{\mathcal{D}_i}$ tal que $\overline{t''}_m \sqsubseteq \overline{t}_m$, $t'' \sqsupseteq t$ y $p^{\mathcal{D}_i}\,\overline{t''}_m \to t''$. Por la condición de radicalidad establecida para $\mathcal{D}_i$, debe haber algún $t' \in \mathcal{U}_{\mathcal{D}_i}$ total tal que $p^{\mathcal{D}_i}\,\overline{t''}_m \to t'$ con $t' \sqsupseteq t''$. Nótese que $t' \sqsupseteq t'' \sqsupseteq t$, y como $\overline{t''}_m \sqsubseteq \overline{t}_m$ y $t' \sqsupseteq t'$, $p^{\mathcal{D}_i}\,\overline{t''}_m \to t'$ implica $p^{\mathcal{S}}\,\overline{t}_m \to t'$ por definición de $p^{\mathcal{S}}$.

   (c) **Buen tipado**: sea $p \in SPF^m$, una instancia monomórfica de p con tipo principal $\overline{\tau'}_m \to \tau'$ y sean $\overline{t}_m, t \in \mathcal{U}_{\mathcal{S}}$ tal que $\Sigma \vdash_{WT} \overline{t}_m :: \overline{\tau'}_m$ y $p^{\mathcal{S}}\,\overline{t}_m \to t$. Si t es $\bot$, entonces se cumple $\Sigma \vdash_{WT} \bot :: \tau'$. En otro caso, por la definición de $p^{\mathcal{S}}$ existe $1 \leq i \leq n$ y $\overline{t'}_m, t' \in \mathcal{U}_{\mathcal{D}_i}$ tal que $\overline{t'}_m \sqsubseteq \overline{t}_m$, $t' \sqsupseteq t$ y $p^{\mathcal{D}_i}\,\overline{t'}_m \to t'$. Además, como $\overline{t'}_m \sqsubseteq \overline{t}_m$ y $\Sigma \vdash_{WT} \overline{t}_m :: \overline{\tau'}_m$ por el lema de lema de preservación de tipos 6 se deduce que $\Sigma \vdash_{WT} \overline{t'}_m :: \overline{\tau'}_m$. Entonces, como $\mathcal{D}_i$ es un dominio donde sus primitivas tienen la propiedad del buen tipado entonces se garantiza $\Sigma \vdash_{WT} t' :: \tau'$, que implica $\Sigma \vdash_{WT} t :: \tau'$ por $t \sqsubseteq t'$ y el lema 6.

   (d) **Igualdad estricta**: La primitiva == (en caso de que pertenezca a SPF) es interpretado como la *igualdad estricta* sobre $\mathcal{U}_{\mathcal{S}}$. Pero esto está garantizado por la construcción de la suma amalgamada.

2. Sean i tal que $1 \leq i \leq n$, un símbolo de función primitiva $p \in SPF_i^m$ y valores $\overline{t}_m, t \in \mathcal{U}_{\mathcal{D}_i}$, se debe probar: $p^{\mathcal{D}_i}\,\overline{t}_m \to t$ iff $p^{\mathcal{S}}\,\overline{t}_m \to t$. Por definición de $p^{\mathcal{S}}$, se sabe que se cumple $p^{\mathcal{S}}\,\overline{t}_m \to t$ si y solamente si existe algún $\overline{t'}_m, t' \in \mathcal{U}_{\mathcal{D}_i}$ tal que $\overline{t'}_m \sqsubseteq \overline{t}_m$, $t' \sqsupseteq t$

y $p^{\mathcal{D}_i}\,\overline{t'}_m \to t'$. Pero esta condición es equivalente a $p^{\mathcal{D}_i}\bar{t}_m \to t$ porque $p^{\mathcal{D}_i}$ satisface la propiedad de polaridad.

3. Sean i tal que $1 \leq i \leq n$, un conjunto de restricciones primitivas $\Pi \subseteq APCon_{\mathcal{D}_i}$ y una valoración $\eta \in Val_{\mathcal{D}_i}$, se debe probar: $\eta \in Sol_{\mathcal{D}_i}(\Pi) \Leftrightarrow \eta \in Sol_{\mathcal{S}}(\Pi)$. Para hacer esta demostración se va a estudiar la siguiente equivalencia:

$$\eta \in Sol_{\mathcal{D}_i}(\Pi) \Leftrightarrow \forall \pi \in \Pi :\ \eta \in Sol_{\mathcal{D}_i}(\pi) \Leftrightarrow \forall \pi \in \Pi :\ \eta \in Sol_{\mathcal{S}}(\pi) \Leftrightarrow \eta \in Sol_{\mathcal{S}}(\Pi)$$

Por lo tanto para demostrar $\eta \in Sol_{\mathcal{D}_i}(\Pi) \Leftrightarrow \eta \in Sol_{\mathcal{S}}(\Pi)$ basta con demostrar la equivalencia: $(\star)\, \eta \in Sol_{\mathcal{D}_i}(\pi) \Leftrightarrow \eta \in Sol_{\mathcal{S}}(\pi)$ para un $\pi \in \Pi$ fijo.

Nótese que π debe tener la forma $p\,\bar{t}_m \to !\,t$ para algún $p \in SPF_i^m$, $\bar{t}_m \in Pat_{\mathcal{D}_i}$ y $t \in Pat_{\mathcal{D}_i}$ total. Si p es ==, $(\star)$ se cumple pues $t_1\eta ==^{\mathcal{D}_i} t_2\eta \to !t\eta$ y $t_1\eta ==^{\mathcal{S}} t_2\eta \to !t\eta$ se cumplen bajo las mismas condiciones, por la definición 1. Si p es distinto de ==, entonces sea $\overline{t'}_m = \bar{t}_m\eta$ y $t' = t\eta$. Si t' no es un patrón total, entonces ni $\eta \in Sol_{\mathcal{D}_i}(\pi)$ ni $\eta \in Sol_{\mathcal{S}}(\pi)$ se cumplen. En otro caso,

$$\eta \in Sol_{\mathcal{D}_i}(\pi) \Leftrightarrow p^{\mathcal{D}_i}\overline{t'}_m \to t' \Leftrightarrow_{(\star\star)} p^{\mathcal{S}}\overline{t'}_m \to t' \Leftrightarrow \eta \in Sol_{\mathcal{S}}(\pi)$$

donde $(\star\star)$ se cumple por el segundo punto de este teorema, ya que $\overline{t'}_m, t' \in \mathcal{U}_{\mathcal{D}_i}$.

4. Sean i tal que $1 \leq i \leq n$, un conjunto de restricciones $\mathcal{D}_i$-específicas $\Pi \subseteq APCon_{\mathcal{D}_i}$ y una valoración $\eta \in Val_{\mathcal{S}}$, se debe probar: $\eta \in Sol_{\mathcal{S}}(\Pi) \Leftrightarrow \mid \eta \mid_{\mathcal{D}_i} \in Sol_{\mathcal{D}_i}(\Pi)$.

Primero se demuestra $\eta \in Sol_{\mathcal{S}}(\Pi) \Leftarrow \mid \eta \mid_{\mathcal{D}_i} \in Sol_{\mathcal{D}_i}(\Pi)$.

Sea $\mid \eta \mid_{\mathcal{D}_i} \in Sol_{\mathcal{D}_i}(\Pi)$, aplicando el punto anterior de este teorema, se obtiene $\mid \eta \mid_{\mathcal{D}_i} \in Sol_{\mathcal{S}}(\Pi)$. Como $\mid \eta \mid_{\mathcal{D}_i} \sqsubseteq \eta$, se puede aplicar el lema de monotonicidad 7 y se obtiene $\eta \in Sol_{\mathcal{S}}(\Pi)$ como se desea.

Ahora se debe probar $\eta \in Sol_{\mathcal{S}}(\Pi) \Rightarrow \mid \eta \mid_{\mathcal{D}_i} \in Sol_{\mathcal{D}_i}(\Pi)$.

Sea $\eta \in Sol_{\mathcal{S}}(\Pi)$, como Π es $\mathcal{D}_i$-específica se tiene que $\eta(X) \in \mathcal{U}_{\mathcal{D}_i}$ para todo $X \in var(\Pi)$. Entonces $\eta(X) = \mid \eta \mid_{\mathcal{D}_i} (X)$ para todo $X \in var(\Pi)$ y por lo tanto $\mid \eta \mid_{\mathcal{D}_i} \in Sol_{\mathcal{S}}(\Pi)$, que implica $\mid \eta \mid_{\mathcal{D}_i} \in Sol_{\mathcal{D}_i}(\Pi)$, por el punto anterior.

c.q.d.

A.6 Demostración del teorema 5 (pag. 85)

Para la demostración del teorema 5 son necesarios los siguientes lemas auxiliares, que están demostrados en el trabajo [EFH+09].

Lema 8. (Lema auxiliar de corrección)

Dado un dominio $\mathcal{D}$, $\Pi \subseteq PCon_{\mathcal{D}}$ y $\sigma, \sigma_1 \in Sub_{\mathcal{D}}$ tal que σ es idempotente y $\Pi\sigma = \Pi$. Entonces $Sol_{\mathcal{D}}(\Pi\sigma_1) \cap Sol_{\mathcal{D}}(\sigma\sigma_1) \subseteq Sol_{\mathcal{D}}(\Pi) \cap Sol_{\mathcal{D}}(\sigma)$.

Lema 9. (Lema auxiliar de completitud)
Sea $\Pi \subseteq PCon_{\mathcal{D}}, \sigma, \sigma_1 \in Sub_{\mathcal{D}}$ y $\eta, \eta' \in Val_{\mathcal{D}}$ tal que $\eta \in Sol_{\mathcal{D}}(\Pi) \cap Sol_{\mathcal{D}}(\sigma)$, $\sigma_1\eta' = \eta'$ y $\eta' =_{\setminus \overline{Y'}} \eta$, donde $\overline{Y'}$ son variables nuevas distintas de $var(\Pi) \cup vdom(\sigma) \cup vran(\sigma)$. Entonces $\sigma\eta' = \eta'$ y $\eta' \in Sol_{\mathcal{D}}(\Pi\sigma_1) \cap Sol_{\mathcal{D}}(\sigma\sigma_1)$.

Primero se demuestra que el sistema de transformación de almacenes con relación de transición $\vdash_{\mathcal{M}_{\mathcal{FD}.\mathcal{R}}}$ definido mediante las reglas de la tabla 5.1 cumple las propiedades de la definición 6.

1. **Variables locales nuevas:** trivial pues no se producen variables nuevas en ninguna de las reglas.

2. **Ramificación finita:** trivial, pues Π es un conjunto finito de restricciones, hay un conjunto finito de reglas para aplicar y no hay elecciones no deterministas.

3. **Terminación:** considérese la posibilidad de un almacén formado por $\Pi \Box \sigma$, donde Π es un conjunto finito de restricciones y σ es una sustitución idempotente. Debemos probar que después de una secuencia finita de pasos $\Pi \Box \sigma \vdash^*_{\mathcal{M}} \Pi_n \Box \sigma_n$ se obtiene un almacén irreducible. Es decir, debemos probar que cada regla puede ser aplicada solo un número finito de veces.

   Sea una secuencia finita de i pasos $\Pi \Box \sigma \vdash^*_{\mathcal{M}} \Pi_i \Box \sigma_i$. Si $\Pi_i \Box \sigma_i$ es reducible (un almacén es reducible si existe una regla de transformación de almacenes que puede ser aplicada para transformarla) entonces se puede dar alguno de los siguientes casos:

   - **M3** puede ser aplicada si Π_i contiene la restricción X `#==` u' con $u' \in \mathbb{R}$ y $\exists u \in \mathbb{Z}$ tal que u `#==`$^{\mathcal{M}_{\mathcal{FD}.\mathcal{R}}}$ $u' \rightarrow true$ y $\sigma_1 = \{X \mapsto u\}$. En este caso, se aplica **M3** y la restricción puente es eliminada. Como Π es finito entonces el número de veces que se puede aplicar **M3** es finito.

     Se puede aplicar un razonamiento similar a las reglas **M5**, **M6** y **M8**.
   - **M4** puede ser aplicada, pero solo una vez, pues en este caso el cómputo termina. Se puede aplicar un razonamiento similar a las reglas **M7** y **M9**.
   - **M1** puede ser aplicada, en este caso se reduce a un nuevo almacén con un puente que, si se procesa, es por una de las reglas comprendidas entre **M3** y **M9**, y como estas reglas se aplican un número finito de veces como se acaba de ver y Π es finito, entonces el número de veces que **M1** puede ser aplicada es finito.

     Se puede aplicar un razonamiento similar a la regla **M2**.

   Por lo tanto solo se puede aplicar un número finito de pasos demostrando así que el cómputo es terminante.

4. **Corrección local:** hay que demostrar que para cualquier almacén $\Pi \Box \sigma$ del dominio $\mathcal{M}_{\mathcal{FD}.\mathcal{R}}$, el conjunto

$\bigcup\{Sol_{\mathcal{M}_{\mathcal{FD}.\mathcal{R}}}(\exists\overline{Y'}(\Pi' \square \sigma')) \mid \Pi \square \sigma \vdash_{\mathcal{M}_{\mathcal{FD}.\mathcal{R}}} \Pi' \square \sigma', \overline{Y'} = var(\Pi' \square \sigma') \setminus var(\Pi \square \sigma)\}$ es un subconjunto de $Sol_{\mathcal{M}_{\mathcal{FD}.\mathcal{R}}}(\Pi \square \sigma)$.

Es decir, hay que probar que para todo almacén $\Pi' \square \sigma'$ que se obtiene aplicando alguna regla definida en la tabla 5.1 al almacén $\Pi \square \sigma$ se cumple:

$Sol_{\mathcal{M}_{\mathcal{FD}.\mathcal{R}}}(\Pi \square \sigma) \supseteq Sol_{\mathcal{M}_{\mathcal{FD}.\mathcal{R}}}(\Pi' \square \sigma')$ con $\overline{Y'} = var(\Pi' \square \sigma') \setminus var(\Pi \square \sigma)\}$

Esta condición se demuestra por una distinción de casos de las reglas definidas en la tabla 5.1. En cada caso, asumimos que los almacenes tienen exactamente la forma mostrada por la correspondiente regla de transformación en la tabla 5.1.

- **M1**: Sea $\eta \in Sol_{\mathcal{M}_{\mathcal{FD}.\mathcal{R}}}((t$ #==$s, \Pi)\sigma_1 \square \sigma\sigma_1)$.
  Tenemos que comprobar que $\eta \in Sol_{\mathcal{M}_{\mathcal{FD}.\mathcal{R}}}((t$ #== $s) \rightarrow!\ B,\ \Pi \square \sigma)$ con $t \in \mathcal{Var} \cup \mathbb{Z}$, $s \in \mathcal{Var} \cup \mathbb{R}$, $B \in \mathcal{Var}$ y $\sigma_1 = \{B \mapsto true\}$.
  Usando el lema auxiliar de corrección (lema 8), mostrado anteriormente, sabemos:

  $$Sol_{\mathcal{M}_{\mathcal{FD}.\mathcal{R}}}((t \text{ \#==s},\Pi)\sigma_1) \cap Sol_{\mathcal{M}_{\mathcal{FD}.\mathcal{R}}}(\sigma\sigma_1) \subseteq Sol_{\mathcal{M}_{\mathcal{FD}.\mathcal{R}}}(t \text{ \#==s},\Pi) \cap Sol_{\mathcal{M}_{\mathcal{FD}.\mathcal{R}}}(\sigma)$$

  Entonces $\eta \in Sol_{\mathcal{M}_{\mathcal{FD}.\mathcal{R}}}((t$ #==$s, \Pi) \square \sigma)$. Pero, según se explicó en el capítulo 3, las restricciones de la forma $p\ e_1 \cdots e_n \rightarrow!\ true$ se abrevian a $p\ e_1 \cdots e_n$. Por lo tanto, t #==s se puede escribir como t #==$s \rightarrow!B$ donde B es una variable que se vincula a `true`. De esta forma $\eta \in Sol_{\mathcal{M}_{\mathcal{FD}.\mathcal{R}}}((t$ #== $s) \rightarrow!\ B,\ \Pi \square \sigma)$
  Se puede aplicar un razonamiento similar a la regla **M2**.
- **M3**: Sea $\eta \in Sol_{\mathcal{M}_{\mathcal{FD}.\mathcal{R}}}(\Pi\sigma_1 \square \sigma\sigma_1)$.
  Tenemos que comprobar que $\eta \in Sol_{\mathcal{M}_{\mathcal{FD}.\mathcal{R}}}$(X #== u', $\Pi \square \sigma)$ donde $u' \in \mathbb{R}$, u #==$^{\mathcal{M}_{\mathcal{FD}.\mathcal{R}}}$ $u' \rightarrow true$ y $\sigma_1 = \{X \mapsto u\}$.
  Usando el lema auxiliar de corrección (lema 8), mostrado anteriormente, sabemos que:

  $$Sol_{\mathcal{M}_{\mathcal{FD}.\mathcal{R}}}(\Pi\sigma_1) \cap Sol_{\mathcal{M}_{\mathcal{FD}.\mathcal{R}}}(\sigma\sigma_1) \subseteq Sol_{\mathcal{M}_{\mathcal{FD}.\mathcal{R}}}(\Pi) \cap Sol_{\mathcal{M}_{\mathcal{FD}.\mathcal{R}}}(\sigma)$$

  Entonces $\eta \in Sol_{\mathcal{M}_{\mathcal{FD}.\mathcal{R}}}(\Pi \square \sigma)$. Además, $\eta \in Sol_{\mathcal{M}_{\mathcal{FD}.\mathcal{R}}}$(X #== u') porque $\eta \in Sol_{\mathcal{M}_{\mathcal{FD}.\mathcal{R}}}(\Pi\sigma_1 \square \sigma\sigma_1)$ con $\sigma_1 = \{X \mapsto u\}$ y donde se cumple u #==$^{\mathcal{M}_{\mathcal{FD}.\mathcal{R}}}$ $u' \rightarrow true$ entonces $\eta \in Sol_{\mathcal{M}_{\mathcal{FD}.\mathcal{R}}}$(X #== u',$\Pi \square \sigma)$.
  Se puede aplicar un razonamiento similar a la regla **M5**.
- **M4**, **M7**, **M9**: Trivial, porque $Sol_{\mathcal{M}_{\mathcal{FD}.\mathcal{R}}}(\blacksquare) = \emptyset$ y el conjunto vacío es subconjunto de cualquier conjunto.
- **M6**: Trivial, ya que u #==$^{\mathcal{M}_{\mathcal{FD}.\mathcal{R}}}$ $u' \rightarrow true$, entonces $\eta \in Sol_{\mathcal{M}_{\mathcal{FD}.\mathcal{R}}}$(u #== u',$\Pi \square \sigma)$ se cumple para cualquier $\eta \in Sol_{\mathcal{M}_{\mathcal{FD}.\mathcal{R}}}(\Pi \square \sigma)$.
  Se puede aplicar un razonamiento similar a la regla **M8**.

Con esta distinción de casos se ha probado que toda solución del almacén $\Pi' \square \sigma'$ es solución del almacén $\Pi \square \sigma$ si se puede obtener $\Pi' \square \sigma'$ desde $\Pi \square \sigma$ aplicando alguna regla definida en la tabla 5.1.

5. **Completitud local:** para cualquier almacén $\Pi \,\square\, \sigma$ del dominio $\mathcal{M}_{\mathcal{FD}.\mathcal{R}}$, el conjunto $WTSol_{\mathcal{M}_{\mathcal{FD}.\mathcal{R}}}(\Pi \,\square\, \sigma)$ es un subconjunto de

   $\bigcup\{WTSol_{\mathcal{M}_{\mathcal{FD}.\mathcal{R}}}(\exists\overline{Y'}(\Pi' \,\square\, \sigma')) \mid \Pi\square\sigma \vdash_{\mathcal{M}_{\mathcal{FD}.\mathcal{R}}} \Pi'\square\sigma', \overline{Y'} = var(\Pi'\square\sigma') \setminus var(\Pi\square\sigma)\}$

   Esta condición se demuestra por una distinción de casos de las reglas definidas en la tabla 5.1. En cada caso, asumimos que los almacenes tienen exactamente la forma mostrada por la correspondiente regla de transformación en la tabla 5.1.

   - **M1**: Sea $\eta \in Sol_{\mathcal{M}_{\mathcal{FD}.\mathcal{R}}}\ ((t \texttt{ \#== } s) \to!\ B,\ \Pi \,\square\, \sigma)$.

     Tenemos que comprobar que $\eta \in Sol_{\mathcal{M}_{\mathcal{FD}.\mathcal{R}}}((t \texttt{ \#==} s, \Pi)\sigma_1 \,\square\, \sigma\sigma_1)$ con $t \in \mathcal{Var} \cup \mathbb{Z}$, $s \in \mathcal{Var} \cup \mathbb{R}$, $B \in \mathcal{Var}$ y $\sigma_1 = \{B \mapsto true\}$.

     Pero como $\sigma_1 = \{B \mapsto true\}$, la restricción $t \texttt{ \#== } s \to! B$ se simplifica a $t \texttt{ \#== } s$, resultando $\eta \in Sol_{\mathcal{M}_{\mathcal{FD}.\mathcal{R}}}\ ((t \texttt{ \#== } s), \Pi \,\square\, \sigma)$

     Usando el lema de completitud (lema 9) con $\overline{Y'} = \emptyset$, $\eta' = \eta$ y $\sigma_1 = \{B \mapsto true\}$, $\sigma_1\eta' = \eta'$ se cumple, entonces $\sigma\eta' = \eta'$ y $\eta' \in WTSol_{\mathcal{D}}((t \texttt{ \#== } s), \Pi)\ \sigma_1) \cap WTSol_{\mathcal{D}}(\sigma\sigma_1)$.

     Se puede aplicar un razonamiento similar a la regla **M2**.

   - **M3**: Sea $\eta \in WTSol_{\mathcal{M}_{\mathcal{FD}.\mathcal{R}}}(\text{X } \texttt{\#==} \text{ u'}, \Pi \,\square\, \sigma)$.

     Tenemos que comprobar que $\eta \in WTSol_{\mathcal{M}_{\mathcal{FD}.\mathcal{R}}}(\Pi\sigma_1 \,\square\, \sigma\sigma_1)$ con $u' \in \mathbb{R}$ y $\exists u \in \mathbb{Z}$ tal que $u \texttt{ \#==}^{\mathcal{M}_{\mathcal{FD}.\mathcal{R}}} u' \to true$ y $\sigma_1 = \{X \mapsto u\}$.

     Puesto que $\eta \in WTSol_{\mathcal{M}_{\mathcal{FD}.\mathcal{R}}}(\text{X } \texttt{\#==} \text{ u'}, \Pi \,\square\, \sigma)$, $\eta \in WTSol_{\mathcal{M}_{\mathcal{FD}.\mathcal{R}}}(\text{X } \texttt{\#==} \text{ u'})$ y $\eta \in WTSol_{\mathcal{M}_{\mathcal{FD}.\mathcal{R}}}(\Pi \,\square\, \sigma)$. Como $u \texttt{ \#==}^{\mathcal{M}_{\mathcal{FD}.\mathcal{R}}} u' \to true$, entonces $\eta(X) = u$.

     Usando el lema de completitud (lema 9) con $\overline{Y'} = \emptyset$, $\eta' = \eta$ y $\sigma_1 = \{X \mapsto u\}$, $\sigma_1\eta' = \eta'$ se cumple, entonces $\sigma\eta' = \eta'$ y $\eta' \in WTSol_{\mathcal{D}}(\Pi\sigma_1) \cap WTSol_{\mathcal{D}}(\sigma\sigma_1)$.

     Se puede aplicar un razonamiento similar a la regla **M5**.

   - **M4**: Como $\nexists u \in \mathbb{Z}$ tal que $u \texttt{ \#==}^{\mathcal{M}_{\mathcal{FD}.\mathcal{R}}} u' \to true$ entonces no hay soluciones y $\emptyset \in WTSol_{\mathcal{M}_{\mathcal{FD}.\mathcal{R}}}(\blacksquare)$.

     Se puede aplicar un razonamiento similar a las reglas **M7** y **M9**.

   - **M6**: Sea $\eta \in WTSol_{\mathcal{M}_{\mathcal{FD}.\mathcal{R}}}(\text{u } \texttt{\#==} \text{ u'}, \Pi \,\square\, \sigma)$ entonces $\eta \in WTSol_{\mathcal{M}_{\mathcal{FD}.\mathcal{R}}}(\Pi \,\square\, \sigma)$.

     Se puede aplicar un razonamiento similar a la regla **M8**.

   Con esta distinción de casos se ha probado que toda solución bien tipada del almacén $\Pi\square\sigma$ es solución del almacén $\Pi'\square\sigma'$ obtenido desde $\Pi\square\sigma$ aplicando alguna regla definida en la tabla 5.1.

Aplicando el lema 1 a las propiedades que se acaban de demostrar se concluye que $solve^{\mathcal{M}_{\mathcal{FD}.\mathcal{R}}}$ satisface todos los requerimientos de los resolutores de la definición 4.

c.q.d.

A.7 Propiedades del cálculo $CCLNC(\mathcal{C}_{\mathcal{FD}.\mathcal{R}})$

En este apéndice se incluyen las demostraciones de los resultados de corrección y completitud limitada que se enunciaron en la sección 5.3. Primero se presentan dos lemas necesarios para las demostraciones en el dominio de coordinación genérico $\mathcal{C}$. La notación $(WT)Sol$ indica que el resultado se cumple tanto para soluciones simples como para soluciones bien tipadas.

Lema 10. (Lema de sustitución)
Para cualquier $\Pi \subseteq PCon_{\mathcal{C}}$, $\sigma \in Sub_{\mathcal{C}}$ y $\eta \in Val_{\mathcal{C}}$, se cumple $\eta \in (WT)Sol_{\mathcal{C}}(\Pi\sigma) \Leftrightarrow \sigma\eta \in (WT)Sol_{\mathcal{C}}(\Pi)$.

Este lema se puede demostrar por inducción sobre la estructura sintáctica de Π.

Lema 11. (Resultado auxiliar que comprueba las soluciones de un objetivo)
Sea un objetivo admisible $G \equiv \exists\overline{U}.\ P \sqcap C \sqcap M \sqcap H \sqcap F \sqcap R$ de un programa $\mathcal{P}$. Sea $\overline{Y'}$ un conjunto de nuevas variables distintas de $\overline{U}$ y de las variables contenidas en G. Sean dos valoraciones $\mu, \hat{\mu} \in Val_{\mathcal{C}}$ tal que $\hat{\mu} =_{\backslash \overline{U}, \overline{Y'}} \mu$ y $\hat{\mu} \in (WT)Sol_{\mathcal{P}}(P \sqcap C \sqcap M \sqcap H \sqcap F \sqcap R)$. Entonces $\mu \in (WT)Sol_{\mathcal{P}}(G)$.

Demostración del lema 11

Sea $\hat{\hat{\mu}} \in Val_{\mathcal{C}}$ unívocamente definida por las dos siguientes condiciones $\hat{\hat{\mu}} =_{\backslash \overline{Y'}} \hat{\mu}$ y $\hat{\hat{\mu}} =_{\overline{Y'}} \mu$. Por hipótesis, $\hat{\mu} \in (WT)Sol_{\mathcal{P}}(P \sqcap C \sqcap M \sqcap H \sqcap F \sqcap R)$ y las variables $\overline{Y'}$ no aparecen en G. Por lo tanto, $\hat{\hat{\mu}} \in (WT)Sol_{\mathcal{P}}(P \sqcap C \sqcap M \sqcap H \sqcap F \sqcap R)$ está garantizado por la construcción de $\hat{\hat{\mu}}$. Por la definición 11 de soluciones de objetivos y sus testigos, solo se necesita demostrar $\hat{\hat{\mu}} =_{\backslash \overline{U}} \mu$ a fin de concluir $\mu \in (WT)Sol_{\mathcal{P}}(G)$. De hecho, dada una variable $X \notin \overline{U}$, o bien $X \in \overline{Y'}$ o bien $X \notin \overline{Y'}$. En el primer caso, $\hat{\hat{\mu}}(X) = \mu(X)$ por la construcción de $\hat{\hat{\mu}}$. En el segundo caso, $\hat{\hat{\mu}}(X) = \hat{\mu}(X)$ por la construcción de $\hat{\hat{\mu}}$ y $\hat{\mu}(X) = \mu(X)$ por hipótesis. *c.q.d.*

A.7.1 Demostración del teorema 7 (pág. 102)

El teorema 7 garantiza la corrección y completitud *local* para un paso de transformación de un objetivo dado. Para demostrar este teorema hay que comprobar lo siguiente:

Sea un programa $\mathcal{P}$ y un objetivo admisible G que no está en forma resuelta. Se elige una regla **RL** aplicable a G y se selecciona una parte γ de G sobre la que se aplica la regla **RL**. Entonces la afirmación de que existe una cantidad finita de transformaciones $G \Vdash_{\mathbf{RL},\gamma,\mathcal{P}} G'_j$ $(1 \leq j \leq k)$ puede ser trivialmente verificada mediante la inspección de todas las reglas en las tablas 4.1, 4.2, 5.2, 5.5 y 5.6. Además hay que demostrar las siguientes afirmaciones para las reglas de dichas tablas:

1. **Corrección local:** $Sol_{\mathcal{P}}(G) \supseteq \bigcup_{j=1}^{k} Sol_{\mathcal{P}}(G'_j)$.

2. **Completitud limitada local:** $WTSol_{\mathcal{P}}(G) \subseteq \bigcup_{j=1}^{k} WTSol_{\mathcal{P}}(G'_j)$, donde la aplicación de la regla **RL** a la parte seleccionada γ de G es *segura* en el siguiente sentido: no es una aplicación opaca de **DC** ni es una aplicación de una regla de las tablas 4.2 y 5.6 que implique una invocación incompleta al resolutor.

Los puntos 1 y 2 deben ser probados para cada regla **RL** por separado. Para algunas reglas se necesita la construcción de ciertos testigos definidos como multiconjuntos de árboles de prueba $CRWL(\mathcal{C}_{\mathcal{FD}.\mathcal{R}})$. La técnica empleada para la construcción de estos testigos se describe en los trabajos [GHLR96, GHLR99] y más tarde, de [LRV04].

Con respecto a la tabla 4.1, se van a tomar como representantes las reglas **DF** y **FC** y con respecto a las tablas 4.2, 5.2, 5.5 y 5.6 se trabajará sobre la mayoría de las reglas, pues están directamente involucradas en la cooperación de dominios de restricciones. Cuando se trata cada regla **RL**, vamos a suponer que G y G'_j son exactamente como el objetivo original y el objetivo transformado tal y como aparece en la presentación de **RL** en las correspondientes tablas.

Reglas seleccionadas de la tabla 4.1

Regla **DF, Defined Function**. En este caso, γ es una producción $f\,\overline{e}_n \to t$.

1. **Corrección local:** Sea $\mu \in Sol_{\mathcal{P}}(G'_j)$ para algún $1 \leq j \leq k$. Entonces existe $\mu' =_{\setminus \overline{Y}, \overline{U}} \mu$ tal que $\mu' \in Sol_{\mathcal{P}}(\overline{e_n \to t_n},\ r \to t,\ P \,\square\, C', C \,\square\, M \,\square\, H \,\square\, F \,\square\, R)$. De esto se deduce que $\mu' \in Sol_{\mathcal{C}_{\mathcal{FD}.\mathcal{R}}}(M \,\square\, H \,\square\, F \,\square\, R)$ y que se puede construir un testigo[1] $\mathcal{M}'$ tal que $\mathcal{M}' : \mathcal{P} \vdash_{CRWL(\mathcal{C}_{\mathcal{FD}.\mathcal{R}})} (\overline{e_n \to t_n},\ r \to t,\ P \,\square\, C', C)\mu'$. Una parte de $\mathcal{M}'$ prueba $(\overline{e_n \to t_n},\ r \to t, C')\mu'$, lo que permite deducir $(f\,\overline{e}_n \to t)\mu'$ usando la regla de inferencia $CRWL(\mathcal{C}_{\mathcal{FD}.\mathcal{R}})$ que trata las funciones definidas. Por lo tanto, $\mathcal{M}'$ se puede utilizar para construir otro testigo $\mathcal{M} : \mathcal{P} \vdash_{CRWL(\mathcal{C}_{\mathcal{FD}.\mathcal{R}})} (f\,\overline{e}_n \to t,\ P \,\square\, C)\mu'$. Como $\mu' =_{\setminus \overline{U}} \mu$, se puede concluir que $\mu \in Sol_{\mathcal{P}}(G)$.

2. **Completitud local limitada:** Sea $\mu \in WTSol_{\mathcal{P}}(G)$. Entonces existe una cierta $\mu' =_{\setminus \overline{U}} \mu$ tal que $\mu' \in WTSol_{\mathcal{P}}(f\,\overline{e}_n \to t,\ P \,\square\, C \,\square\, M \,\square\, H \,\square\, F \,\square\, R)$. Entonces, $\mu' \in WTSol_{\mathcal{C}_{\mathcal{FD}.\mathcal{R}}}(M \,\square\, H \,\square\, F \,\square\, R)$ y existe un testigo $\mathcal{M}$ tal que $\mathcal{M} : \mathcal{P} \vdash_{CRWL(\mathcal{C}_{\mathcal{FD}.\mathcal{R}})} (f\,\overline{e}_n \to t,\ P \,\square\, C)\mu'$. Hay que tener en cuenta que $\mathcal{M}$ debe incluir un árbol de prueba $\mathcal{T}$ que demuestra $(f\,\overline{e}_n \to t)\mu'$ usando alguna instancia de $Rl : f\,\overline{t_n} \to r \Leftarrow C'$, convenientemente elegida como una variante de alguna regla de $\mathcal{P}$ con variables nuevas $\overline{Y} = var(Rl)$. Sea G'_j el resultado de aplicar **DF** con $f\,\overline{e}_n \to t$ como la parte seleccionada de G y Rl como la regla seleccionada de $\mathcal{P}$ para f. Se puede construir una valoración $\mu'' \in Val_{\mathcal{C}_{\mathcal{FD}.\mathcal{R}}}$ bien tipada que instancia las variables en $\overline{Y}$ como se requiere por el árbol de prueba $\mathcal{T}$, e instancia cualquier otra variable V a $\mu'(V)$. Con una

[1] La notación $\mathcal{M} : \mathcal{P} \vdash_{CRWL(\mathcal{C})} (P \,\square\, C)\mu'$ indica que el testigo $\mathcal{M}$ es un multiconjunto formado por árboles de prueba $CRWL(\mathcal{C})$ que demuestra $(P \,\square\, C)\mu'$ para el programa $\mathcal{P}$ utilizando las reglas de inferencia de la lógica $CRWL(\mathcal{C})$ presentadas en [LRV07, dVV08]

adecuada reutilización de fragmentos del testigo de $\mathcal{M}$ es posible construir un testigo $\mathcal{M}' : \mathcal{P} \vdash_{CRWL(\mathcal{C}_{\mathcal{FD}.\mathcal{R}})} (\overline{e_n \to t_n},\, r \to t,\, P \,\square\, C', C)\mu''$. Como $\mu'' =_{\backslash \overline{Y}, \overline{U}} \mu$ se puede concluir que $\mu \in WTSol_{\mathcal{P}}(G'_j)$.

Regla **FC, Flatten Constraint**. En este caso, γ es una restricción atómica $p\,\overline{e}_n \to\!!\, t$ de tal manera que algunos e_i no son patrones y $k = 1$. Utilizamos G' en lugar de G'_1. Para simplificar, consideramos $p\,e_1\,t_2 \to\!!\, t$, donde e_1 no es un patrón. La presentación de la regla es entonces como en la tabla 4.1 con $n = 2$, $m = 1$.

1. **Corrección local:** Sea $\mu \in Sol_{\mathcal{P}}(G')$. Entonces existe $\mu' =_{\backslash V_1, \overline{U}} \mu$ tal que $\mu' \in Sol_{\mathcal{P}}(e_1 \to V_1,\, P \,\square\, p\,V_1\,t_2 \to\!!\, t,\, C \,\square\, M \,\square\, H \,\square\, F \,\square\, R)$. Por lo tanto, $\mu' \in Sol_{\mathcal{C}_{\mathcal{FD}.\mathcal{R}}}(M \,\square\, H \,\square\, F \,\square\, R)$ y existe un testigo $\mathcal{M}'$ tal que $\mathcal{M}' : \mathcal{P} \vdash_{CRWL(\mathcal{C}_{\mathcal{FD}.\mathcal{R}})} (e_1 \to V_1,\, P \,\square\, p\,V_1\,t_2 \to\!!\, t,\, C)\mu'$. Una parte de $\mathcal{M}'$ prueba $(e_1 \to V_1\ p\,V_1\,t_2 \to\!!\, t)\mu'$, lo que permite deducir $(p\,e_1\,t_2 \to\!!\, t)\mu'$ usando la regla de inferencia $CRWL(\mathcal{C}_{\mathcal{FD}.\mathcal{R}})$ que trata las funciones primitivas. Por lo tanto, $\mathcal{M}'$ se puede utilizar para construir otro testigo $\mathcal{M} : \mathcal{P} \vdash_{CRWL(\mathcal{C}_{\mathcal{FD}.\mathcal{R}})} (P \,\square\, p\,e_1\,t_2 \to\!!\, t,\, C)\mu'$. Como $\mu' =_{\backslash \overline{U}} \mu$, se concluye que $\mu \in Sol_{\mathcal{P}}(G)$.

2. **Completitud local limitada:** Sea $\mu \in WTSol_{\mathcal{P}}(G)$. Entonces existe una cierta $\mu' =_{\backslash \overline{U}} \mu$ tal que $\mu' \in WTSol_{\mathcal{P}}(P \,\square\, p\,e_1\,t_2 \to\!!\, t,\, C \,\square\, M \,\square\, H \,\square\, F \,\square\, R)$. Entonces, $\mu' \in WTSol_{\mathcal{C}_{\mathcal{FD}.\mathcal{R}}}(M \,\square\, H \,\square\, F \,\square\, R)$ y existe un testigo $\mathcal{M}$ tal que $\mathcal{M} : \mathcal{P} \vdash_{CRWL(\mathcal{C}_{\mathcal{FD}.\mathcal{R}})} (P \,\square\, p\,e_1\,t_2 \to\!!\, t,\, C)\mu'$. Se ha de tener en cuenta que $\mathcal{M}$ debe incluir un árbol de prueba $\mathcal{T}$ que demuestra la restricción atómica $(p\,e_1\,t_2 \to\!!\, t)\mu'$. Una parte de $\mathcal{T}$ debe demostrar una producción de la forma $e_1\mu' \to t_1$ por algún patrón adecuado t_1. Considérese $\mu'' \in Val_{\mathcal{C}_{\mathcal{FD}.\mathcal{R}}}$ bien tipado tal que $\mu''(V_1) = t_1$ y $\mu'' =_{\backslash V_1} \mu'$. Reutilizando adecuadamente partes del testigo $\mathcal{M}$, es posible construir un testigo $\mathcal{M}' : \mathcal{P} \vdash_{CRWL(\mathcal{C}_{\mathcal{FD}.\mathcal{R}})} (e_1 \to V_1,\, P \,\square\, p\,V_1\,t_2 \to\!!\, t,\, C)\mu''$. Como $\mu'' =_{\backslash V_1, \overline{U}} \mu$, se concluye que $\mu \in WTSol_{\mathcal{P}}(G')$.

Reglas de la tabla 5.2

Regla **SB, Set Bridges**. En este caso, γ es una restricción primitiva atómica π que se utiliza para computar los puentes y $k = 1$. Utilizaremos G' en vez de G'_1. La aplicación de esta regla calcula $\exists \overline{V'}\, B' = bridges^{\mathcal{D} \to \mathcal{D}'}(\pi, \Pi_M) \neq \emptyset$, donde Π_M es el conjunto de puentes del almacén M y $\mathcal{D} = \mathcal{FD}$ y $\mathcal{D}' = \mathcal{R}$ o viceversa, de acuerdo con los dos casos (i) y (ii) de la tabla 5.2.

1. **Corrección local:** Sea $\mu \in Sol_{\mathcal{P}}(G')$. Entonces existe $\mu' =_{\backslash \overline{V'}, \overline{U}} \mu$ tal que $\mu' \in Sol_{\mathcal{P}}(P \,\square\, \pi,\, C \,\square\, M' \,\square\, H \,\square\, F \,\square\, R)$. Por lo tanto, $\mu' \in Sol_{\mathcal{C}_{\mathcal{FD}.\mathcal{R}}}(M' \,\square\, H \,\square\, F \,\square\, R)$ y existe un testigo $\mathcal{M}'$ tal que $\mathcal{M}' : \mathcal{P} \vdash_{CRWL(\mathcal{C}_{\mathcal{FD}.\mathcal{R}})} (P \,\square\, \pi,\, C)\mu'$. Como M' es B', M entonces $\mu' \in Sol_{\mathcal{C}_{\mathcal{FD}.\mathcal{R}}}(M \,\square\, H \,\square\, F \,\square\, R)$, además $\mathcal{M}' : \mathcal{P} \vdash_{CRWL(\mathcal{C}_{\mathcal{FD}.\mathcal{R}})} (P \,\square\, \pi,\, C)\mu'$. Ambos implican $\mu \in Sol_{\mathcal{P}}(G)$ por el lema 11.

2. **Completitud local limitada:** Sea $\mu \in WTSol_{\mathcal{P}}(G)$. Entonces existe $\mu' =_{\backslash \overline{U}} \mu$ tal que $\mu' \in WTSol_{\mathcal{P}}(P \,\square\, \pi,\, C \,\square\, M \,\square\, H \,\square\, F \,\square\, R)$. Por lo tanto, $\mu' \in WTSol_{\mathcal{C}_{\mathcal{FD}.\mathcal{R}}}(M \,\square\, H \,\square\, F \,\square\, R)$ y existe un testigo $\mathcal{M}$ tal que $\mathcal{M} : \mathcal{P} \vdash_{CRWL(\mathcal{C}_{\mathcal{FD}.\mathcal{R}})} (P \,\square\, \pi,\, C)\mu'$. Como π es primitiva, estas condiciones implican $\mu' \in WTSol_{\mathcal{C}_{\mathcal{FD}.\mathcal{R}}}(\pi \wedge \Pi_M)$. Por el punto 2 de la proposición 6, se sabe que $WTSol_{\mathcal{C}_{\mathcal{FD}.\mathcal{R}}}(\pi \wedge \Pi_M) \subseteq WTSol_{\mathcal{C}_{\mathcal{FD}.\mathcal{R}}}(\exists \overline{V'}(\pi \wedge \Pi_M \wedge B'))$, donde $\overline{V'}$

son variables nuevas. De esto podemos concluir que $\mu' \in WTSol_{\mathcal{C}_{\mathcal{FD}.\mathcal{R}}}(\exists\overline{V'}(\pi \wedge \Pi_M \wedge B'))$ y por lo tanto hay una cierta $\mu'' =_{\setminus\overline{V'}} \mu'$ tal que $\mu'' \in WTSol_{\mathcal{C}_{\mathcal{FD}.\mathcal{R}}}(\pi \wedge \Pi_M \wedge B')$. Como $\overline{V'}$ son variables nuevas que no aparecen en G se puede comprobar que $\mu'' \in WTSol_{\mathcal{C}_{\mathcal{FD}.\mathcal{R}}}(M' \,\square\, H \,\square\, F \,\square\, R)$ y $\mathcal{M} : \mathcal{P} \vdash_{CRWL(\mathcal{C}_{\mathcal{FD}.\mathcal{R}})} (P \,\square\, \pi,\, C)\mu''$, lo que asegura $\mu \in WTSol_{\mathcal{P}}(G')$.

Regla **PP, Propagate Projections**. En este caso, γ es una restricción primitiva atómica π que se usa para computar la proyección y $k = 1$. Utilizaremos G' en lugar de G'_1. La aplicación de la regla obtiene G' de G evaluando $\exists\overline{V'}\,\Pi' = proj^{\mathcal{D}\to\mathcal{D}'}(\pi, \Pi_M) \neq \emptyset$, donde Π_M es el conjunto de puentes del almacén M y $\mathcal{D} = \mathcal{FD}$ y $\mathcal{D}' = \mathcal{R}$ o viceversa, de acuerdo con los dos casos (i) y (ii) de la tabla 5.2. Los razonamientos de corrección local y completitud local limitada son bastante similares a los utilizados en el caso de la Regla **SB**, excepto que se utiliza el punto 3 de la Proposición 6 en lugar del punto 2.

Regla **SC, Submit Constraints**. En este caso, γ es una restricción primitiva atómica π y $k = 1$. Utilizaremos G' en lugar de G'_1.

1. **Corrección local:** Sea $\mu \in Sol_{\mathcal{P}}(G')$. Entonces existe $\mu' =_{\setminus\overline{U}} \mu$ tal que $\mu' \in Sol_{\mathcal{P}}(P \,\square\, C \,\square\, M' \,\square\, H' \,\square\, F' \,\square\, R')$.

   Por lo tanto, $\mu' \in Sol_{\mathcal{C}_{\mathcal{FD}.\mathcal{R}}}(M' \,\square\, H' \,\square\, F' \,\square\, R')$ y existe un testigo $\mathcal{M}'$ tal que $\mathcal{M}' : \mathcal{P} \vdash_{CRWL(\mathcal{C}_{\mathcal{FD}.\mathcal{R}})} (P \,\square\, C)\mu'$.

   Debido la relación sintáctica entre G y G', $\mu' \in Sol_{\mathcal{C}_{\mathcal{FD}.\mathcal{R}}}(M' \,\square\, H' \,\square\, F' \,\square\, R')$ equivale a $\mu' \in Sol_{\mathcal{C}_{\mathcal{FD}.\mathcal{R}}}(M \,\square\, H \,\square\, F \,\square\, R)$ y también $\mu' \in Sol_{\mathcal{C}_{\mathcal{FD}.\mathcal{R}}}(\pi)$. Como $\mu' \in Sol_{\mathcal{C}_{\mathcal{FD}.\mathcal{R}}}(\pi)$, el testigo $\mathcal{M}'$ se puede ampliar a otro testigo $\mathcal{M} : \mathcal{P} \vdash_{CRWL(\mathcal{C}_{\mathcal{FD}.\mathcal{R}})} (P \,\square\, \pi,\, C)\mu'$.

   A partir de $\mathcal{M}$ se tiene $\mu' \in Sol_{\mathcal{P}}(P \,\square\, \pi,\, C \,\square\, M \,\square\, H \,\square\, F \,\square\, R)$ y por lo tanto $\mu \in Sol_{\mathcal{P}}(G)$.

2. **Completitud local limitada:** Sea $\mu \in WTSol_{\mathcal{P}}(G)$. Entonces existe $\mu' =_{\setminus\overline{U}} \mu$ tal que $\mu' \in WTSol_{\mathcal{P}}(P \,\square\, \pi,\, C \,\square\, M \,\square\, H \,\square\, F \,\square\, R)$.

   Por lo tanto, $\mu' \in WTSol_{\mathcal{C}_{\mathcal{FD}.\mathcal{R}}}(M \,\square\, H \,\square\, F \,\square\, R)$ y existe un testigo $\mathcal{M}$ tal que $\mathcal{M} : \mathcal{P} \vdash_{CRWL(\mathcal{C}_{\mathcal{FD}.\mathcal{R}})} (P \,\square\, \pi,\, C)\mu'$. Debido a la relación sintáctica entre G y G' y al hecho de que π es primitiva, se puede concluir que $\mu' \in WTSol_{\mathcal{C}_{\mathcal{FD}.\mathcal{R}}}(M' \,\square\, H' \,\square\, F' \,\square\, R')$. Sea $\mathcal{M}'$ el testigo construido a partir de $\mathcal{M}$ omitiendo el árbol de prueba para $\pi\mu'$ que es parte de $\mathcal{M}$. Entonces $\mathcal{M}' : \mathcal{P} \vdash_{CRWL(\mathcal{C}_{\mathcal{FD}.\mathcal{R}})} (P \,\square\, C)\mu'$. Esto permite llegar a la conclusión $\mu' \in WTSol_{\mathcal{P}}(P \,\square\, C \,\square\, M' \,\square\, H' \,\square\, F' \,\square\, R')$ y por lo tanto $\mu \in WTSol_{\mathcal{P}}(G')$.

Reglas de la tabla 5.5

Regla **IE, Infer Equalities**. Esta regla incluye dos casos similares. Aquí vamos a tratar sólo el primer caso, el segundo es completamente análogo. La parte seleccionada γ consta de un par de puentes de la forma X `#==` RX, X' `#==` RX y $k = 1$. Utilizaremos G' en lugar de G'_1.

1. **Corrección local:** Sea $\mu \in Sol_{\mathcal{P}}(G')$. Entonces existe $\mu' =_{\setminus\overline{U}} \mu$ tal que $\mu' \in Sol_{\mathcal{P}}(P \,\square\, C \,\square\, X$ `#==` $RX,\, M \,\square\, H \,\square\, X$ `==` $X',\, F \,\square\, R)$. Esto implica dos hechos: en primer lugar que existe un testigo $\mathcal{M}'$ tal que $\mathcal{M}' : \mathcal{P} \vdash_{CRWL(\mathcal{C}_{\mathcal{FD}.\mathcal{R}})} (P \,\square\, C)\mu'$; y en segundo

lugar, $\mu' \in Sol_{\mathcal{C}_{\mathcal{FD}.\mathcal{R}}}(X \texttt{ \#== } RX,\ M \square H \square X \texttt{ == } X',\ F \square R)$. El segundo hecho implica $\mu' \in Sol_{\mathcal{C}_{\mathcal{FD}.\mathcal{R}}}(X \texttt{ \#== } RX,\ X' \texttt{ \#== } RX,\ M \square H \square F \square R)$. Junto con el testigo $\mathcal{M}'$, esta condición garantiza $\mu' \in Sol_{\mathcal{P}}(P \square C \square X \texttt{ \#== } RX,\ X' \texttt{ \#== } RX,\ M \square H \square F \square R)$ y por lo tanto $\mu \in Sol_{\mathcal{P}}(G)$.

2. **Completitud local limitada:** Sea $\mu \in WTSol_{\mathcal{P}}(G)$. Entonces existe $\mu' =_{\setminus \overline{U}} \mu$ tal que $\mu' \in WTSol_{\mathcal{P}}(P \square C \square X \texttt{ \#== } RX,\ X' \texttt{ \#== } RX,\ M \square H \square F \square R)$. Esto implica dos hechos: en primer lugar que existe un testigo $\mathcal{M}$ tal que $\mathcal{M} : \mathcal{P} \vdash_{CRWL(\mathcal{C}_{\mathcal{FD}.\mathcal{R}})} (P \square C)\mu'$; y en segundo lugar, $\mu' \in WTSol_{\mathcal{C}_{\mathcal{FD}.\mathcal{R}}}(X \texttt{ \#== } RX,\ X' \texttt{ \#== } RX,\ M \square H \square F \square R)$. El segundo hecho implica $\mu' \in WTSol_{\mathcal{C}_{\mathcal{FD}.\mathcal{R}}}(X \texttt{ \#== } RX,\ M \square H \square X \texttt{ == } X',\ F \square R)$. Entonces, $\mu' \in WTSol_{\mathcal{P}}(P \square C \square X \texttt{ \#== } RX,\ M \square H \square X \texttt{ == } X',\ F \square R)$ se cumple gracias al mismo testigo $\mathcal{M}$, y por lo tanto $\mu \in Sol_{\mathcal{P}}(G')$.

Regla **ID, Infer Disequalities**. Esta regla incluye dos casos similares. Aquí vamos a tratar sólo la primera, la segunda es completamente análoga. La parte seleccionada γ es un antipuente de la forma $X \texttt{ \#/== } u'$ situado dentro del almacén M, y $k = 1$. Utilizaremos G' en lugar de G'_1.

La aplicación de la regla obtiene G' de G quitando $X \texttt{ \#/== } u'$ de M añadiendo una restricción de desigualdad semánticamente equivalente $X \texttt{ /= } u$ al almacén F. Los razonamientos de corrección local y completitud local limitado son muy similares a los utilizados en el caso de la Regla **IE**.

Reglas de las tablas 4.2 y 5.6

En la tabla 4.2 se presentan las reglas $\mathbf{D}_i\mathbf{S}$, **MS**, **HS** y **SF**. La primera regla expresa el caso general de enviar una restricción a su correspondiente resolutor. Esta regla se particulariza en el calculo $CCLNC(\mathcal{C}_{\mathcal{FD}.\mathcal{R}})$ con las reglas **FDS** y **RS** presentadas en la tabla 5.6. En esta demostración se probarán únicamente dos reglas: **FDS** y **SF**. La técnica que se aplica a la regla **FDS** se puede aplicar a las reglas **MS** y **RS**. En la demostración de la regla **FDS** se debe tener en cuenta que las propiedades corrección y completitud del resolutor de $\mathcal{FD}$ se refieren a las valoraciones sobre el universo $\mathcal{U}_{\mathcal{FD}}$, que debe estar relacionado con las valoraciones sobre el universo $\mathcal{U}_{\mathcal{C}_{\mathcal{FD}.\mathcal{R}}}$ por medio del teorema 4, como veremos a continuación.

La regla **HS** puede ser también tratada de manera similar a **FDS**, pero en este caso el teorema 4 no es necesario porque las propiedades corrección y completitud del resolutor extensible $\mathcal{H}$ se refiere directamente a las valoraciones sobre el universo $\mathcal{U}_{\mathcal{C}_{\mathcal{FD}.\mathcal{R}}}$.

Regla **FDS $\mathcal{FD}$-Constraint Solver**. La parte seleccionada γ es el almacén F.

1. **Corrección local:** Sea G' uno de los posible objetivos G'_j tal que $G \Vdash_{\mathbf{FDS},\gamma,\mathcal{P}} G'_j$. Entonces $G' = \exists\overline{Y'}, \overline{U}.(P \square C \square M \square H \square (\Pi' \square \sigma_F) \square R)@_{\mathcal{FD}}\sigma'$ para algún $\exists\overline{Y'}(\Pi' \square \sigma')$ elegido como una de las posibles alternativas computadas por el resolutor $\mathcal{FD}$, es decir, $\Pi_F \Vdash_{solve^{\mathcal{FD}}} \exists\overline{Y'}(\Pi' \square \sigma')$ donde $solve^{\mathcal{FD}}$ se ha definido como la secuenciación de dos resolutores: $solve^{\mathcal{FD}^T} \diamond solve^{\mathcal{FD}^N}$. Sea ahora $\mu \in Sol_{\mathcal{P}}(G')$. Entonces existe $\mu' =_{\setminus \overline{Y'},\overline{U}} \mu$ tal que

$$\mu' \in Sol_{\mathcal{P}}((P \square C \square M \square H \square (\Pi' \square \sigma_F) \square R)@_{\mathcal{FD}}\sigma')$$

para algún $\exists\overline{Y'}(\Pi' \square \sigma')$ tal que $\Pi_F \Vdash_{solve^{\mathcal{FD}}} \exists\overline{Y'}(\Pi' \square \sigma')$. Como $\Pi' \square \sigma'$ es un almacén, podemos suponer que $\Pi'\sigma' = \Pi'$ y deducir las siguientes condiciones:

$$(0)\text{ existe un testigo } \mathcal{M}' \text{ tal que } \mathcal{M}' : \mathcal{P} \vdash_{CRWL(\mathcal{C}_{\mathcal{FD}.\mathcal{R}})} (P \square C)\sigma'\mu'$$
$$(1)\, \mu' \in Sol_{\mathcal{C}_{\mathcal{FD}.\mathcal{R}}}(\Pi_M\sigma' \square \sigma_M \star \sigma')$$
$$(2)\, \mu' \in Sol_{\mathcal{C}_{\mathcal{FD}.\mathcal{R}}}(\Pi_H\sigma' \square \sigma_H \star \sigma')$$
$$(3)\, \mu' \in Sol_{\mathcal{C}_{\mathcal{FD}.\mathcal{R}}}(\Pi'\sigma' \square \sigma_F\sigma') \text{ donde } \Pi'\sigma' = \Pi'$$
$$(4)\, \mu' \in Sol_{\mathcal{C}_{\mathcal{FD}.\mathcal{R}}}(\Pi_R\sigma' \square \sigma_R \star \sigma')$$

En particular, (3) implica $\mu' \in Sol(\sigma_F\sigma')$, es decir

$$(5)\, \sigma_F\sigma'\mu' = \mu'$$

Con el fin de concluir que $\mu \in Sol_{\mathcal{P}}(G)$, se muestra que la hipótesis del lema auxilia 11 se cumple para $\hat{\mu} = \mu'$. Así, $\hat{\mu} =_{\setminus\overline{U},\overline{Y'}} \mu$ y las nuevas variables $\overline{Y'}$ son distintas de $\overline{U}$ y de las demás variables de G. Todavía tenemos que demostrar que $\mu' \in Sol_{\mathcal{P}}(P \square C \square M \square H \square F \square R)$.

- Demostración de $\mu' \in Sol_{\mathcal{P}}(P \square C)$: Debido a las propiedades invariantes de objetivos admisibles (véase pág. 74), $(P \square C) = (P \square C)\sigma_F$. Usando esta igualdad y (5) se obtiene $(P \square C)\sigma'\mu' = (P \square C)\sigma_F\sigma'\mu' = (P \square C)\mu'$. Por lo tanto, $\mathcal{M}' : \mathcal{P} \vdash_{CRWL(\mathcal{C}_{\mathcal{FD}.\mathcal{R}})} (P \square C)\mu'$ se deduce de (0).
- Demostración de $\mu' \in Sol_{\mathcal{C}_{\mathcal{FD}.\mathcal{R}}}(S)$, S siendo cualquiera de los almacenes M, H, R: De acuerdo con la elección de los S se puede utilizar (1), (2) o bien (4) para concluir
$$(6)\, \mu' \in Sol_{\mathcal{C}_{\mathcal{FD}.\mathcal{R}}}(\Pi_S\sigma')$$
$$(7)\, \mu' \in Sol(\sigma_S \star \sigma') \text{ es decir, } (\sigma_S \star \sigma')\mu' = \mu'$$
  - Demostración de $\mu' \in Sol_{\mathcal{C}_{\mathcal{FD}.\mathcal{R}}}(\Pi_S)$: Debido a las propiedades invariantes de objetivos admisibles, $\Pi_S = \Pi_S\sigma_F$. Entonces (6) es equivalente a $\mu' \in Sol_{\mathcal{C}_{\mathcal{FD}.\mathcal{R}}}(\Pi_S\sigma_F\sigma')$. Aplicando el lema de sustitución 10 se deduce $\sigma_F\sigma'\mu' \in Sol_{\mathcal{C}_{\mathcal{FD}.\mathcal{R}}}(\Pi_S)$, lo que equivale a $\mu' \in Sol_{\mathcal{C}_{\mathcal{FD}.\mathcal{R}}}(\Pi_S)$ por (5).
  - Demostración de $\mu' \in Sol(\sigma_S)$: Sea cualquier variable $X \in vdom(\sigma_S)$. Entonces
$$X\mu' = X\sigma_S\sigma'\mu' = X\sigma_S\sigma_F\sigma'\mu' = X\sigma_S\mu'$$
    donde la primera igualdad se cumple debido a (7), la segunda igualdad se cumple porque las propiedades de admisibilidad de G garantizan $\sigma_S \star \sigma_F = \sigma_S$, y la tercera igualdad se cumple debido a (5).
- Demostración de $\mu' \in Sol_{\mathcal{C}_{\mathcal{FD}.\mathcal{R}}}(F)$: En primer lugar, afirmamos que
$$(8)\ \ |\, \sigma'\mu' \,|_{\mathcal{FD}} \in Sol(\sigma') \text{ es decir } \sigma' \ |\, \sigma'\mu' \,|_{\mathcal{FD}} = |\, \sigma'\mu' \,|_{\mathcal{FD}}$$
  donde $|\cdot|_{\mathcal{D}}$ es el operador de truncamiento de una valoración (definición 10).
  Para probar la afirmación, se asume cualquier $X \in vdom(\sigma')$. Debido al postulado 2 hay dos casos posibles:

(a) $\sigma'(X)$ es un valor entero n. Entonces:

$$X\sigma' \mid \sigma'\mu' \mid_{\mathcal{FD}} = n = \mid X\sigma'\mu' \mid_{\mathcal{FD}} = X \mid \sigma'\mu' \mid_{\mathcal{FD}}$$

(b) $X \in var(\Pi_F)$ y $\sigma'(X)$ es una variable $X' \in var(\Pi_F)$. Entonces $\sigma'(X') = X'$ porque σ' es idempotente, y:

$$X\sigma' \mid \sigma'\mu' \mid_{\mathcal{FD}} = X' \mid \sigma'\mu' \mid_{\mathcal{FD}} = \mid X'\sigma'\mu' \mid_{\mathcal{FD}} =$$

$$\mid X'\mu' \mid_{\mathcal{FD}} = \mid X\sigma'\mu' \mid_{\mathcal{FD}} = X \mid \sigma'\mu' \mid_{\mathcal{FD}}$$

Continuamos nuestro razonamiento utilizando (8).

- Demostración de $\mu' \in Sol_{\mathcal{C}_{\mathcal{FD}.\mathcal{R}}}(\Pi_F)$: Por (3) y el lema de sustitución 10 se obtiene $\sigma'\mu' \in Sol_{\mathcal{C}_{\mathcal{FD}.\mathcal{R}}}(\Pi')$. Debido al postulado 2 podemos asumir que todas las restricciones que pertenece a Π' son $\mathcal{FD}$-específicas. Entonces, se puede aplicar el punto 4 del teorema 4 para concluir $\mid \sigma'\mu' \mid_{\mathcal{FD}} \in Sol_{\mathcal{FD}}(\Pi')$. Usando (8) se obtiene $\mid \sigma'\mu' \mid_{\mathcal{FD}} \in Sol_{\mathcal{FD}}(\Pi' \,\square\, \sigma')$, que implica $\mid \sigma'\mu' \mid_{\mathcal{FD}} \in Sol_{\mathcal{FD}}(\exists\overline{Y'}(\Pi' \,\square\, \sigma'))$. Debido a la propiedad de la corrección de resolutor de $\mathcal{FD}$ (ver definición 4 y postulado 2) se obtiene $\mid \sigma'\mu' \mid_{\mathcal{FD}} \in Sol_{\mathcal{FD}}(\Pi_F)$. Aplicando de nuevo el punto 4 del teorema 4, se obtiene $\sigma'\mu' \in Sol_{\mathcal{C}_{\mathcal{FD}.\mathcal{R}}}(\Pi_F)$. Como $\Pi_F \,\square\, \sigma_F$ es un almacén, $\Pi_F = \Pi_F\sigma_F$ y por consiguiente $\sigma'\mu' \in Sol_{\mathcal{C}_{\mathcal{FD}.\mathcal{R}}}(\Pi_F\sigma_F)$. Entonces, el lema de sustitución 10 produce $\sigma_F\sigma'\mu' \in Sol_{\mathcal{C}_{\mathcal{FD}.\mathcal{R}}}(\Pi_F)$, que es el mismo que $\mu' \in Sol_{\mathcal{C}_{\mathcal{FD}.\mathcal{R}}}(\Pi_F)$ por (5).
- Demostración de $\mu' \in Sol(\sigma_F)$: $\mu' = \sigma_F\mu'$ se deduce de la siguiente cadena de igualdades, que se basa en (5) la idempotencia de σ_F:

$$\mu' = \sigma_F\sigma'\mu' = \sigma_F\sigma_F\sigma'\mu' = \sigma_F\mu'$$

2. **Completitud local limitada:**

En este punto se asume que la regla **FDS** se puede aplicar a G de una manera segura, es decir, que la invocación al resolutor $solve^{\mathcal{FD}}(\Pi_F)$ satisface la propiedad de completitud para resolutores establecida en la definición 4. Sea $\mu \in WTSol_{\mathcal{P}}(G)$. Entonces existe algún $\mu' =_{\backslash\overline{U}} \mu$ tal que $\mu' \in WTSol_{\mathcal{P}}(P \,\square\, C \,\square\, M \,\square\, H \,\square\, F \,\square\, R)$. Por consiguiente, asumimos:

(9) $(P \,\square\, C)\mu'$ está bien tipada y $\mathcal{M} : \mathcal{P} \vdash_{CRWL(\mathcal{C}_{\mathcal{FD}.\mathcal{R}})} (P \,\square\, C)\mu'$ para algún testigo $\mathcal{M}$

$$(10)\ \mu' \in WTSol_{\mathcal{C}_{\mathcal{FD}.\mathcal{R}}}(M)$$

$$(11)\ \mu' \in WTSol_{\mathcal{C}_{\mathcal{FD}.\mathcal{R}}}(H)$$

$$(12)\ \mu' \in WTSol_{\mathcal{C}_{\mathcal{FD}.\mathcal{R}}}(F)$$

$$(13)\ \mu' \in WTSol_{\mathcal{C}_{\mathcal{FD}.\mathcal{R}}}(R)$$

En particular, (12) implica $\mu' \in WTSol_{\mathcal{C}_{\mathcal{FD}.\mathcal{R}}}(\Pi_F)$. Por los lemas 1 y 2 y el postulado 2 se puede asumir que Π_F es $\mathcal{FD}$-específica y aplicando el punto 4 del teorema

4 se concluye $\mid \mu' \mid_{\mathcal{FD}} \in WTSol_{\mathcal{FD}}(\Pi_F)$. Por completitud de la invocación al resolutor $solve^{\mathcal{FD}}(\Pi_F)$, donde $solve^{\mathcal{FD}}$ es la secuenciación de dos resolutores: $solve^{\mathcal{FD}^{\mathcal{T}}} \diamond solve^{\mathcal{FD}^{N}}$, existe alguna alternativa $\exists\overline{Y'}(\Pi' \square \sigma')$ calculada por el resolutor (es decir, tal que $\Pi_F \vdash_{solve^{\mathcal{FD}}} \exists\overline{Y'}(\Pi' \square \sigma')$) que verifica

$$(14) \quad \mid \mu' \mid_{\mathcal{FD}} \in WTSol_{\mathcal{FD}}(\exists\overline{Y'}(\Pi' \square \sigma'))$$

Entonces $G' = \exists\overline{Y'}, \overline{U}.(P \square C \square M \square H \square (\Pi' \square \sigma_F) \square R)@_{\mathcal{FD}}\sigma'$ es uno de los objetivos G'_j tal que $G \Vdash_{\mathbf{FDS},\gamma,\mathcal{P}} G'_j$. En el resto de la demostración vamos a mostrar que $\mu \in WTSol_{\mathcal{P}}(G')$ mediante $\mu'' =_{\setminus\overline{Y'},\overline{U}} \mu$ tal que

$$(\dagger)\, \mu'' \in Sol_{\mathcal{P}}((P \square C \square M \square H \square (\Pi' \square \sigma_F) \square R)@_{\mathcal{FD}}\sigma').$$

Por (14) existe $\hat{\hat{\mu}} \in Val_{\mathcal{FD}}$ tal que

$$(15) \quad \mid \mu' \mid_{\mathcal{FD}} =_{\setminus\overline{Y'}} \hat{\hat{\mu}} \in WTSol_{\mathcal{FD}}(\Pi' \square \sigma')$$

Sea $\mu'' \in Val_{\mathcal{C}_{\mathcal{FD}.\mathcal{R}}}$ unívocamente definido por las condiciones $\mu'' =_{\overline{Y'}} \hat{\hat{\mu}}$ y $\mu'' =_{\setminus\overline{Y'}} \mu'$. Como $\mu =_{\setminus\overline{U}} \mu'$, resulta que $\mu'' =_{\setminus\overline{Y'},\overline{U}} \mu$. Es más, $\mid \mu'' \mid_{\mathcal{FD}} = \hat{\hat{\mu}}$, porque para cualquier variable $X \in \mathcal{V}ar$ hay dos casos posibles: o bien $X \in \overline{Y'}$ y entonces $\mid \mu'' \mid_{\mathcal{FD}} (X) = \mid \hat{\hat{\mu}} \mid_{\mathcal{FD}} (X) = \hat{\hat{\mu}}(X)$, ya que $\hat{\hat{\mu}} \in Val_{\mathcal{FD}}$; o bien $X \notin \overline{Y'}$ y entonces $\mid \mu'' \mid_{\mathcal{FD}} (X) = \mid \mu' \mid_{\mathcal{FD}} (X) = \hat{\hat{\mu}}(X)$, ya que $\mid \mu' \mid_{\mathcal{FD}} =_{\setminus\overline{Y'}} \hat{\hat{\mu}}$. Por (15) y $\mid \mu'' \mid_{\mathcal{FD}} = \hat{\hat{\mu}}$ se obtiene $\mu'' \in WTSol_{\mathcal{FD}}(\Pi')$ aplicando el punto 4 del teorema 4.

Ahora afirmamos:

$$(16)\, \mu'' \in WTSol_{\mathcal{FD}}(\Pi' \square \sigma')$$

Para justificar esta afirmación es suficiente para probar $\mu'' \in Sol(\sigma')$, es decir, $\sigma'\mu'' = \mu''$. Para probar esto, supongamos cualquier $X \in vdom(\sigma')$. Por el postulado 2, hay dos casos posibles:

(a) $\sigma'(X)$ es un valor entero n. Por (15) se sabe que $\hat{\hat{\mu}} \in Sol(\sigma')$ y por lo tanto $\hat{\hat{\mu}}(X) = n$. Como $\mid \mu'' \mid_{\mathcal{FD}} = \hat{\hat{\mu}}$, resulta que $\mu''(X) = n$, y entonces $X\sigma'\mu'' = n = X\mu''$.

(b) $X \in var(\Pi_F)$ y $\sigma'(X)$ es una variable $X' \in var(\Pi_F)$. Entonces:

$$\begin{array}{lll}
 & X\sigma'\mu'' = X'\mu'' & \\
= & X'\mu' & \text{(utilizando } \mu'' =_{\setminus\overline{Y'}} \mu' \text{ y } X' \notin \overline{Y'}) \\
= & \mid X'\mu' \mid_{\mathcal{FD}} & \text{(utilizando el hecho de que } \Pi_F \text{ es } \mathcal{FD}\text{-específica y (12))} \\
= & X' \mid \mu' \mid_{\mathcal{FD}} & \text{(definición 10)} \\
= & X'\hat{\hat{\mu}} & \text{(utilizando (15) y } X' \notin \overline{Y'}) \\
= & X\sigma'\hat{\hat{\mu}} = X\hat{\hat{\mu}} & \text{(utilizando (15))} \\
= & X \mid \mu' \mid_{\mathcal{FD}} & \text{(utilizando (15) y } X \notin \overline{Y'}) \\
= & \mid X\mu' \mid_{\mathcal{FD}} = X\mu' & \text{(utilizando el hecho de que } \Pi_F \text{ es } \mathcal{FD}\text{-específica y (12))} \\
= & X\mu'' & \text{(utilizando } \mu'' =_{\setminus\overline{Y'}} \mu' \text{ y } X \notin \overline{Y'})
\end{array}$$

Ahora estamos en condiciones de probar (†), terminando con ello la demostración:

- Demostración de $\mu'' \in WTSol_{\mathcal{P}}(P \,\square\, C)\sigma'$: Por el lema de sustitución 10, esto es equivalente a $\sigma'\mu'' \in WTSol_{\mathcal{P}}(P \,\square\, C)$. Por (16), $\sigma'\mu'' = \mu''$. Como $\mu' =_{\setminus \overline{Y'}} \mu''$ y las variables $\overline{Y'}$ no están en $P \,\square\, C$, $\mu'' \in WTSol_{\mathcal{P}}(P \,\square\, C)$ es equivalente a $\mu' \in WTSol_{\mathcal{P}}(P \,\square\, C)$, que está garantizado por el mismo testigo $\mathcal{M}$ dado por (9).
- Demostración de $\mu'' \in WTSol_{\mathcal{C}_{\mathcal{FD}.\mathcal{R}}}(S \star \sigma')$, S siendo cualquiera de los almacenes M, H, R: De acuerdo con la elección de S se puede utilizar (10), (11) o bien (13) para concluir

$$(17)\, \mu' \in WTSol_{\mathcal{C}_{\mathcal{FD}.\mathcal{R}}}(\Pi_S) \;\text{ y }\; (18)\, \mu' \in Sol(\sigma_S) \text{ es decir, } \sigma_S\mu' = \mu'$$

  - Demostración de $\mu'' \in WTSol_{\mathcal{C}_{\mathcal{FD}.\mathcal{R}}}(\Pi_S\sigma')$: Como $\mu'' =_{\setminus \overline{Y'}} \mu'$ y las variables $\overline{Y'}$ no están en Π_S, (17) implica $\mu'' \in WTSol_{\mathcal{C}_{\mathcal{FD}.\mathcal{R}}}(\Pi_S)$, que es equivalente a $\sigma'\mu'' \in WTSol_{\mathcal{C}_{\mathcal{FD}.\mathcal{R}}}(\Pi_S)$ por (16). Entonces, $\mu'' \in WTSol_{\mathcal{C}_{\mathcal{FD}.\mathcal{R}}}(\Pi_S\sigma')$ se deduce del lema de sustitución 10.
  - Demostración de $\mu'' \in WTSol(\sigma_S \star \sigma')$: Sea cualquier variable $X \in vdom(\sigma_S)$. Entonces

$$X\sigma_S\sigma'\mu'' = X\sigma_S\mu'' = X\sigma_S\mu' = X\mu' = X\mu''$$

    donde la primera igualdad se cumple por (16), la segunda igualdad se cumple porque $\mu'' =_{\setminus \overline{Y'}} \mu'$ y las variables $\overline{Y'}$ no están en $X\sigma_S$, la tercera igualdad se cumple por (18), la cuarta igualdad se cumple porque $\mu'' =_{\setminus \overline{Y'}} \mu'$ y las variables $\overline{Y'}$ no incluyen X.

- Demostración de $\mu'' \in WTSol_{\mathcal{C}_{\mathcal{FD}.\mathcal{R}}}(\Pi'\sigma' \,\square\, \sigma_F\sigma')$:
  - Demostración de $\mu'' \in WTSol_{\mathcal{C}_{\mathcal{FD}.\mathcal{R}}}(\Pi'\sigma')$: Esto es consecuencia de (16), ya que $\Pi'\sigma' = \Pi'$ (porque $\Pi' \,\square\, \sigma'$ es un almacén).
  - Demostración de $\mu'' \in Sol(\sigma_F\sigma')$: Por (16) se puede asumir $\mu'' \in Sol(\sigma')$, es decir, $\sigma'\mu'' = \mu''$. Se debe probar $\sigma_F\sigma'\mu'' = \mu''$. Sea cualquier variable $X \in vdom(\sigma_F\sigma')$. Debido a las propiedades invariantes de objetivos admisibles (véase pág. 74), hay tres posibles casos:
    (a) $X \in vdom(\sigma_F)$ y $\sigma_F(X)$ es un valor entero n. Por (12), se sabe que $\mu' \in Sol(\sigma_F)$ y por lo tanto $X\sigma_F\mu' = n = X\mu'$. Además, $X\mu'' = X\mu' = n$ porque $\mu'' =_{\setminus \overline{Y'}} \mu'$ y las variables $\overline{Y'}$ no incluyen X. Entonces se puede concluir que $X\sigma_F\sigma'\mu'' = n = X\mu''$.
    (b) $X \in vdom(\sigma_F)$ y $\sigma_F(X) = X' \in var(\Pi_F)$. Entonces:

$$\begin{aligned} & X\sigma_F\sigma'\mu'' = X'\sigma'\mu'' \\ = \;& X'\mu'' && \text{(por (16))} \\ = \;& X'\mu' && \text{(por } \mu'' =_{\setminus \overline{Y'}} \mu' \text{ y } X' \notin \overline{Y'}) \\ = \;& X\sigma_F\mu' = X\mu' && \text{(por (12))} \\ = \;& X\mu'' && \text{(por } \mu'' =_{\setminus \overline{Y'}} \mu' \text{ y } X \notin \overline{Y'}) \end{aligned}$$

    (c) $X \notin vdom(\sigma_F)$. Entonces $X\sigma_F = X$, y se puede utilizar $\mu'' \in Sol(\sigma')$ para deducir que $X\sigma_F\sigma'\mu'' = X\sigma'\mu'' = X\mu''$.

Regla **SF, Solving Failure**. La parte seleccionada γ es uno de los cuatro almacenes del objetivo, el número k de posibles transformaciones $G \Vdash_{\mathbf{RL},\gamma,\mathcal{P}} G'_j$ de G en un objetivo que no ha fallado G'_j es 0, y por lo tanto $\bigcup_{j=1}^{k} WTSol_{\mathcal{P}}(G'_j) = \emptyset$.

1. **Corrección local:** La inclusión $Sol_{\mathcal{P}}(G) \supseteq \emptyset$ es trivial.

2. **Completitud local limitada:** La inclusión $WTSol_{\mathcal{P}}(G) \subseteq \emptyset$ es equivalente a $WTSol_{\mathcal{P}}(G) = \emptyset$. Para probar esto, se supone que la aplicación de **SF** para G se ha basado en una invocación completa del resolutor $\mathcal{D}$.

   Como la invocación al resolutor ha fallado (es decir,, $\Pi_S \Vdash_{solve^{\mathcal{D}}_{\mathcal{X}}} \blacksquare$) pero se supone que es completa, sabemos que $WTSol_{\mathcal{D}}(\Pi_S) = \emptyset$. De esto podemos concluir $WTSol_{\mathcal{C}_{\mathcal{FD}.\mathcal{R}}}(\Pi_S) = \emptyset$, por el punto 4 del teorema 4 en el caso que $\mathcal{D}$ no es $\mathcal{H}$. Finalmente, $WTSol_{\mathcal{P}}(G) = \emptyset$ es consecuencia de $WTSol_{\mathcal{C}_{\mathcal{FD}.\mathcal{R}}}(\Pi_S) = \emptyset$.

A.7.2 Demostración del lema de progreso (lema 4, pág 104).

Considérese un objetivo admisible $G \equiv \exists\overline{U}.\ P \sqcap C \sqcap M \sqcap H \sqcap F \sqcap R$ para un programa $\mathcal{P}$, una solución $\mu \in WTSol_{\mathcal{P}}(G)$ bien tipada y un testigo $\mathcal{M}$ para $\mu \in Sol_{\mathcal{P}}(G)$. Se asume que ni $\mathcal{P}$ ni G tienen apariciones de variables libres de orden superior y que G no está en forma resuelta.

1. Demostremos que debe haber alguna regla **RL** aplicable a G que no es una regla de fallo, pues los pasos de transformación que producen fallo no tienen soluciones. Es decir, para pasos de transformación de la forma $G \Vdash_{\mathbf{RL},\mathcal{P}} \blacksquare$ se obtiene que $WTSol_{\mathcal{P}}(G) = \emptyset$, y se puede aplicar la condición de completitud a la regla **RL** de una forma segura.

   Entonces, demostremos que debe haber alguna regla **RL** aplicable a G que no es una regla de fallo. Como G no está en forma resuelta, se sabe que o bien $P \neq \emptyset$, o bien $C \neq \emptyset$, o bien algunas de las transformaciones mostradas en las Tablas 5.5, 4.2 y 5.6 se pueden aplicar a G. Obsérvese que por una parte la regla **CF** de la tabla 4.1 no se puede aplicar a G porque G tiene soluciones. Por otra parte, si la regla de fallo **SF** se puede aplicar a G, entonces algunas de las otras reglas de la tabla 4.2 o 5.6 sería aplicable también.

   Sea $\mathcal{PR}$ el conjunto de las reglas de transformación de la tabla 4.1 que son distintas de **CF**, **EL** y **FC**. En los puntos siguientes se analizan diferentes casos según la forma de G. En cada caso, o encontramos alguna regla **RL** que se puede aplicar a G o hacemos algún supuesto que se puede utilizar para razonar en los casos siguientes.

   En el último paso se aplica la regla **EL** si los puntos anteriores no se ha aplicado alguna regla.

   (a) Si algunas de las reglas de transformación de las tablas 5.5 y 4.2 o 5.6 se pueden aplicar a G, entonces ya se ha terminado. En los siguientes puntos se supone que este no es el caso.

(b) Si $P \neq \emptyset$ y alguna regla $\mathbf{RL} \in \mathcal{PR}$ puede ser aplicada a G, entonces ya se ha terminado. En los siguientes puntos se supone que este no es el caso.

(c) Debido a la hipótesis de que G no tiene apariciones de variables libres de orden superior, desde ahora se puede asumir que cada producción de P debe tener una de las tres formas siguientes:

i. $h\,\overline{e}_m \rightarrow X$, con $h\,\overline{e}_m$ pasiva pero no es un patrón.
ii. $f\overline{e}_n\,\overline{a}_k \rightarrow X$, con $f \in DF^n$ y $k \geq 0$.
iii. $p\,\overline{e}_n \rightarrow X$, con $p \in PF^n$.

Si este no fuera el caso, entonces P incluiría alguna producción $e \rightarrow t$ y un análisis de la forma sintáctica de $e \rightarrow t$ llevaría a la conclusión de que alguna regla $\mathbf{RL} \in \mathcal{PR}$ se podría aplicar.

(d) Si $C \neq \emptyset$ e incluye alguna restricción atómica α que no es primitiva, entonces la regla **FC** de la tabla 4.1 se puede aplicar a α, y ya estaría demostrado. En los siguientes puntos se supone que este no es el caso.

(e) Si $C \neq \emptyset$ y sólo incluye restricciones primitivas atómicas, una de ellas es π, entonces se puede aplicar a G al menos la regla **SC** de la tabla 5.2 (y tal vez también las reglas **SB** y **PP**) tomando π como la parte seleccionada y ya se habría terminado la demostración. En los siguientes puntos se supone que $C = \emptyset$.

(f) En este punto, si hay alguna variable $X \in pvar(P) \cap odvar(G)$, entonces X está en el lado derecho de alguna producción de P con una de las tres formas 1) o 2) o 3) del punto anterior c) y una de las tres reglas **IM** o **DF** o **PC** podría aplicarse, lo cual contradice las hipótesis formuladas en el punto b). Por lo tanto, a partir de ahora se puede asumir $pvar(P) \cap odvar(G) = \emptyset$.

(g) Sea $S = \Pi_S \,\square\, \sigma_S$ uno de los cuatro almacenes $\mathcal{M}_{\mathcal{FD}.\mathcal{R}}$, $\mathcal{H}$, $\mathcal{FD}$ o $\mathcal{R}$. Sea $\mathcal{D}$ el dominio correspondiente y sea $\chi = pvar(P) \cap var(\Pi_S)$. Debido a las suposiciones hechas en el punto a), S debe estar en forma resuelta con respecto a χ y la propiedad de discriminación del resolutor $solve^{\mathcal{D}}$ asegura que una de las dos condiciones siguientes se debe cumplir:

i. $\chi \cap odvar_{\mathcal{D}}(\Pi_S) \neq \emptyset$, es decir, $pvar(P) \cap var(\Pi_S) \cap odvar_{\mathcal{D}}(\Pi_S) \neq \emptyset$.
ii. $\chi \cap var_{\mathcal{D}}(\Pi_S) = \emptyset$, es decir, $pvar(P) \cap var(\Pi_S) = \emptyset$.

Como 1) contradice la suposición $pvar(P) \cap odvar(G) = \emptyset$ hecha en el punto f), entonces se cumple 2) para los cuatro almacenes. Por otro lado, las propiedades invariantes de objetivos admisibles garantizan que las variables producidas no pueden darse en las sustituciones σ_S.

(h) En este punto, debido a las suposiciones hechas en los puntos anteriores, se puede asumir $C = \emptyset$, los cuatro almacenes están en forma resuelta y no contienen variables producidas, y todas las producciones de P tienen la forma $e \rightarrow X$, donde X es una variable. Como G no está resuelto, entonces $P \neq \emptyset$.

Obsérvese que $pvar(P)$ es finito y no vacío. Además, el cierre transitivo $\gg_P^+$ de la relación de producción $\gg_P$ entre las variables producidas debe ser irreflexiva,

debido a las propiedades invariantes de objetivos admisibles. Por lo tanto, existe alguna producción $(e \to X) \in P$ tal que X es mínima con respecto a $\gg_P$.

La variable X no puede darse en e porque esto implicaría $X \gg_P X$, contradiciendo la propiedad irreflexiva de $\gg_P^+$. Para cualquier otra producción $(e' \to X') \in P$, X debe ser diferente de X' debido a las propiedades invariantes de objetivos admisibles, y X no está contenida en e' porque esto implicaría $X \gg_P X'$, contradiciendo la minimalidad de X con respecto a $\gg_P$. Además, X no está contenida en los almacenes porque estos no incluyen variables producidas.

Por lo tanto, X no aparece en el resto del objetivo y la regla **EL** se puede aplicar para eliminar la producción $e \to X$.

2. Supongamos ahora cualquier elección de una regla **RL** (distinta de una regla de fallo) y una parte γ de G, tal que **RL** se puede aplicar a γ de manera segura, es decir, que no implique ni una aplicación opaca de **DC** ni una invocación incompleta a un resolutor.

   Se tiene que demostrar la existencia de un cálculo finito $G \Vdash^+_{\mathbf{RL},\gamma,\mathcal{P}} G'$ y un testigo $\mathcal{M}' : \mu \in WTSol_{\mathcal{P}}(G')$ tal que $(G, \mathcal{M}) \rhd (G', \mathcal{M}')$. Por el punto 2 del teorema de completitud local limitada para $CCLNC(\mathcal{C}_{\mathcal{FD}.\mathcal{R}})$ (teorema 7), existe un paso $G \Vdash_{\mathbf{RL},\gamma,\mathcal{P}} G'_1$ tal que $\mathcal{M}' : \mu \in WTSol_{\mathcal{P}}(G')$ es un testigo construido como se ha esbozado en la demostración del teorema 7. Definimos el cálculo finito deseado por distinción de los casos de la siguiente manera utilizando las reglas **SB** **PP** y **SC** de la tabla 5.2.

   (a) Si **RL** no es ni la regla **SB** ni **PP**, entonces el cálculo finito es elegido como $G \Vdash_{\mathbf{RL},\gamma,\mathcal{P}} G'_1$ y G' es G'_1.

   (b) Si **RL** es **SB** y **PP** es aplicable a γ, entonces el cálculo finito es elegido como $G \Vdash_{\mathbf{SB},\gamma,\mathcal{P}} G'_1 \Vdash_{\mathbf{PP},\gamma,\mathcal{P}} G'_2 \Vdash_{\mathbf{SC},\gamma,\mathcal{P}} G'_3$ y G' es G'_3.

   (c) Si **RL** es **SB** y **PP** no es aplicable a γ, entonces el cálculo finito es elegido como $G \Vdash_{\mathbf{SB},\gamma,\mathcal{P}} G'_1 \Vdash_{\mathbf{SC},\gamma,\mathcal{P}} G'_2$ y G' es G'_2.

   (d) Si **RL** es **PP** y **SB** es aplicable a γ, entonces el cálculo finito es elegido como $G \Vdash_{\mathbf{PP},\gamma,\mathcal{P}} G'_1 \Vdash_{\mathbf{SB},\gamma,\mathcal{P}} G'_2 \Vdash_{\mathbf{SC},\gamma,\mathcal{P}} G'_3$ y G' es G'_3.

   (e) Si **RL** es **PP** y **SB** no es aplicable a γ, entonces el cálculo finito es elegido como $G \Vdash_{\mathbf{PP},\gamma,\mathcal{P}} G'_1 \Vdash_{\mathbf{SC},\gamma,\mathcal{P}} G'_2$ y G' es G'_2.

   En todos los casos, la completitud local limitada de $CCLNC(\mathcal{C}_{\mathcal{FD}.\mathcal{R}})$ permite encontrar pasos de cómputo y un testigo $\mathcal{M}' : \mu \in WTSol_{\mathcal{P}}(G')$ que cumple $(G, \mathcal{M}) \rhd (G', \mathcal{M}')$, es decir, $||(G, \mathcal{M})|| >_{lex} ||(G', \mathcal{M}')||$. Esto se justifica por la tabla A.1. Cada fila de esta tabla corresponde a una posibilidad de que la regla **RL** utilice un solo paso de cálculo finito $G \Vdash^+_{\mathbf{RL},\gamma,\mathcal{P}} G'$ del tipo a), a excepción de una fila que corresponde a $G \Vdash^+_{\mathbf{RL},\gamma,\mathcal{P}} G'$ del tipo b), c), d) o e). Cada columna $1 \leq i \leq 7$ muestra la variación de cada O_i del orden de progreso bien fundado definido en la pág. 103, de acuerdo al orden $>_i$ cuando se avanza de $||(G, \mathcal{M})||$ a $||(G', \mathcal{M}')||$ por medio del correspondiente cálculo finito. Por ejemplo, la fila de **IE** muestra que la aplicación de esta regla no incrementa O_i para $1 \leq i \leq 5$ y disminuye en O_6.

RULES	$\mathbf{O}_1$	$\mathbf{O}_2$	$\mathbf{O}_3$	$\mathbf{O}_4$	$\mathbf{O}_5$	$\mathbf{O}_6$	$\mathbf{O}_7$
DC	$\geq_{mul}$	$\geq$	$\geq$	$>$			
SP	$\geq_{mul}$	$\geq$	$\geq$	$>$			
IM	$\geq_{mul}$	$\geq$	$>$				
EL	$\geq_{mul}$	$\geq$	$\geq$	$>$			
DF	$>_{mul}$						
PC	$\geq_{mul}$	$\geq$	$\geq$	$>$			
FC	$\geq_{mul}$	$>$					
(b),(c),(d),(e)	$>_{mul}$						
IE	$\geq_{mul}$	$\geq$	$\geq$	$\geq$	$\geq$	$>$	
ID	$\geq_{mul}$	$\geq$	$\geq$	$\geq$	$\geq$	$\geq$	$>$
MS	$\geq_{mul}$	$\geq$	$\geq$	$\geq$	$>$		
HS	$\geq_{mul}$	$\geq$	$\geq$	$\geq$	$>$		
FS	$\geq_{mul}$	$\geq$	$\geq$	$\geq$	$>$		
RS	$\geq_{mul}$	$\geq$	$\geq$	$\geq$	$>$		

Tabla A.1: $\rhd$: orden de progreso bien fundado para $CCLNC(\mathcal{C}_{\mathcal{FD}.\mathcal{R}})$

Solo queda por demostrar que la información que se muestra en la tabla A.1 es correcta. Aquí nos limitamos a explicar las ideas clave.

- Para cada regla **RL**, la aplicación de **RL** no incrementa O_1, como se muestra en la primera columna de la tabla. Esto sucede porque el testigo $\mathcal{M}'$ puede construirse a partir $\mathcal{M}$ de tal manera que todos los pasos de inferencia en $\mathcal{M}'$ que tratan de funciones primitivas y definidas que corresponden a $\mathcal{M}$.
- La aplicación de cualquier regla **DF** disminuye estrictamente O_1. La razón es que el testigo $\mathcal{M}$ incluye un árbol de prueba $\mathcal{T}$ para una instancia apropiada de una producción de la forma $f\,\overline{e}_n \rightarrow t$. La raíz infiere que este árbol de prueba contribuye al tamaño restringido de $\mathcal{M}$ y desaparece en el testigo $\mathcal{M}'$ construido desde $\mathcal{M}$ como se indica en A.7.1. Por lo tanto, el tamaño restringido de $\mathcal{M}'$ disminuye en uno con respecto al tamaño restringido de $\mathcal{M}$.
- La tabla también muestra que los cálculos finitos de tipo b), c), d) o e) estrictamente disminuyen O_1. La razón es que tales cálculos finitos siempre trabajan con una restricción primitiva atómica fija π que se mueve en última instancia del *pool* de restricciones C a uno de los almacenes de G' cuando se realiza el último paso **SC**. El testigo $\mathcal{M} : \mu \in WTSol_{\mathcal{P}}(G)$ incluye un árbol de prueba para una instancia apropiada de π, mientras que el correspondiente árbol de prueba no es necesario en el testigo $\mathcal{M}'$. Por lo tanto, el tamaño restringido de $\mathcal{M}'$ disminuye

en cierta cantidad positiva.

- La aplicación de la regla **PC** decrementa O_4, porque G incluye una producción $p\,\overline{e}_n \rightarrow t$ que desaparece en G' y el resto de producciones se mantienen.
- La aplicación de la regla **FC** decrementa O_2, porque G incluye una restricción $p\,\overline{e}_n \rightarrow !t$ con $||p\,\overline{e}_n|| > 0$, que se sustituye en G' por una restricción primitiva atómica $p\,\overline{t}_n \rightarrow !\,t$ con $||p\,\overline{t}_n|| = 0$ y algunas nuevas producciones $e_i \rightarrow V_i$ cuya contribución a la medida O_2 de G' debe ser menor que $||p\,\overline{e}_n||$.
- Con respecto al resto de reglas correspondientes al estrechamiento perezoso (reglas **DC**, **SP**, **IM**, **EL** y **FC**), la demostración de que estas reglas decrementan se puede encontrar en el trabajo [LRV04].
- La aplicación de la regla **IE** decrementa O_6 y no incrementa O_i para $1 \leq i \leq 5$. En este caso, el testigo $\mathcal{M}'$ puede ser elegido como $\mathcal{M}$ y las medidas O_2, O_3, O_4 y O_5 no están afectadas por **IE**. La medida O_6 decrece en 1 cuando se aplica **IE**.
- Por razones similares, la aplicación de la regla **ID** decrece O_7 y no incrementa O_i para $1 \leq i \leq 6$.
- Sea **RL** cualquiera de las cuatro reglas de transformación de resolución de restricciones **MS**, **HS**, **FS** y **RS**. Se puede garantizar que existe el testigo $\mathcal{M}' : \mu \in WTSol_{\mathcal{P}}(G')$ sólo si la invocación resolutor ha sido completa. En este caso, $\mathcal{M}'$ puede ser elegido como el mismo testigo $\mathcal{M}$, y por lo tanto la medida O_1 no incrementa cuando se pasa de G a G'. Las medidas O_2, O_3 y O_4 no se ven afectadas por las vinculaciones creadas por las invocaciones del resolutor (ya que sustituyen variables por patrones). La medida O_5 decrece, pues para el almacén que se ha resuelto sf_S desciende de 1 a 0.

c.q.d.

A.8 Demostración del Teorema 10 (pág. 123)

Primero se demuestra que el sistema de transformación de almacenes con relación de transición $\vdash_{\mathcal{M}_{\mathcal{FD}.\mathcal{FS}}}$ definido mediante las reglas de la tabla 6.1 cumple las propiedades de la definición 6.

1. **Variables locales nuevas:** no existen reglas que introduzcan variables locales nuevas.
2. **Ramificación finita:** trivial, pues Π es un conjunto finito de restricciones, hay un conjunto finito de reglas para aplicar y no hay elecciones no deterministas.
3. **Terminación:** considérese la posibilidad de un almacén formado por $\Pi\Box\sigma$, donde Π es un conjunto finito de restricciones y σ es una sustitución idempotente. Debemos probar que después de una secuencia finita de pasos se obtiene un almacén irreducible. Es decir, que se puede obtener una derivación de la forma $\Pi\Box\sigma \vdash^*_{\mathcal{M}} \Pi_n\Box\sigma_n$ con $\Pi_n\Box\sigma_n$

irreducible. Por lo tanto, es suficiente probar que cada regla sólo se puede aplicar un número finito de veces, pues si el número de reglas es finito y se aplican un número finito de veces a un conjunto finito de restricciones entonces se llega a un almacén irreducible en un número finito de pasos.

Sea una secuencia finita de i pasos $\Pi \square \sigma \Vdash^*_{\mathcal{M}} \Pi_i \square \sigma_i$. Si $\Pi_i \square \sigma_i$ es reducible (un almacén es reducible si existe una regla de transformación de almacenes que puede ser aplicada para transformarla) entonces se puede dar alguno de los siguientes casos:

- O bien se puede ser aplicar **M1**, es decir, Π_i contiene la restricción X `#--` u' con $u' \in \mathcal{B}^{\mathcal{FS}}_{\texttt{set}}$ y $X \in Var$. En este caso, se aplica **M1** y la restricción puente es eliminada. Como Π es finito y ninguna regla introduce puentes en Π, entonces el número de veces que **M1** puede ser aplicada es finito.

  Se puede aplicar un razonamiento similar a las reglas **M2** y **M3**.
- O bien se puede aplicar **M4**, pero en este caso solo se puede aplicar una vez porque el cómputo termina.

Por lo tanto, cada regla sólo se puede aplicar un número finito de veces demostrando así que el cómputo es terminante.

4. **Corrección local:** hay que demostrar que para cualquier almacén $\Pi \square \sigma$ del dominio $\mathcal{M}_{\mathcal{FD}.\mathcal{FS}}$, el conjunto

   $\bigcup\{Sol_{\mathcal{M}_{\mathcal{FD}.\mathcal{FS}}}(\exists \overline{Y'}(\Pi' \square \sigma')) \mid \Pi \square \sigma \Vdash_{\mathcal{M}_{\mathcal{FD}.\mathcal{FS}}} \Pi' \square \sigma', \overline{Y'} = var(\Pi' \square \sigma') \setminus var(\Pi \square \sigma)\}$

   es un subconjunto de $Sol_{\mathcal{M}_{\mathcal{FD}.\mathcal{FS}}}(\Pi \square \sigma)$.

   Es decir, hay que probar que para todo almacén $\Pi' \square \sigma'$ que se obtiene aplicando alguna regla definida en la tabla 6.1 al almacén $\Pi \square \sigma$ se cumple:

   $Sol_{\mathcal{M}_{\mathcal{FD}.\mathcal{FS}}}(\Pi \square \sigma) \supseteq Sol_{\mathcal{M}_{\mathcal{FD}.\mathcal{FS}}}(\Pi' \square \sigma')$ con $\overline{Y'} = var(\Pi' \square \sigma') \setminus var(\Pi \square \sigma)\}$

   Esta condición se demuestra por una distinción de casos de las reglas definidas en la tabla 6.1. En cada caso, suponemos que los almacenes tienen exactamente la forma mostrada en la correspondiente regla de transformación en la tabla 6.1.

   - **M1**: Sea $\eta \in Sol_{\mathcal{M}_{\mathcal{FD}.\mathcal{FS}}}(\Pi\sigma_1 \ \square \ \sigma\sigma_1)$.

     Tenemos que comprobar que $\eta \in Sol_{\mathcal{M}_{\mathcal{FD}.\mathcal{FS}}}(X$ `#--` $u', \Pi \ \square \ \sigma)$ con $u' \in \mathcal{B}^{\mathcal{FS}}_{\texttt{set}}$, $X \in Var$, $u \in \mathbb{Z}^+$ tal que u `#--`$^{\mathcal{M}_{\mathcal{FD}.\mathcal{FS}}}$ u' se cumple y $\sigma_1 = \{X \mapsto u\}$.

     Usando el lema auxiliar de corrección (lema 8), mostrado anteriormente, se sabe que

     $$Sol_{\mathcal{M}_{\mathcal{FD}.\mathcal{FS}}}(\Pi\sigma_1) \cap Sol_{\mathcal{M}_{\mathcal{FD}.\mathcal{FS}}}(\sigma\sigma_1) \subseteq Sol_{\mathcal{M}_{\mathcal{FD}.\mathcal{FS}}}(\Pi) \cap Sol_{\mathcal{M}_{\mathcal{FD}.\mathcal{FS}}}(\sigma)$$

     Además, $\eta \in Sol_{\mathcal{M}_{\mathcal{FD}.\mathcal{FS}}}(X$ `#--` $u')$ porque $\eta \in Sol_{\mathcal{M}_{\mathcal{FD}.\mathcal{FS}}}(\Pi\sigma_1 \ \square \ \sigma\sigma_1)$ con $\sigma_1 = \{X \mapsto u\}$ y u `#--`$^{\mathcal{M}_{\mathcal{FD}.\mathcal{FS}}}$ u' se cumple, entonces $\eta \in Sol_{\mathcal{M}_{\mathcal{FD}.\mathcal{FS}}}(X$ `#--` $u', \Pi \ \square \ \sigma)$.

     Se puede aplicar un razonamiento similar a la regla **M2**.

- **M3**: Trivial, como u #--$^{\mathcal{M}_{\mathcal{FD}.\mathcal{FS}}}$ u', entonces $\eta \in Sol_{\mathcal{M}_{\mathcal{FD}.\mathcal{FS}}}(u$ #-- $u', \Pi \square \sigma)$ se cumple para cualquier $\eta \in Sol_{\mathcal{M}_{\mathcal{FD}.\mathcal{FS}}}(\Pi \square \sigma)$.
- **M4**: Trivial, porque $Sol_{\mathcal{M}_{\mathcal{FD}.\mathcal{FS}}}(\blacksquare) = \emptyset$ es un subconjunto de cualquier conjunto.

Con esta distinción de casos se ha probado que toda solución bien tipada del almacén $\Pi' \square \sigma'$ es solución del almacén $\Pi \square \sigma$ si se puede obtener $\Pi' \square \sigma'$ desde $\Pi \square \sigma$ aplicando alguna regla definida en la tabla 6.1.

5. **Completitud local:** para cualquier almacén $\Pi \square \sigma$ del dominio $\mathcal{M}_{\mathcal{FD}.\mathcal{FS}}$, el conjunto $WTSol_{\mathcal{M}_{\mathcal{FD}.\mathcal{FS}}}(\Pi \square \sigma)$ es un subconjunto de

   $\bigcup\{WTSol_{\mathcal{M}_{\mathcal{FD}.\mathcal{FS}}}(\exists\overline{Y'}(\Pi' \square \sigma')) \mid \Pi\square\sigma \vdash_{\mathcal{M}_{\mathcal{FD}.\mathcal{FS}}} \Pi'\square\sigma', \overline{Y'} = var(\Pi'\square\sigma')\backslash var(\Pi\square\sigma)\}$

   Esta condición también se demuestra por una distinción de casos de las reglas definidas en la tabla 6.1. En cada caso, asumimos que los almacenes tienen exactamente la forma mostrada por la correspondiente regla de transformación.

   - **M1**: sea $\eta \in WTSol_{\mathcal{M}_{\mathcal{FD}.\mathcal{FS}}}(X$ #-- $u', \Pi \square \sigma)$.

     Tenemos que comprobar que $\eta \in WTSol_{\mathcal{M}_{\mathcal{FD}.\mathcal{FS}}}(\Pi\sigma_1 \square \sigma\sigma_1)$ con $u' \in \mathcal{B}^{\mathcal{FS}}_{\mathtt{set}}$, $X \in Var$, $u \in \mathbb{Z}^+$ tal que u #--$^{\mathcal{M}_{\mathcal{FD}.\mathcal{FS}}}$ u' y $\sigma_1 = \{X \mapsto u\}$.

     Puesto que $\eta \in WTSol_{\mathcal{M}_{\mathcal{FD}.\mathcal{FS}}}(X$ #-- $u', \Pi \square \sigma)$, se cumple $\eta \in WTSol_{\mathcal{M}_{\mathcal{FD}.\mathcal{FS}}}(X$ #-- $u')$ y $\eta \in WTSol_{\mathcal{M}_{\mathcal{FD}.\mathcal{FS}}}(\Pi \square \sigma)$. Como u #--$^{\mathcal{M}_{\mathcal{FD}.\mathcal{FS}}}$ u' entonces $\eta(X) = u$.

     Usando el lema auxiliar de completitud (lema 9) con $\overline{Y'} = \emptyset$, $\eta' = \eta$ y $\sigma_1 = \{X \mapsto u\}$, $\sigma_1\eta' = \eta'$ se cumple $\sigma\eta' = \eta'$ y $\eta' \in WTSol_{\mathcal{M}_{\mathcal{FD}.\mathcal{FS}}}(\Pi\sigma_1) \cap WTSol_{\mathcal{M}_{\mathcal{FD}.\mathcal{FS}}}(\sigma\sigma_1)$.

     Consecuentemente, $\eta \in WTSol_{\mathcal{M}_{\mathcal{FD}.\mathcal{FS}}}(\Pi\sigma_1 \square \sigma\sigma_1)$.

     Se puede aplicar un razonamiento similar a la regla **M2**.

   - **M3**: trivial, sea $\eta \in WTSol_{\mathcal{M}_{\mathcal{FD}.\mathcal{FS}}}(u$ #-- $u', \Pi \square \sigma)$ entonces $\eta \in WTSol_{\mathcal{M}_{\mathcal{FD}.\mathcal{FS}}}(\Pi \square \sigma)$.
   - **M4**: trivial, puesto que u #--$^{\mathcal{M}_{\mathcal{FD}.\mathcal{FS}}}$ u' no se cumple, entonces no hay soluciones.

   Con esta distinción de casos se ha probado que toda solución bien tipada del almacén $\Pi\square\sigma$ es solución del almacén $\Pi'\square\sigma'$ si se puede obtener $\Pi'\square\sigma'$ desde $\Pi\square\sigma$ aplicando alguna regla definida en la tabla 6.1.

Aplicando el lema 1 a las propiedades que se acaban de demostrar se concluye que $solve^{\mathcal{M}_{\mathcal{FD}.\mathcal{FS}}}$ satisface todos los requerimientos de los resolutores de la definición 4.

c.q.d.

A.9 Propiedades del cálculo $CCLNC(\mathcal{C}_{\mathcal{FD}.\mathcal{FS}})$

Las demostraciones que se incluyen en este apéndice corresponden a los resultados de corrección y completitud limitada de la sección 6.3. En concreto se demuestra la corrección local y completitud limitada local y el lema de progreso que se enuncia para poder demostrar la completitud global. Estas demostraciones siguen la misma metodología que las demostraciones del apéndice A.7.

A.9.1 Demostración del teorema 11 (pág. 130)

Este teorema garantiza la corrección local y completitud limitada local del cálculo para un paso de transformación de un objetivo dado. Para demostrar este teorema hay que comprobar lo siguiente:

Sea $\mathcal{P}$ un programa $CFLP(\mathcal{C}_{\mathcal{FD}.\mathcal{FS}})$ y G un objetivo admisible que no está en forma resuelta. Se elige una regla **RL** de las tablas 4.1, 4.2, 6.2, 6.5 y 6.6 que sea aplicable a G y se selecciona una parte γ de G sobre la que se aplica la regla **RL**. Entonces la afirmación de que existe una cantidad finita de transformaciones $G \vdash_{\mathbf{RL},\gamma,\mathcal{P}} G'_j$ $(1 \leq j \leq k)$ es trivial explorando todas las reglas. Además hay que demostrar las condiciones de corrección local y completitud limitada local para todas las reglas. Las tablas 4.1 y 4.2 ya se trataron en el apéndice A.7. Las demostraciones correspondientes a las tablas 6.2 y 6.6 son semejantes a las análogas (5.2 y 5.6) del cálculo $CCLNC(\mathcal{C}_{\mathcal{FD}.\mathcal{R}})$. Aunque es muy similar a la demostración de las reglas de la tabla 5.5 del teorema 7, a continuación se demuestra la corrección local y completitud limitada local de las reglas de la tabla 6.5.

Regla **IE, Infer Equalities**. La parte seleccionada γ es un par de puentes de la forma I_1 `#--` S, I_2 `#--` S y $k = 1$. Utilizaremos G' en vez de G'_1.

1. **Corrección local:** Sea $\mu \in Sol_{\mathcal{P}}(G')$. Entonces existe $\mu' =_{\backslash \overline{U}} \mu$ tal que $\mu' \in Sol_{\mathcal{P}}(P \square C \square I_1$ `#--` $S\ M \square H \square I_1$ `==` $I_2, F \square S)$. Esto implica que existe un testigo $\mathcal{M}'$ tal que $\mathcal{M}' : \mathcal{P} \vdash_{CRWL(\mathcal{C}_{\mathcal{FD}.\mathcal{FS}})} (P \square C)\mu'$ y $\mu' \in Sol_{\mathcal{C}_{\mathcal{FD}.\mathcal{FS}}}(I_1$ `#--` $S, M \square H \square I_1$ `==` $I_2, F \square S)$. Por tanto, $\mu' \in Sol_{\mathcal{C}_{\mathcal{FD}.\mathcal{FS}}}(\ I_1$ `#--` S, I_2 `#--` $S, M \square H \square F \square S)$, que junto con el testigo $\mathcal{M}'$ se obtiene $\mu' \in Sol_{\mathcal{P}}(P \square C \square\ I_1$ `#--` S, I_2 `#--` $S, M \square H \square F \square S)$, y por lo tanto $\mu \in Sol_{\mathcal{P}}(G)$.

2. **Completitud local limitada:** Sea $\mu \in WTSol_{\mathcal{P}}(G)$. Entonces existe $\mu' =_{\backslash \overline{U}} \mu$ tal que $\mu' \in WTSol_{\mathcal{P}}(P \square C \square I_1$ `#--` S, I_2 `#--` $S, M \square H \square F \square S)$. Esto implica que existe un testigo $\mathcal{M}'$ tal que $\mathcal{M} : \mathcal{P} \vdash_{CRWL(\mathcal{C}_{\mathcal{FD}.\mathcal{FS}})} (P \square C)\mu'$ y $\mu' \in WTSol_{\mathcal{C}_{\mathcal{FD}.\mathcal{FS}}}(\ I_1$ `#--` S, I_2 `#--` $S, M \square H \square F \square S)$. El segundo hecho implica $\mu' \in WTSol_{\mathcal{C}_{\mathcal{FD}.\mathcal{FS}}}(\ I_1$ `#--` $S, M \square H \square I_1$ `==` $I_2, F \square S)$. Entonces, $\mu' \in WTSol_{\mathcal{P}}(P \square C \square\ I_1$ `#--` $S, M \square H \square I_1$ `==` $I_2, F \square R)$ se cumple gracias al mismo testigo $\mathcal{M}$, y por lo tanto $\mu \in Sol_{\mathcal{P}}(G')$.

IF Infer Failure

La parte seleccionada γ es un par de puentes de la forma I_1 `#--` S, I_2 `#--` S y las restricciones I_1 `/=` I_2 y S_1 `==` S_2. Además, $k = 1$ y $Sol_{\mathcal{P}}(G') = \emptyset$.

1. **Corrección local:** la inclusión $\emptyset \subseteq Sol_{\mathcal{P}}(G)$ se cumple trivialmente.

2. **Completitud local limitada:** Sea $\mu \in WTSol_{\mathcal{P}}(G)$. Entonces existe $\mu' =_{\setminus \overline{U}} \mu$ tal que $\mu' \in WTSol_{\mathcal{P}}(\exists \overline{U}.P \,\square\, C \,\square\, I_1$ `#--` S_1,I_2 `#--` S_2, $M \,\square\, H \,\square\, I_1$`/=`$I_2$, $F \,\square\, S_1$`==`S_2, $S)$.

   Esto implica $\mathcal{M} : \mathcal{P} \vdash_{CRWL(\mathcal{C}_{\mathcal{FD}.\mathcal{FS}})} (P \,\square\, C)\mu'$ y además $\mu' \in WTSol_{\mathcal{C}_{\mathcal{FD}.\mathcal{FS}}}(I_1$ `#--` S_1,I_2 `#--` S_2,$M \,\square\, H \,\square\, I_1$`/=`$I_2$, $F \,\square\, S_1$`==`S_2, $S)$. Pero no existe ninguna μ' que cumpla la segunda condición así que $WTSol_{\mathcal{P}}(G) = \emptyset$

c.q.d.

A.9.2 Demostración del lema de progreso (lema 5, pág. 132)

Sea un objetivo G para un programa $\mathcal{P}$ y un testigo $\mathcal{M} : \mu \in WTSol_{\mathcal{P}}(G)$. Se asume que ni $\mathcal{P}$ ni G tienen variables libres de orden superior y además G no está en forma resuelta. Entonces hay que demostrar

1. Existe alguna regla `RL` aplicable a G que no es una regla de fallo. Este punto no se demuestra pues es semejante a la correspondiente demostración del apéndice A.7.2.

2. Para cualquier regla **RL** que no sea una regla de fallo y una parte γ de G, tal que **RL** se aplica a γ de una manera segura, existe un número finito de computaciones $G \Vdash^{+}_{\mathbf{RL},\gamma,\mathcal{P}} G'$ tal que:

   - $\mu \in WTSol_{\mathcal{P}}(G')$.
   - Existe un testigo $\mathcal{M}' : \mu \in WTSol_{\mathcal{P}}(G')$ que verifica $(G, \mathcal{M}) \blacktriangleright (G', \mathcal{M}')$.

   Para demostrar este segundo punto se toma una regla **RL** de las tablas 4.1, 4.2, 6.2, 6.5 y 6.6, distinta de una regla de fallo, tal que se puede aplicar **RL** a una parte γ de G de manera segura, es decir, que no implique ni una aplicación opaca de **DC** ni una invocación incompleta a un resolutor. Para este tipo de reglas se tiene que demostrar que existe un cálculo finito $G \Vdash^{+}_{\mathbf{RL},\gamma,\mathcal{P}} G'$ y un testigo $\mathcal{M}' : \mu \in WTSol_{\mathcal{P}}(G')$ tal que $(G, \mathcal{M}) \blacktriangleright (G', \mathcal{M}')$.

   Por el teorema 11, que enuncia la completitud local limitada, existe un paso de computación $G \Vdash_{\mathbf{RL},\gamma,\mathcal{P}} G'_1$ con un testigo adecuado $\mathcal{M}' : \mu \in WTSol_{\mathcal{P}}(G')$. El cálculo finito deseado se define por distinción de los casos de la misma manera que en la demostración A.7.2, pero hay que tener en cuenta que en este caso las reglas **SB**, **PP** y **SC** corresponden a la tabla 6.2, que no han cambiado de nombre por no saturar la notación pero la creación y uso de los puentes varía de un cálculo a otro. El cálculo finito deseado se define como:

   (a) Si **RL** no es ni la regla **SB** ni **PP**, entonces el cálculo finito es elegido como $G \Vdash_{\mathbf{RL},\gamma,\mathcal{P}} G'_1$ y G' es G'_1.

   (b) Si **RL** es **SB** y **PP** es aplicable a γ, entonces el cálculo finito es elegido como $G \Vdash_{\mathbf{SB},\gamma,\mathcal{P}} G'_1 \Vdash_{\mathbf{PP},\gamma,\mathcal{P}} G'_2 \Vdash_{\mathbf{SC},\gamma,\mathcal{P}} G'_3$ y G' es G'_3.

(c) Si **RL** es **SB** y **PP** no es aplicable a γ, entonces el cálculo finito es elegido como $G \Vdash_{\mathbf{SB},\gamma,\mathcal{P}} G'_1 \Vdash_{\mathbf{SC},\gamma,\mathcal{P}} G'_2$ y G' es G'_2.

(d) Si **RL** es **PP** y **SB** es aplicable a γ, entonces el cálculo finito es elegido como $G \Vdash_{\mathbf{PP},\gamma,\mathcal{P}} G'_1 \Vdash_{\mathbf{SB},\gamma,\mathcal{P}} G'_2 \Vdash_{\mathbf{SC},\gamma,\mathcal{P}} G'_3$ y G' es G'_3.

(e) Si **RL** es **PP** y **SB** no es aplicable a γ, entonces el cálculo finito es elegido como $G \Vdash_{\mathbf{PP},\gamma,\mathcal{P}} G'_1 \Vdash_{\mathbf{SC},\gamma,\mathcal{P}} G'_2$ y G' es G'_2.

En todos los casos, la completitud local limitada de $CCLNC(\mathcal{C}_{\mathcal{FD}.\mathcal{FS}})$ permite encontrar pasos de cómputo y un testigo $\mathcal{M}' : \mu \in WTSol_{\mathcal{P}}(G')$ con $(G, \mathcal{M}) \blacktriangleright (G', \mathcal{M}')$, es decir, $||(G, \mathcal{M})|| >_{lex} ||(G', \mathcal{M}')||$. Esto se justifica mediante la tabla A.2. Cada fila de esta tabla corresponde a un paso de la forma $G \Vdash_{\mathbf{RL},\gamma,\mathcal{P}} G'_1$ realizado con la regla **RL**, excepto la fila marcada como (b),(c),(d),(e). Cada columna muestra la variación en O_i de cada regla para el orden de progreso definido en la página 131. En esta tabla las filas que corresponden al estrechamiento perezoso (**DC**, **SP**, **IM**, **EL**, **DF**, **PC** y **FC**) no varían con respecto a la tabla A.1. Igualmente no varía la fila correspondiente a las aplicaciones (b),(c),(d),(e) de las reglas **SB**, **PP** y **SC**. Con respecto a la regla **IE** que infiere igualdades a partir de puentes, lo único que varía es la forma de los puentes, pero en ambos casos el número de puentes desciende. En la tabla A.1 se tiene la regla **ID** que no existe en el cálculo $CCLNC(\mathcal{C}_{\mathcal{FD}.\mathcal{FS}})$. Sin embargo en este cálculo se tiene la regla **IF** que no se trata por ser una regla de fallo. Finalmente, con respecto a la invocación de los resolutores, las filas de las reglas **MS**, **HS** y **FS** no varían y la fila correspondiente a **SetS** es construida de forma semejante.

Reglas	$\mathbf{O}_1$	$\mathbf{O}_2$	$\mathbf{O}_3$	$\mathbf{O}_4$	$\mathbf{O}_5$	$\mathbf{O}_6$
DC	$\geq_{mul}$	$\geq$	$\geq$	$>$		
SP	$\geq_{mul}$	$\geq$	$\geq$	$>$		
IM	$\geq_{mul}$	$\geq$	$>$			
EL	$\geq_{mul}$	$\geq$	$\geq$	$>$		
DF	$>_{mul}$					
PC	$\geq_{mul}$	$\geq$	$\geq$	$>$		
FC	$\geq_{mul}$	$>$				
(b),(c),(d),(e)	$>_{mul}$					
IE	$\geq_{mul}$	$\geq$	$\geq$	$\geq$	$\geq$	$>$
MS	$\geq_{mul}$	$\geq$	$\geq$	$\geq$	$>$	
HS	$\geq_{mul}$	$\geq$	$\geq$	$\geq$	$>$	
FS	$\geq_{mul}$	$\geq$	$\geq$	$\geq$	$>$	
SetsS	$\geq_{mul}$	$\geq$	$\geq$	$\geq$	$>$	

Tabla A.2: $\blacktriangleright$: orden de progreso bien fundado para las reglas de $\mathcal{C}_{\mathcal{FD}.\mathcal{FS}}$

Solo queda por demostrar que la información que se muestra en la tabla A.2 es correcta para las reglas que difieren con respecto a la tabla A.1. La demostración formal se hace sobre la base de una construcción detallada de los testigos $\mathcal{M}' : \mu \in WTSol_{\mathcal{P}}(G')$. A continuación se explican únicamente las ideas clave.

La aplicación de la regla **IE** de la tabla 6.5 decrementa O_6 pues representa el número de puentes que aparecen en el almacén del dominio mediador de G. Se toma el testigo $\mathcal{M}'$ igual que el testigo $\mathcal{M}$. La aplicación de la regla **IE** decrementa O_6 y no incrementa O_1 pues no trata ninguna función primitiva ni definida, y tampoco incrementa O_i con $2 \leq i \leq 5$ pues igualmente estas medidas no están afectadas por **IE**. Por lo tanto se verifica $(G, \mathcal{M}) \blacktriangleright (G', \mathcal{M}')$.

La aplicación de la regla **SetS** de la tabla 6.6 decrementa la medida O_5 porque es definida como la suma $sf_M + sf_H + sf_{FD} + sf_{FS}$, donde sf_{FS} es el valor 1 si la regla **SetS** puede ser aplicada y 0 en otro caso. Se puede garantizar que existe el testigo $\mathcal{M}' : \mu \in WTSol_{\mathcal{P}}(G')$ sólo si la invocación resolutor de $\mathcal{FS}$ ha sido completa. En este caso, se toma $\mathcal{M}'$ como el mismo testigo $\mathcal{M}$, y por lo tanto la medida O_1 no incrementa cuando se pasa de G a G'. Las medidas O_2, O_3 y O_4 no se ven afectadas por las vinculaciones creadas por las invocaciones del resolutor (ya que sustituyen variables por patrones).

c.q.d.

Appendix B

English Summary

B.1 Outline

This thesis presents a computational model and its implementation for the cooperation of constraint domains in $\mathcal{TOY}$ [TOY12, ACE+07], a multiparadigm programming language and system designed to integrate the main declarative programming styles and their combination. This summary in English follows the works [EFH+09, ECS12], co-authored by the Ph.D. candidate.

$\mathcal{TOY}$ relies on the Constraint Functional Logic Programming scheme $CFLP(\mathcal{D})$ [LRV07] and the goal solving calculus $CLNC(\mathcal{D})$ (Constrained Lazy Narrowing Calculus over a parametrically given constraint domain $\mathcal{D}$) [LRV04]. This scheme must be instantiated to a concrete constraint domain $\mathcal{D}$ which provides specific data values, constraints based on specific primitive operations, and a constraint solver. Therefore, there are different instances $CFLP(\mathcal{D})$ of the $CFLP$ scheme for various choices of $\mathcal{D}$, whose instances provide a declarative framework for any chosen domain $\mathcal{D}$. The computational model formalized in this thesis for the cooperation of constraint domains is established over a special kind of hybrid domain, namely the *coordination domain*:

$$\mathcal{C} = \mathcal{M} \oplus \mathcal{D}_1 \oplus \cdots \oplus \mathcal{D}_n$$

This coordination domain is built as the combination of pure domains $\mathcal{D}_1, \ldots, \mathcal{D}_n$ and a *Mediatorial* domain $\mathcal{M}$ that is used for communicating those domains. In particular, this thesis presents two specific coordination domains tailored to the cooperation of the different pure domains:

$$\mathcal{C}_{\mathcal{FD}.\mathcal{R}} = \mathcal{M}_{\mathcal{FD}.\mathcal{R}} \oplus \mathcal{H} \oplus \mathcal{R} \oplus \mathcal{FD} \quad \text{and} \quad \mathcal{C}_{\mathcal{FD}.\mathcal{FS}} = \mathcal{M}_{\mathcal{FD}.\mathcal{FS}} \oplus \mathcal{H} \oplus \mathcal{FD} \oplus \mathcal{FS}$$

The *pure* domains $\mathcal{H}$, $\mathcal{R}$, $\mathcal{FD}$ and $\mathcal{FS}$ supply equality and disequality constraints over symbolic terms; arithmetic constraints over real numbers; finite domain constraints over integers; and finite domain constraints over sets of integers, respectively.

Constraints in the mediatorial domain are called *bridges* and are a special kind of hybrid constraints which allow to exchange information between pure domains. Bridge constraints

can be used to *project* constraints to other domains. Projected constraints are submitted to their corresponding store, with the aim of improving the performance of the corresponding solver. In this way, projections behave as an important cooperation mechanism, enabling certain solvers to profit from (the projected forms) of constraints originally intended for other solvers. For example, the cooperation between the domains $\mathcal{R}$ and $\mathcal{FD}$ is established by the bridge defined as `X #== Y`. In this bridge constraint, the real variable `Y :: real` takes an integral real value equivalent to the integer variable `X :: int`, and is used to *project* constraints involving the variable `X` into constraints involving the variable `Y`, or vice versa. For instance, the $\mathcal{R}$ constraint `RX <= 3.4` (based on the inequality primitive `<=` – 'less or equal' – for the type `real`) can be projected into the $\mathcal{FD}$ constraint `X #<= 3` (based on the inequality primitive `#<=` – 'less or equal' – for the type `int`) in case that the bridge `X #== RX` is available.

Building upon the coordination domain, this thesis presents the formal goal solving calculus $CCLNC(\mathcal{C})$ (Cooperative Constraint Lazy Narrowing Calculus over $\mathcal{C}$) which is sound and complete with respect to the instance $CFLP(\mathcal{C})$ of the generic $CFLP$ scheme. $CCLNC(\mathcal{C})$ embodies computation rules for creating bridges, invoking constraint solvers, performing constraint projections, and new approaches to detect in advance the failure of goals that involve constraints in the domains $\mathcal{FD}$ and $\mathcal{FS}$, as well as other more *ad hoc* operations for communication among different constraint stores. Moreover, $CCLNC(\mathcal{C})$ uses lazy narrowing (a combination of lazy evaluation and unification) for processing calls to program defined functions, ensuring that function calls are evaluated only as far as demanded by the resolution of the constraints involved in the current goal. The computational model is sound and complete (with limitations) with respect to the declarative semantics provided by the $CFLP$ scheme. Soundness condition requires that no new spurious solution is introduced (any computed solution is correct), while the completeness condition requires that no well-typed solution is lost (any well-typed correct solution can be computed). Completeness can be ensured if the following does not occur:

- Higher-order free variables in goals and program rules.
- Incomplete solver invocations.
- 'Opaque decompositions' that may produce ill-typed goals. A type which can be written as $\overline{\tau}_m \rightarrow \tau$ is called *opaque* iff $tvar(\overline{\tau}_m) \nsubseteq tvar(\tau)$ (the set of type variables occurring in τ is written $tvar(\tau)$).

A prototype of this communication has been developed on top of the $CFLP$ system $\mathcal{TOY}$, using the SICStus and ECL^{i}PS^e libraries for the finite domain constraints, arithmetic real constraints and finite sets of integers constraints as black-box solvers. The prototype has been tested with a set of benchmarks.

This summary has been made, for the most part, from the works [EFH$^+$09] and [ECS12] and the thesis thoroughly revises, expands and elaborates previous related publications in many respects. In fact, [EFH$^+$07b] was a very preliminary work which focused on presenting bridges and providing evidence for their usefulness. Building upon these ideas, [EFH$^+$07a]

introduced coordination domains and a cooperative goal solving calculus over an arbitrary coordination domain, proving local soundness and completeness results, while [EFH+08] further elaborated the cooperative goal solving calculus, providing stronger soundness and completeness results and experimental data on an implementation tailored to the cooperation of the domains $\mathcal{H}$, $\mathcal{FD}$ and $\mathcal{R}$. Significant novelties in the article [EFH+09] include: technical improvements in the formalization of domains; a new notion of solver taking care of critical variables and well-typed solutions; a new notion of domain-specific constraint to clarify the behaviour of coordination domains; various elaborations in the cooperative goal solving transformations needed to deal with critical variables and domain-specific constraints; a more detailed presentation of the implementation results previously reported in [EFS07, EFS08, EFH+08]; and quite extensive comparisons to other related approaches. The work [ECS12] treats the cooperation of the domains $\mathcal{H}$, $\mathcal{FD}$ y $\mathcal{FS}$. In this paper, besides presenting the theoretical basis for this new cooperation, also presents the corresponding adaptation of the calculus, as well as the theoretical results of soundness and limited completeness. A first approach to this new cooperation was outlined in the works [EFS09a, EFS09b], but the work [ECS12] developed the initial idea with many improvements and a new implementation.

B.2 Aims of this Thesis

The first aim of this thesis is to present a computational model for the cooperation of constraint domains in the $CFLP$ context and the formal goal solving calculus $CCLNC(\mathcal{C})$ where $\mathcal{C}$ is instantiated for two concrete cooperations. This calculus is sound and complete with respect to the instance $CFLP(\mathcal{C})$ of the generic $CFLP$ scheme.

A second objective of the thesis is to implement the extension of $\mathcal{TOY}$ which supports the cooperation of solvers via bridges and projections for the Herbrand domain $\mathcal{H}$ and the two numeric domains $\mathcal{R}$ and $\mathcal{FD}$. The implementation follows the techniques summarized in previous papers [EFS07, EFS08]. It has been developed on top of the $\mathcal{TOY}$ system [ACE+07], which is in turn implemented on top of SICStus Prolog [SIC11]. The $\mathcal{TOY}$ system already supported non-cooperative $CFLP$ programming using the $\mathcal{FD}$ and $\mathcal{R}$ solvers provided by SICStus along with Prolog code for the $\mathcal{H}$ solver. This former system has been extended, including a store for bridges and implementing mechanisms for computing bridges and projections according to the $CCLNC(\mathcal{C}_{\mathcal{FD}.\mathcal{R}})$ computation model.

A third objective verifies the feasibility of this approach with the implementation of the extension of $\mathcal{TOY}$ which supports the cooperation of solvers via bridges and projections for the Herbrand domain $\mathcal{H}$ and the finite domains $\mathcal{FD}$ and $\mathcal{FS}$. This prototype has been developed in the $\mathcal{TOY}$ system, using the ECL^{i}PS^e libraries for sets of integers and finite domain as black-box solvers. In fact, the $\mathcal{FD}$ and $\mathcal{FS}$ solvers for $\mathcal{TOY}$ has been developed on top of the solvers available in ECL^{i}PS^e, and has been extended with some primitive functions and enhanced with failure checking when handling disequality constraints. For this reason, these solvers have been split in two different layers: First, a glass-box solver developed in $\mathcal{TOY}$ for dealing with disequality constraints, and second, the black-box solvers which invoke the corresponding solvers available in ECL^{i}PS^e.

The implementation of this extension of $\mathcal{TOY}$ has been developed for working in different

modes or strategies: The first mode, referred to as *interactive*, is oriented to model reasoning, i.e., when the model can be modified during solving. It is based on an interactive communication between $\mathcal{TOY}$ and ECL^{i}PS^e. When $\mathcal{TOY}$ requires a constraint to be solved in ECL^{i}PS^e, $\mathcal{TOY}$ posts it and blocks until the ECL^{i}PS^e server returns an answer. This approach takes advantage of all features of the $\mathcal{TOY}$ language. The second, *batch* mode, is intended for classic CP applications, where constraints are first specified and then posted and solved. This approach delays the computation of some or all constraints of the current goal, sending them to ECL^{i}PS^e at the end of the $\mathcal{TOY}$ narrowing step. This mode avoids the interactive communication between both processes that may slow down the execution of goals. However, communication is inevitable when more answers are demanded for a given goal. The backtracking mechanism forces interactive communication between both processes, even though the constraints are sent in batch mode. The third mode is the ECL^{i}PS$^e_{gen}$ mode. It measures the time that ECL^{i}PS^e takes in solving the set of constraints sent by $\mathcal{TOY}$, removing any overhead produced by narrowing and inter-process communication. This has been done by generating an ECL^{i}PS^e Prolog program that contains all atomic primitive constraints created by $\mathcal{TOY}$ in the narrowing process. For the first two modes, an ECL^{i}PS^e program executes in a different process acting as a server, accepting and executing requests from the $\mathcal{TOY}$ process. During the execution of a $\mathcal{TOY}$ goal, a request is issued to the ECL^{i}PS^e server for every constraint that needs to be evaluated. The server evaluates the constraint and returns the result to the $\mathcal{TOY}$ process. For communicating both processes, standard input/output library predicates have been used operating on a pipe that interconnects them.

Finally, another important aim of the thesis is to provide some evidence on the practical use and performance of our implementation. To this purpose, we present some illustrative examples and a set of benchmarks tailored to test the performance of the cooperation of constraint domains implemented in $\mathcal{TOY}$

B.3 Related Work

The cooperation of constraints has been addressed already in the CFLP context with different approaches and solvers. One of these approaches is the work of Mircea Marin and Tetsuo Ida [Mar00, MI00], which defined a scheme CFLP(X,S,C) for the integration of a higher-order lazy narrowing calculus C with the solver cooperation over a constraint domain X. The solver cooperation is defined by a collection of constraint solvers which cooperated in accordance with a strategy S. The functional logic component C provides support for defining user-defined functions and predicates over the constraint domain X.

An instance of this cooperative constraint functional logic programming scheme is the system `CFLP` [MIS01]. This distributed implementation can solve problems that involve constraints such as systems of linear, polynomial, and differential equations, and equations with invertible functions. An extension of `CFLP` is the `Open CFLP` system [KMI01, KMI03] where the constraint solvers were distributed in an open environment such as the Internet. This scheme allows to embed the results of the solvers into a Functional Logic Programming language using a higher-order lazy narrowing calculus similar to [LRV07].

This work and our work differ mainly in the nature of the solvers. Their approach uses

specialized solvers for solving equations over very specific domains, such as a polynomial solver and a differential equations solver, while our approach deals with the cooperation between more general-purpose solvers, in this case $\mathcal{H}$, $\mathcal{FD}$ and $\mathcal{FS}$.

Another approach is the work of Petra Hofstedt, [Hof01, Hof00b]. Within the scope of the CFLP languages, Hofstedt defines a general scheme for the cooperation of constraint solvers that uses an interface which allows a fine-grain formal specification of the information exchange between constraint solvers. On top of this interface, there is an open and flexible combination mechanism for constraint solvers. The combination is open in the sense that a new constraint system, with its associated constraint solver (satisfying certain properties), can be easily incorporated into the overall system independently of its domain and the language in which the constraint solver was implemented. And it is flexible because it is possible to define different strategies for the cooperation of the single solvers. In [HP07] a general approach was introduced for the integration of declarative languages and constraint programming. The idea was based on three steps: First, to identify the language; next, to extend the syntax of the language (and the evaluation mechanism) with constraints of other domains; and finally, to define the interface functions of the particular language solver. Therefore, the overall system for cooperating solvers allowed the handling of hybrid constraints over different domains.

An implementation of this framework, called `Meta-S`, is described in [FHM03a, FHM03b, FHR05]. `Meta-S` allows the integration of external solvers and their cooperation with various strategies. `Meta-S` connects a meta-solver framework with attached solvers from several domains. In particular, some finite domain solvers for different domain types (floats, strings and rationals), a solver for linear arithmetic, and an interval arithmetic solver. Therefore, only the cooperation of the $\mathcal{R}$ and $\mathcal{FD}$ solvers in $\mathcal{TOY}$ can be compared with `Meta-S`, since it has not incorporated set solvers. Both systems share that projections play a key role, and in the cooperation between $\mathcal{FD}$ and $\mathcal{R}$ there is a significant speed-up in $\mathcal{TOY}$ caused by the activation of projections. An extended discussion on analogies and differences between `Meta-S` and $\mathcal{TOY}$ can be found in [EFH$^+$09].

Also in the context of CFLP, the programming language `Curry` [Cur12] combines features from functional programming, logic programming, and concurrent programming. `Curry` and its prolog-based implementation, PAKCS, do not incorporate a solver for set constraints and, to the best of our knowledge, the cooperation of constraints has not been considered in `Curry`.

With respect to the cooperation of the $\mathcal{FD}$ and $\mathcal{FS}$ solvers we follow the idea developed by Carmen Gervet [Ger94, Ger97]. She defines a framework for a new logic programming language over finite domains of sets. This language, called `Conjunto`, combines aspects of declarative languages as Prolog with constraint solvers. To define this new framework, Gervet defines the set variable domain using the greater lower bound and least upper bound as ground bounds, as well as a partial order relation (set inclusion $\subseteq$). This set representation has been the basis of several subsets bound solvers, as for example: ECL^{i}PS^e [ECL], `FaCiLe` [FaC], `Mozart-OZ` [Moz], `B-Prolog` [B-P], and `CHOCO` [CHO]. Later, a different approach is the one proposed in [AB00], that adds cardinality information to subset-domains. This idea is applied to a new finite set solver called `Cardinal`, which was implemented in ECL^{i}PS^e [AW07]. [SG04] proposed a hybrid set domain with three components: a subset-bound domain, the cardinality, and a lexicographic representation of the domain. The consistency of the three

domains is maintained by intra-domain constraints, which were complex and computationally expensive. Finally, [GV06] proposed for the same system the length-lex representation which provides a total order on sets, first by length and then lexicographically.

B.4 The Language $\mathcal{TOY}$

$\mathcal{TOY}$ [ACE+07, CCES12] is a functional logic language and system that solves goals by means of a *demand driven lazy narrowing strategy* [LLR93] combined with constraint solving. A $\mathcal{TOY}$ program is a collection of definitions of datatypes, operators, lazy defined functions in Haskell style, as well as definitions of predicates in Prolog style. A *predicate* is a particular kind of function which returns a Boolean true value. A *function* includes an optional *polymorphic type declaration* f :: τ_1 -> ... τ_n -> τ, and one or more defining rules with *curried notation*:

$$f\ t_1\ \ldots\ t_n = r \texttt{ <== } C_1, \ldots, C_m$$

A defined function must be *linear* in the *left-hand side*, i.e., $t_1 \ldots t_n$ cannot include repeated variables. The *right-hand side* r can be any expression built from variables, built-in operations, data constructors, and functions. *Conditions* C_i, and *local definitions* D_j are optional. The intended meaning of a defining rule is the same as in functional programming languages, namely: an expression of the form $f\ e_1 \ldots e_n = r$ can be narrowed by narrowing the actual parameters e_k until they match the patterns t_i, checking that the conditions C_i are satisfied, and then narrowing the right hand side r (affected by the pattern matching substitution). Local definitions D_j are used to obtain the values of locally defined variables. Functions can be *non-deterministic* and *higher-order*. A *goal* in $\mathcal{TOY}$ is a conjunction of conditions C_1, ..., C_m where each condition C_i is interpreted as a constraint to be solved.

Some features of CFLP languages as $\mathcal{TOY}$ [ACE+07] and Curry [Cur12], such as higher-order and partial function applications, facilitate programming in the sense of clarity and neatness, resulting in some cases in more understandable and maintainable programs.

Example 1. To illustrate the $\mathcal{TOY}$ language, let us define the following function `atMostOne` which, given a list of sets, restricts it to a list of triples such that, given any two sets in the list, they have at most one element in common. The expression `S #-- C` relates a set `S` with its cardinality `C`.

```
atMostOne :: [set]->bool
atMostOne [] = true
atMostOne [X|Xs] = atMostOne Xs <==
a    3 #-- X,
b    andL (map (sendCon X) Xs)
```

```
sendCon :: set -> set -> bool
sendCon SX SY = true <==
c    intersect SX SY SZ,
d    Z #-- SZ, Z #<= 1

andL :: [bool] -> bool
andL = foldr (/\) true
```

The function `atMostOne` restricts the cardinality of the sets to the value 3 (line a). In b, uses the higher-order built-in library function `map`, which applies a partial application `sendCon X` to each element of the list `Xs`, returning a list of Boolean results. Lines c and d constrain the sets to have at most one element in common. Finally, `andL` is defined as a folding of conjunctions. A possible goal that invokes `atMostOne` is:

```
L==[S1,S2],domainSets L {} {1,2,3,4,5},atMostOne L,labelingSets L
```

where `domainSets L {} {1,2,3,4,5}` constrains set variables `S1` and `S2` to be sets in the lattice defined between `{}` and `{1,2,3,4,5}`, and `labelingSets` enumerates all ground values of `S1` and `S2`. Some answers of this goal are:

```
S1 ↦ {1,2,3}, S2 ↦ {1,4,5}
S1 ↦ {1,2,3}, S2 ↦ {2,4,5}
        .....
S1 ↦ {2,3,5}, S2 ↦ {1,2,4}
```

□

As illustrated by Example 1, the language $\mathcal{TOY}$ is strongly typed, with all the well-known advantages of a type checking process. The principal type of a function can include type variables (*polymorphic types*). Type declarations are optional and are nevertheless inferred and checked. It is also possible to define types (*constructed types*), along with *data constructors* for their values, and *type synonyms* to declare a new identifier as a shorthand for a complex type.

$\mathcal{TOY}$ solves goals by means of a *demand driven lazy narrowing strategy* combined with constraint solving. Lazy narrowing avoids computations which are not demanded, allowing to handle infinite structures. For instance, let us consider the goal

```
take 3 infList == R, intSets R 1 6, atMostOne R, labelingSets R
```

where `infList` is defined as: `infList = [_|infList]`.

`R` is constructed with 3 elements taken from an infinite list of set variables, with elements between 1 and 6, and which have at most one element in common. After the labeling, all solutions are obtained on backtracking. For example: `R` ↦ `[{1,2,3},{1,4,5},{2,4,6}]`.

Another feature of the language $\mathcal{TOY}$ is the possibility of using *higher-order functions* and *partial function applications*. It refers to the process of fixing the first arguments of a function, producing another function with the remaining arguments. $\mathcal{TOY}$ allows partial function applications because functions are defined in curried form. Therefore, if a function is called with few parameters, then a partially applied function is obtained. This is a neat way to create functions that can be passed to another function as data. A function that takes other functions as arguments or returns a function as a result is called a higher-order function. $\mathcal{TOY}$ supports higher-order programming, although λ-abstractions are not allowed. These $\mathcal{TOY}$ features allow modeling problems in a simple and compact way.

For instance, the function `atMostOne` of Example 1 uses the function `map`, which, given a partial function application `(sendCon X)`, applies it to each element of the list `Xs`. The application of the function `sendCon` is partial because only one argument is provided, while `sendCon` demands two arguments. The evaluation of the function application is suspended until function `map` provides the second argument from the list `Xs`.

Other language features are detailed in [TOY12, ACE+07].

B.5 Motivating Examples of the Cooperation

This section shows the advantages of the cooperation between solvers by means of motivating examples.

B.5.1 Examples of the Cooperation of the Domains $\mathcal{R}$ and $\mathcal{FD}$

The following program written in $\mathcal{TOY}$ [EFH+09] solves the problem of searching for a $2D$ point lying in the intersection of a discrete grid and a continuous region. The program includes type declarations, equations for defining functions and clauses for defining predicates. Type declarations are similar to those used in functional languages such as Haskell [Pey02]. Function applications use *curried notation*, also typical of Haskell and other higher-order functional languages.

```
% Discrete versus continuous points:
type dPoint = (int, int)
type cPoint = (real, real)

% Sets and membership:
type setOf A = A -> bool
isIn :: setOf A -> A -> bool
isIn Set Element = Set Element

% Grids and regions as sets of points:
type grid = setOf  dPoint
type region = setOf  cPoint

% Predicate for computing intersections of regions and grids:
bothIn :: region -> grid -> dPoint -> bool
```

```
bothIn Region Grid (X, Y) :- X #== RX, Y #== RY,
    isIn Region (RX, RY), isIn Grid (X,Y), labeling [ ] [X,Y]

% Square grid (discrete):
square :: int -> grid
square N (X,Y) :- domain [X,Y] 0 N

% Triangular region (continuous):
triangle :: cPoint -> real  -> real -> region
triangle (RX0,RY0)  B H (RX,RY) :-
  RY >= RY0 - H,
  B * RY - 2 * H * RX <= B * RY0 - 2 * H * RX0,
  B * RY + 2 * H * RX <= B * RY0 + 2 * H * RX0

% Diagonal segment (discrete):
diagonal :: int -> grid
diagonal N (X,Y) :- domain [X,Y] 0 N, X == Y

% Parabolic line (continuous):
parabola :: cPoint ->  region
parabola (RX0,RY0)  (RX,RY) :- RY - RY0 == (RX - RX0) * (RX - RX0)
```

Predicate `bothIn` is intended to check if a given discrete point `(X,Y)` belongs to the intersection of the continuous region `Region` and the discrete grid `Grid` given as parameters, and the constraints occurring as conditions are designed to this purpose. More precisely, the two bridge constraints `X #== RX, Y #== RY` ensure that the discrete point `(X,Y)` and the continuous point `(RX,RY)` are equivalent; the two strict equality constraints `isIn Region (RX, RY) == true, isIn Grid (X,Y) == true` ensure membership to `Region` and `Grid`, respectively; and finally the $\mathcal{FD}$ constraint `labeling [ ] [X,Y]` ensures that the variables `X` and `Y` are bound to integer values.

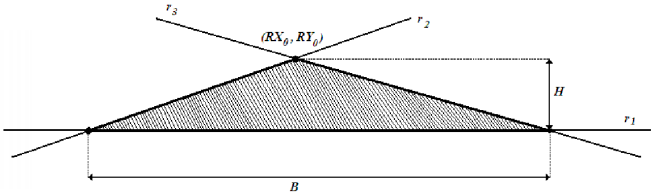


Figure B.1: Triangular Region

Note that both grids and regions are represented as sets, represented themselves as Boolean functions. They can be passed as parameters because higher-order is supported. The program also defines two functions `square` and `triangle`, intended to compute representations of square grids and triangular regions, respectively. Let us discuss them in

turn. We first note that the type declaration for `triangle` can be written in the equivalent form `triangle :: cPoint -> real -> real -> (cPoint -> bool)`. A function call of the form `triangle (RX0,RY0) B H` is intended to return a Boolean function representing the region of all continuous $2D$ points lying within the isosceles triangle with upper vertex `(RX0,RY0)`, base `B` and height `H`. Applying this Boolean function to the argument `(RX,RY)` yields a function call written as `triangle (RX0,RY0) B H (RX,RY)` and expected to return `true` in case that `(RX,RY)` lies within the intended isosceles triangle, whose three vertices are `(RX0,RY0)`, `(RX0-B/2,RY0-H)` and `(RX0+B/2,RY0-H)`. The three sides of the triangle are characterized by the equations `RY = RY0-H`, `B*RY-2*H*RX = B*RY0-2*H*RX0` and `B*RY+2*H*RX = B*RY0+2*H*RX0` (corresponding to the lines r_1, r_2 and r_3 in Fig. B.1, respectively). Therefore, the conjunction of three linear inequality $\mathcal{R}$ constraints occurring as conditions in the clause for `triangle` succeeds for those points `(RX,RY)` lying within the intended triangle.

Similarly, the type declaration for `square` can be written in the equivalent form `square :: int -> (dPoint -> bool)`, and a function call of the form `square N` is intended to return a Boolean function representing the grid of all discrete $2D$ points with coordinates belonging to the interval of integers between `0` and `N`. Therefore, a function call of the form `square N (X,Y)` must return `true` in case that `(X,Y)` lies within the intended grid, and for this reason the single $\mathcal{FD}$ constraint placed as a condition in the clause for `square` has been chosen to impose the interval of integers between `0` and `N` as the domain of possible values for the variables `X` and `Y`.

Finally, the last two functions `diagonal` and `parabola` are defined in such a way that `diagonal N` returns a Boolean function representing the diagonal of the grid represented by `square N`, while `parabola (RX0,RY0)` returns a Boolean function representing the parabola whose equation is `RY-RY0 = (RX-RX0)*(RX-RX0)`. The type declarations and clauses for these functions can be understood similarly to the case of `square` and `triangle`.

Different *goals* can be posed and solved using the small program just described and the cooperative goal solving calculus $CCLNC(\mathcal{C})$ as implemented in the $\mathcal{TOY}$ system. For the sake of discussing some of them, assume two fixed positive integer values `d` and `n` such that `n = 2*d`. Then `(d,d)` is the middle point of the grid `(square n)`, which includes `(n+1)`^{2} discrete points. The three following goals ask for points in the intersection of this fixed square grid with three different triangular regions:

- **Goal 1**: `bothIn (triangle (d, d+0.75) n 0.5) (square n) (X,Y)`.
  This goal fails.

- **Goal 2**: `bothIn (triangle (d, d+0.5) 2 1) (square n) (X,Y)`.
  This goal computes one solution for `(X,Y)`, corresponding to the point `(d,d)`.

- **Goal 3**: `bothIn (triangle (d, d+0.5) (2*n) 1) (square n) (X,Y)`.
  This goal computes `n+1` solutions for `(X,Y)`, corresponding to the points `(0,d)`, `(1,d)`, `...`, `(n,d)`.

These three goals are illustrated in Fig. B.2 for the particular case `n = 4` and `d = 2`, although the shapes and positions of the three triangles with respect to the middle point of

the grid would be the same for any even positive integer `n = 2*d`. The expected solutions for each of the three goals are clear from the figures.

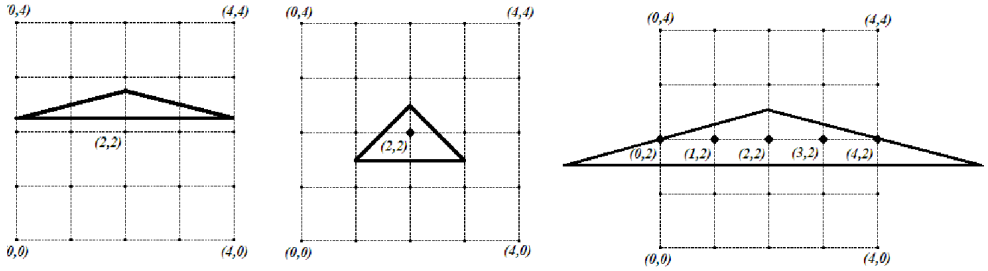


Figure B.2: Intersection of a Fixed Square Grid with Three Different Triangular Regions

In the three cases, cooperation between the $\mathcal{R}$ solver and the $\mathcal{FD}$ solver is crucial for the efficiency of the computation. In the case of **Goal 2**, cooperative goal solving implemented in $\mathcal{TOY}$ according to the $CCLNC(\mathcal{C})$ computation model uses the clauses in the program and eventually reduces the problem of solving the goal to the problem of solving a constraint system that, suitably simplified, becomes:

```
X #== RX, Y #== RY,
RY >= d-0.5, RY-RX <= 0.5, RY+RX <= n+0.5,
domain [X,Y] 0 n, labeling [ ] [X,Y]
```

The $\mathcal{TOY}$ system has the option to enable or disable the computation of projections. When projections are disabled, the two bridges do still work as constraints, and the last $\mathcal{FD}$ constraint `labeling [] [X,Y]` forces the enumeration of all possible values for `X` and `Y` within their domains, eventually finding the unique solution `X = Y = d` after $\mathcal{O}(\mathtt{n}^2)$ steps. When projections are enabled, the available bridges are used to project the $\mathcal{R}$ constraints `RY >= d-0.5, RY-RX <= 0.5, RY+RX <= n+0.5` into the $\mathcal{FD}$ constraints `Y #>= d, Y#-X #<= 0, Y#+X #<= n`. Since `n = 2*d`, the only possible solution of these inequalities is `X = Y = d`. Therefore, the $\mathcal{FD}$ solver drastically prunes the domains of `X` and `Y` to the singleton set `{d}`, and solving the last labeling constraint leads to the unique solution with no effort. For a big value of `n = 2*d` the performance of the computation is greatly enhanced in comparison to the case where projections are disabled, as confirmed by the experimental results. The expected speed-up in execution time corresponds to the improvement from the $\mathcal{O}(\mathtt{n}^2)$ steps needed to execute the labeling constraint `labeling [] [X,Y]` when the domains of both `X` and `Y` have size $\mathcal{O}(\mathtt{n})$, to the $\mathcal{O}(1)$ steps needed to execute the same constraint when the domains of both `X` and `Y` have been pruned to size $\mathcal{O}(1)$. Similar speedups are observed when solving **Goal 1** (which finitely fails, and where the expected execution time also improves from $\mathcal{O}(\mathtt{n}^2)$ to $\mathcal{O}(1)$) and **Goal 3** (which has just `n+1` solutions, and where the expected execution time reduces from $\mathcal{O}(\mathtt{n}^2)$ to $\mathcal{O}(\mathtt{n})$).
The three goals just discussed mainly illustrate the benefits obtained by the $\mathcal{FD}$ solver from the projection of $\mathcal{R}$ constraints. In fact, when $\mathcal{TOY}$ solves these three goals according to

the cooperative computation model $CCLNC(\mathcal{C})$, the available bridge constraints also allow to project the $\mathcal{FD}$ constraint `domain [X,Y] 0 n` into the conjunction of the $\mathcal{R}$ constraints `0 <= RX, RX <= n, 0 <= RY, RY <= n`. These constraints, however, are not helpful for optimizing the resolution of the previously computed $\mathcal{R}$ constraints `RY >= d-0.5, RY-RX <= 0.5, RY+RX <= n+0.5`.

In general, it seems easier for the $\mathcal{FD}$ solver to profit from the projection of $\mathcal{R}$ constraints than the other way round. This is because the solution of many practical problems is arranged to finish with solving $\mathcal{FD}$ labeling constraints, which means enumerating values for integer variables, and this process can greatly benefit from a reduction of the variables' domains due to previous projections of $\mathcal{R}$ constraints. However, the projection of $\mathcal{FD}$ constraints into $\mathcal{R}$ constraints can help to define the intended solutions even if the performance of the $\mathcal{R}$ solver does not improve. For instance, assume that the value chosen for `n = 2*d` is big, and consider the goal

- **Goal 4**: `bothIn (triangle (d,d) n d) (square 4) (X,Y).`

whose resolution eventually reduces to the problem of solving a constraint system that, suitably simplified, becomes:

```
X #== RX, Y #== RY,
RY >= 0, RY-RX <= 0, RY+RX <= n,
domain [X,Y] 0 4, labeling [ ] [X,Y]
```

The solutions correspond to the points lying in the intersection of a big isosceles triangle and a tiny square grid. Projecting `RY >= 0, RY-RX <= 0, RY+RX <= n` into $\mathcal{FD}$ constraints via the two bridges `X #== RX, Y #== RY` brings no significant gains to the $\mathcal{R}$ solver whose task is anyhow trivial. The $\mathcal{R}$ constraints projected from `domain [X,Y] 0 4` (i.e., `0 <= RX, RX <= 4, 0 <= RY, RY <= 4`) do not improve the performance of the $\mathcal{R}$ solver either, but they help to define the intended solutions. In this example, the last labeling constraint eventually enumerates the right solutions even if the projection of the domain constraint to $\mathcal{R}$ does not take place, but this projection would allow the $\mathcal{R}$ solver to compute suitable constraints as solutions in case that the labeling constraint were removed.

There are also some cases where the performance of the $\mathcal{R}$ solver can benefit from the cooperation with the $\mathcal{FD}$ domain. Consider for instance the goal

- **Goal 5**: `bothIn (parabola (2,0)) (diagonal 4) (X,Y).`

asking for points in the intersection of the discrete diagonal segment of size 4 and a parabola with vertex `(2,0)` (see Fig. B.3). Solving this goal eventually reduces to solving a constraint system that, suitably simplified, becomes:

```
X #== RX, Y #== RY,
RY == (RX-2)*(RX-2),
domain [X,Y] 0 4, X == Y, labeling [ ] [X,Y]
```

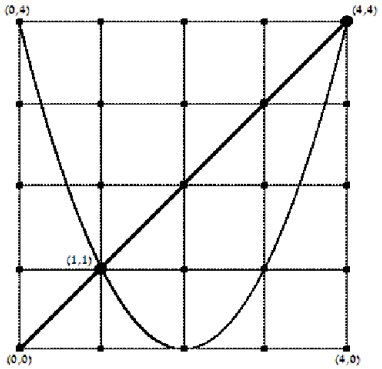


Figure B.3: Intersection of a Parabolic Line and a Diagonal Segment

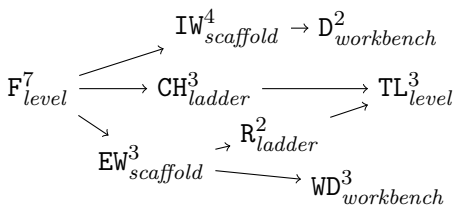
Cooperative goal solving as implemented in $\mathcal{TOY}$ processes the constraints within the current goal in left-to right order, performing projections whenever possible, and sometimes delaying a constraint that cannot be managed by the available solvers. In this case, the quadratic $\mathcal{R}$ constraint `RY == (RX-2)*(RX-2)` is delayed because the $\mathcal{R}$ solver used by $\mathcal{TOY}$ (inherited form SICStus Prolog) cannot solve non-linear constraints. However, since this strict equality relates expressions of type `real`, it is accepted as an $\mathcal{R}$ constraint and projected via the available bridges, producing the $\mathcal{FD}$ constraint `Y == (X-2)*(X-2)`, which is submitted to the $\mathcal{FD}$ solver. Next, projecting the $\mathcal{FD}$ constraint `domain [X,Y] 0 4` and solving `X == Y` causes the $\mathcal{R}$ constraints `0 <= RX, RX <= 4, 0 <= RY, RY <= 4` to be submitted to the $\mathcal{R}$ solver, and the variable `X` to be substituted in place of `Y` all over the goal. The bridges `X #== RX, Y #== RY` become then `X #== RX, X #== RY`, and the labeling constraint becomes `labeling [] [X,X]`. An especial mechanism called *bridge unification* infers from the two bridges `X #== RX, X #== RY` the strict equality constraint `RX == RY`, which is solved by substituting `RX` for `RY` all over the current goal. At this point, the delayed $\mathcal{R}$ constraint becomes `RX == (RX-2)*(RX-2)`. Finally, the $\mathcal{FD}$ constraint `labeling [] [X,X]` is solved by enumerating all the possible values for `X` allowed by its domain, and continuing a different alternative computation with each of them. Due to the bridge `X #== RX`, each integer value `v` assigned to `X` by the labeling process causes the variable `RX` to be bound to the integral real number `rv` equivalent to `v` (in our computation model, this is part of the behaviour of a solver in charge of solving bridge constraints). The binding of `RX` to `rv` awakens the delayed constraint `RX == (RX-2)*(RX-2)`, which becomes the linear (and moreover ground) constraint `rv == (rv-2)*(rv-2)` and succeeds if `rv` is an integral solution of the delayed quadratic equation. In this way, the two solutions of **Goal 5** are eventually computed, corresponding to the two points `(X,Y)` lying in the intersection of the parabolic line and the diagonal segment: `(1,1)` and `(4,4)`, as seen in Fig. B.3.

All the computations described in this subsection can be actually executed in the $\mathcal{TOY}$ system and also formally represented in the cooperative goal solving calculus $CCLNC(\mathcal{C})$. The formal representation of goal solving computations in $CCLNC(\mathcal{C})$ performs quite many detailed intermediate steps. In particular, constraints are transformed into a flattened form (without nested calls to primitive functions) before performing projections, and especial mech-

anisms for creating new bridges in some intermediate steps are provided. Detailed explanations and examples are given in Section B.9.

B.5.2 Example of the Cooperation of the Domains $\mathcal{FD}$ and $\mathcal{FS}$

Let us suppose that we need to plan the project of building a house, a modified version of the problem exposed in [MS98] (page 17). This problem is split into smaller tasks as: laying foundations, build walls and a chimney, place doors and windows, and putting the roof tiles. Each task has a certain duration that is measured in a number of days and is performed in full days, but may be on non-consecutive days. Some tasks precede others and others require the same resources and therefore can not be performed simultaneously. Figure B.4 shows a precedence graph for this problem where superscripts and subscripts stand for the number of days needed to complete the tasks, and the resource required for performing the task, respectively.



F:	Foundations		
IW:	Interior Walls		
CH:	Chimney	D:	Doors
EW:	Exterior Walls	R:	Roof
WD:	Windows		
TL:	Tiles		

Figure B.4: Building a House

When tasks are performed on consecutive days, coding this problem in the $\mathcal{FD}$ domain is simple by following the idea shown in [MS98]. But if we allow that tasks can be performed on non-consecutive days (i.e., interruptions are allowed), then the task representation and the overall problem modelling become more complicated.

An alternative model is to use a representation of the problem based on sets of integers. Each task can be represented by a set variable whose elements are the days that the task is executed. For instance, the set `{1,3,4}` means that the task is performed in the days `1`, `3` and `4`.

The issue we encountered when modelling this problem with sets is to establish precedence constraints with the basic set operations ($\subseteq$, $\cup$, $\cap$, ...). To overcome this issue we have introduced a new constraint `<<`, where Set_1 `<<` Set_2 forces all elements of Set_1 to be less than all the elements of Set_2. Now, to model this problem is simple in the finite integer set domain.

For modelling this problem, each task is represented as a set variable whose domain ranges between 1 and the number of available days. The duration of a task is the cardinal of the corresponding set. Tasks that require the same resource are restricted to be disjoint and precedence constraints are established by the new constraint `<<`. A $\mathcal{TOY}$ code excerpt implementing this is:

```
% Main Function
sch_house :: int -> [iset] -> bool
sch_house  Days Sets = true <==
```

```
% List of tasks
Sets == [F,IW,CH,EW,D,R,TL,WD],

% Each task is a set of integer numbers from 1 to Days
intSets Sets 1 Days,

% The duration of a task corresponds to the cardinal of its set
andL (zipWith (#--) [7,4,3,3,2,2,3,3] Sets),

% Precedence constraints
F<<IW, F<<CH, F<<EW, IW<<D, CH<<TL, EW<<R, EW<<WD, R<<TL,

% No overlap tasks
disjoints [F,TL], disjoints [IW,EW],
disjoints [D,WD], disjoints [CH,R],

% Enumerating solutions
labelingSets Sets
```

For this code, the goal `sch_house 16 Set` binds `Set` to the list: `[[1,2,3,4,5,6,7], [8,12,13,14], [8,9,10], [9,10,11], [15,16], [12,13], [14,15,16], [12,13,14]]`.

In addition to modeling this example in a simple way, it is possible to improve the runtime of the goals if the finite domain solver cooperates with the set solver. Suppose the number of days available for all tasks is 14. In this case it is easy to see that it is not possible the planning for the tasks `F`, `EW`, `R` and `TL` because of their precedence constraints, since the sum of the durations of these tasks (15) exceeds the available number of days (14). With the above constraints, the finite set solver is not able to infer that there is no solution and it therefore tests all possible combinations. However, exploring all combinations can be avoided if the information of precedence of these tasks would be projected to the $\mathcal{FD}$ domain with some sort of bridge constraint. Let's assume that we could use in $\mathcal{FD}$ constraints the 'function' $min(S)$ that relates its integer outcome with the minimum value in the set variable S. Then, we could write the following $\mathcal{FD}$ constraints in order for the $\mathcal{FD}$ solver to participate in the solving:

```
1 1≤min(F),          3 min(EW)+3≤min(R),    5 min(TL)+3≤14+1
2 min(F)+7≤min(EW),  4 min(R)+2≤min(TL),
```

In this system of inequalities, the minimum and maximum of each set ranges between 1 and 14. Besides, from the inequalities 1, 2 and 3 the inequality $11 \leq min(R)$ is inferred. In 4 the value $min(TL)$ is therefore at least 13, which does not satisfy the inequality $min(TL) + 3 \leq 15$. So, the solver $\mathcal{FD}$ detects the inconsistency and anticipates failure. However, to establish this communication between the $\mathcal{FD}$ and $\mathcal{FS}$ solvers it is necessary to define new constraints like the cardinal that relate variables between different domains, acting as bridges. This thesis develops this by introducing the bridge constraints `minSet` and `maxSet` for limiting the minimum and maximum values in set variables, but first constraints domains and solvers in our $CFLP$ scheme are formally defined.

B.6 Constraint Domains and Solvers

The formal definition of pure constraint domains and their solvers are defined with respect to the computational model for the cooperation of constraint domains described in [EFH$^+$08, EFH$^+$09]. First let us recall some aspects.

A constraint domain has a specific signature $\Sigma = \langle TC, SBT, DC, DF, SPF\rangle$ consisting of pairwise disjoint sets of *Type Constructors* (TC), *Specific Base Types* (SBT), *Data Constructors* (DC), *Defined Function Symbols* (DF) and *Specific Primitive Function Symbols* (SPF). Base types and primitive function symbols are related to specific constraint domains, while type constructors, data constructors, and defined function symbols are provided by user programs.

In our computational model, a *constraint domain* $\mathcal{D}$ with signature Σ is a structure $\mathcal{D} = \langle \mathcal{B}^{\mathcal{D}}, \{p^{\mathcal{D}}\}_{p\in SPF}\rangle$ where $\mathcal{B}^{\mathcal{D}} = \bigcup_{d\in SBT} \mathcal{B}^{\mathcal{D}}_d$ is the set of *base values* and $p^{\mathcal{D}}$ is the *interpretation* of each primitive function symbol $p \in SPF$. Interpretation $p^{\mathcal{D}}\bar{t}_n \to t$ means that the primitive function p with given arguments $t_1, \ldots, t_n$ returns t in the domain $\mathcal{D}$.

We define an *expression* with the syntax $e ::= X \mid \bot \mid u \mid h \mid (e\, e_1)$, where X is a variable, $\bot$ is a special data constructor that represents an undefined value that belongs to any type, $u \in \mathcal{B}^{\mathcal{D}}$ is a base value, $h \in DC \cup DF \cup SPF$, and $(e\, e_1)$ stands for the application of e to e_1. A *pattern* is a particular expression representing a data value that does not need to be evaluated. Its syntax is $t ::= X \mid \bot \mid u \mid c\,\bar{t}_m \mid f\,\bar{t}_m \mid p\,\bar{t}_m$, where X is a variable, $u \in \mathcal{B}^{\mathcal{D}}$, $c \in DC^n$ for some $m \leq n$, $f \in DF^n$ for some $m < n$, and $p \in SPF^n$ for some $m < n$, representing partial applications. Expressions and patterns without any occurrence of $\bot$ are called total.

An *atomic constraint* $\pi \in ACon_{\mathcal{D}}$ over a given domain $\mathcal{D}$ is defined either as $\lozenge$ (standing for *truth*), or $\blacklozenge$ (standing for *falsity*), or $p\ \bar{e}_n \to\!!\ t$ with $p \in SPF^n$, where each e_i is an expression and t is a total pattern. By convention, constraints of the form $p\ \bar{e}_n \to\!!\ true$ are abbreviated as $p\ \bar{e}_n$. In particular, strict equality constraints e_1 `==` e_2 and strict disequality constraints e_1 `/=` e_2 are understood as abbreviations of `(==)` $e_1\ e_2 \to\!!$ `true` and `(==)` $e_1\ e_2 \to\!!$ `false`, respectively. *Atomic primitive constraints* $\pi \in APCon_{\mathcal{D}}$ over a given domain $\mathcal{D}$ are atomic constraints where $\bar{e}_n$ are patterns.

A *substitution* $\sigma \in Sub_{\mathcal{D}}$ over a given domain $\mathcal{D}$ is a set of mappings from variables to patterns. A *valuation* η over a given domain $\eta \in Val_{\mathcal{D}}$, is a ground substitution that maps variables to values. The valuations that satisfy a given constraint π are said to be *solutions of* π, $Sol_{\mathcal{D}}(\pi) \subseteq Val_{\mathcal{D}}$.

A *constraint store* is a pair $S = \Pi\Box\sigma$ for a domain $\mathcal{D}$, where $\Pi \subset APCon_{\mathcal{D}}$ is a set of atomic primitive constraints and σ is an idempotent substitution such that domain variables of σ and variables of Π are disjoint. The symbol $\Box$ is interpreted as a conjunction. *Solutions of constraint stores* are defined as $Sol_{\mathcal{D}}(\Pi \Box \sigma) = Sol_{\mathcal{D}}(\Pi) \cap Sol_{\mathcal{D}}(\sigma)$ where $Sol_{\mathcal{D}}(\sigma) = \{\eta \in Val_{\mathcal{D}} \mid \eta = \sigma\eta\}$. If σ is empty $Sol_{\mathcal{D}}(\Pi)$ is used.

Constraint domains are equipped with their respective solvers, which process the constraints arising in the course of a computation. We consider a constraint solver for the domain $\mathcal{D}$ as modeled by a function $solve^{\mathcal{D}}$. From a user viewpoint, a solver can behave as a black-box or a glass-box. We use a convenient abstract technique for specifying the

behaviour of glass-box solvers named a *store transformation system*. The idea is to specify a set of *store transformation rules* which describe different ways to transform a given store. A store is called *irreducible* iff no store transformation rules can be applied to transform it. A rule is not applicable if the store is not transformed by the rule in any way.

A *transformation step* for a given rule is denoted as $\vdash_{\mathcal{D}}$ and

- $\pi, \Pi \,\square\, \sigma \vdash_{\mathcal{D}} \Pi' \,\square\, \sigma'$ indicates that the store $\pi, \Pi \,\square\, \sigma$, which includes the atomic constraint π plus other constraints Π, is transformed into $\Pi' \square\, \sigma'$ in one step.
- $\Pi \,\square\, \sigma \vdash^*_{\mathcal{D}} \Pi' \,\square\, \sigma'$ indicates that $\Pi \,\square\, \sigma$ can be transformed into $\Pi' \,\square\, \sigma'$ in finitely many steps.
- $\Pi \,\square\, \sigma \vdash_{\mathcal{D}} \blacksquare$ indicates a failing transformation step.

If substitutions are not relevant for the explanation, they will be omitted, and we will use (π, Π_Z) to refer to the store $(\pi, \Pi_Z \,\square\, \sigma_Z)$. Solvers reduce primitive constraints to *solved forms*, which are simpler and are shown as computed answers to the users. Formally:

$$SF_{\mathcal{D}}(\Pi\square\ \sigma) = \{\Pi' \ \square\ \sigma' \mid \Pi \ \square\ \sigma \vdash^*_{\mathcal{D}} \Pi' \ \square\ \sigma', \text{ and } \Pi' \ \square\ \sigma' \text{ is irreducible}\}$$

A constraint solver for a domain $\mathcal{D}$ is modeled as a function $solve^{\mathcal{D}}$ which deals with a set Π of $\mathcal{D}$ constraints. A solver invocation $solve^{\mathcal{D}}(\Pi)$ returns a finite disjunction of existentially quantified constraint stores composed of constraints and substitutions. Formally,

$$solve^{\mathcal{D}}(\Pi) = \bigvee_{j=1}^{k} \{\exists \overline{Y}_j (\Pi_j \ \square\ \sigma_j) \mid \Pi_j \ \square\ \sigma_j \in SF_{\mathcal{D}}(\Pi), \text{ where } \overline{Y}_j = var(\Pi_j \ \square\ \sigma_j) \setminus var(\Pi)\}$$

Alternative constraint stores which are returned by solver invocations are usually explored in sequential order using backtracking.

B.6.1 Constraint Domain and Solver for the $\mathcal{R}$ Domain

The $\mathcal{R}$ domain supporting computation with arithmetic constraints over real numbers is a familiar idea, used in the well-known instance $CLP(\mathcal{R})$ of the CLP scheme [JMSY92]. In the context of our $CFLP$ framework, the $\mathcal{R}$ domain is defined with respect to the signature $\Sigma_{\mathcal{R}} = \langle TC,\ SBT_{\mathcal{R}},\ DC,\ DF,\ SPF_{\mathcal{R}} \rangle$ where $SBT_{\mathcal{R}} = \{\texttt{real}\}$, and the set of base values is the set of real numbers $\mathbb{R}$. Primitive functions in $SPF_{\mathcal{R}}$ are the following:

- `== :: A -> A -> bool`
- `+, -, *, / :: real -> real -> real`
- `<= :: real -> real -> bool`

The real solver of $\mathcal{TOY}$ is not able of solving non-linear constraints since it is built on the solver SICStus, which has the same limitation. For example, $\mathcal{TOY}$ can solve the goal `X + X == 4.0`, where the variable `X` is bound to the value `2`. However, $\mathcal{TOY}$ is unable to resolve the goal `X * X == 4.0` and the constraint `4.0-X^2.0==0.0` is shown as answer. Despite this limitation $\mathcal{TOY}$ can solve interesting problems as it is shown in the motivating examples and the cooperation with the $\mathcal{FD}$ solver.

B.6.2 Constraint Domain and Solver for the $\mathcal{FD}$ Domain

The $\mathcal{FD}$ domain is defined with respect to the signature $\Sigma_{\mathcal{FD}} = \langle TC, SBT_{\mathcal{FD}}, DC, DF, SPF_{\mathcal{FD}} \rangle$ where $SBT_{\mathcal{FD}} = \{\texttt{int}\}$, and the set of base values is the set of integer numbers $\mathbb{Z}$. Primitive functions in $SPF_{\mathcal{FD}}$ are the following:

- `== :: A -> A -> bool`
- `#<= :: int -> int -> bool`
- `#+, #-, #*, #/ :: int -> int -> int`
- `domain :: [int] -> int -> int -> bool`
- `all_different :: [int] -> bool`
- `exactly :: int -> [int] -> int -> bool`
- `belongs :: int -> [int] -> bool`
- `labeling :: [labelingType] -> [int] -> bool`

The constraint `domain L A B` returns `true` if each integer in the list `L` belongs to the interval `[A,B]` in $\mathbb{Z}$ and also if each $\mathcal{FD}$ variable in the list `L` is constrained to have values in the integer interval `[A,B]`. `all_different L` returns `true` if the elements of the list `L` are pairwise different and `false` in other case. `exactly X L N` returns `true` if `X` occurs `N` times in the list `L`, and `labeling L` instantiates all variables of the list `L`.

The solver $solve^{\mathcal{FD}}$ has been implemented in two different systems, SICStus Prolog and ECL^{i}PS^e. The first is involved in the cooperation of $\mathcal{R}$ with $\mathcal{FD}$, and the second in the cooperation of $\mathcal{FD}$ and $\mathcal{FS}$.

The $\mathcal{FD}$ solver has been split in two different layers: First, a glass-box solver $\mathcal{FD}^{\mathcal{T}}$ developed in $\mathcal{TCY}$ for dealing with disequality constraints, and second, the black-box solver $\mathcal{FD}^{N}$ which invokes the solver for integers available in ECL^{i}PS^e. The glass-box solver $\mathcal{FD}^{\mathcal{T}}$ has been formalized using a store transformation system. Its rules are shown in Table B.1, where a given store $\Pi \Box \sigma$ is detected as inconsitent in one rewriting step.

FD1	F_1 `==` F_2, F_1 `/=` F_2, $\Pi \Box \sigma \vdash_{\mathcal{FD}^{\mathcal{T}}} \blacksquare$
FD2	F_1 `#<=` F_2, F_2 `#<=` F_1, F_1 `/=` F_2, $\Pi \Box \sigma \vdash_{\mathcal{FD}^{\mathcal{T}}} \blacksquare$

Table B.1: Store Transformation Rules for $\vdash_{\mathcal{FD}^{\mathcal{T}}}$

As semantic results, the thesis presents soundness and limited completeness of the glass box solver $\mathcal{FD}^{\mathcal{T}}$. Regarding the black-box solver $\mathcal{FD}^{N}$, we can assume that $solve^{\mathcal{FD}^{N}}$ reduces primitive constraints to a *solved form*, in the sense that they can not be further reduced. We assume that $solve^{\mathcal{FD}^{N}}$ is sound, and the completeness is limited by the ECL^{i}PS^e solver.

Therefore, both SICStus and ECL^{i}PS^e solvers are not able to detect that the next goal fails, and the suspended constraints are just returned as answer instead:

```
domain [X,Y] 1 100000, X #<= Y, Y #<= X, X /= Y
```

The $\mathcal{FD}$ solver propagates the constraints but domains can not be reduced. Then, it is necessary to use the `labeling` constraint in order to check the infeasibility of the constraints.

```
domain [X,Y] 1 100000, X #<= Y, Y #<= X, X /= Y, labeling [] [X,Y]
```

The process of labeling the variables is in the complexity order $\mathcal{O}(\mathtt{N} * \mathtt{M})$ where `N` and `M` are the sizes of the domains of the variables `X` and `Y`, respectively. However, $\mathcal{TOY}$ anticipates the failure detecting the inconsistency of constraints `X # <= Y, Y # <= X, X /= Y`, and reducing the cost from order $\mathcal{O}(\mathtt{N} * \mathtt{M})$ to $\mathcal{O}(1)$. This mechanism is activated in $\mathcal{TOY}$ by the command `/fs`.

B.6.3 Constraint Domain and Solver for the $\mathcal{FS}$ Domain

The $\mathcal{FS}$ domain has the signature $\Sigma_{\mathcal{FS}} = \langle TC,\ SBT_{\mathcal{FS}},\ DC,\ DF,\ SPF_{\mathcal{FS}} \rangle$ where $SBT_{\mathcal{FS}}$ = {`elem`, `set`}, and the set of base values of base type `elem` is a denumerable set $\mathcal{B}^{\mathcal{FS}}_{\mathtt{elem}}$ with a strict total order $\prec$, and the set of base values for `set` contains all finite subsets of $\mathcal{B}^{\mathcal{FS}}_{\mathtt{elem}}$, $\mathcal{B}^{\mathcal{FS}}_{\mathtt{set}} = \mathcal{P}_f(\mathcal{B}^{\mathcal{FS}}_{\mathtt{elem}})$. A set of elements of type `elem` is represented as $\{e_1, \ldots, e_n\}$ where $e_1 \prec \ldots \prec e_n$. Primitive functions in $SPF_{\mathcal{FS}}$ are the following:

- `== :: A -> A -> bool`
- `domainSets :: [set] -> set -> set -> bool`
- `intSets L A B :: [set] -> int -> int -> bool`
- `intSet :: set -> int -> int -> bool`
- `subset, superset :: set -> set -> bool`
- `intersect, union, diff, symdiff :: set -> set -> set -> bool`
- `intersections, unions :: [set] -> set -> bool`
- `disjoints :: [set] -> bool`
- `isIn, isNotIn :: int -> set -> bool`
- `labelingSets :: [set] -> bool`

`domainSets` constrains the domain of each set variable of the list w.r.t. a given lower and upper set bounds forming a lattice of possible values for each variable. `intSets L A B` is a particular case of `domainSets`, where the lower bound is the empty set and the upper bound is a set whose range is limited by the second and the third arguments. For example, if the elements are integers, `intSets [S] 3 6` defines the set `S` as the set in the lattice `{}` and `{3,4,5,6}`. `intSet :: set -> elem -> elem -> bool` is the previous function simplified to a single set. `subset` and `superset` correspond to usual set inclusion operations. `subset A B` and `superset B A` impose `A` $\subseteq$ `B`. `intersect A B C` imposes `C` = `A` $\cap$ `B`, `union A B C`

imposes `C` = `A` ∪ `B`, `diff A B C` imposes `C` = `A` \ `B`, and `symdiff A B C` imposes `C` = `A` \ `B` ∪ `B` \ `A`. `intersections` and `unions` extend the corresponding previous functions to a set lists. `disjoints` constrains the list of sets to have no element in common. `isIn, isNotIn :: elem -> set -> bool` constrain an element to be, or not to be, resp., a member of a set. `labelingSets` enumerates all ground instantiations of each set expression in the list.

In addition to the primitives defined above for the domain $\mathcal{FS}$, we define a new primitive constraint `<<` that has no correspondence with the primitive constraints of the system ECL^{i}PS^e Prolog:

- `<< :: set -> set -> bool`.

This operator is related to the strict partial order between two finite set variables. In particular, S_1 `<<` S_2 means that the greatest element in the set S_1 is lower than the lowest element in the set S_2.

The $\mathcal{FS}$ solver for $\mathcal{TOY}$ has been developed on top of the solver for sets of integers available in ECL^{i}PS^e, and has been extended with several additional features, such as disequality handling and constraint deduction. For this reason, the $\mathcal{FS}$ solver has been split in two different layers: First, a glass-box solver $\mathcal{FS}^{\mathcal{T}}$ in $\mathcal{TOY}$ for dealing with those additional features, and second, the black-box solver $\mathcal{FS}^{N}$ available in ECL^{i}PS^e. Observe that, in the formal description of the $\mathcal{TOY}$ extension, a general type `elem` is used, but ECL^{i}PS^e sets are only allowed to contain integer elements. It can be easily extended to any other denumerable domain by means of a bijection into $\mathbb{Z}$. In what follows, we will assume that `elem = int`.

The glass-box solver $\mathcal{FS}^{\mathcal{T}}$ has been formalized using a store transformation system with the store transformation rules described in Table B.2. Rules from **S1** to **S10** infer new constraints in order to generate either equalities (`==`), disequalities (`/=`), or `subset` constraints that can anticipate failure by means of the rule **S11**. For example, **S1** propagates the information S_1 `/= {}`,..., S_n `/= {}` when the constraint `domainSets` $[S_1, \ldots, S_n]$ `lb ub` is processed and `lb` is not empty. Rules **S2**, **S3** and **S10** also infer that some sets are different to the empty set. Rules **S4**, **S5**, and **S8** infer equalities, and rules **S6**, **S7**, and **S9**, infer `subset` constraints. Additional rules based on set theory could be added.

$\mathcal{FS}^{\mathcal{T}}$ complements the solver already existing in ECL^{i}PS^e, especially for anticipating failure by handing disequalities. For instance, $\mathcal{TOY}$ detects inconsistency in the goal `intSets [X,Y] 1 10000, subset X Y, subset Y X, X /= Y` using rules **S5** and **S11**, whilst other systems, such as ECL^{i}PS^e, delay the detection of this inconsistency until labeling variables.

As semantic results, the thesis presents soundness and limited completeness of the glass box $\mathcal{FS}^{\mathcal{T}}$ solver. Regarding the black-box solver $\mathcal{FS}^{N}$, we can assume that $solve^{\mathcal{FS}^N}$ reduces primitive constraints to a *solved form*, in the sense that they can not be further reduced. We also assume that $solve^{\mathcal{FS}^N}$ is sound, and the completeness result is limited by ECL^{i}PS^e solver.

B.7 The Coordination Domain $\mathcal{C}$ and the Calculus $CCLNC(\mathcal{C})$

With the pure domains $\mathcal{D}_1, \ldots, \mathcal{D}_n$ it is possible to build the amalgamated sum $\mathcal{C} = \mathcal{M} \oplus \mathcal{D}_1 \oplus \cdots \oplus \mathcal{D}_n$ [EFH$^+$09]. This 'hybrid' domain supports the communication among the

S1	`domainSets` $[S_1, \ldots, S_n]$ `lb ub`, `lb/={}`, $\Pi \Box \sigma \Vdash_{\mathcal{FS}^\mathcal{T}}$
	`domainSets` $[S_1, \ldots, S_n]$ `lb ub`, `lb/={}`, S_1`/={},`$\ldots$, S_n `/= {}`, $\Pi \Box \sigma$
	si $S_1, \ldots, S_n \in \mathcal{V}ar$
S2	`isIn` X S, $\Pi \Box \sigma \Vdash_{\mathcal{FS}^\mathcal{T}}$ `isIn` X S, S `/= {}`, $\Pi \Box \sigma$
S3	`subset` S_1 S_2, S_1 `/= {}`, $\Pi \Box \sigma \Vdash_{\mathcal{FS}^\mathcal{T}}$
	`subset` S_1 S_2, S_1 `/= {}`, S_2 `/= {}`, $\Pi \Box \sigma$
S4	`subset` S_1 S_2, S_2 `== {}`, $\Pi \Box \sigma \Vdash_{\mathcal{FS}^\mathcal{T}}$ S_2 `== {}`, S_1 `== {}`, $\Pi \Box \sigma$
S5	`subset` S_1 S_2, `subset` S_2 S_1, $\Pi \Box \sigma \Vdash_{\mathcal{FS}^\mathcal{T}}$ S_1 `==` S_2, $\Pi \Box \sigma$
S6	`union` S_1 S_2 S_1, $\Pi \Box \sigma \Vdash_{\mathcal{FS}^\mathcal{T}}$ `union` S_1 S_2 S_1, `subset` S_2 S_1, $\Pi \Box \sigma$
S7	`union` S_1 S_2 S_2, $\Pi \Box \sigma \Vdash_{\mathcal{FS}^\mathcal{T}}$ `union` S_1 S_2 S_2, `subset` S_1 S_2, $\Pi \Box \sigma$
S8	`union` S_1 S_2 S_3, S_3 `== {}`, $\Pi \Box \sigma \Vdash_{\mathcal{FS}^\mathcal{T}}$
	S_1 `== {}`, S_2 `== {}`, S_3 `== {}` $\Pi \Box \sigma$
S9	`union` S_1 S_2 S_3 $\Pi \Box \sigma \Vdash_{\mathcal{FS}^\mathcal{T}}$
	`union` S_1 S_2 S_3, `subset` S_1 S_3, `subset` S_2 S_3, $\Pi \Box \sigma$
S10	`intersect` S_1 S_2 S_3, S_3`/={}`, $\Pi \Box \sigma \Vdash_{\mathcal{FS}^\mathcal{T}}$
	`intersect` S_1 S_2 S_3, S_3`/={}`, S_1 `/= {}`, S_2 `/= {}`, $\Pi \Box \sigma$
S11	S_1 `==` S_2, S_1 `/=` S_2, $\Pi \Box \sigma \Vdash_{\mathcal{FS}^\mathcal{T}}$ ■

Table B.2: Store Transformation Rules for $\Vdash_{\mathcal{FS}^\mathcal{T}}$

domains $\mathcal{D}_i$ via bridge constraints provided by the *mediatorial domain* $\mathcal{M}$. Therefore, $\mathcal{M}$ supplies communication among the domains $\mathcal{D}_1, \ldots, \mathcal{D}_n$.

In practice, it is advisable to include the Herbrand domain $\mathcal{H}$ as one of the component domains $\mathcal{D}_i$ when building a coordination domain $\mathcal{C}$. In application programs over such a coordination domain, the $\mathcal{H}$ solver is typically helpful for solving symbolic equality and disequality constraints over user defined datatypes. The solvers of other component domains $\mathcal{D}_i$ whose specific signatures include the primitive `==` may be helpful for computing with equalities and disequalities related to $\mathcal{D}_i$'s specific base types.

The framework for cooperative programming and the cooperative goal solving calculus $CCLNC(\mathcal{C})$ presented in Section B.9 essentially rely on the coordination domain $\mathcal{C}$, and the instance $CFLP(\mathcal{C})$ of the $CFLP$ scheme [LRV07], and provides a declarative semantics for proving the soundness and completeness of $CCLNC(\mathcal{C})$.

B.8 The Coordination Domain $\mathcal{C}_{\mathcal{FD}.\mathcal{R}}$

In this subsection, we explain the construction of a coordination domain for cooperation among the three pure domains $\mathcal{H}$, $\mathcal{FD}$ and $\mathcal{R}$.

First, we define a mediatorial domain $\mathcal{M}$ suitable to this purpose. The notation $\mathcal{M}$ is used instead of $\mathcal{M}_{\mathcal{FD}.\mathcal{R}}$ to simplify the presentation. This mediatorial domain is built with specific signature $\langle TC, SBT_\mathcal{M}, DC, DF, SPF_\mathcal{M} \rangle$, where $SBT_\mathcal{M} = \{int, real\}$ and $SPF_\mathcal{M} = \{$`#==`$\}$. The equivalence primitive `#==` is interpreted with respect to the total injective mapping

$inj_{int,real} :: \mathbb{Z} \to \mathbb{R}$, which maps each integer value into an equivalent real value.

Next, we use this mediatorial domain for building the coordination domain $\mathcal{C}_{\mathcal{FD}.\mathcal{R}} = \mathcal{M} \oplus \mathcal{H} \oplus \mathcal{FD} \oplus \mathcal{R}$. Note that bridges `X #== RX` and antibridges `X #/== RX` can be useful just as constraints; in particular, `X #== RX` acts as an *integrality constraint* over the value of the real variable `RX`. More importantly, the mediatorial domain $\mathcal{C}_{\mathcal{FD}.\mathcal{R}}$ will serve as a basis for useful cooperation facilities, including the projection of $\mathcal{R}$ constraints to the $\mathcal{FD}$ solver (and vice versa) using bridges, the specialization of $\mathcal{H}$ constraints to become $\mathcal{R}$-specific or $\mathcal{FD}$-specific in some computational contexts, and some other special mechanisms designed for processing the mediatorial constraints occurring in computations. In particular, computation rules for simplifying mediatorial constraints will be needed.

Although $\mathcal{M}$ is not a pure domain, simplifying $\mathcal{M}$-constraints is most conveniently thought of as the task of a $\mathcal{M}$-solver. This solver is expected to deal with $\mathcal{M}$-specific constraint sets $\Pi \subseteq APCon_{\mathcal{M}}$ consisting of atomic primitive constraints π of the form t `#==`$s \to\!! b$, where b is either a variable or a Boolean constant and each of the two patterns t and s is either a variable or a numeric value of the proper type (`int` for t and `real` for s).

We define a glass-box solver $solve^{\mathcal{M}}$ by means of the store transformation technique, using the store transformation rules for $\mathcal{M}$-stores shown in Table B.3. One-step transformations of $\mathcal{M}$-stores have the form $\pi, \Pi \,\square\, \sigma \vdash_{\mathcal{M}} \Pi' \,\square\, \sigma'$, indicating the transformation of any store $\pi, \Pi \,\square\, \sigma$, which includes the atomic constraint π plus other constraints Π; no sequential ordering is intended. We say that π is the *selected atomic constraint* for this transformation step.

The thesis ensures in particular that the store transformation rules for $\mathcal{M}$-stores can be accepted as a correct specification of a glass-box solver for the domain $\mathcal{M}$.

As we will see, some cooperative goal solving rules in $CCLNC(\mathcal{C}_{\mathcal{FD}.\mathcal{R}})$ rely on the identification of certain atomic primitive Herbrand constraints π as $\mathcal{FD}$-specific or $\mathcal{R}$-specific, respectively, on the basis of the mediatorial constraints available in a given $\mathcal{M}$-store M. The notations $M \vdash \pi$ `in` $\mathcal{FD}$ and $M \vdash \pi$ `in` $\mathcal{R}$ defined below serve to this purpose.

Definición 12 (Inference of domain-specific extended Herbrand constraints)**.** Assume a mediatorial store M and a well-typed atomic extended Herbrand constraint π having the form t_1 `==` t_2 or t_1 `/=` t_2, where each of the two patterns t_1 and t_2 is either a numeric constant v or a variable V. Then we define:

1. $M \vdash \pi$ `in` $\mathcal{FD}$ (read as 'M allows to infer that π is $\mathcal{FD}$-specific') iff some of the three following conditions holds:
   (a) Either t_1 or t_2 is an integer constant, or
   (b) t_1 or t_2 is a variable that occurs as the left argument of the operator `#==` within some mediatorial constraint belonging to M, or
   (c) t_1 or t_2 is a variable that has been recognized to have type `int` by some implementation dependent device.
2. $M \vdash \pi$ `in` $\mathcal{R}$ (read as 'M allows to infer that π is $\mathcal{R}$-specific') iff some of the three following conditions holds:

M1 $(t \texttt{ \#== } s) \rightarrow\!!\ B,\ \Pi \square \sigma \vdash_{\mathcal{M}} (t \texttt{ \#== } s,\ \Pi)\sigma_1 \square \sigma\sigma_1$
if $t \in \mathcal{V}ar \cup \mathbb{Z}$, $s \in \mathcal{V}ar \cup \mathbb{R}$, $B \in \mathcal{V}ar$, where $\sigma_1 = \{B \mapsto true\}$.

M2 $(t \texttt{ \#== } s) \rightarrow\!!\ B,\ \Pi \square \sigma \vdash_{\mathcal{M}} (t \texttt{ \#/== } s,\ \Pi)\sigma_1 \square \sigma\sigma_1$
if $t \in \mathcal{V}ar \cup \mathbb{Z}$, $s \in \mathcal{V}ar \cup \mathbb{R}$, $B \in \mathcal{V}ar$, where $\sigma_1 = \{B \mapsto false\}$.

M3 $X \texttt{ \#==} u',\ \Pi \square \sigma \vdash_{\mathcal{M}} \Pi\sigma_1 \square \sigma\sigma_1$
if $u' \in \mathbb{R}$, and there is $u \in \mathbb{Z}$ such that $u \texttt{ \#==}^{\mathcal{M}} u' \rightarrow true$, where $\sigma_1 = \{X \mapsto u\}$.

M4 $X \texttt{ \#== } u',\ \Pi \square \sigma \vdash_{\mathcal{M}} \blacksquare$
if $u' \in \mathbb{R}$, and there is no $u \in \mathbb{Z}$ such that $u \texttt{ \#==}^{\mathcal{M}} u' \rightarrow true$.

M5 $u \texttt{ \#== } RX,\ \Pi \square \sigma \vdash_{\mathcal{M}} \Pi\sigma_1 \square \sigma\sigma_1$
if $u \in \mathbb{Z}$ and $u' \in \mathbb{R}$ is so chosen that $u \texttt{ \#==}^{\mathcal{M}} u' \rightarrow true$, where $\sigma_1 = \{RX \mapsto u'\}$.

M6 $u \texttt{ \#== } u',\ \Pi \square \sigma \vdash_{\mathcal{M}} \Pi \square \sigma$ if $u \in \mathbb{Z}$, $u' \in \mathbb{R}$, and $u \texttt{ \#==}^{\mathcal{M}} u' \rightarrow true$.

M7 $u \texttt{ \#== } u',\ \Pi \square \sigma \vdash_{\mathcal{M}} \blacksquare$ if $u \in \mathbb{Z}$, $u' \in \mathbb{R}$, and $u \texttt{ \#==}^{\mathcal{M}} u' \rightarrow false$.

M8 $u \texttt{ \#/== } u',\ \Pi \square \sigma \vdash_{\mathcal{M}} \Pi \square \sigma$ if $u \in \mathbb{Z}$, $u' \in \mathbb{R}$, and $u \texttt{ \#==}^{\mathcal{M}} u' \rightarrow false$

M9 $u \texttt{ \#/== } u',\ M \square \sigma \vdash_{\mathcal{M}} \blacksquare$ if $u \in \mathbb{Z}$, $u' \in \mathbb{R}$, and $u \texttt{ \#==}^{\mathcal{M}} u' \rightarrow true$.

Table B.3: Store Transformations for $solve^{\mathcal{M}}$

(a) Either t_1 or t_2 is a real constant, or

(b) t_1 or t_2 is a variable that occurs as the right argument of the operator `#==` within some mediatorial constraint belonging to M, or

(c) t_1 or t_2 is a variable that has been recognized to have type `real` by some implementation dependent device.

B.9 Cooperative Programming and Goal Solving in $CFLP(\mathcal{C}_{\mathcal{FD}.\mathcal{R}})$

This section presents the cooperative computation model for goal solving. After introducing programs and goals in the first subsection, the subsequent subsections deal with goal solving rules, an illustrative computation example, and results concerning the formal properties of the computation model.

Our goal solving rules work by transforming initial goals into final goals in solved form which serve as computed answers, as in the previously published *Constrained Lazy Narrowing Calculus* $CLNC(\mathcal{D})$ [LRV04], which works over any parametrically given domain $\mathcal{D}$ equipped with a solver. We have substantially extended $CLNC(\mathcal{D})$ with various mechanisms for do-

main cooperation via bridges, projections and some more *ad hoc* operations. The result is a *Cooperative Constrained Lazy Narrowing Calculus* $CCLNC(\mathcal{C})$ which is sound and complete (with some limitations) w.r.t. the instance $CFLP(\mathcal{C})$ of the generic $CFLP$ scheme [LRV07]. For the sake of simplicity, we have restricted our presentation of $CCLNC(\mathcal{C})$ to the cases $\mathcal{C}_{\mathcal{FD}.\mathcal{R}} = \mathcal{M} \oplus \mathcal{H} \oplus \mathcal{FD} \oplus \mathcal{R}$ and $\mathcal{C}_{\mathcal{FD}.\mathcal{FS}} = \mathcal{M} \oplus \mathcal{H} \oplus \mathcal{FD} \oplus \mathcal{FS}$ although it could be easily extended to other coordination domains.

B.9.1 Programs and Goals

Programs are sets of constrained rewrite rules that define the behaviour of possibly higher-order and/or non-deterministic lazy functions, called *program rules*. As mentioned before, a program rule Rl for a defined function symbol $f \in DF^n_\Sigma$ with principal type $f :: \overline{\tau}_n \to \tau$ has the form $f\,\overline{t}_n \to r \Leftarrow \Delta$, where $\overline{t}_n$ is a linear sequence of patterns, r is an expression, and Δ is a finite conjunction $\delta_1, \ldots, \delta_m$ of atomic constraints. Each program rule Rl is required to be well-typed, i.e., there must exist some type environment Γ for the variables occurring in Rl such that $\Sigma, \Gamma \vdash_{WT} t_i :: \tau_i$ for all $1 \leq i \leq n$, $\Sigma, \Gamma \vdash_{WT} r :: \tau$ and $\Sigma, \Gamma \vdash_{WT} \delta_i$ for all $1 \leq i \leq m$.

The left-linearity requirement is quite common in functional and functional logic programming [Han97]. As in constraint logic programming, the conditional part of a program rule needs no explicit occurrences of existential quantifiers. A program rule Rl is said to include *free occurrences of higher-order logic variables* iff there is some variable X which does not occur in the left-hand side of Rl but has some occurrence in a context of the form $X\,\overline{e}_m$ (with $m > 0$) somewhere else in Rl. A program $\mathcal{P}$ includes free occurrences of higher-order logic variables iff some of the program rules in $\mathcal{P}$ does. The design of $CCLNC(\mathcal{C})$ is tailored to programs and goals having non-free occurrences of higher-order logic variables. As shown in [GHR01], goal solving rules for dealing with free higher-order logic variables give rise to ill-typed solutions very often. If desired, they could be easily added to our present setting.

As in functional languages such as Haskell [Pey02], our program rules can deal with higher-order functions and are not expected to be always terminating. Moreover, in contrast to Haskell and most other functional languages, we do not require program rules to be confluent. Therefore, some program defined functions can be *non-deterministic* and return several values for a fixed choice of arguments in some cases.

Programs are used to solve *goals* using a cooperative goal solving calculus which will be described below. Goals over the coordination domain $\mathcal{C}_{\mathcal{FD}.\mathcal{R}}$ have the general form $G \equiv \exists\overline{U}.\ P \,\square\, C \,\square\, M \,\square\, H \,\square\, F \,\square\, R$, where the symbol $\square$ is interpreted as conjunction and:

- $\overline{U}$ is a finite set of so-called *existential variables*, intended to represent local variables in the computation.
- P is a set of so-called *productions* of the form $e_1 \to t_1, \ldots, e_m \to t_m$, where $e_i \in Exp_{\mathcal{C}}$ and $t_i \in Pat_{\mathcal{C}}$ for all $1 \leq i \leq m$. The set of *produced variables* of G is defined as the set $pvar(P)$ of variables occurring in $t_1 \ldots t_m$. During goal solving, productions are used to compute values for the produced variables insofar as demanded, using the goal solving rules for constrained lazy narrowing presented in Subsection B.9.2.

- C is the so-called *constraint pool*, a finite set of constraints to be solved, possibly including active occurrences of defined functions symbols.
- $M = \Pi_M \,\square\, \sigma_M$ is the *mediatorial store*, including a finite set of atomic primitive constraints $\Pi_M \subseteq APCon_{\mathcal{M}}$ and a substitution σ_M. We will write $B_M \subseteq \Pi_M$ for the set of all $\pi \in \Pi_M$ which are *bridges* t `#==` s, where each of the two patterns t and s can be either a variable or a numeric constant.
- $H = \Pi_H \,\square\, \sigma_H$ is the *Herbrand store*, including a finite set of atomic primitive constraints $\Pi_H \subseteq APCon_{\mathcal{H}}$ and a substitution σ_H.
- $F = \Pi_F \,\square\, \sigma_F$ is the *finite domain store*, including a finite set of atomic primitive constraints $\Pi_F \subseteq APCon_{\mathcal{FD}}$ and a substitution σ_F.
- $R = \Pi_R \,\square\, \sigma_R$ is the *real arithmetic store*, including a finite set of atomic primitive constraints $\Pi_R \subseteq APCon_{\mathcal{R}}$ and a substitution σ_R.

A goal G is said to have *free occurrences of higher-order logic variables* iff there is some variable X occurring in G in some context of the form $X\,\overline{e}_m$, with $m > 0$. Two special kinds of goals are particularly interesting. *Initial goals* just consist of a well-typed constraint pool C. More precisely, the existential prefix $\overline{U}$, productions in P, and stores M, H, F and R are empty. *Solved goals* (also called *solved forms*) have empty P and C parts and cannot be transformed by any goal solving rule. Therefore, they are used to represent *computed answers*. We will also write ■ to denote an *inconsistent goal*.

Example 2. Initial and Solved Goals

Consider the initial goals **Goal 1**, **Goal 2** and **Goal 3** presented in $\mathcal{TOY}$ syntax in Subsection B.5, for the choice `d = 2`, `n = 4`. When written with the abstract syntax for general $CFLP(\mathcal{C}_{\mathcal{FD}.\mathcal{R}})$-goals they become

1) □ bothIn (triangle (2, 2.75) 4 0.5) (square 4) (X,Y) □□□□
2) □ bothIn (triangle (2, 2.5) 2 1) (square 4) (X,Y) □□□□
3) □ bothIn (triangle (2, 2.5) 8 1) (square 4) (X,Y) □□□□

The resolution of these example goals in our cooperative goal solving calculus $CCLNC(\mathcal{C}_{\mathcal{FD}.\mathcal{R}})$ will be discussed in detail in Subsection 5. The respective solved forms obtained as computed answers (restricted to the variables in the initial goal) will be:

1) ■
2) □□□□ ($\Diamond$ □ $\{X \mapsto 2, Y \mapsto 2\}$) □
3) □□□□ ($\Diamond$ □ $\{X \mapsto 0, Y \mapsto 2\}$) □
   □□□□ ($\Diamond$ □ $\{X \mapsto 1, Y \mapsto 2\}$) □
   □□□□ ($\Diamond$ □ $\{X \mapsto 2, Y \mapsto 2\}$) □
   □□□□ ($\Diamond$ □ $\{X \mapsto 3, Y \mapsto 2\}$) □
   □□□□ ($\Diamond$ □ $\{X \mapsto 4, Y \mapsto 2\}$) □

Next subsections present the goal solving rules that have been designed as an extension of the goal solving calculus for the $CFLP$ scheme [LRV04, LRV07].

B.9.2 Constrained Lazy Narrowing Rules

Rules displayed in Table B.4 model the behaviour of constrained lazy narrowing ignoring domain cooperation and solver invocation. They have been adapted from [LRV04] and can be classified as follows: The first four rules encode unification transformations similar to those found in the $\mathcal{H}$ *sts* and other related formalisms; rule **EL** discards unneeded suspensions, rule **DF** (presented in two cases in order to optimize the $k = 0$ case) applies program rules to deal with calls to program defined functions; rule **PC** transforms demanded calls to primitive functions into atomic constraints that are placed in the pool; and rule **FC**, working in interplay with **PC**, transforms the atomic constraints in the pool into a flattened form consisting of a conjunction of atomic primitive constraints with new existential variables.

The behaviour of the main rules in Table B.4 will be illustrated in Subsection 5. Example 3 below focuses on the transformation rules **PC** and **FC**. Their iterated application flattens the atomic $\mathcal{R}$-constraint `(RX + 2*RY)*RZ <= 3.5` into a conjunction of four atomic primitive $\mathcal{R}$-constraints involving three new existential variables, that are placed in the constraint pool. Note that [LRV04] and other previous related calculi also include rules that can be used to achieve constraint flattening, but the resulting atomic primitive constraints are placed in a constraint store. In our present setting, they are kept in the pool in order that the domain cooperation rules described in the next subsection can process them.

Example 3. Constraint Flattening

$\square\ \underline{(RX + 2 * RY) * RZ \texttt{ <= } 3.5}\ \square\square\square\square \vdash_{\mathbf{FC}}$

$\exists RA.\ \underline{(RX + 2 * RY) * RZ \ \rightarrow\ RA}\ \square\ RA \texttt{ <= } 3.5\ \square\square\square\square \vdash_{\mathbf{PC}}$

$\exists RA.\ \square\ \underline{(RX + 2 * RY) * RZ \ \rightarrow!\ RA}, RA \texttt{ <= } 3.5\ \square\square\square\square \vdash_{\mathbf{FC}}$

$\exists RB,\ RA. \underline{RX + 2 * RY \rightarrow RB}\ \square\ RB * RZ \rightarrow! RA, RA \texttt{ <= } 3.5\ \square\square\square\square \vdash_{\mathbf{PC}}$

$\exists RB, RA.\ \square\ \underline{RX + 2 * RY \ \rightarrow!\ RB}, RB * RZ \rightarrow!\ RA, RA \texttt{ <= } 3.5\ \square\square\square\square \vdash_{\mathbf{FC}}$

$\exists RC, RB, RA. \underline{2 * RY \rightarrow RC}\ \square\ RX + RC \rightarrow! RB, RB * RZ \rightarrow! RA, RA \texttt{ <= } 3.5\ \square\square\square\square \vdash_{\mathbf{PC}}$

$\exists RC, RB, RA.\ \square\ 2 * RY \rightarrow! RC, RX + RC \rightarrow! RB, RB * RZ \rightarrow! RA, RA \texttt{ <= } 3.5\ \square\square\square\square$

Note that suspensions $e \rightarrow X$ can be discarded by rule **EL** in case that X does not occur in the rest of the goal. Otherwise, they must wait until X gets bound to a non-variable pattern or becomes obviously demanded, and then they can be processed by using either rule **DF** or rule **PC**, according to the syntactic form of e. Moreover, all the substitutions produced by the transformations bind variables X to patterns t, standing for computed values that are shared by all the occurrences of t in the current goal. In this way, the goal transformation rules encode a lazy narrowing strategy.

The goal solving rules presented in the rest of this section has been designed as an extension of an existing goal solving calculus for the $CFLP$ scheme [LRV04], adding the new features needed to support solver coordination via bridge constraints.

B.9.3 Domain Cooperation Rules

This subsection presents the goal transformation rules in $CCLNC(\mathcal{C}_{\mathcal{FD}.\mathcal{R}})$ which take care of domain cooperation. The core of the subsection deals with bridges, which allow to ex-

DC DeComposition

$\exists\overline{U}.\ h\ \overline{e_m} \to h\ \overline{t_m},\ P \sqcap C \sqcap M \sqcap H \sqcap F \sqcap R \vdash_{\mathbf{DC}} \exists\overline{U}.\ \overline{e_m \to t_m},\ P \sqcap C \sqcap M \sqcap H \sqcap F \sqcap R$

CF Conflict Failure

$\exists\overline{U}.\ e \to t, P \sqcap C \sqcap M \sqcap H \sqcap F \sqcap R \vdash_{\mathbf{CF}} \blacksquare$

If e is rigid and passive, $t \notin \mathcal{V}ar$, e and t have conflicting roots.

SP Simple Production

$\exists\overline{U}.\ s \to t,\ P \sqcap C \sqcap M \sqcap H \sqcap F \sqcap R \vdash_{\mathbf{SP}} \exists\overline{U}'.\ (P \sqcap C \sqcap M \sqcap H \sqcap F \sqcap R)@_{\mathcal{H}}\sigma'$

If $s = X \in \mathcal{V}ar$, $t \notin \mathcal{V}ar$, $\sigma' = \{X \mapsto t\}$ and $\overline{U'} = \overline{U}$ or else $s \in Pat_{\mathcal{C}}$, $t = X \in \mathcal{V}ar$, $\sigma' = \{X \mapsto s\}$ and $\overline{U}' = \overline{U}\backslash\{X\}$.

IM IMitation

$\exists X, \overline{U}.h\overline{e_m} \to X, P \sqcap C \sqcap M \sqcap H \sqcap F \sqcap R \vdash_{\mathbf{IM}} \exists\overline{X_m}, \overline{U}.(\overline{e_m \to X_m}, P \sqcap C \sqcap M \sqcap H \sqcap F \sqcap R)\sigma'$

If $h\ \overline{e_m} \notin Pat_{\mathcal{C}}$ is passive, $X \in odvar(G)$ and $\sigma' = \{X \mapsto h\ \overline{X_m}\}$.

EL ELimination

$\exists X, \overline{U}.\ e \to X,\ P \sqcap C \sqcap M \sqcap H \sqcap F \sqcap R \vdash_{\mathbf{EL}} \exists\overline{U}.\ P \sqcap C \sqcap M \sqcap H \sqcap F \sqcap R$

If X does not occur in the rest of the goal.

DF Defined Function

$\exists\overline{U}.\ f\ \overline{e_n} \to t,\ P \sqcap C \sqcap M \sqcap H \sqcap F \sqcap R \vdash_{\mathbf{DF}_f}$
$\exists\overline{Y}, \overline{U}.\ \overline{e_n \to t_n}, r \to t,\ P \sqcap C',\ C \sqcap M \sqcap H \sqcap F \sqcap R$

If $f \in DF^n$, $t \notin \mathcal{V}ar$ or $t \in odvar(G)$ and $Rl : f\ \overline{t_n} \to r \Leftarrow C'$ is a fresh variant of a rule in $\mathcal{P}$, with $\overline{Y} = var(Rl)$ new variables.

$\exists\overline{U}.\ f\ \overline{e_n a_k} \to t,\ P \sqcap C \sqcap M \sqcap H \sqcap F \sqcap R \vdash_{\mathbf{DF}_f}$
$\exists X, \overline{Y}, \overline{U}.\ \overline{e_n \to t_n}, r \to X, X\ \overline{a_k} \to t,\ P \sqcap C',\ C \sqcap M \sqcap H \sqcap F \sqcap R$

If $f \in DF^n$ $(k > 0)$, $t \notin \mathcal{V}ar$ or $t \in odvar(G)$ and $Rl : f\ \overline{t_n} \to r \Leftarrow C'$ is a fresh variant of a rule in $\mathcal{P}$, with $\overline{Y} = var(Rl)$ and X new variables.

PC Place Constraint

$\exists\overline{U}.\ p\ \overline{e_n} \to t,\ P \sqcap C \sqcap M \sqcap H \sqcap F \sqcap R \vdash_{\mathbf{PC}} \exists\overline{U}.\ \ P \sqcap p\ \overline{e_n} \to!\ t,\ C \sqcap M \sqcap H \sqcap F \sqcap R$

If $p \in PF^n$ and $t \notin \mathcal{V}ar$ or $t \in odvar(G)$.

FC Flatten Constraint

$\exists\overline{U}.\ P \sqcap p\ \overline{e_n} \to!\ t,\ C \sqcap M \sqcap H \sqcap F \sqcap R \vdash_{\mathbf{FC}}$
$\exists\overline{V}_m, \overline{U}.\ \overline{a_m \to V_m}, P \sqcap p\ \bar{t}_n \to!\ t,\ C \sqcap M \sqcap H \sqcap F \sqcap R$

If $p \in PF^n$, some $e_i \notin Pat_{\mathcal{C}}$, $\overline{a_m}$ $(m \leq n)$ are those e_i which are not patterns, $\overline{V_m}$ are new variables, and $p\ \overline{t_n}$ is obtained from $p\ \overline{e_n}$ by replacing each e_i which is not a pattern by V_i.

Table B.4: Rules for Constrained Lazy Narrowing

change information between pure domains, and projections, which take place whenever a constraint is posted to its corresponding solver. This process builds mate constraints considering the available bridge constraints, and posts them to the mate solver. A few more *ad hoc* cooperation rules are presented at the end of this subsection.

Given a goal G whose pool C includes an atomic primitive constraint $\pi \in APCon_{\mathcal{FD}}$ and whose mediatorial store M includes a set of bridges B_M, we will consider three possible goal transformations intended to convey useful information from π to the $\mathcal{R}$-solver:

- To compute new bridges $bridges^{\mathcal{FD}\to\mathcal{R}}(\pi, B_M)$ to add to M, by means of a *bridge generation* operation $bridges^{\mathcal{FD}\to\mathcal{R}}$ defined to this purpose.
- To compute projected $\mathcal{R}$-constraints $proj^{\mathcal{FD}\to\mathcal{R}}(\pi, B_M)$ to be added to R, by means of a *projection* operation $proj^{\mathcal{FD}\to\mathcal{R}}$ defined to this purpose.
- To place π into the $\mathcal{FD}$ store F.

Similar goal transformations based on two operations $bridges^{\mathcal{R}\to\mathcal{FD}}$ and $proj^{\mathcal{R}\to\mathcal{FD}}$ can be used to convey useful information from a primitive atomic constraint $\pi \in PCon_{\mathcal{R}}$ to the $\mathcal{FD}$-solver. Rules **SB**, **PP** and **SC** in Table B.5 formalize these transformations, while tables B.6 and B.7 give an effective specification of the bridge generation and projection operations.

The formulation of **SB**, **PP** and **SC** in Table B.5 relies on the identification of certain atomic primitive Herbrand constraints π as $\mathcal{FD}$-specific or $\mathcal{R}$-specific, as indicated by the notations $M \vdash \pi$ `in` $\mathcal{FD}$ and $M \vdash \pi$ `in` $\mathcal{R}$, definition 12. The notation Π, S is used at several places to indicate the new store obtained by adding the set of constraints Π to the constraints within store S. The notation π, S (where π is a single constraint) must be understood similarly. In practice, **SB**, **PP** and **SC** are best applied in this order. Note that **PP** places the projected constraints in their corresponding stores, while constraints in the pool that are not useful anymore for computing additional bridges or projections will be eventually placed into their stores by means of transformation **SC**.

The functions $bridges^{\mathcal{C}\to\mathcal{C}'}$ and $proj^{\mathcal{C}\to\mathcal{C}'}$ are specified in Table B.6 for the case $\mathcal{C} = \mathcal{FD}$, $\mathcal{C}' = \mathcal{R}$ and in Table B.7 for the case $\mathcal{C} = \mathcal{R}$, $\mathcal{C}' = \mathcal{FD}$. Note that the primitive `#/` is not considered in Table B.6 because integer division constraints cannot be projected into real division constraints. The notations $\lceil a \rceil$ (resp. $\lfloor a \rfloor$) used in Table B.7 stand for the least integer upper bound (resp. the greatest integer lower bound) of $a \in \mathbb{R}$. Constraints t_1 `>` t_2, t_1 `>=` t_2 are not explicitly considered in Table B.7; they are treated as t_2 `<` t_1, t_2 `<=` t_1, respectively. In tables B.6 and B.7, the existential quantification of the new variables $\overline{V'}$ is left implicit, and results displayed as an empty set of constraints must be read as an empty (and thus trivially true) conjunction.

Example 4 below illustrates the operation of the goal transformation rules from Table B.5 for computing bridges and projections with the help of the functions specified in Tables B.6 and B.7.

SB Set Bridges

$$\exists \overline{U}.\ P \sqcap \pi,\ C \sqcap M \sqcap H \sqcap F \sqcap R \vdash_{\mathbf{SB}} \exists \overline{V}', \overline{U}.\ P \sqcap \pi,\ C \sqcap M' \sqcap H \sqcap F \sqcap R$$

If π is a primitive atomic constraint and either (i) or (ii) holds, where

(i) π is a proper $\mathcal{FD}$-constraint or else an extended $\mathcal{H}$-constraint such that $M \vdash \pi$ `in` $\mathcal{FD}$, and $M' = B', M$, where $\exists \overline{V'}\, B' = bridges^{\mathcal{FD}\to\mathcal{R}}(\pi, B_M) \neq \emptyset$.

(ii) π is a proper $\mathcal{R}$-constraint or else an extended $\mathcal{H}$-constraint such that $M \vdash \pi$ `in` $\mathcal{R}$, and $M' = B', M$, where $\exists \overline{V'}\, B' = bridges^{\mathcal{R}\to\mathcal{FD}}(\pi, B_M) \neq \emptyset$.

PP Propagate Projections

$$\exists \overline{U}.\ P \sqcap \pi,\ C \sqcap M \sqcap H \sqcap F \sqcap R \vdash_{\mathbf{PP}} \exists \overline{V'}, \overline{U}.\ P \sqcap \pi,\ C \sqcap M \sqcap H \sqcap F' \sqcap R'$$

If π is a primitive atomic constraint and either (i) or (ii) holds, where

(i) π is a proper $\mathcal{FD}$-constraint or else an extended $\mathcal{H}$-constraint such that $M \vdash \pi$ `in` $\mathcal{FD}$, $\exists \overline{V'}\, \Pi' = proj^{\mathcal{FD}\to\mathcal{R}}(\pi, B_M) \neq \emptyset$, $F' = F$, and $R' = \Pi', R$, or else,

(ii) π is a proper $\mathcal{R}$-constraint or else an extended $\mathcal{H}$-constraint such that $M \vdash \pi$ `in` $\mathcal{R}$, $\exists \overline{V'}\, \Pi' = proj^{\mathcal{R}\to\mathcal{FD}}(\pi, B_M) \neq \emptyset$, $F' = \Pi', F$, and $R' = R$.

SC Submit Constraints

$$\exists \overline{U}.\ P \sqcap \pi,\ C \sqcap M \sqcap H \sqcap F \sqcap R \vdash_{\mathbf{SC}} \exists \overline{U}.\ P \sqcap C \sqcap M' \sqcap H' \sqcap F' \sqcap R'$$

If π is a primitive atomic constraint and one of the following cases applies:

(i) π is a $\mathcal{M}$-constraint, $M' = \pi, M$, $H' = H$, $F' = F$, and $R' = R$.

(ii) π is an extended $\mathcal{H}$-constraint such that neither $M \vdash \pi$ `in` $\mathcal{FD}$ nor $M \vdash \pi$ `in` $\mathcal{R}$, $M' = M$, $H' = \pi, H$, $F' = F$, and $R' = R$.

(iii) π is a proper $\mathcal{FD}$-constraint or else an extended $\mathcal{H}$-constraint such that $M \vdash \pi$ `in` $\mathcal{FD}$, $M' = M$, $H' = H$, $F' = \pi, F$, and $R' = R$.

(iv) π is a proper $\mathcal{R}$-constraint or else an extended $\mathcal{H}$-constraint such that $M \vdash \pi$ `in` $\mathcal{R}$, $M' = M$, $H' = H$, $F' = F$, and $R' = \pi, R$.

Table B.5: Rules for Bridges and Projections

Example 4. Computation of Bridges and Projections

The initial goal in this current example is an extension of the initial goal in Example 3. The first six steps of the current computation are similar to those in Example 3, taking care of flattening the $\mathcal{R}$-constraint `(RX+2*RY)*RZ <= 3.5`. The subsequent steps use the transformation rules from Table B.5 until no further bridges and projections can be computed and no constraint remains in the constraint pool.

$\sqcap\ \underline{(RX + 2 * RY) * RZ \texttt{ <= } 3.5} \sqcap\ X \texttt{ \#== } RX, Y \texttt{ \#== } RY, Z \texttt{ \#== } RZ \sqcap\sqcap\sqcap \vdash_{\mathbf{FC^3},\mathbf{PC^3}}$

π	$bridges^{\mathcal{FD}\to\mathcal{R}}(\pi, B)$	$proj^{\mathcal{FD}\to\mathcal{R}}(\pi, B)$
`domain` $[X_1, \ldots, X_n]$ a b	$\{X_i$ `#==` $RX_i \mid 1 \leq i \leq n$, X_i has no bridge in B and RX_i new$\}$	$\{a$ `<=` RX_i, RX_i `<=` $b \mid 1 \leq i \leq n$ and $(X_i$ `#==` $RX_i) \in B\}$
`belongs` X $[a_1, \ldots, a_n]$	$\{X$ `#==` $RX \mid X$ has no bridge in B and RX new$\}$	$\{min(a_1, \ldots, a_n)$ `<=` RX, RX `<=` $max(a_1, \ldots, a_n) \mid 1 \leq i \leq n$ and $(X$ `#==` $RX) \in B\}$
t_1 `#<` t_2 (resp. `#<=`, `#>`, `#=>`)	$\{X_i$ `#==` $RX_i \mid 1 \leq i \leq 2$, t_i is a variable X_i with no bridge in B, and RX_i new$\}$	$\{t_1^{\mathcal{R}}$ `<` $t_2^{\mathcal{R}} \mid$ For $1 \leq i \leq 2$: Either t_i is an integer constant n and $t_i^{\mathcal{R}}$ is the integral real n, or else t_i is a variable X_i with $(X_i$ `#==` $RX_i) \in B$, and $t_i^{\mathcal{R}}$ is $RX_i\}$
t_1 `==` t_2	$\{X$ `#==` $RX \mid$ either t_1 is an integer constant and t_2 is a variable X with no bridges in B (or vice versa) and RX is new$\}$	$\{t_1^{\mathcal{R}}$ `==` $t_2^{\mathcal{R}} \mid$ For $1 \leq i \leq 2$: $t_i^{\mathcal{R}}$ is determined as in the `#<` case$\}$
t_1 `/=` t_2	$\{X$ `#==` $RX \mid$ either t_1 is an integer constant and t_2 is a variable X with no bridges in B (or vice versa) and RX is new$\}$	$\{t_1^{\mathcal{R}}$ `/=` $t_2^{\mathcal{R}} \mid$ For $1 \leq i \leq 2$: $t_i^{\mathcal{R}}$ is determined as in the `#<` case$\}$
t_1 `#+` $t_2 \to!$ t_3 (resp. `#-`, `#*`)	$\{X_i$ `#==` $RX_i \mid 1 \leq i \leq 3$, t_i is a variable X_i with no bridge in B and RX_i new$\}$	$\{t_1^{\mathcal{R}}$ `+` $t_2^{\mathcal{R}} \to!$ $t_3^{\mathcal{R}} \mid$ For $1 \leq i \leq 3$: $t_i^{\mathcal{R}}$ is determined as in the `#<` case$\}$

Table B.6: Computing Bridges and Projections from $\mathcal{FD}$ to $\mathcal{R}$

$\exists RC, RB, RA.\ \square\ \underline{2 * RY \ \to!\ RC}, \underline{RX + RC \ \to!\ RB}, \underline{RB * RZ \ \to!\ RA}, RA$ `<=` $3.5\ \square$
X `#==` RX, Y `#==` RY, Z `#==` $RZ\ \square\square\square \vdash_{\mathbf{SB^3}}$

$\exists C, B, A, RC, RB, RA.\ \square\ \underline{2 * RY \ \to!\ RC}, \underline{RX + RC \ \to!\ RB}, \underline{RB * RZ \ \to!\ RA}, \underline{RA}$ `<=` $\underline{3.5}\ \square$
C `#==` RC, B `#==` RB, A `#==` RA, X `#==` RX, Y `#==` RY, Z `#==` $RZ\ \square\square\square \vdash_{\mathbf{PP^4}}$

$\exists C, B, A, RC, RB, RA.\ \square\ \underline{2 * RY \ \to!\ RC}, \underline{RX + RC \ \to!\ RB}, \underline{RB * RZ \ \to!\ RA}, \underline{RA}$ `<=` $\underline{3.5}\ \square$
C `#==` RC, B `#==` RB, A `#==` RA, X `#==` RX, Y `#==` RY, Z `#==` $RZ\ \square\square$
2 `#*` $Y \to!\ C, X$ `#+` $C \to!\ B, B$ `#*` $Z \to!\ A, A$ `#<=` $3\ \square \vdash_{\mathbf{SC^4}}$

$\exists C, B, A, RC, RB, RA.\ \square\square\ C$ `#==` RC, B `#==` RB, A `#==` RA, X `#==` RX, Y `#==` RY, Z `#==` RZ
$\square\square\ 2$ `#*` $Y \to!\ C, X$ `#+` $C \to!\ B, B$ `#*` $Z \to!\ A, A$ `#<=` $3\ \square$
2 `*` $RY \to!\ RC, RX$ `+` $RC \to!\ RB, RB$ `*` $RZ \to!\ RA, RA$ `<=` 3.5

To finish this subsection, we present the goal transformation rules in Table B.8, which can be used to infer $\mathcal{H}$-constraints from the $\mathcal{M}$-constraints placed in the store M. The inferred $\mathcal{H}$-constraints happen to be $\mathcal{FD}$-specific or $\mathcal{R}$-specific, according to the case, and can be placed in the corresponding store. Therefore, the rules in this group model domain cooperation mechanisms other than bridges and projections.

π	$bridges^{\mathcal{R}\to\mathcal{FD}}(\pi, B)$	$proj^{\mathcal{R}\to\mathcal{FD}}(\pi, B)$
RX < RY	$\emptyset$ (*no bridges are created*)	$\{X$ #< $Y \mid (X$ #== $RX)$,$(Y$ #== $RY) \in B\}$
RX < a	$\emptyset$ (*no bridges are created*)	$\{X$ #< $\lceil a \rceil \mid a \in \mathbb{R}, (X$ #== $RX) \in B\}$
a < RY	$\emptyset$ (*no bridges are created*)	$\{\lfloor a \rfloor$ #< $Y \mid a \in \mathbb{R}, (Y$ #== $RY) \in B\}$
RX <= RY	$\emptyset$ (*no bridges are created*)	$\{X$ #<= $Y \mid (X$ #== $RX)$,$(Y$ #== $RY) \in B\}$
RX <= a	$\emptyset$ (*no bridges are created*)	$\{X$ #<= $\lfloor a \rfloor \mid a \in \mathbb{R}, (X$ #== $RX) \in B\}$
a <= RY	$\emptyset$ (*no bridges are created*)	$\{\lceil a \rceil$ #<= $Y \mid a \in \mathbb{R}, (Y$ #== $RY) \in B\}$
t_1 == t_2	$\{X$ #== $RX \mid$ either t_1 is an integral real constant and t_2 is a variable RX with no bridges in B (or vice versa) and X is new$\}$	$\{t_1^{\mathcal{FD}}$ == $t_2^{\mathcal{FD}} \mid$ For $1 \le i \le 2$: Either t_i is an integral real constant n and $t_i^{\mathcal{FD}}$ is the integer n, or else t_i is a variable RX_i with $(X_i$ #== $RX_i) \in B$, and $t_i^{\mathcal{FD}}$ is $X_i\}$
t_1 /= t_2	$\emptyset$ (*no bridges are created*)	$\{t_1^{\mathcal{FD}}$ /= $t_2^{\mathcal{FD}} \mid$ For $1 \le i \le 2$: Either t_i is an integral real constant n and $t_i^{\mathcal{FD}}$ is the integer n, or else t_i is a variable RX_i with $(X_i$ #== $RX_i) \in B$, and $t_i^{\mathcal{FD}}$ is $X_i\}$
t_1 + $t_2 \to!$ t_3 (resp. -, *)	$\{X$ #== $RX \mid t_3$ is a variable RX with no bridge in B, X new, for $1 \le i \le 2$, t_i is either an integral real constant or a variable RX_i with bridge $(X_i$ #== $RX_i) \in B\}$	$\{t_1^{\mathcal{FD}}$ #+ $t_2^{\mathcal{FD}} \to!$ $t_3^{\mathcal{FD}} \mid$ For $1 \le i \le 3$: $t_i^{\mathcal{FD}}$ is determined as in the previous case$\}$
t_1 / $t_2 \to!$ t_3	$\emptyset$ (*no bridges are created*)	$\{t_2^{\mathcal{FD}}$ #* $t_3^{\mathcal{FD}} \to!$ $t_1^{\mathcal{FD}} \mid$ For $1 \le i \le 3$ is determined as in the previous case$\}$

Table B.7: Computing Bridges and Projections from $\mathcal{R}$ to $\mathcal{FD}$

IE Infer Equalities

$\exists\overline{U}.\ P \square C \square X$ #== RX, X' #== $RX, M \square H \square F \square R \vdash_{\mathbf{UB}}$
$\exists\overline{U}.\ P \square C \square X$ #== $RX, M \square H \square X$ == $X', F \square R.$

$\exists\overline{U}.\ P \square C \square X$ #== RX, X #== $RX', M \square H \square F \square R \vdash_{\mathbf{UB}}$
$\exists\overline{U}.\ P \square C \square X$ #== $RX, M \square H \square F \square RX$==$RX', R.$

ID Infer Disequalities

$\exists\overline{U}.\ P \square C \square X$#/==$u', M \square H \square F \square R \vdash_{\mathbf{ID}} \exists\overline{U}.\ P \square C \square M \square H \square X$/=$u, F \square R$
if $u \in \mathbb{Z}$, $u' \in \mathbb{R}$ and u #==$^{\mathcal{M}}$ $u' \to true.$

$\exists\overline{U}.\ P \square C \square u$#/==$RX, M \square H \square F \square R \vdash_{\mathbf{ID}} \exists\overline{U}.\ P \square C \square M \square H \square F \square RX$/=$u', R$
if $u \in \mathbb{Z}$, $u' \in \mathbb{R}$ and u #==$^{\mathcal{M}}$ $u' \to true.$

Table B.8: Rules for Inferring $\mathcal{H}$-constraints from $\mathcal{M}$-constraints

B.9.4 Constraint Solving Rules

The rules shown in Table B.9 describe the possible transformation of a goal by a solver's invocation. Each time a new constraint from the pool is placed into its store by means of the transformation **SC**, the corresponding solver is invoked by means of the rules in this table. The solvers for the four domains $\mathcal{M}$, $\mathcal{H}$, $\mathcal{FD}$ and $\mathcal{R}$ involved in the coordination domain $\mathcal{C}$ are considered. The availability of the $\mathcal{M}$-solver means that solving mediatorial constraints contributes to the cooperative goal solving process, in addition to the role of bridges for guiding projections.

MS	$\mathcal{M}$**-Constraint Solver (*glass-box*)**
	$\exists\overline{U}.\ P \sqcap C \sqcap M \sqcap H \sqcap F \sqcap R \Vdash_{\mathbf{MS}} \exists\overline{Y'},\overline{U}.\ (P \sqcap C \sqcap (\Pi' \sqcap \sigma_M) \sqcap H \sqcap F \sqcap R)@_{\mathcal{M}}\sigma'$
	If $pvar(P) \cap var(\Pi_M) = \emptyset$, $(\Pi_M \sqcap \sigma_M)$ is not solved, $\Pi_M \Vdash_{solve^{\mathcal{M}}} \exists\overline{Y'}(\Pi' \sqcap \sigma')$.
FS	$\mathcal{FD}$**-Constraint Solver (*black-box*)**
	$\exists\overline{U}.\ P \sqcap C \sqcap M \sqcap H \sqcap F \sqcap R \Vdash_{\mathbf{FS}} \exists\overline{Y'},\overline{U}.\ (P \sqcap C \sqcap M \sqcap H \sqcap (\Pi' \sqcap \sigma_F) \sqcap R)@_{\mathcal{FD}}\sigma'$
	If $pvar(P) \cap var(\Pi_F) = \emptyset$, $(\Pi_F \sqcap \sigma_F)$ is not solved, $\Pi_F \Vdash_{solve^{\mathcal{FD}}} \exists\overline{Y'}(\Pi' \sqcap \sigma')$.
RS	$\mathcal{R}$**-Constraint Solver (*black-box*)**
	$\exists\overline{U}.\ P \sqcap C \sqcap M \sqcap H \sqcap F \sqcap R \Vdash_{\mathbf{RS}} \exists\overline{Y'},\overline{U}.\ (P \sqcap C \sqcap M \sqcap H \sqcap F \sqcap (\Pi' \sqcap \sigma_R))@_{\mathcal{R}}\sigma'$
	If $pvar(P) \cap var(\Pi_R) = \emptyset$, $(\Pi_R \sqcap \sigma_R)$ is not solved, $\Pi_R \Vdash_{solve^{\mathcal{R}}} \exists\overline{Y'}(\Pi' \sqcap \sigma')$.

Table B.9: Rules for $\mathcal{M}$, $\mathcal{H}$, $\mathcal{FD}$ and $\mathcal{R}$ Constraint Solving

At this point, we can precise the notion of *solved goal* as follows: a goal G is solved iff it has the form $\exists\overline{U}.\ \sqcap\ \sqcap\ M \sqcap H \sqcap F \sqcap R$ (with empty P and C) and the $CLNC(\mathcal{C})$-transformations in Tables B.8 and B.9 cannot be applied to G.

The $CCLNC(\mathcal{C})$ calculus leaves ample room for choosing a particular goal transformation at each step, so that many different computations are possible in principle. However, the $\mathcal{TOY}$ implementation follows a particular strategy. The part $P \sqcap C$ of the current goal is treated as a sequence and processed from left to right, with the only exception of suspensions $e \to X$ that are delayed until they can be safely eliminated by means of rule **EL** or the goal is so transformed that they are no suspensions anymore. As long as the current goal is not in solved form, a subgoal is selected and processed according to a strategy which can be roughly described as follows:

1. If P includes some production which can be handled by the constrained lazy narrowing rules in Table B.4, the leftmost production is selected and processed. Note that the selected production must be either a suspension $e \to X$ that can be discarded by rule **EL**, or else a production that is not a suspension. The applications of rule **DF** are performed in an optimized way by using definitional trees [Vad05, Vad07].

2. If P is empty or consists only of productions $e \to X$ that cannot be processed by means of the constrained lazy narrowing rules in Table B.4, and moreover some of the stores M, H, F or R are not in solved form and its constraints include no produced variables, then the solvers for such stores are invoked, choosing the set $\mathcal{X}$ of critical variables as explained in Table B.9.

3. If neither of the two previous items applies and C is not empty, the leftmost atomic constraint δ in C is selected. In case it is not primitive, the flattening rule **FC** from Table B.4 is applied. Otherwise, δ is a primitive atomic constraint π, and exactly one of the following cases applies:

   (a) If π is a proper $\mathcal{FD}$-constraint or else an extended $\mathcal{H}$-constraint such that $M \vdash \pi$ `in` $\mathcal{FD}$, then π is processed by means of the rules **SB**, **PP** and **SC** from Table B.5. This generates bridges and projected constraints π', if possible, and submits π to the store F. Then, the rules from Table B.9 are used for invoking the $\mathcal{FD}$-solver (in case that the constraints in F include no produced variables) and the $\mathcal{R}$-solver (in case that the constraints in R include no produced variables).

   (b) If π is a proper $\mathcal{R}$-constraint or else an extended $\mathcal{H}$-constraint such that $M \vdash \pi$ `in` $\mathcal{R}$, then π is processed by means of the rules **SB**, **PP** and **SC** from Table B.5. This generates bridges and projected constraints π', if possible, and submits π to the store R. Then, the rules from Table B.9 are used for invoking the $\mathcal{R}$-solver (in case that the constraints in R include no produced variables) and the $\mathcal{FD}$-solver (in case that the constraints in F include no produced variables).

   (c) If π is an extended $\mathcal{H}$-constraint such that neither $M \vdash \pi$ `in` $\mathcal{FD}$ nor $M \vdash \pi$ `in` $\mathcal{R}$, then π is submitted to the store H by means of rule **SC**, and the $\mathcal{H}$-solver is invoked in case that the constraints in H include no obviously demanded produced variables.

   (d) If π is a $\mathcal{M}$-constraint, then π is submitted to the store M by means of rule **SC**, the rules of Table B.8 are applied if possible, and the $\mathcal{M}$-solver is invoked in case that the constraints in M include no produced variables.

Example 5. Example of cooperative goal solving

In order to illustrate the overall behaviour of our cooperative goal solving calculus, we present a $CCLNC(\mathcal{C})$ computation solving the goal **Goal 2** discussed in Subsection B.5. The reader is referred to Figure B.2 for a graphical representation of the problem and to Subsection B.9.1 for a formulation of the goal and the expected solution in the particular case `d = 2`, `n = 4`. However, the solution is the same for any choice of positive integer values `d` and `n` such that `n = 2*d`, and here we will discuss the general case.

The series of goals G_0 up to G_{12} displayed below correspond to the initial goal, the final solved goal and a selection of intermediate goals in a computation which roughly models the strategy of the $\mathcal{TOY}$ implementation, working with the projection functionality activated. In the initial goal, `d` and `n` are arbitrary positive integers such that `n = 2*d` and `d' = d+0.5`.

$G_0 : \square\ \underline{bothIn\,(triangle\,(d,d')\,2\,1)\,(square\,n)\,(X,Y)\ \texttt{==}\ \texttt{true}}\ \square\square\square\square \vdash_{\mathbf{FC}}$

$G_1 : \exists\overline{U_1}.\ bothIn\,(triangle\,(d,d')\,2\,1)\,(square\,n)\,(X,Y) \rightarrow A\ \square\ \underline{A\ \texttt{==}\ \texttt{true}}\ \square\square\square\square \vdash_{\mathbf{SC(ii)}}$

$G_2 : \exists\overline{U_2}.\ \underline{bothIn\,(triangle\,(d,d')\,2\,1)\,(square\,n)\,(X,Y)} \to A \;\square\; \square\square\; A \texttt{ == true}\square\square \vdash_{\mathbf{DF}_{bothIn}}$

$G_3 : \exists\overline{U_3}.\underline{triangle\,(d,d')\,2\,1 \to R},\ \underline{square\,n \to G},\ \underline{(X,Y) \to (X',Y')},\ \underline{\texttt{true} \to A}\ \square$
$X' \texttt{ \#== } RX,\ Y' \texttt{ \#== } RY,\ isIn\,R\,(RX,RY) \texttt{ == true},\ isIn\,G\,(X',Y') \texttt{ == true},$
$labeling\,[\,]\,[X',Y']\ \square\square\ \underline{A \texttt{ == true}}\ \square\square \vdash^*_{\mathbf{SP^2},\mathbf{DC},\mathbf{SP^3},\mathbf{HS}}$

$G_4 : \exists\overline{U_4}.\ \square\underline{X \texttt{ \#== } RX},\ \underline{Y \texttt{ \#== } RY},\ isIn\,(triangle\,(d,d')\,2\,1)\,(RX,RY) \texttt{ == true},$
$isIn\,(square\,n)\,(X,Y) \texttt{ == true},\ labeling\,[\,]\,[X,Y]\ \square\square\ \sigma_H\ \square\square \vdash^*_{\mathbf{SC(i)^2},\mathbf{MS}}$

$G_5 : \exists\overline{U_5}.\ \square\ \underline{isIn\,(triangle\,(d,d')\,2\,1)\,(RX,RY) \texttt{ == true}},\ \underline{isIn\,(square\,n)\,(X,Y) \texttt{ == true}},$
$labeling\,[\,]\,[X,Y]\ \square X \texttt{ \#== } RX,\ Y \texttt{ \#== } RY\square\ \sigma_H\ \square\square \vdash^*_{\mathbf{CLN}}$

$G_6 : \exists\overline{U_6}.\ \square\ \underline{RY \texttt{>=} d'-1},\ 2*RY - 2*1*RX \texttt{<=} 2*d' - 2*1*d,$
$2*RY + 2*1*RX \texttt{<=} 2*d' + 2*1*d,\ domain\,[X,Y]\,0\,n,$
$labeling\,[\,]\,[X,Y]\ \square\ X \texttt{ \#== } RX,\ Y \texttt{ \#== } RY\square\ \sigma'_H\ \square\square \vdash^*_{\mathbf{FC},\mathbf{PC}}$

$G_7 : \exists\overline{U_7}.\ \square\ \underline{d'-1 \to !RA},\ RY \texttt{>=} RA,\ 2*RY - 2*1*RX \texttt{<=} 2*d' - 2*1*d,$
$2*RY + 2*1*RX \texttt{<=} 2*d' + 2*1*d,\ domain\,[X,Y]\,0\,n,$
$labeling\,[\,]\,[X,Y]\ \square\ X \texttt{ \#== } RX,\ Y \texttt{ \#== } RY\square\ \sigma'_H\ \square\square \vdash^*_{\mathbf{SC(iv)},\mathbf{RS}}$

$G_8 : \exists\overline{U_8}.\ \square\ \underline{RY \texttt{>=} d''},\ 2*RY - 2*1*RX \texttt{<=} 2*d' - 2*1*d,$
$2*RY + 2*1*RX \texttt{<=} 2*d' + 2*1*d,\ domain\,[X,Y]\,0\,n,$
$labeling\,[\,]\,[X,Y]\ \square\ X \texttt{ \#== } RX,\ Y \texttt{ \#== } RY\square\ \sigma'_H\ \square\square\ S_R \vdash^*_{\mathbf{BP},\mathbf{CS}}$

$G_9 : \exists\overline{U_9}.\ \square\ \underline{2*RY - 2*1*RX \texttt{<=} 2*d' - 2*1*d},$
$\underline{2*RY + 2*1*RX \texttt{<=} 2*d' + 2*1*d},\ domain\,[X,Y]\,0\,n,\ labeling\,[\,]\,[X,Y]\ \square$
$X \texttt{ \#== } RX,\ Y \texttt{ \#== } RY\ \square\ \sigma'_H\ \square\ Y \texttt{\#>=} d\ \square\ RY \texttt{>=} d'',\ S_R \vdash^*_{\mathbf{FR},\mathbf{BP}}$

$G_{10} : \exists\overline{U_{10}}.\ \square\ domain\,[X,Y]\,0\,n,\ labeling\,[\,]\,[X,Y]\ \square$
$\underline{X \texttt{ \#== } RX,\ Y \texttt{ \#== } RY\ B \texttt{ \#== } RB,\ C \texttt{ \#== } RC,\ S'_M}\ \square\ \sigma'_H\ \square$
$\underline{Y \texttt{\#>=} d,\ 2 \texttt{ \#*} Y \texttt{\#-} 2 \texttt{\#*} X \to !B,\ B \texttt{ \#<= } 1,\ 2 \texttt{ \#*} Y \texttt{\#+} 2 \texttt{\#*} X \to !C,\ C \texttt{ \#<= } n',\ S'_F}\ \square$
$\underline{RY \texttt{>=} d'',\ 2\texttt{*}RY \texttt{-} 2\texttt{*}RX \to !RB,\ RB \texttt{<=} 1,\ 2\texttt{*}RY \texttt{+} 2\texttt{*}RX \to !RC,\ RC \texttt{<=} n',\ S'_R} \vdash^*_{\mathbf{CS}}$

$G_{11} : \exists\overline{U_{11}}.\ \square\ \underline{domain\,[d,d]\,0\,n},\ \underline{labeling\,[\,]\,[d,d]}\ \square\ S''_M\ \square\ \sigma'_H\ \square\ S''_F\ \square\ S''_R \vdash^*_{\mathbf{SC(iii)},\mathbf{FS},\mathbf{SC(iii)},\mathbf{FS}}$

$G_{12} : \exists\overline{U_{12}}.\ \square\ \square\ S''_M\ \square\ \sigma'_H\ \square\ S''_F\ \square\ S''_R$

The local existential variables $\exists\overline{U_i}$ of each goal G_i are not explicitly displayed, and the notation $G_{i-1} \vdash^*_{\mathcal{RS}} G_i$ is used to indicate the transformation of G_{i-1} into G_i using the goal solving rules indicated by $\mathcal{RS}$. At some steps, $\mathcal{RS}$ indicates a particular sequence of individual rules, named as explained in the previous subsections. In other cases, namely for $i = 6$ and $9 \leq i \leq 11$, $\mathcal{RS}$ indicates sets of goal transformation rules, named according to the following conventions:

- **CLN** names the set of constrained lazy narrowing rules presented in Table B.4.

- **FR** names the set consisting of the two rules **FC** and **PC** displayed at the end of Table B.4, used for constraint flattening.
- **BP** names the set of rules for bridges and projections presented in Table B.5.
- **CS** names the set of constraint solving rules presented in Table B.9.

We finish with some comments on the computation steps:

- Transition from G_0 to G_1: The only constraint in C is flattened, giving rise to one suspension and one flat constraint in the new goal. The produced variable `A` is not obviously demanded because the constraint `A == true` is not yet placed in the $\mathcal{H}$-store.
- Transition from G_1 to G_2: The only suspension is delayed, and the only constraint in the pool is processed by submitting it to the $\mathcal{H}$-store. However, the $\mathcal{H}$-solver cannot be invoked at this point, because A has become an obviously demanded variable that is also produced.
- Transition from G_2 to G_3: The former suspension has become a production which is processed by applying the program rule defining the function `bothIn`, which introduces new productions in P and new constraints in C.
- Transition from G_3 to G_4: The four productions in P are processed by binding propagations and decompositions (rules **SP** and **DC**), until P becomes empty. Then the $\mathcal{H}$-solver can be invoked. At this point, the $\mathcal{H}$-store just contains a substitution σ_H resulting from the previous binding steps.
- Transition from G_4 to G_5: P is empty, and the two first constraints in C are bridges. They are submitted to the $\mathcal{M}$-store and the $\mathcal{M}$-solver is invoked, which has no effect in this case.
- Transition from G_5 to G_6: There are no productions, and the two first constraints in the pool are processed by steps similar to those used in the transition going from G_0 to G_4. Upon completing this process, the new pool includes a number of new constraints coming from the conditions in the program rules defining the functions `isIn`, `triangle` and `square`, and the substitution stored in H has changed. At this point, P is empty again and the constraints in C plus the bridges in M amount to a system equivalent to the one used in Subsection B.5 for an informal discussion of the resolution of **Goal 2**.
- Transition from G_6 to G_7 and from G_7 to G_8: There are no productions, and flattening the first constraint in C gives rise to the primitive constraint `d'-1` $\rightarrow!$ `RA`. This is submitted to the $\mathcal{R}$-store and the $\mathcal{R}$-solver is invoked, which computes `d''` as the numeric value of `d'-1` and propagates the variable binding `RA` $\mapsto$ `d''` to the whole goal, possibly causing some other internal changes in the $\mathcal{R}$-store.

- Transition from G_8 to G_9: There are no productions, and the first constraint in C is now `RY >= d''`. Since `d'' = d'-1 = d+0.5-1 = d-0.5`, we have $\lceil$`d''`$\rceil$ = `d`. Therefore, projecting `RY >= d''` with the help of the available bridges (including `Y #== RY`) allows to compute `Y #>= d` as a projected $\mathcal{FD}$-constraint. Both `RY >= d''` and `Y #>= d` are submitted to their respective stores and the two solvers are invoked, having no effect in this case.

- Transition from G_9 to G_{10}: There are no productions, and the two first atomic constraints in the pool of G_9 (two $\mathcal{R}$-constraints δ_1 and δ_2) are processed by steps similar to those used in the transition going from G_6 to G_9, except that the solver invocations are delayed to the transition from G_{10} to G_{11} and commented in the next item. (Actually, the $\mathcal{TOY}$ implementation would invoke the solvers two times: The first time when processing δ_1 and the second time when processing δ_2. Here we explain the overall effect of the two invocations for the sake of simplicity.) Upon completing this process, G_{10} stays as follows: P is empty, C includes the two other constraints which were there in G_9, and the stores M, F and R have changed because of new bridges and projections. In fact, the constraints within the stores F and R in G_{10} would be equivalent but not identical to the ones shown in this presentation, due to intermediate flattening steps that we have not shown explicitly. In particular, the $\mathcal{R}$-constraint `2*RY-2*RX` $\rightarrow!$ `RB` and its $\mathcal{FD}$-projection `2#*Y#-2#*X` $\rightarrow!$ `B` would really not occur in this form, but a conjunctions of primitive constraints obtained by flattening them would occur at their place.

- Transition from G_{10} to G_{11}: At this point, the $\mathcal{FD}$-solver is able to infer that the constraints in the $\mathcal{FD}$ store imply one single solution for the variables `X` and `Y`, namely `{X` $\mapsto$ `d, Y` $\mapsto$ `d}`. Therefore, the $\mathcal{FD}$-solver propagates these bindings to the whole goal, affecting in particular to the bridges in M. Then, the $\mathcal{M}$-solver propagates the corresponding bindings `{RX` $\mapsto$ `rd, RY` $\mapsto$ `rd}` (`rd` being the representation of `d` as an integral real number), and the $\mathcal{R}$-solver succeeds.

- Transition from G_{11} to G_{12}: The two constraints in C have now become trivial. Submitting them to their stores and invoking the respective solvers leads to a solved goal, whose restriction to the variables in the initial goal is the computed answer $\square\square\square\square$ ($\Diamond$ $\square$ `{X` $\mapsto$ `d, Y` $\mapsto$ `d}`) $\square$. Note that no labeling whatsoever has been performed, independently of the size of `n`.

The semantic results of *soundness* and *limited completeness* of the cooperative goal solving calculus $CCLNC(\mathcal{C})$ w.r.t. the declarative semantics of $CFLP(\mathcal{C})$ are given in the thesis.

Following to the notion of coordination domain $\mathcal{C}_{\mathcal{FD}.\mathcal{R}}$ and the corresponding goal solving calculus $CCLNC(\mathcal{C}_{\mathcal{FD}.\mathcal{R}})$, next sections present coordination domain $\mathcal{C}_{\mathcal{FD}.\mathcal{FS}}$ and the goal solving calculus $CCLNC(\mathcal{C}_{\mathcal{FD}.\mathcal{FS}})$ for the cooperation of the $\mathcal{FD}$ and $\mathcal{FS}$ domains. In the thesis, the cooperation of the domains $\mathcal{FD}$ and $\mathcal{FS}$ have been presented in two separate chapters. However, in this summary has been preferred to unite them, and to give an overview without details.

B.10 The Coordination Domain $\mathcal{C}_{\mathcal{FD}.\mathcal{FS}}$

The coordination domain $\mathcal{C}_{\mathcal{FD}.\mathcal{FS}} = \mathcal{M}_{\mathcal{FD}.\mathcal{FS}} \oplus \mathcal{H} \oplus \mathcal{FD} \oplus \mathcal{FS}$ is constructed analogously to the coordination domain $\mathcal{C}_{\mathcal{FD}.\mathcal{R}}$ of Section B.8. In particular, these two coordination domains differ in their corresponding mediatorial domains as it is detailed below.

The $\mathcal{FD}$ and $\mathcal{FS}$ domains can be joined by means of the *amalgamated sum* defined as a new domain $\mathcal{S} = \mathcal{FD} \oplus \mathcal{FS}$ with signature $\Sigma_{\mathcal{S}} = \langle TC,\ SBT_{\mathcal{FD}} \cup SBT_{\mathcal{FS}},\ DC,\ DF,\ SPF_{\mathcal{FD}} \cup SPF_{\mathcal{FS}} \rangle$. According to [EFH+09], $\mathcal{FD}$ and $\mathcal{FS}$ are joinable, and $\mathcal{S}$ is a conservative extension of both domains. However, this new domain $\mathcal{S}$ has no mechanisms for the communication between both pure domains. In order to allow cooperation between these domains we need a new *Mediatorial domain*, which supplies the so called *cardinality bridge constraints* and *minimum (resp. maximum) bridge constraints*. For the sake of reading, we will write $\mathcal{M}$ instead of $\mathcal{M}_{\mathcal{FD}.\mathcal{FS}}$. Therefore the mediatorial domain $\mathcal{M}$ for the communication between $\mathcal{FD}$ and $\mathcal{FS}$ has signature $\Sigma_{\mathcal{M}} = \langle TC,\ SBT_{\mathcal{M}},\ DC,\ DF,\ SPF_{\mathcal{M}} \rangle$ and it is defined as follows:

- $SBT_{\mathcal{M}} = \{\texttt{int}, \texttt{set}\} \subseteq SBT_{\mathcal{FD}} \cup SBT_{\mathcal{FS}}$.
- Each set of base values of the mediatorial domain corresponds to a set of base values of each pure domain: $\mathcal{B}^{\mathcal{M}}_{\texttt{set}} = \mathcal{B}^{\mathcal{FS}}_{\texttt{set}}$ and $\mathcal{B}^{\mathcal{M}}_{\texttt{int}} = \mathcal{B}^{\mathcal{FD}}_{\texttt{int}}$.
- $SPF_{\mathcal{M}} = \{$ `#--` , `minSet`, `maxSet`$\}$. The interpretation of the cardinality bridge constraint is i `#--`$^{\mathcal{M}_{\mathcal{FD}.\mathcal{FS}}}$ $s \rightarrow t$, where `#--`$^{\mathcal{M}_{\mathcal{FD}.\mathcal{FS}}}$ is a subset of the Cartesian product $\mathbb{Z} \times \mathcal{B}^{\mathcal{FS}}_{\texttt{set}}$, defined to hold iff any of the following cases holds: either s is a set, i is the cardinality of s and $t = \texttt{true}$; or s is a set, i is not the cardinality of s and $t = \texttt{false}$; or $t = \bot$. The interpretation of the `minSet` (resp. `maxSet`) bridge constraint is `minSet`$^{\mathcal{M}}$ $s\ m \rightarrow t$ where `minSet`$^{\mathcal{M}}$ is a subset of the Cartesian product $\mathcal{B}^{\mathcal{FS}}_{\texttt{int}} \times \mathbb{Z}$, defined to hold iff any of the following cases holds: either s is a non-empty set, m is the smallest integer of the set s and t is `true`; or s is a non-empty set, m is not the smallest integer of the set s and t is `false`; or $t = \bot$.

We define $solve^{\mathcal{M}}$ as a store transformation system, using the same abstract technique for glass-box solvers described previously. Store transformation rules are defined in Table B.10, rules from **M1** to **M5** correspond to the cardinality bridge, and rules from **M6** to **M9** correspond to the bridge `minSet`. Rules to the bridge `maxSet` are similar to rules from **M6** to **M9**.

Rule **M1** represents the case of a cardinality bridge with a ground set, in which the variable X is bound to the cardinality of the set. In **M2** the cardinality of set variable S is zero and therefore S is bound to the empty set. Rules **M3** and **M4** correspond to the case in which the set and the cardinality are ground. If the constraint is satisfied then **M3** is applied else **M4** is applied. Rule **M5** fails because it is not possible to assign a minimum value to the empty set. Rule **M6** correspond to the case of a ground set not empty s and a variable Min. In this case the variable Min is substituted by the integer m that represent the minimum value of the set. When both arguments are ground, then rule **M7** checks if m is the minimum of s and rule **M8** checks otherwise.

M1	X #-- u', $\Pi \square \sigma \vdash_{\mathcal{M}} \Pi\sigma_1 \square \sigma\sigma_1$
	if $u' \in \mathcal{B}^{\mathcal{FS}}_{\mathtt{set}}$, $X \in Var$ and $\exists u \in \mathbb{Z}^+$ s.t. u #--$^{\mathcal{M}_{\mathcal{FD}.\mathcal{FS}}}$ u' and $\sigma_1 = \{X \mapsto u\}$.
M2	u #-- S, $\Pi \square \sigma \vdash_{\mathcal{M}} \Pi\sigma_1 \square \sigma\sigma_1$
	if $u = 0$, $S \in Var$ and $\sigma_1 = \{S \mapsto \{\}\}$
M3	u #-- u', $\Pi \square \sigma \vdash_{\mathcal{M}} \Pi \square \sigma$
	if $u \in \mathbb{Z}^+$, $u' \in \mathcal{B}^{\mathcal{FS}}_{\mathtt{set}}$ and u #--$^{\mathcal{M}_{\mathcal{FD}.\mathcal{FS}}}$ u'.
M4	u #-- u', $\Pi \square \sigma \vdash_{\mathcal{M}} \blacksquare$
	if $u \in \mathbb{Z}^+$, $u' \in \mathcal{B}^{\mathcal{FS}}_{\mathtt{set}}$ and u #--$^{\mathcal{M}_{\mathcal{FD}.\mathcal{FS}}}$ u' does not hold.
M5	`minSet` $\emptyset$ Min, $\Pi \square \sigma \vdash_{\mathcal{M}'_{\mathcal{FD}.\mathcal{FS}}} \blacksquare$
M6	`minSet` s Min, $\Pi \square \sigma \vdash_{\mathcal{M}'_{\mathcal{FD}.\mathcal{FS}}} \Pi\sigma_1 \square \sigma\sigma_1$
	if $s \in \mathcal{B}^{\mathcal{FS}}_{\mathtt{set}}$, $Min \in \mathcal{V}ar$ and $\exists m \in \mathbb{Z}$ s.t. `minSet`$^{\mathcal{M}'_{\mathcal{FD}.\mathcal{FS}}}$ s $m \rightarrow$ `true` and $\sigma_1 = \{Min \mapsto m\}$
M7	`minSet` s m, $\Pi \square \sigma \vdash_{\mathcal{M}'_{\mathcal{FD}.\mathcal{FS}}} \Pi \square \sigma$
	if $s \in \mathcal{B}^{\mathcal{FS}}_{\mathtt{set}}$, $m \in \mathbb{Z}$ and `minSet`$^{\mathcal{M}'_{\mathcal{FD}.\mathcal{FS}}}$ s $m \rightarrow true$
M8	`minSet` s m, $\Pi \square \sigma \vdash_{\mathcal{M}'_{\mathcal{FD}.\mathcal{FS}}} \blacksquare$
	if $s \in \mathcal{B}^{\mathcal{FS}}_{\mathtt{set}}$, $m \in \mathbb{Z}$ and `minSet`$^{\mathcal{M}'_{\mathcal{FD}.\mathcal{FS}}}$ s $m \rightarrow false$

Table B.10: Store Transformation Rules for $\vdash_{\mathcal{M}_{\mathcal{FD}.\mathcal{FS}}}$ and $\vdash_{\mathcal{M}'_{\mathcal{FD}.\mathcal{FS}}}$

The definition of the identification of atomic primitive constraints as $\mathcal{FS}$-specific is similar to the definition of the given for $\mathcal{FD}$-specific and $\mathcal{R}$-specific in the Definition 12.

B.11 Cooperative Programming and Goal Solving in $CFLP(\mathcal{C}_{\mathcal{FD}.\mathcal{FS}})$

In order to adapt the calculus to the new coordination domain $\mathcal{C}_{\mathcal{FD}.\mathcal{FS}}$, some new rules are needed and other rules are similar to rules of Section B.9. This section describes rules that are different while only comments on similar rules.

Rules are divided in three parts: the constrained lazy narrowing rules that are exactly the same because they model the behaviour of the lazy narrowing; the domain cooperation which basically changes; and the solver invocation rules that are similar.

Regarding to the domain cooperation rules that set bridges, propagate projections and submit constraints are similar to rules of the table B.5. Rule **SB** generates cardinality bridge constraints in M for variables involved in $\mathcal{FS}$ constraints. However, **SB** do not generate minimum and maximum bridge constraints because is not possible to assure that it is different from the empty set. Rule **PP** projects constraints from $\mathcal{FD}$ to $\mathcal{FS}$ and vice versa. Rule **SC** places the constraints in the corresponding solver.

Functions *bridges* and *proj* are used by **SB** and **PP** for obtaining bridges and projections, respectively. Given a pool of constraints that includes an $\mathcal{FS}$ atomic primitive constraint π and a Mediatorial store M, with a set of bridge constraints B, we define the function $bridges^{\mathcal{FS}\rightarrow\mathcal{FD}}(\pi, B)$ to generate new bridges for all set variables involved in π as long as they are not already available in B. Each set variable has an associated finite domain variable

which represents the cardinality of the set.

$$bridges^{\mathcal{FS}\to\mathcal{FD}}(\pi, B) = \{C_i \texttt{ \#-- } S_i \mid S_i \text{ has no bridge in } B,\ C_i \text{ fresh}\}$$

Projections defined by the functions $proj^{\mathcal{FS}\to\mathcal{FD}}(\pi, B)$ and $proj^{\mathcal{FD}\to\mathcal{FS}}(\pi, B)$ take place whenever a constraint π is posted to its corresponding solver. This process builds mate constraints considering the available bridge constraints, and posts them to the mate solver. Tables 6.3 and 6.4 give a specification of projection generation for each constraint π.

π	$proj^{\mathcal{FD}\to\mathcal{FS}}(\pi, B)$
C_1 `#<` C_2 (or `#>` or `/=`)	$\{S_1$ `/=` $S_2 \mid (C_1$ `#--` $S_1), (C_2$ `#--` $S_2) \in B\}$
Min_1 `/=` Min_2 (resp. `#<`, `#>`, `all_different`)	$\{S_1$ `/=` $S_2 \mid$ `minSet` $M_1\ S_1$, `minSet` $M_2\ S_2 \in B\}$

Table B.11: Computing Projections from $\mathcal{FD}$ to $\mathcal{FS}$

π	$proj^{\mathcal{FS}\to\mathcal{FD}}(\pi, B)$
S_1 `==` S_2	$\{C_1$ `==` $C_2 \mid (C_1$ `#--` $S_1), (C_2$ `#--` $S_2) \in B\ \}$
`domainSets [`S_1`,..,`S_n`] l u`	$\{$`c #<=` C_i, C_i `#<= c'` $\mid$ `(c #--` `l`$), ($`c'` `#--` `u`$), (C_i$ `#--` $S_i) \in B$, $1 \le i \le n\}$
`intSets [`S_1`,..,`S_n`] n m`	$\{$`0 #<=` C_i, C_i `#<= m-n+1` $\mid (C_i$ `#--` $S_i) \in B$, $1 \le i \le n\}$
`subset` $S_1\ S_2$	$\{C_1$ `#<=` $C_2 \mid (C_1$ `#--` $S_1), (C_2$ `#--` $S_2) \in B\ \}$
`intersect` $S_1\ S_2\ S_3$	$\{C_3$ `#<=` C_1`,` C_3 `#<=` $C_2\}$ $\mid (C_i$ `#--` $S_i) \in B$, $1 \le i \le 3\}$
`union` $S_1\ S_2\ S_3$	$\{C_3$ `#<=` C_1 `#+` C_2, C_1 `#<=` C_3, C_2 `#<=` $C_3 \mid (C_i$ `#--` $S_i) \in B$, $1 \le i \le 3\}$
`disjoints` $[S_1 \dots S_n]$	$\{\ C_1$ `#+` $\dots$ `#+` C_n`==`$C \mid \cup_{k=1}^{n} S_k = S$, $(C$ `#--` $S)$, $(C_i$ `#--` $S_i)$ $\in B$, $1 \le i \le n\}$
S_1 `==` S_2	$\{\ M_1$ `==` $M_2 \mid$ `(minSet` $S_1\ M_1)$, `(minSet` $S_2\ M_2) \in B\ \}$ (resp. `maxSet`)
`subset` $S_1\ S_2$ (resp. `superset`)	$\{\ Min_1$ `#>=` Min_2, Max_1 `#<=` $Min_2 \mid$ `(minSet` $Min_1\ S_1)$, `(minSet` $S_2\ Min_2)$, `(maxSet` $S_1\ Max_1) \in B\}$
`intersect` $S_1\ S_2\ S_3$ (resp. `intersections`)	$\{\ Min_3$ `#>=` Min_1, Min_3 `#>=` Min_2, Max_3 `#<=` Max_1, Max_3 `#<=` Max_2, $\mid$ `(minSet` $S_i\ Min_i)$, `(maxSet` $S_i\ Max_i) \in B$, $1 \le i \le 3\}$
`union` $S_1\ S_2\ S_3$ (resp. `unions`)	$\{\ Min_3$ `#<=` Min_1, Min_3 `#<=` Min_2, Max_3 `#>=` Max_1, Max_3 `#>=` Max_2, $\mid$ `(minSet` $S_i\ Min_i)$, `(maxSet` $S_i\ Max_i) \in B$, $1 \le i \le 3\}$
S_1 `<<` S_2	$\{\ Max_1$ `#<` Min_2, Min_1 `#+` $Card_1$ `#<=` Min_2, Max_1 `#<` Max_2 `#-` $Card_2$, $\mid$ `(minSet` $S_i\ Min_i)$, `(maxSet` $S_i\ Max_i)$, $(Card_i$ `#--` $S_i) \in B$, $1 \le i \le 2\}$

Table B.12: Computing Projections from $\mathcal{FS}$ to $\mathcal{FD}$

Other rules that change from the calculus $CCLNC(\mathcal{C}_{\mathcal{FD}.\mathcal{R}})$ are those that infer information from the mediatorial domain. In particular, rule **IE** of the Table B.13 infers equalities from cardinality bridges already existing in M, and rule **IF** infers failure from disequalities detected in bridges in M.

IE Infer Equalities

$\exists\overline{U}.P \sqcup C \sqcup(I_1$ `#--` S,I_2 `#--` $S,M)\sqcup H \sqcup F \sqcup S \vdash_{\mathbf{IE}}$

$\exists\overline{U}.P \sqcup C \sqcup(I_1$ `#--` $S,M) \sqcup H \sqcup (I_1$`==`$I_2,F) \sqcup S$

$\exists\overline{U}.P \sqcup C \sqcup$ (`minSet` S_1 M_1, `minSet` S_1 M_2, $M)\sqcup H \sqcup F \sqcup S \vdash_{\mathbf{IE}}$

$\exists\overline{U}.P \sqcup C \sqcup$ (`minSet` S_1 M_1, $M) \sqcup H \sqcup(M_1$`==`$M_2,F)\sqcup S$

IF Infer Failure

$\exists\overline{U}.P\sqcup C\sqcup$ (I_1 `#--` S_1,I_2 `#--` $S_2,M) \sqcup H \sqcup (I_1$`/=`$I_2,F) \sqcup (S_1$`==`$S_2,S) \vdash_{\mathbf{IF}}\blacksquare$

$\exists\overline{U}.P \sqcup C \sqcup$ (`minSet` S_1 M_1, `minSet` S_1 M_2, $M) \sqcup H\sqcup$ (M_1`/=`M_2, $F) \sqcup \vdash_{\mathbf{IF}} \blacksquare$

Table B.13: Inference Rules of Equality and Failure from Bridges Constraints

Example 6. Let us see how this calculus is applied to the evaluation of the goal in Example 1. Substitutions are not shown in the goal to avoid overloading the notation. Initially, the goal is as follows:

$$\emptyset \sqcup \overbrace{\texttt{L==[S1,S2]}}^{\pi_1}, \overbrace{\texttt{domainSets L \{\} \{1,2,3,4,5\}}}^{\pi_2}, \overbrace{\texttt{atMostOne L}}^{\pi_3}, \texttt{labelingSets L} \sqcup \emptyset \sqcup \emptyset \sqcup \emptyset \sqcup \emptyset$$

The constraint π_1 is sent to the $\mathcal{H}$ store and the $\mathcal{H}$ solver is invoked producing the substitution $\sigma_1 =\{\texttt{L} \mapsto \texttt{[S1,S2]}\}$. Next, constraint π_2 is processed: First, rule **SB** generates the bridge constraints `I1 #-- S1` and `I2 #-- S2` with `I1` and `I2` fresh variables, according to the function $bridges^{\mathcal{FS}\to\mathcal{FD}}$ defined in page 249.

$$bridges^{\mathcal{FS}\to\mathcal{FD}}(\pi_2,\emptyset) = \{\texttt{I1 \#-- S1}, \texttt{I2 \#-- S2}\}$$
$$B = \{\texttt{I1 \#-- S1}, \texttt{I2 \#-- S2}\}$$

Next, rule **PP** builds $\mathcal{FD}$ constraints corresponding to the projection of π_2:

$$proj^{\mathcal{FS}\to\mathcal{FD}}(\pi_2,B) = \{\texttt{0\#<=I1, I1\#<=5, 0\#<=I2, I2\#<=5}\}$$

Rule **SC** sends π_2 to the $\mathcal{FS}$ store and fs solver is invoked constraining the domains of variables `S1` and `S2` to be lattices of lower bound `{}` and upper bound `{1,2,3,4,5}`. Then, the goal is transformed to:

$$\exists\ \texttt{I1,I2}.\emptyset \sqcup \texttt{atMostOne [S1,S2]}, \texttt{labelingSets [S1,S2]} \sqcup \texttt{I1 \#-- S1}, \texttt{I2 \#-- S2} \sqcup \emptyset \sqcup \texttt{0\#<=I1,I1\#<=5,0\#<=I2,I2\#<=5} \sqcup \pi_2$$

Now π_3 is flattened, producing new atomic primitive constraints:

$$\exists\ \texttt{I1,I2,S12,I12}.\emptyset \square \overbrace{\texttt{3 \#-- S1}}^{\pi_4}, \overbrace{\texttt{3 \#-- S2}}^{\pi_5}, \overbrace{\texttt{intersect S1 S2 S12}}^{\pi_6}, \overbrace{\texttt{I12 \#-- S12}}^{\pi_7}, \overbrace{\texttt{I12 \#<= 1}}^{\pi_8},$$
$$\overbrace{\texttt{labelingSets [S1,S2]}}^{\pi_9} \square \texttt{I1 \#-- S1, I2 \#-- S2} \square \emptyset \square \texttt{0\#<=I1, I1\#<=5, 0\#<=I2, I2\#<=5} \square$$
$$\pi_2$$

π_4 is sent to the Mediatorial store, and $solve^{\mathcal{M}}$ is invoked. Likewise proceeds with π_5. Next, the constraint π_6 is processed in a similar way to π_2, generating a new bridge and new $\mathcal{FD}$ constraints:

$$bridges^{\mathcal{FS}\to\mathcal{FD}}(\pi_6, B) = \{\texttt{I12 \#-- S12}\} = B'$$
$$proj^{\mathcal{FS}\to\mathcal{FD}}(\pi_6, B'') = \{\texttt{I12\#<= I1, I12\#<= I2}\}, \text{ where } B'' = B \cup B'$$

Then, rule **SC** is successively applied to π_7, π_8 and π_9, submitting them to the $\mathcal{M}$, $\mathcal{FD}$ and $\mathcal{FS}$ stores, respectively. Finally, P and C are empty.

$$\exists\ \texttt{I1,I2,S12,I12}.\emptyset \square \emptyset \square \texttt{I1 \#-- S1, I2 \#-- S2}, \pi_4, \pi_5, \pi_7 \square \emptyset \square \pi_8, \texttt{0\#<=I1, I1\#<=5,}$$
$$\texttt{0\#<=I2, I2\#<=5, I12\#<=I1, I12\#<=I2} \square \pi_2, \pi_6, \pi_9$$

Taking into consideration `I1 #-- S1`, $\pi_4 \equiv$ `3 #-- S1`, `I2 #-- S2`, and $\pi_5 \equiv$ `3 #-- S2`, it is possible to infer the equalities `I1 == 3` and `I2 == 3` applying rule **IE** of Table B.13 twice. These equalities are posted to the $\mathcal{H}$ store, producing the substitution $\sigma_4 = \{$`I1` $\mapsto$ `3`, `I2` $\mapsto$ `3`$\}$.

$$\exists\ \texttt{S12,I12}.\emptyset \square \emptyset \square \texttt{3 \#-- S1, 3 \#-- S2}, \pi_7 \square \emptyset \square \pi_8, \texttt{I12\#<=3} \square \pi_2, \pi_6, \pi_9.$$

If any store is not in solved form yet, its respective solver is appropriately invoked. In $\mathcal{FS}$, `labelingSets` enumerates all ground values for `S1` and `S2`. A first attempt is `S1` $\mapsto$ {1,2,3}, `S2` $\mapsto$ {1,2,3}, but this substitution does not satisfy π_6, π_7 and π_8. The rest of the valuations are checked on backtracking until a solution is obtained, as for example `S1` $\mapsto$ {1,2,3}, `S2` $\mapsto$ {1,4,5}. □

The semantic results for the cooperative goal solving calculus $CCLNC(\mathcal{C}_{\mathcal{FD}.\mathcal{FS}})$: *soundness* and *limited completeness* are decribed in the thesis.

B.12 Implementation

The $CCLNC(\mathcal{C}_{\mathcal{FD}.\mathcal{R}})$ and $CCLNC(\mathcal{C}_{\mathcal{FD}.\mathcal{FS}})$ computational models detailed in the previous sections have been implemented into the $\mathcal{TOY}$ system. This section sketches this implementation and the reader is referred to [ACE+07, EM04, EFS06, EFS07, EFS08] for more details. The prototype and the code implementation of examples are available at `http://gpd.sip.ucm.es/sonia/Prototype.html`.

Programs in $\mathcal{TOY}$ are compiled and executed in Prolog. The compilation generates Prolog

code that implements goal solving by constrained lazy narrowing guided by *definitional trees*, a well known device for ensuring an optimal behaviour of lazy narrowing [LLR93, AEH94, AEH00, Vad07]. $\mathcal{TOY}$ relies on an efficient Prolog system, SICStus Prolog [SIC11], which provides many libraries, including constraint solvers for the domains $\mathcal{FD}$ and $\mathcal{R}$.

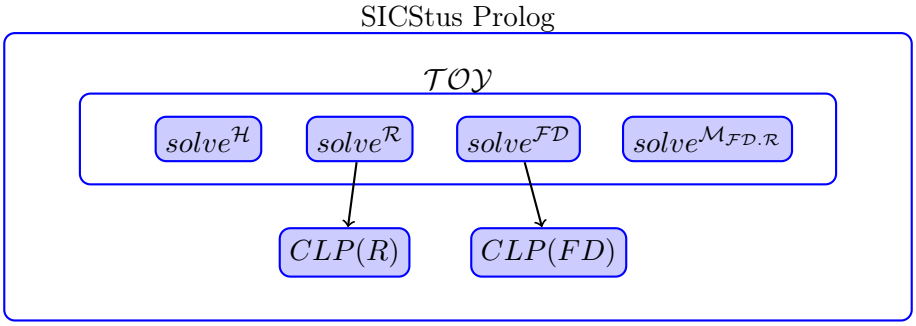


Figure B.5: Architectural Components of the Cooperation $\mathcal{C}_{\mathcal{FD}.\mathcal{R}}$ in $\mathcal{TOY}$

Figure B.5 shows the architectural components of the cooperation of the three pure constraint domains $\mathcal{H}$, $\mathcal{R}$ and $\mathcal{FD}$ which are combined with a mediatorial domain $\mathcal{M}_{\mathcal{FD}.\mathcal{R}}$ to yield the coordination domain $\mathcal{C}_{\mathcal{FD}.\mathcal{R}} = \mathcal{M}_{\mathcal{FD}.\mathcal{R}} \oplus \mathcal{H} \oplus \mathcal{FD} \oplus \mathcal{R}$. The solvers and constraint stores for the domains $\mathcal{FD}$ and $\mathcal{R}$ are provided by the SICStus Prolog constraint libraries. The impedance mismatch problem among the host language constraint primitives and these solvers is tackled by glue code. Proper $\mathcal{FD}$ and $\mathcal{R}$ constraints, as well as Herbrand constraints specific to $\mathcal{FD}$ and $\mathcal{R}$ are posted to the respective stores and handled by the respective SICStus Prolog solvers. On the other hand, the stores and solvers for the domains $\mathcal{H}$ and $\mathcal{M}_{\mathcal{FD}.\mathcal{R}}$ are built into the code of the $\mathcal{TOY}$ implementation, rather than being provided by the underlying SICStus Prolog system. In order to deal with $\mathcal{H}$ and $\mathcal{M}_{\mathcal{FD}.\mathcal{R}}$ constraints, the $\mathcal{TOY}$ system uses a so-called *mixed store* which keeps a representation of the $\mathcal{H}$ and $\mathcal{M}_{\mathcal{FD}.\mathcal{R}}$ stores as one single Prolog structure. It includes encodings of $\mathcal{H}$-constraints in solved form (i.e., totality constraints `X == X` and disequality constraints `X /= t`), as well as encodings of bridges and antibridges. The implementation of the $\mathcal{H}$ and $\mathcal{M}$ solvers in $\mathcal{TOY}$ is plugged into the Prolog code of various predicates which control the transformation of the mixed store (passed as argument) by means of two auxiliary arguments `Cin` and `Cout`.

Regarding projections, the $\mathcal{TOY}$ implementation has been designed to support two modes of use: A *disabled projections* mode which allows to solve mediatorial constraints, but computes no projections; and a *enabled projections* mode which also computes projections. For each particular problem, the user can analyze the trade-off between communication flow and performance gain and decide the best option to execute a goal in the context of a given program.

Figure B.6 shows the architectural components of the cooperation of the three pure constraint domains $\mathcal{H}$, $\mathcal{FD}$ and $\mathcal{FS}$ which are combined with a mediatorial domain $\mathcal{M}$ to yield the coordination domain $\mathcal{C}_{\mathcal{FD}.\mathcal{R}} = \mathcal{M} \oplus \mathcal{H} \oplus \mathcal{FD} \oplus \mathcal{FS}$. The $CCLNC(\mathcal{C}_{\mathcal{FD}.\mathcal{FS}})$ calculus uses lazy narrowing for processing calls to program defined functions, ensuring that function calls are evaluated only as far as demanded by the evaluation of primitive constraints. Primitive

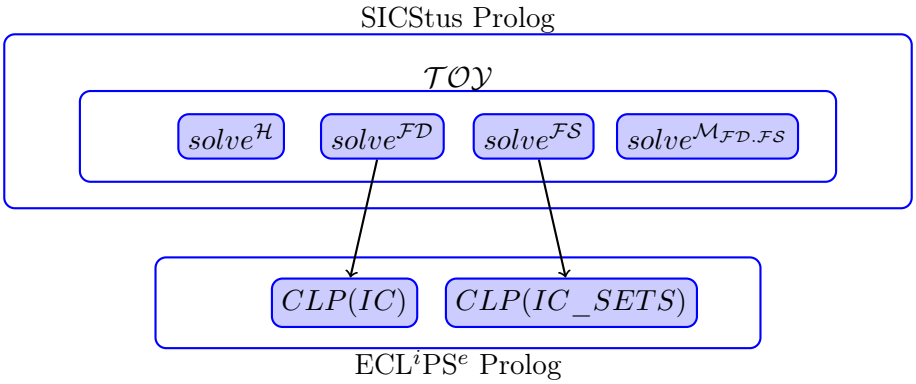


Figure B.6: Architectural Components of the Cooperation $\mathcal{C}_{\mathcal{FD}.\mathcal{R}}$ in $\mathcal{TOY}$

constraint evaluation is done by the constraint solvers available in ECL^{i}PS^e libraries `ic` and `ic_sets`. Therefore, there is a constraint translation process from $\mathcal{TOY}$ to ECL^{i}PS^e for similar constraints in both systems. Additional constraints, such as `<<`, have been implemented using daemons, a special feature available in ECL^{i}PS^e. The operator `<<` is related to the strict partial order between two finite set variables. In particular, `S1 << S2` means that the greatest element in the set `S1` is lower than the lowest element in the set `S2`. This constraint must hold during the computation: when one of the arguments becomes ground, the bounds of the other argument are set and the daemon is removed. That is, when the set `S1` is ground, the corresponding daemon wakes up, reduces the domain of `S2` appropriately, and ends.

For performing the communication between both processes, standard input/output library predicates (i.e., `write` predicates) have been used operating on a pipe that interconnects them. This scheme can be easily moved to a socket-based communication to allow more complex scenarios (for example, executing processes in different machines). Requests and results must be properly arranged for sending them through the streams that connect both processes. In particular, several issues must be addressed in order to make the system work. The first issue is that requests are independent, this means that if a request contains a variable, standard `write` predicates will produce a representation of such variable that when read by the ECL^{i}PS^e server a fresh variable will be produced. Two requests referring to the same variable in $\mathcal{TOY}$ will contain different fresh variables when evaluated by ECL^{i}PS^e. To address this issue, variables are represented by a ground term `'$setvar'(Id)` or `'$fdvar'(Id)` where `Id` is an integer identifying the corresponding variable. Therefore, a request has the form of a Prolog term without variables. For instance, for evaluating the intersection of two set variables, a request of the form `intersect(X,Y,Z)` is sent to ECL^{i}PS^e, where `X`, `Y` and `Z` are either representations of sets as Prolog lists or variable representations. For example, `intersect('$setvar'(3), [1,3,5], '$setvar'(6))` requests the intersection of the set variable with identifier 3 with the set `[1,3,5]`, and unifying the result with the variable with identifier 6.

Both $\mathcal{TOY}$ and the ECL^{i}PS^e server maintain a table with the state of the variables used in previous requests in order to relate different requests and keep the constraints on the variables just used. In $\mathcal{TOY}$ set variables are marked with a specific attribute with the

identifier to be used in the requests to ECL^{i}PS^e. Set variables are actually constrained in the ECL^{i}PS^e process only, and $\mathcal{TOY}$ only keeps the identifier to refer to it in future requests to ECL^{i}PS^e.

An important issue is related to the cooperation between constraint domains. There are two cases in which set variables cooperate with other constraint domains. The first case happens when obtaining the cardinality of a set by means of a bridge constraint, whereas the second case occurs when a set variable is bound to a list in the Herbrand domain, e.g. during labeling. Since ECL^{i}PS^e includes a finite domain tightly integrated with the integer sets domain, the finite domain already existing in $\mathcal{TOY}$ (provided by the SICStus implementation) has been replaced by the ECL^{i}PS^e finite domain. Therefore, primitive operations in the finite domain of $\mathcal{TOY}$ have been replaced by requests to the ECL^{i}PS^e server regarding finite domain variables (primitive functions shown in Subsection B.6.2).

Another issue that must be addressed is related to backtracking. In the presence of backtracking, requests performed in ECL^{i}PS^e must be undone. The ECL^{i}PS^e server has been designed to accept special requests for backtracking that unconstraint set and finite domain variables. Backtracking during execution of the $\mathcal{TOY}$ program is mimicked in ECL^{i}PS^e server by issuing a backtracking command that performs backtracking in ECL^{i}PS^e constraints accordingly. Failure in the application of a constraint in ECL^{i}PS^e is propagated to $\mathcal{TOY}$ in the same way.

The prototype for the cooperation between the domains $\mathcal{FD}$ and $\mathcal{FS}$ includes different implementation approaches.

First approach, the *interactive mode*, is oriented to model reasoning, i.e., when the model can be modified during solving. It is based on an interactive communication between $\mathcal{TOY}$ and ECL^{i}PS^e. When $\mathcal{TOY}$ begins execution, ECL^{i}PS^e executes in a different process as a server, accepting and executing requests from the $\mathcal{TOY}$ process. For every primitive function call that needs to be evaluated on sets of integers or on finite domain, a request is issued to the ECL^{i}PS^e server. The server evaluates the function call and returns the result to the $\mathcal{TOY}$ process. Therefore, when $\mathcal{TOY}$ requires a constraint to be solved in ECL^{i}PS^e, $\mathcal{TOY}$ posts it and blocks until ECL^{i}PS^e server returns an answer. This approach takes advantage of all features of the $\mathcal{TOY}$ language with the $\mathcal{FD}$ and $\mathcal{FS}$ domains. However, the communication between $\mathcal{TOY}$ and ECL^{i}PS^e for each constraint has a negative effect on the efficiency of the program. An alternative approach was the batch mode, intended for classic CP applications, where constraints are first specified and then posted and solved.

Second approach, the *batch mode*, delays the computation of some or all constraints of the current goal, sending them to ECL^{i}PS^e at the end of the $\mathcal{TOY}$ narrowing procedure. This mode avoids the interactive communication between both processes that may slow down the execution of goals. However, communication is inevitable when more answers are demanded for a given goal. The backtracking mechanism forces interactive communication between both processes, even though the constraints are sent in batch mode. In addition, for classic CP applications, we measured the time that ECL^{i}PS^e takes in solving the set of constraints sent by $\mathcal{TOY}$, but removing any overhead produced by the inter-process communication.

Third approach, the *ECL^{i}PS$^e_{gen}$ mode*. This has been done by generating an ECL^{i}PS^e Prolog program that contains all atomic primitive constraints created by $\mathcal{TOY}$ in the nar-

rowing process. This ECL^{i}PS^e Prolog program is performed in a different process.

B.13 Experiments

The implementation has been tested in two different ways. The first for the cooperation of $\mathcal{R}$ and $\mathcal{FD}$ domains and the second for the cooperation of $\mathcal{FD}$ and $\mathcal{FS}$ domains. This Section only presents the conclusions of the performances of the prototypes for the cooperation. These conclusions have been extracted from the works [EFH$^+$09, ECS12].

Experiments on the prototype of cooperation between $\mathcal{R}$ and $\mathcal{FD}$ lead us to several conclusions. Firstly, we conclude that the activation of the domain cooperation mechanisms between $\mathcal{FD}$ and $\mathcal{R}$ does not penalize the execution time in problems which can be solved by using the domain $\mathcal{FD}$ alone. Secondly, we also conclude that the cooperation mechanism using projections helps to speed-up the execution time in problems where a real cooperation between $\mathcal{FD}$ and $\mathcal{R}$ is needed. Thirdly, our experiments show a good performance of our implementation with respect to the closest related system we are aware of. In summary, we conclude that our approach to the cooperation of constraint domains has been effectively implemented in a practical system.

The implementation of the cooperation between $\mathcal{FD}$ and $\mathcal{FS}$ has been tested in all modes and the conclusions are: $\mathcal{TOY}$ batch mode is faster than the interactive mode, in which lazy narrowing computation in $\mathcal{TOY}$ is interleaved with constraint solving in ECL^{i}PS^e. The former case does not require to establish a communication from the ECL^{i}PS^e server to $\mathcal{TOY}$ every time a constraint is posted, and therefore computation in ECL^{i}PS^e is performed without interruption for communicating intermediate results to $\mathcal{TOY}$. Batch mode in these particular examples is very appropriate, since the structure of the solution is first generated in $\mathcal{TOY}$ together with all constraints, which are solved in a second step, in a similar way usual CP problems are specified and solved. It is remarkable that in all cases ECL^{i}PS$^e_{gen}$ mode takes approximately the same time or improve than the original ECL^{i}PS^e program, even for examples that generate a very high number of solutions. Therefore, $\mathcal{TOY}$ does not introduce a noticeable overhead with respect to the tests directly performed in ECL^{i}PS^e. Moreover, in some cases, projections prune the search tree to improve dramatically the system performance.

Bibliografía

[AB00] Francisco Azevedo and Pedro Barahona. Modelling digital circuits problems with set constraints. In *Proceedings of CL 2000*, volume 1861 of *LNCS*, pages 414–428. Springer, 2000.

[ACE+07] Puri Arenas, Rafael Caballero, Sonia Estévez, Antonio J. Fernández, Ana Gil, Francisco J. López-Fraguas, Mario Rodríguez-Artalejo, Jaime Sánchez, and Fernando Sáenz-Pérez. $\mathcal{TOY}$. a multiparadigm declarative language. version 2.3.1, 2007. Rafael Caballero and Jaime Sánchez, Editors. Available at `http://toy.sourceforge.net`.

[ACI14] ACIDE web site. `http://www.fdi.ucm.es/profesor/fernan/acide/`, 2014.

[AEH94] Sergio Antoy, Rachid Echahed, and Michael Hanus. A needed narrowing strategy. In *Proceedings of POPL 1994*, pages 268–279. ACM Press, 1994.

[AEH00] Sergio Antoy, Rachid Echahed, and Michael Hanus. A needed narrowing strategy. *Journal of the ACM*, 74(4):776–822, 2000.

[AG97] Jesús M. Almendros-Jiménez and Ana Gil-Luezas. Lazy narrowing with parametric order sorted types. In *Proceedings of ALP/HOA 1997*, volume 1298 of *LNCS*, pages 159–173. Springer, 1997.

[AGG96] Jesús M. Almendros-Jiménez, Antonio Gavilanes-Franco, and Ana Gil-Luezas. Algebraic semantics for functional logic programming with polymorphic order-sorted types. In *Proceedings of ALP 1996*, volume 1139 of *LNCS*, pages 299–313. Springer, 1996.

[AGL94] Puri Arenas-Sánchez, Ana Gil-luezas, and Francisco J. López-Fraguas. Combining lazy narrowing with disequality constraints. In *Proceedings of PLILP 1994*, LNCS, pages 385–399. Springer, 1994.

[AH10] Sergio Antoy and Michael Hanus. Functional logic programming. *Communications of the ACM*, 53(4):74–85, 2010.

[AHH+02] Elvira Albert, Michael Hanus, Frank Huch, Javier Oliver, and Germán Vidal. Operational semantics for functional logic languages. *Electronic Notes in Theoretical Computer Science*, 76:1–19, 2002.

[AHLU96] Puri Arenas-Sánchez, María T. Hortalá-González, Francisco J. López-Fraguas, and Eva Ullán. Real constraints within a functional logic language. In *Proceedings of APPIA-GULP-PRODE 1996*, pages 451–464, 1996.

[ALR98] Puri Arenas-Sánchez, Francisco J. López-Fraguas, and Mario Rodríguez-Artalejo. Embedding multiset constraints into a lazy functional logic language. In *Proceedings of PLILP/ALP 1998*, volume 1490 of *LNCS*, pages 429–444. Springer, 1998.

[ALR99] Puri Arenas-Sánchez, Francisco J. López-Fraguas, and Mario Rodrúguez-Artalejo. Functional plus logic programming with built-in and symbolic constraints. In *Proceedings of PPDP 1999*, volume 1702 of *LNCS*, pages 152–169. Springer, 1999.

[AR97a] Puri Arenas-Sánchez and Mario Rodríguez-Artalejo. A lazy narrowing calculus for functional logic programming with algebraic polymorphic types. In *Proceedings of ILPS 1997*, pages 53–67. MIT Press, 1997.

[AR97b] Puri Arenas-Sánchez and Mario Rodríguez-Artalejo. A semantic framework for functional logic programming with algebraic polymorphic types. In *Proceedings of TAPSOFT 1997*, volume 1214 of *LNCS*, pages 453–464. Springer, 1997.

[AR01] Puri Arenas-Sánchez and Mario Rodríguez-Artalejo. A general framework for lazy functional logic, programming with algebraic polymorphic types. *Theory and Practice of Logic Programming*, 1(2):185–245, 2001.

[AW07] Krzysztof R. Apt and Mark Wallace. *Constraint Logic Programming using Eclipse*. Cambridge University Press, New York, NY, USA, 2007.

[Aze07a] Francisco Azevedo. An attempt to dynamically break symmetries in the social golfers problem. In *Proceedings of CSCLP 2006*, volume 4651 of *LNCS*, pages 33–47. Springer, 2007.

[Aze07b] Francisco Azevedo. Cardinal: A finite sets constraint solver. *Constraints*, 12(1):93–129, 2007.

[B-P] B-PROLOG, CLP system. `http://www.picat-lang.org/bprolog`.

[Bar11] Roman Barták. *History of Constraint Programming*. Wiley Encyclopedia of Operations Research and Management Science, 2011.

[BB94] Wray L. Buntine and Hans-Jürgen Bürckert. On solving equations and disequations. *Journal of the ACM*, 41(4):591–629, 1994.

[BN98] Franz Baader and Tobias Nipkow. *Term rewriting and all that*. Cambridge University Press, New York, USA, 1998.

[Cab04] Rafael Caballero. *Técnicas de diagnóstico y depuración declarativa para lenguajes lóico-funcionales*. PhD thesis, Departamento de Sistemas Informáticos y Programación, Universidad Complutense de Madrid, 2004.

[Cas14] Ignacio Castiñeiras-Pérez. *Mejora de la eficiencia de resolución del sistema $\mathcal{TOY}(\mathcal{FD})$ y su aplicación a problemas reales de la industria.* PhD thesis, Departamento de Sistemas Informáticos y Computación, Universidad Complutense de Madrid, 2014.

[CCES12] Ignacio Castiñeiras-Pérez, Jesús Correas-Fernández, Sonia Estévez-Martín, and F. Sáenz-Pérez. TOY: A CFLP Language and System. *ALP Newsletter*, 2012.

[CD96] Philippe Codognet and Daniel Diaz. Compiling constraints in CLP(FD). *The Journal of Logic Programming*, 27(3):185–226, 1996.

[CHO] CHOCO: Java Library for Constraint Programming. `http://choco-solver.org`.

[CL89] Hubert Comon and Pierre Lescanne. Equational problems and disunification. *Journal of Symbolic Computation*, 7(3/4):371–425, 1989.

[CLR01] Rafael Caballero, Francisco J. López-Fraguas, and Mario Rodríguez-Artalejo. Theoretical Foundations for the Declarative Debugging of Lazy Functional Logic Programs. In *Proceedings of FLOPS 2001*, volume 2024 of *LNCS*, pages 170–184. Springer, 2001.

[CM03] William F. Clocksin and Christopher S. Mellish. *Programming in Prolog.* Springer, Berlin, 5 edition, 2003.

[Col84] Alain Colmerauer. Equations and inequations on finite and infinite trees. In *Proceedings of FGCS 1984*, pages 85–99, 1984.

[Col90] Alain Colmerauer. An introduction to PROLOG III. *Communications of the ACM*, 33(7):69–90, 1990.

[Com91] Hubert Comon. Disunification: A survey. In *Computational Logic - Essays in Honor of Alan Robinson*, pages 322–359. MIT Press, 1991.

[CR02] Rafael Caballero and Mario Rodríguez-Artalejo. A Declarative Debugging System for Lazy Functional Logic Programs. *Electronic Notes in Theoretical Computer Science*, 64:113–175, 2002.

[CR04] Rafael Caballero and Mario Rodríguez-Artalejo. DDT: a declarative debugging tool for functional-logic languages. In *Proceedings of FLOPS 2004*, volume 2998 of *LNCS*, pages 70–84. Springer, 2004.

[Cur12] Curry Web Site. `http://www-ps.informatik.uni-kiel.de/currywiki`, 2012.

[DDPR03] Alessandro Dal Palú, Agostino Dovier, Enrico Pontelli, and Gianfranco Rossi. Integrating finite domain constraints and CLP with sets. In *Proceedings of PPDP 2003*, pages 219–229. ACM, 2003.

[DGH+99a] Bart Demoen, María J. García de la Banda, Warwick Harvey, Kim Marriott, and Peter J. Stuckey. Herbrand constraint solving in HAL. In *Proceedings of ICLP 1999*, pages 260–274. MIT Press, 1999.

[DGH+99b] Bart Demoen, María J. García de la Banda, Warwick Harvey, Kim Marriott, and Peter J. Stuckey. An overview of HAL. In *Proceedings of CP 1999*, volume 1713 of *LNCS*, pages 174–188. Springer, 1999.

[DGP91] John Darlington, Yike Guo, and Helen Pull. Introducing constraint functional logic programming. In *Procceedings of DP 1991*, pages 20–34. Springer, 1991.

[DGP92] John Darlington, Yike Guo, and Helen Pull. A new perspective on integrating functional and logic languages. In *Proceedings of FGCS 1992*, pages 682–693, 1992.

[DL86] Doug DeGroot and Gary Lindstrom. *Logic Programming: Functions, Relations, and Equations*. Prentice-Hall, Upper Saddle River, NJ, USA, 1986.

[DM82] Luís Damas and Robin Milner. Principal type-schemes for functional programs. In *Proceedings of POPL 1982*, pages 207–212. ACM Press, 1982.

[DOPR91] Agostino Dovier, Eugenio G. Omodeo, Enrico Pontelli, and Gianfranco Rossi. {log}: A logic programming language with finite sets. In *Proceedings of ICLP 1991*, pages 111–124. MIT Press, 1991.

[DPPR00] Agostino Dovier, Carla Piazza, Enrico Pontelli, and Gianfranco Rossi. Sets and constraint logic programming. *ACM Transactions on Programming Languages and Systems*, 22(5):861–931, 2000.

[DSGH03] Gregory J. Duck, Peter J. Stuckey, María J. García de la Banda, and Christian Holzbaur. Extending arbitrary solvers with constraint handling rules. In *Proceedings of PPDP 2003*, pages 79–90. ACM, 2003.

[dVV08] Rafael del Vado-Vírseda. *Un Esquema de Programación Lóico-Funcional con Restricciones: Marco Teórico y Aplicación a la Depuración Declarativa*. PhD thesis, Departamento de Sistemas Informáticos y Computación, Universidad Complutense de Madrid, 2008.

[ECL] ECLiPSe Web Site. `http://eclipseclp.org`.

[ECS12] Sonia Estévez-Martín, Jesús Correas-Fernández, and Fernando Sáenz-Pérez. Extending the $\mathcal{TOY}$ System with the ECL^{i}PS^e Solver over Sets of Integers. In *Proceedings of FLOPS 2012*, volume 7294 of *LNCS*, pages 120–135. Springer, 2012.

[EFH+07a] Sonia Estévez-Martín, Antonio J. Fernández-Leiva, María T. Hortalá-González, Mario Rodríguez-Artalejo, and Rafael del Vado-Vírseda. A fully sound goal solving calculus for the cooperation of solvers in the $cflp$ scheme. *Electronic Notes in Theoretical Computer Science*, 177:235–252, 2007.

[EFH+07b] Sonia Estévez-Martín, Antonio J. Fernández-Leiva, María T. Hortalá-González, Mario Rodríguez-Artalejo, Fernando Sáenz-Pérez, and Rafael del Vado-Vírseda. A proposal for the cooperation of solvers in constraint functional logic programming. *Electronic Notes in Theoretical Computer Science*, 188:37–51, 2007.

[EFH+08] Sonia Estévez-Martín, Antonio J. Fernández, María T. Hortalá-González, Mario Rodríguez-Artalejo, Fernando Sáenz-Pérez, and Rafael del Vado-Vírseda. Cooperation of constraint domains in the $\mathcal{TOY}$ system. In *Proceedings of PPDP 2008*, pages 258–268. ACM Press, 2008.

[EFH+09] Sonia Estévez-Martín, Antonio J. Fernández, María T. Hortalá-González, Mario Rodríguez-Artalejo, Fernando Sáenz-Pérez, and Rafael Del Vado-Vírseda. On the Cooperation of the Constraint Domains $\mathcal{H}$, $\mathcal{R}$ and $\mathcal{FD}$ in $CFLP$. *Theory and Practice of Logic Programming*, 9:415–527, 2009.

[EFS06] Sonia Estévez-Martín, Antonio Fernández-Leiva, and Fernando Sáenz-Pérez. Implementing $\mathcal{TOY}$, a Constraint Functional Logic Programming with Solver Cooperation. Research Report LCC ITI 06-8, Universidad de Málaga, 2006.

[EFS07] Sonia Estévez-Martín, Antonio J. Fernández-Leiva, and Fernando Sáenz-Pérez. About implementing a constraint functional logic programming system with solver cooperation. In *Proceedings of CICLOPS 2007*, pages 57–71, 2007.

[EFS08] Sonia Estévez-Martín, Antonio J. Fernández-Leiva, and Fernando Sáenz-Pérez. Playing with $\mathcal{TOY}$: Constraints and Domain Cooperation. In *Proceedings of ESOP 2008*, volume 4960 of *LNCS*, pages 112–115. Springer, 2008.

[EFS09a] Sonia Estévez-Martín, Antonio J. Fernández-Leiva, and Fernando Sáenz-Pérez. Cooperation of the Finite Domain and Set Solvers in $\mathcal{TOY}$. In *Proceedings of Prole 2009*, pages 217–226, 2009.

[EFS09b] Sonia Estévez-Martín, Antonio J. Fernández-Leiva, and Fernando Sáenz-Pérez. $\mathcal{TOY}$: A system for experimenting with cooperation of constraint domains. *Electronic Notes in Theoretical Computer Science*, 258(1):79–91, 2009.

[EM04] Sonia Estévez-Martín. Restricciones sobre dominios finitos en un lenguaje lógico funcional, 2004. Master's Thesis, University Complutense of Madrid.

[EV05] Sonia Estévez-Martín and Rafael Del Vado-Vírseda. Designing an efficient computation strategy in $CFLP(FD)$ using definitional trees. In *Proceedings of WCFLP 2005*, pages 23–31, 2005.

[FaC] FaCiLe Web Site. `http://www.recherche.enac.fr/log/facile`.

[Fer92] Maribel Fernández. Narrowing based procedures for equational disunification. *Applicable Algebra in Engineering, Communication and Computing*, 3:1–26, 1992.

[Fer00] Antonio J. Fernández. *A generic, collaborative framework for interval constraint solving.* PhD thesis, Departamento de Lenguajes y Ciencias de la Computación, University of Málaga, 2000.

[FH99] Antonio J. Fernández and Patricia M. Hill. An interval lattice-based constraint solving framework for lattices. In *Proceedings of FLOPS 1999*, volume 1722 of *LNCS*, pages 194–208. Springer, 1999.

[FH04] Antonio J. Fernández and Patricia M. Hill. An interval constraint system for lattice domains. *ACM Transactions on Programming Languages and Systems*, 26(1):1–46, 2004.

[FH06] Antonio J. Fernández and Patricia M. Hill. An interval constraint branching scheme for lattice domains. *Journal of Universal Computer Science*, 12(11):1466–1499, 2006.

[FHM03a] Stephan Frank, Petra Hofstedt, and Pierre R. Mai. A flexible meta-solver framework for constraint solver collaboration. In *Proceedings of KI 2003*, volume 2821 of *LNCS*, pages 520–534. Springer, 2003.

[FHM03b] Stephan Frank, Petra Hofstedt, and Pierre R. Mai. Meta-S: A strategy-oriented meta-solver framework. In *Proceedings of FLAIRS 2003*, pages 177–181. AAAI Press, 2003.

[FHR05] Stephan Frank, Petra Hofstedt, and Dirk Reckman. Meta-S, combining solver cooperation and programming languages. In *Proceedings of W(C)LP 2005*, pages 159–162. Universität Ulm, Germany, 2005.

[FHS05a] Antonio J. Fernández, María T. Hortalá-González, and Fernando Sáenz-Pérez. Programming with $\mathcal{TOY}(\mathcal{FD})$. In *Proceedings of CP 2005*, volume 3709 of *LNCS*, pages 878–878. Springer, 2005.

[FHS05b] Antonio J. Fernández, María T. Hortalá-González, and Fernando Sáenz-Pérez. Solving FD constraints in $\mathcal{TOY}(\mathcal{FD})$. In *Proceedings of BeyondFD 2005*, pages 77–91, 2005.

[FHSV07] Antonio J. Fernández-Leiva, María T. Hortalá-González, Fernando Sáenz-Pérez, and Rafael Del Vado-Vírseda. Constraint functional logic programming over finite domains. *Theory and Practice of Logic Programming*, 7(5):537–582, 2007.

[Gar14] Yolanda García Ruiz. *Técnicas de Detección y Diagnosis de Errores en Consultas de Bases de Datos*. PhD thesis, Departamento de Sistemas Informáticos y Computación, Universidad Complutense de Madrid, 2014.

[Gec] Gecode: C++ library. `http://www.gecode.org`.

[Ger94] Carmen Gervet. Conjunto: constraint logic programming with finite set domains. In *Proceedings of LP 1994*, pages 339–358. MIT Press, 1994.

[Ger97] Carmen Gervet. Interval propagation to reason about sets: Definition and implementation of a practical language. *Constraints*, 1(3):191–244, 1997.

[GHLR96] Juan C. González-Moreno, María T. Hortalá-González, Francisco J. López-Fraguas, and M. Rodríguez-Artalejo. A rewriting logic for declarative programming. In *Proceedings of ESOP 1996*, volume 1058 of *LNCS*, pages 156–172. Springer, 1996.

[GHLR99] Juan C. González-Moreno, María T. Hortalá-González, Francisco J. López-Fraguas, and M. Rodríguez-Artalejo. An approach to declarative programming based on a rewriting logic. *Journal of Logic Programming*, 40:47–87, 1999.

[GHR97] Juan C. González-Moreno, María T. Hortalá-González, and Mario Rodríguez-Artalejo. A higher-order rewriting logic for functional logic programming. In *Proceedings of ICLP 1997*, pages 153–167. MIT Press, 1997.

[GHR01] Juan C. González-Moreno, María T. Hortalá-González, and Mario Rodríguez-Artalejo. Polymorphic types in functional logic programming. *Journal of Functional and Logic Programming*, 2001(1):1–71, 2001.

[GJM+01] María J. García de la Banda, David Jeffery, Kim Marriott, Nicholas Nethercote, Peter J. Stuckey, and Christian Holzbaur. Building constraint solvers with HAL. In *Proceedings of ICLP 2001*, volume 2237 of *LNCS*, pages 90–104. Springer, 2001.

[GMB01] Laurent Granvilliers, Eric Monfroy, and Frédéric Benhamou. Cooperative solvers in constraint programming: a short introduction. *ALP Newsletter*, 14(2), 2001.

[GV06] Carmen Gervet and Pascal Van Hentenryck. Length-lex ordering for set CSPs. In *Proceedings of AAAI 2006*, pages 48–53. AAAI Press, 2006.

[Hal03] Hal Web Site. `http://www.csse.monash.edu.au/~mbanda/hal/`, 2003.

[Han92] Michael Hanus. On the completeness of residuation. In *Proceedings of JICSLP 1992*, pages 192–206. MIT Press, 1992.

[Han94] Michael Hanus. The integration of functions into logic programming: From theory to practice. *Journal of Logic Programming*, 19/20:583–628, 1994.

[Han97] Michael Hanus. Teaching functional and logic programming with a single computational model. In *Proceedings of PLILP 1997*, volume 1292 of *LNCS*, pages 335–350. Springer, 1997.

[Han07] Michael Hanus. Multi-paradigm declarative languages. In *Proceedings of ICLP 2007*, volume 4670 of *LNCS*, pages 45–75. Springer, 2007.

[Han13] M. Hanus. Functional logic programming: From theory to Curry. In *Programming Logics - Essays in Memory of Harald Ganzinger*, volume 7797 of *LNCS*, pages 123–168. Springer, 2013.

[HBC+12] Manuel V. Hermenegildo, Francisco Bueno, Manuel Carro, Pedro López-García, Edison Mera, José F. Morales, and Germán Puebla. An overview of Ciao and its design philosophy. *Theory and Practice of Logic Programming*, 12(1-2):219–252, 2012.

[Hin69] Roger Hindley. The principal type-scheme of an object in combinatory logic. *Transactions of the American Mathematical Society*, 146:29–60, 1969.

[HK96] Michael Hanus and Herbert Kuchen. Integration of functional and logic programming. *ACM Computing Surveys*, 28(2):306–308, 1996.

[HKW04] Brahim Hnich, Zeynep Kiziltan, and Toby Walsh. Combining symmetry breaking with other constraints: Lexicographic ordering with sums. In *Proceedings of AMAI 2004*, 2004.

[HLSU97] María T. Hortalá-González, Francisco J. López-Fraguas, Jaime Sánchez-Hernández, and Eva Ullán-Hernández. Declarative Programming with Real Constraints. Research Report SIP 5997, Universidad Complutense Madrid, 1997.

[Hof00a] Petra Hofstedt. Better communication for tighter cooperation. In *Proceedings of CL 2000*, volume 1861 of *LNCS*, pages 342–357. Springer, 2000.

[Hof00b] Petra Hofstedt. Cooperating constraint solvers. In *Proceedings of CP 2000*, volume 1894 of *LNCS*, pages 520–524. Springer, 2000.

[Hof01] Petra Hofstedt. *Cooperation and Coordination of Constraint Solvers*. PhD thesis, Technischen Universität Dresden, Fakultät Informatik, 2001.

[HP07] Petra Hofstedt and Peter Pepper. Integration of declarative and constraint programming. *Theory and Practice of Logic Programming*, 7(1-2):93–121, 2007.

[HS98] Michael Hanus and Frank Steiner. Controlling search in declarative programs. In *Proceedings of PLILP/ALP 1998*, volume 1490 of *LNCS*, pages 374–390. Springer, 1998.

[HS05] Peter Hawkins and Peter J. Stuckey. Solving set constraint satisfaction problems using ROBDDs. *Journal of Artificial Intelligence Research*, 24:109–156, 2005.

[Hud89] Paul Hudak. Conception, evolution, and application of functional programming languages. *ACM Computing Surveys*, 21(3):359–411, 1989.

[Hus93] Heinrich Hussmann. *Nondeterminism in Algebraic Specification and Algebraic Programs*. Progress in Theoretical Computer Science Series. Birkhäuser, 1993.

[JL87] Joxan Jaffar and Jean-Louis Lassez. Constraint logic programming. In *Proceedings of POPL 1987*, pages 111–119. ACM Press, 1987.

[JM94] Joxan Jaffar and Michael J. Maher. Constraint Logic Programming: A Survey. *Journal of Logic Programming*, 19/20:503–581, 1994.

[JMMS98] Joxan Jaffar, Michael J. Maher, Kim Marriott, and Peter J. Stuckey. The semantics of constraints logic programs. *Journal of Logic Programming*, 37(1-3):1–46, 1998.

[JMSY92] Joxan Jaffar, Spiro Michaylov, Peter J. Stuckey, and Roland H.C. Yap. The CLP(R) language and system. *ACM Transactions on Programming Languages and Systems*, 14(3):339–395, 1992.

[JSe12] JSetL: Java library. `http://cmt.math.unipr.it/jsetl`, 2012.

[KLMR92] Herbert Kuchen, Francisco J. López-Fraguas, Juan J. Moreno-Navarro, and Mario Rodríguez-Artalejo. Implementing a lazy functional logic language with disequality constraints. In *Proceedings of JICSLP 1992*, pages 207–221. MIT Press, 1992.

[KMI01] Norio Kobayashi, Mircea Marin, and Tetsuo Ida. Collaborative constraint functional logic programming in an open environment. In *Proceedings of APLAS 2001*, pages 49–59, 2001.

[KMI03] Norio Kobayashi, Mircea Marin, and Tetsuo Ida. Collaborative constraint functional logic programming system in an open environment. *IEICE Transactions on Information and Systems*, E86-D(1):223–230, 2003.

[KMIC02] Norio Kobayashi, Mircea Marin, Tetsuo Ida, and Zhanbin Che. Open CFLP: An open system for collaborative constraint functional logic programming. In *Proceedings of WFLP 2002*, pages 229–232, 2002.

[LF92] Francisco J. López-Fraguas. A general scheme for constraint functional logic programming. In *Proceedings of ALP 1992*, volume 632 of *LNCS*, pages 213–227. Springer, 1992.

[LF94] Francisco J. López-Fraguas. *Programación Funcional y Lógica con Restricciones.* PhD thesis, Departamento de Informática y Automática, Universidad Complutense de Madrid, 1994.

[Llo84] John W. Lloyd. *Foundations of Logic Programming.* Springer-Verlag New York, Inc., New York, USA, 1984.

[Llo95] John W. Lloyd. Declarative programming in Escher. Technical Report CSTR-95-013, Department of Computer Science, University of Bristol, June 1995.

[LLR93] Rita Loogen, Francisco J. López-Fraguas, and Mario Rodríguez-Artalejo. A demand driven computation strategy for lazy narrowing. In *Proceedings of PLILP 1993*, volume 714 of *LNCS*, pages 184–200. Springer, 1993.

[LMM88] Jean-Louis Lassez, Michael J. Maher, and Kim Marriot. Unification revisited. In *Proceedings of FLFP 1986*, volume 306 of *LNCS*, pages 67–113. Springer, 1988.

[LPB+87] Giorgio Levi, Catuscia Palamidessi, Pier Giorgio Bosco, Elio Giovannetti, and Corrado Moiso. A complete semantic characterization of k-leaf: A logic language with partial functions. In *Proceedings of LP 1987*, pages 318–327. IEEE-CS, 1987.

[LRV04] Francisco J. López-Fraguas, Mario Rodríguez-Artalejo, and Rafael Del Vado-Vírseda. A Lazy Narrowing Calculus for Declarative Constraint Programming. In *Proceedings of PPDP 2004*, pages 43–54. ACM Press, 2004.

[LRV05] Francisco J. López-Fraguas, Mario Rodríguez-Artalejo, and Rafael Del Vado-Vírseda. Constraint functional logic programming revisited. *Electronic Notes in Theoretical Computer Science*, 117:5–50, 2005.

[LRV07] Francisco J. López-Fraguas, Mario Rodríguez-Artalejo, and Rafael Del Vado-Vírseda. A new generic scheme for functional logic programming with constraints. *Higher-Order and Symbolic Computation*, 20(1/2):73–122, 2007.

[LS99a] Francisco J. López-Fraguas and Jaime Sánchez-Hernández. Disequalities may help to narrow. In *Proceedings of APPIA-GULP-PRODE 1999*, pages 89–104, 1999.

[LS99b] Francisco J. López-Fraguas and Jaime Sánchez-Hernández. $\mathcal{TOY}$: A Multiparadigm Declarative System. In *Proceedings of RTA 1999*, volume 1631 of *LNCS*, pages 244–247. Springer, 1999.

[LS00] Francisco J. López-Fraguas and Jaime Sánchez-Hernández. Proving failure in functional logic programs. In *Proceedings of CL 2000*, volume 1861 of *LNCS*, pages 179–193. Springer, 2000.

[LS02] Francisco J. López-Fraguas and Jaime Sánchez-Hernández. Narrowing failure in functional logic programming. In *Proceedings of FLOPS 2002*, volume 2441 of *LNCS*, pages 212–227. Springer, 2002.

[LS03] Francisco J. López-Fraguas and Jaime Sánchez-Hernández. Failure and equality in functional logic programming. *Electronic Notes in Theoretical Computer Science*, 86(3):123–143, 2003.

[LS04a] Vitaly Lagoon and Peter J. Stuckey. Set domain propagation using ROBDDs. In *Proceedings of CP 2004*, volume 3258 of *LNCS*, pages 347–361. Springer, 2004.

[LS04b] Francisco J. López-Fraguas and Jaime Sánchez-Hernández. A proof theoretic approach to failure in functional logic programming. *Theory and Practice of Logic Programming*, 4(1-2):41–74, 2004.

[Mac77] Alan K. Mackworth. Consistency in networks of relations. *Artificial Intelligence*, 8(1):99–118, 1977.

[Mah88] Michael J. Maher. Complete axiomatizations of the algebras of finite, rational and infinite trees. In *Proceedings of LICS 1988*, pages 348–357. IEEE Computer Society, 1988.

[Mar00] Mircea Marin. *Functional Logic Programming with Distributed Constraint Solving.* PhD thesis, Johannes Kepler Universitat Linz, 2000.

[Mar12] Enrique Martín Martín. *Sistemas de tipos en lenguajes lógico-funcionales.* PhD thesis, Departamento de Sistemas Informáticos y Computación, Universidad Complutense de Madrid, 2012.

[Mes92] José Meseguer. Conditioned rewriting logic as a united model of concurrency. *Theoretical Computer Science*, 96(1):73–155, 1992.

[MH86] Roger Mohr and Thomas C. Henderson. Arc and path consistence revisited. *Artificial Intelligence*, 28(2):225–233, 1986.

[MI00] Mircea Marin and Tetsuo Ida. Cooperative constraint functional logic programming. In *Proceedings of WFLP 2000*, pages 382–390, 2000.

[Mil78] Robin Milner. A theory of type polymorphism in programming. *Journal of Computer and Systems Sciences*, 17(3):348–375, 1978.

[MIS01] Mircea Marin, Tetsuo Ida, and Wolfgang Schreiner. CFLP: a mathematica implementation of a distributed constraint solving system. *The Mathematica Journal*, 8(2):287–300, 2001.

[Mon74] Ugo Montanari. Networks of constraints: Fundamental properties and applications to picture processing. *Information Sciences*, 7(0):95–132, 1974.

[Mon96] Eric Monfroy. *Solver collaboration for constraint logic programming.* PhD thesis, Centre de Recherche en Informatique de Nancy, 1996.

[Mor89] Juan J. Moreno-Navarro. *Diseño, semántica e implementación de BABEL: un lenguaje que integra la programación funcional y lógica.* PhD thesis, Universidad Politécnica de Madrid, 1989.

[Moz] Mozart-Oz Programming System. `http://www.mozart-oz.org`.

[MR92] Juan J. Moreno-Navarro and Mario Rodríguez-Artalejo. Logic programming with functions and predicates: The language BABEL. *Journal of Logic Programming*, 12(3/4):191–223, 1992.

[MR01] Joaquín Mateos-Lago and Mario Rodríguez-Artalejo. A declarative framework for object-oriented programming with genetic inheritance. *Theoretical Computer Science*, 269(1-2):363–417, 2001.

[MR12] Enrique Martin-Martin and Juan Rodríguez-Hortalá. Transparent function types: clearing up opacity. In *Proceedings of PPDP 2012*, pages 127–138. ACM, 2012.

[MRS95] Eric Monfroy, Michael Rusinowitch, and René Schott. Implementing non-linear constraints with cooperative solvers. Research Report 2747, CRI Nancy, 1995.

[MS98] Kim Marriott and Peter J. Stuckey. *Programming with Constraints: An Introduction.* MIT Press, 1998.

[Nar99] Guy Alain Narboni. From prolog III to prolog IV: the logic of constraint programming revisited. *Constraints*, 4(4):313–335, 1999.

[Pal02] Miguel Palomino Tarjuelo. Comparing Meseguer's rewriting logic with the logic CRWL. *Electronic Notes in Theoretical Computer Science*, 64:255–276, 2002.

[Pal07] Miguel Palomino Tarjuelo. A comparison between two logical formalisms for rewriting. *Theory and Practice of Logic Programming*, 7(1-2):183–213, 2007.

[Pey87] Simon L. Peyton-Jones. *The Implementation of Functional Programming Languages*. Prentice-Hall, 1987.

[Pey02] Simon L. Peyton-Jones. Haskell 98 language and libraries: the revised report. Technical report, 2002. `http://www.haskell.org/onlinereport/`.

[Rod02] Mario Rodríguez-Artalejo. Functional and constraint logic programming. In *Proceedings of FLOPS 2002*, volume 2441 of *LNCS*. Springer, 2002.

[Rom11] Carlos Romero Díaz. *Programación Lógica Cuantitativa y su Implementación en $\mathcal{TOY}$*. PhD thesis, Departamento de Sistemas Informáticos y Computación, Universidad Complutense de Madrid, 2011.

[RR14] Adrián Riesco and Juan Rodríguez-Hortalá. Singular and plural functions for functional logic programming. *Theory and Practice of Logic Programming*, 14(1):65–116, 2014.

[RS82] Robinson and Sibert. LOGLISP: Motivation, design and implementation. In *Logic Programming*, pages 299–313. Academic Press, London, 1982.

[SG01] Andrew Sadler and Carmen Gervet. Global reasoning on sets. In *Proceedings of FORMUL 2001*, 2001.

[SG04] Andrew Sadler and Carmen Gervet. Hybrid set domains to strengthen constraint propagation and reduce symmetries. In *Proceedings of CP 2004*, volume 3258 of *LNCS*, pages 604–618. Springer, 2004.

[SG08] Andrew Sadler and Carmen Gervet. Enhancing set constraint solvers with lexicographic bounds. *Journal of Heuristics*, 14(1):23–67, 2008.

[SGD02] Tom Schrijvers, María J. García de la Banda, and Bart Demoen. Trailing analysis for HAL. In *Proceedings of ICLP 2002*, volume 2401 of *LNCS*, pages 38–53. Springer, 2002.

[SH98] Jaime Sánchez-Hernández. $\mathcal{TOY}$: Un Lenguaje Lógico funcional con restricciones, 1998. Master's Thesis, University Complutense of Madrid.

[SH06] Jaime Sánchez-Hernández. Constructive failure in functional-logic programming: From theory to implementation. *Journal of Universal Computer Science*, 12(11):1574–1593, 2006.

[SIC11] SICStus Prolog Web Site. `http://www.sics.se`, 2011.

[SS94] Leon Sterling and Ehud Shapiro. *The Art of Prolog (2Nd Ed.): Advanced Programming Techniques*. MIT Press, Cambridge, USA, 1994.

[SY86] P. A. Subrahmanyam and Jia-Hui You. FUNLOG: A computational model integrating logic programming and functional programming. In *Logic Programming: Functions, Relations, and Equations*, pages 157–198. 1986.

[TOY12] TOY Web Site. `https://gpd.sip.ucm.es/trac/gpd/wiki/ToySystem`, 2012.

[Vad03] Rafael Del Vado-Vírseda. A demand-driven narrowing calculus with overlapping definitional trees. In *Proceedings of PPDP 2003*, pages 213–227. ACM, 2003.

[Vad05] Rafael Del Vado-Vírseda. Declarative constraint programming with definitional trees. In *Proceedings of FroCoS 2005*, volume 3717 of *LNCS*, pages 184–199. Springer, 2005.

[Vad07] Rafael Del Vado-Vírseda. A higher-order demand-driven narrowing calculus with definitional trees. In *Proceedings of ICTAC 2007*, volume 4711 of *LNCS*, pages 165–179. Springer, 2007.

[vEK76] Maarten H. van Emden and Robert A. Kowalski. The semantics of predicate logic as a programming language. *Journal of the ACM*, 23(4):733–742, 1976.

[vHSD94] Pascal van Hentenryck, Helmut Simonis, and Mehmet Dincbas. Constraint satisfaction using constraint logic programming. *Artificial Intelligence*, 58:113–159, 1994.

[vHSD98] P. van Hentenryck, V. Saraswat, and Y. Deville. Design, implementation and evaluation of the constraint language cc(FD). *Journal of Logic Programming*, 37:139–164, 1998.

[VYGD08] Pascal Van Hentenryck, Justin Yip, Carmen Gervet, and Grégoire Dooms. Bound consistency for binary length-lex set constraints. In *Proceedings of AAAI 2008*, pages 375–380. AAAI Press, 2008.

Índice alfabético