Programación funcional y lógica con restricciones

Francisco Javier López Fraguas
Departamento de Informática y Automática
Universidad Complutense de Madrid

Memoria presentada para optar al título de
Doctor en Ciencias Matemáticas

Dirigida por
Mario Rodríguez Artalejo

Setiembre de 1994

Resumen

En este trabajo presentamos e investigamos el esquema teórico $CFLP(X)$ para la programación lógico funcional perezosa con restricciones. Cada estructura con restricciones X, que consiste en un dominio de Scott como soporte más un conjunto de operaciones predefinidas continuas, determina una instancia del esquema. Los programas en $CFLP(X)$ están constituidos por reglas de reescritura con restricciones para definir nuevas funciones. Se desarrolla una semántica declarativa de modelo mínimo, caracterizado también como mínimo punto fijo, y una semántica operacional basada en un mecanismo de cómputo – estrechamiento perezoso por restricciones – del que se prueban resultados de corrección y completitud con respecto a la semántica declarativa. Probamos también que, bajo hipótesis razonables, el estrechamiento por restricciones se puede combinar con un sistema de resolución de restricciones, preservando la corrección y la completitud. En una segunda parte se aplican los resultados obtenidos al lenguaje $SFL_{\neq}$, que incorpora restricciones de desigualdad a la programación lógico funcional perezosa. Se muestra que $SFL_{\neq}$ puede ser concebido como la instancia de $CFLP(X)$ que tiene al universo de Herbrand infinitario como soporte, dotado de restricciones de igualdad y desigualdad, más las operaciones primitivas necesarias para expresar de forma continua la unificación. De este modo, hereda las propiedades generales del esquema. Finalmente se presenta, en forma de cláusulas, una especificación de la semántica operacional, que se convierte en un programa Prolog ejecutable al adoptar una adecuada representación de las expresiones, dando lugar a una implementación del lenguaje.

Agradecimientos

Por fin terminé la tesis. No me equivocaré mucho si adjudico estas palabras a buena parte de los muchos doctores que en el mundo han sido y serán. Es un pensamiento que encierra sentimientos muy diferentes — alivio, fastidio por lo que has pasado, satisfacción porque también *tu* tesis ha resultado ser terminante y, presumiblemente, correcta — en el momento de culminar un proceso en el que has tenido momentos (cortos) de lucidez que te hacen comerte el mundo, y períodos (largos) de una torpeza, casi rayana en el coma intelectual, que te ponen de los nervios. Aunque ya sé que esto no es más que una tesis doctoral, lo cierto es que el momento es personalmente importante, porque me hace sentir que se cierra un capítulo de mi vida, más dilatado de lo que corresponde a la mera realización de la tesis, que arranca desde el momento en que, tras dejar plantadas a las topologías débiles en espacios de Banach (¿o me plantaron ellas a mí?), me convertí en informático converso. Al poco de aquello recaí en el tabaco (después de cinco años de abstinencia) y cuando hace unos meses lo volví a dejar, supe que estaba a punto de terminar.

Hay mucha gente que me ha ayudado durante estos años. Creo que he llegado a abusar en ocasiones de su amistad y compañerismo, y espero poder corresponderles. En primer lugar, estoy muy contento de pasar a engrosar la cada vez más nutrida lista (y ya vienen detrás pisando fuerte) de los que debemos un sincero agradecimiento a Mario Rodríguez Artalejo, por su dedicación y generosidad que hacen que los demás también saquemos provecho de su sabiduría. Trabajar a su lado es, aparte de un gran privilegio, un placer. Teresa Hortalá tuvo la culpa de todo: fui a pedirle trabajo y a los dos días ya tenía medio metro de material; un tiempo antes me dio (gracias a un suspenso por un quítame allá ese `REPEAT`) la primera lección sobre lenguajes de programación: que requieren una sintaxis. Ana Gil me ha ofrecido su apoyo y amistad en todo momento; hemos mantenido además muchas apasionantes (y apasionadas) discusiones (no sólo científicas). A Juan Carlos González he acudido en montones de ocasiones en busca de ayuda de todo tipo, y siempre le he encontrado dispuesto a echar una mano, a pesar de que él también tenía lo suyo. Sin la colaboración de Antonio Gavilanes, cumplir mis obligaciones docentes estos dos últimos años se me habría hecho muy cuesta arriba. La complicidad de 'proto-ser' con Lourdes Araujo me ha hecho sentir más arropado en muchos momentos. Puri Arenas 'Martínez' ha trabajado con entusiasmo, junto con Ana, para la implementación de $SFL_{\neq}$. Con Susana Nieva compartí mis primeros escarceos con Prolog, y Javier Leach siempre ejerció una influencia benéfica sobre mí. No puedo olvidarme de mis amigos *losdematemáticas* de Vallecas, a los que debo algo indefinible, pero valioso. A mis padres les agradezco que siempre hayan confiado en mí, y nunca me hayan regañado cuando he abandonado los confortables brazos del Estado.

La realización de esta tesis, desde su génesis, la he vivido como algo compartido con Rosma, a quien va dedicada. Ella ha cargado los últimos años con el peso de las cosas, y encima ha sido capaz de sacar tiempo hasta para ayudarme a teclear. De todas las cosas importantes que nos han pasado, y que nos incumben a ella y a mí, no resisto la tentación de recordar aquí una curiosa forma de investigar en Pisa. A Paco e Isabel estoy deseando comérmelos a besos, que es lo que se merecen.

Para terminar, agradeceré al amable lector que lo siga siendo durante la lectura de esta tesis, y al lector astuto, que sea clemente con los errores (espero que no graves) que pueda encontrar.

Índice General

Preliminares

La combinación de diferentes paradigmas de programación es un lugar común de muchas investigaciones actuales en el mundo de la programación declarativa, en un intento de obtener lenguajes de programación que sean cada vez más expresivos, que acepten implementaciones eficientes, y que dispongan al tiempo de sólida fundamentación teórica. Dos combinaciones han suscitado particular interés en la última década: Programación lógica + Programación con restricciones, y Programación lógica + Programación funcional. Como máximo exponente de la primera combinación está $CLP(X)$, un esquema que generaliza los principios de la programación lógica, reemplazando la unificación por la noción más general de resolución de restricciones. Hay más heterogeneidad en las propuestas para la segunda combinación. Los llamados lenguajes lógico-funcionales utilizan sistemas de reescritura condicionales con disciplina de constructoras como programas, y estrechamiento como mecanismo de cómputo para resolver objetivos. En el caso de lenguajes perezosos, en los que se contempla la posibilidad de hacer cómputos con objetos infinitos, se ha propuesto en ocasiones una semántica declarativa que considera como dominio de los cómputos un universo de Herbrand con árboles posiblemente infinitos o parcialmente definidos.

El *primer*, y principal, *objetivo* de nuestro trabajo es el desarrollo de un marco teórico para la integración de los tres paradigmas citados: Programación lógica + Programación funcional + Programación con restricciones. El diseño de tal marco teórico está guiado desde el principio por un supuesto de trabajo: la amalgama de los tres paradigmas debe poder realizarse, partiendo de la programación lógico funcional perezosa, de un modo similar al que permite pasar de la progamación lógica a la programación lógica con restricciones. Es decir, nuestro objetivo básico puede ser reformulado del siguiente modo: perseguimos la definición de un esquema general, que denominaremos $CFLP(X)$, para la programación lógico funcional perezosa con restricciones (*Constraint Functional Logic Programming*). Este esquema jugará un papel, con relación a la programación lógico funcional, similar al del esquema $CLP(X)$ con relación a la programación lógica. Estas consideraciones nos permiten establecer de antemano algunas de las características que esperamos del esquema $CFLP(X)$

- De modo similar al caso de $CLP(X)$, obtendremos una *instancia* del esquema $CFLP(X)$ al fijar una estructura determinada X, cuyas características describiremos en seguida. Una instancia del esquema determina un lenguaje de programación lógico funcional perezoso, sobre la estructura X que se haya fijado.

- El tipo de *estructuras* a considerar está muy determinado por la naturaleza perezosa que se pretende para la componente funcional de las instancias del esquema. Parece natural generalizar la idea del universo de Herbrand infinitario que aparece en la semántica de algunos lenguajes lógico funcionales, y considerar 'estructuras continuas' con *dominios de Scott* como soporte, y dotadas de una serie de operaciones primitivas continuas (en el sentido de la teoría de dominios).

- Los *programas* serán también generalización directa de los sistemas de reescritura condicionales de la programación lógico funcional. Consideraremos reglas de reescritura condicionales con restricciones, en los que la unificación – que no tiene sentido en este marco general – queda reemplazada por la satisfacción de restricciones en X.

- De nuevo como en el caso de $CLP(X)$ con relación a la programación lógica, pretendemos que $CFLP(X)$ sea una *extensión suave* de la programación lógico funcional. Por 'suavidad' entendemos que las principales ideas, intuiciones y resultados acerca de las semánticas declarativa y operacional sean preservadas, así como la metodología de trabajo para el estudio de las propiedades del esquema. Es más, esperamos que, al estar liberados de los tecnicismos que siempre requiere la unificación, se simplifique en muchos casos la formulación de los conceptos y la demostración de los resultados.

Como *segundo* gran *objetivo* de nuestro trabajo está el estudio de una instancia interesante del esquema $CFLP(X)$: el lenguaje $SFL_{\neq}$, que resulta de incorporar restricciones de desigualdad a un lenguaje lógico funcional perezoso. El uso de desigualdades (en programas y respuestas) incrementa el poder expresivo de un lenguaje, pues la condición representada por una desigualdad puede requerir infinitas igualdades (o sustituciones) para ser expresada de modo equivalente. En el contexto de la programación lógica con restricciones es relativamente habitual el uso de desigualdades entre términos – más en general, entre algún tipo de árboles –, pero es bastante novedoso entre los lenguajes lógico funcionales. En el trabajo se pretende analizar con bastante detalle el lenguaje $SFL_{\neq}$, cubriendo no sólo sus fundamentos teóricos, sino también aspectos más prácticos, incluyendo una propuesta de implementación.

Aparte de la utilidad del lenguaje en sí, el estudio de $SFL_{\neq}$ tiene gran interés para nosotros, porque constituye un magnífico test de la aplicabilidad del esquema $CFLP(X)$. No esperamos que la caracterización de $SFL_{\neq}$ como instancia de $CFLP(X)$ dé respuesta a todos los problemas relativos al lenguaje concreto, aunque sí a bastantes de ellos. La relación inversa (instancia - esquema) también puede resultar interesante: de lo que ocurra en $SFL_{\neq}$, podremos extraer conclusiones útiles acerca de $CFLP(X)$.

Organización de la memoria

La organización y resumen del trabajo es como sigue.

La memoria está dividida en dos partes, correspondientes a los dos grandes temas que tratamos: el esquema $CFLP(X)$, y el lenguaje $SFL_{\neq}$. Pasamos a describir la primera parte.

En un primer capítulo delimitamos el contexto en el que se sitúa esta tesis, haciendo un rápido repaso de las propuestas más importantes de combinación de paradigmas declarativos, bastante abundantes en los casos de la programación lógica con restricciones y la programación lógico-funcional, mucho más escasas en el caso de la programación funcional y lógica con restricciones; en este epígrafe hemos incluido también las propuestas, casi anecdóticas, de programación funcional con restricciones.

En el capítulo 2, primero de los dedicado al esquema $CFLP(X)$, se introducen las nociones básicas acerca del tipo de *estructuras* 'continuas' que consideramos para las distintas instancias del esquema, y se define la *sintaxis* de los $CFLP$-programas. Acto seguido nos detenemos a analizar algunas *instancias* del esquema, mostrando en particular cómo la programación lógica con restricciones y la programación lógico-funcional son expresables en $CFLP(X)$.

El capítulo 3 contiene la *semántica declarativa* de $CFLP(X)$. Tras introducir las nociones de modelo e interpretación, se prueba, bajo hipótesis de la mayor generalidad posible (consistencia, i. e., existencia de algún modelo), la existencia de *modelo mínimo*. A una

demostración muy sencilla, basada en la 'propiedad de la intersección' que cumplen los modelos, se suma otra en la que se caracteriza el modelo mínimo como *mínimo punto fijo* de un operador 'de consecuencias inmediatas'. Debido a la generalidad de las hipótesis, este operador tiene la característica, poco habitual en este tipo de semánticas, de estar sólo parcialmente definido sobre el conjunto de las interpretaciones. Un último epígrafe muestra que la consistencia de un programa es una propiedad indecidible, y discute algunas cuestiones relacionadas, como la no ambigüedad.

El capítulo 4, probablemente el más importante del trabajo, se encarga de la *semántica operacional*. El mecanismo de cómputo que consideramos, que denominamos *estrechamiento por restricciones*, consta inicialmente de una sola regla, cuya *corrección* probamos. Para obtener completitud nos debemos circunscribir a una clase de programas, que incluye a los programas semánticamente no ambiguos. Para reducir el espacio de búsqueda determinado por la regla de estrechamiento, se introduce la noción de cómputos *perezosos*, en los que el estrechamiento se realiza en posiciones *demandadas*. Estas nociones son de carácter más bien semántico, al estar basadas en información extraída del modelo mínimo, que se utiliza para construir unas ciertas 'aproximaciones' a expresiones y objetivos. Sobre ellas se define un orden bien fundado en el que se decrece al dar un paso de estrechamiento perezoso, lo que nos permitirá probar la completitud. En la última parte de este capítulo se analizan las limitaciones de la semántica operacional propuesta, y se concluye la necesidad de combinar el estrechamiento con algún sistema de resolución de restricciones específico de la estructura de base. Se proponen condiciones naturales y generales bajo las que la combinación preserva la corrección y la completitud, si bien ésta en un sentido relativamente débil.

Con el capítulo 5 comienza la segunda parte de esta memoria, en la que se estudia el lenguaje $SFL_{\neq}$. Primeramente se fija la sintaxis, por la que los programas son reglas de reescritura con patrones lineales en las cabezas, y sistemas de igualdades estrictas $==$ y desigualdades $=/=$ como condiciones. A continuación se caracteriza el lenguaje como la instancia $CFLP(\mathcal{H}_{\neq})$, siendo $\mathcal{H}$ el universo de Herbrand infinitario determinado por un conjunto de constructoras, y donde los predicados predefinidos $==$ y $=/=$ se interpretan como aproximaciones continuas maximales a la igualdad 'de verdad' $=$ en $\mathcal{H}$ (que no es continua) y a su negación $\neq$. Se introducen, también como operaciones primitivas, unas ciertas 'funciones selectoras'y 'predicados reconocedores' que permiten expresar la unificación de forma continua. De modo alternativo, y más conveniente, se sugiere la posibilidad de expresar la unificación mediante un uso restringido, 'continuo', de la igualdad no continua $=$. La equivalencia de ambos enfoques es probada. A continuación se propone para $SFL_{\neq}$ un sistema de resolución de restricciones, para ser combinado con el estrechamiento, que permite heredar para $SFL_{\neq}$ los resultados generales del esquema. Mediante un análisis más detallado de los cómputos, obtenemos después un resultado mucho más potente de completitud.

Finalmente, en el capítulo 6 se aborda la cuestión de la implementación de $SFL_{\neq}$. Se proporciona en primer lugar una especificación, en forma de cláusulas, de la semántica operacional, y que depende de algunos predicados inicialmente no especificados. Tras una adecuada elección de la representación de expresiones y restricciones, se pueden definir tales predicados, para obtener una especificación completa, que en definitiva resulta definir una compilación de $SFL_{\neq}$-programas a programas Prolog ejecutables.

Algunas conclusiones y sugerencias para trabajos futuros cierran la memoria.

Parte I

El esquema $CFLP(X)$

1 Introducción

En esta sección haremos un breve repaso de distintas propuestas para la combinación de paradigmas de programación declarativa; más en concreto, de las que tienen relación más directa con nuestro trabajo, por involucrar parte o todos los ingredientes de nuestra propuesta. Revisaremos pues las siguientes combinaciones:

- Programación lógica (LP) + Programación con restricciones (CP)
- LP + Programación funcional (FP)
- LP + FP + CP

No somos ambiciosos, especialmente en lo que se refiere a los dos primeros puntos, en cuanto a la extensión, ni en cuanto a la agudeza u originalidad de nuestros comentarios. Existen excelentes referencias ([30, 12, 71, 72, 81] para CLP, [10, 65, 2, 114, 66, 69] para FLP) cuya lectura proporcionará sin duda mayor información al respecto. Como referencias estándar sobre LP y FP pueden citarse [100, 5], [75, 9].

1.1 Programación lógica con restricciones

La noción de 'programación con restricciones' no forma parte en realidad de los paradigmas bien establecidos de programación declarativa. Con esa denominación se incluyen de modo bastante impreciso lenguajes de programación que utilizan técnicas de propagación de restricciones para la resolución de problemas combinatorios [98, 146] o técnicas de Investigación Operativa. Ejemplo típico de tal clase de lenguajes es *THINGLAB* [18].

En un sentido genérico, una *restricción* es una condición, expresada en algún lenguaje determinado, impuesta sobre un conjunto de variables V. Al tiempo que expresa una condición, una restricción sirve también para representar de modo implícito al conjunto de sus soluciones, es decir al conjunto de valoraciones de las variables de V que *satisfacen* la condición. Las restricciones juegan pues un múltiple papel: como representación de datos sirven para efectuar entrada/salida, como representación de condiciones sirven para dirigir el flujo de control de la ejecución. La noción básica que subyace es la de *satisfactibilidad* de una restricción, y el mecanismo operacional básico, el de *resolución de restricciones*. La resolución de restricciones tiene, a su vez, una doble función:

- verificar la satisfactibilidad de una restricción.
- producir formas simplificadas de las restricciones, que sirvan ya como respuesta final, o contribuyan a la incrementalidad del proceso de resolución, si nuevas restricciones son añadidas en el futuro.

Está implícito en todo lo anterior la existencia de un *dominio prefijado* al que se refieren todas las nociones de satisfactibilidad de condiciones, resolución de restricciones, etc. Y la combinación dominio/lenguaje sugiere la idea de 'estructura'. Así, en un sentido aún más general podemos concebir la computación con restricciones como *computación sobre*

una estructura determinada, y su interés radica en la posibilidad de aprovechar mecanismos eficientes, específicos de la estructura, que se conozcan previamente o se diseñen al efecto.

La conexión, siquiera circunstancial, entre la programación lógica y la resolución de restricciones aparece desde los mismos orígenes de Prolog [1]. En sus trabajos sobre Prolog II [31, 32, 33], Colmerauer considera programas que constan de cláusulas de Horn, pero reemplaza la noción de unificación sintáctica entre términos por la de resolución de sistemas de ecuaciones y desigualdades, cuyas soluciones deben entenderse en el universo de los árboles regulares [2], y para los que se proporciona un sistema de resolución y paso a forma resuelta. Como vemos, aparecen ya – dentro de un contexto de programación lógica – las dos nociones básicas de la programación lógica con restricciones:

- Sustituir el universo de Herbrand de los árboles finitos como dominio de cómputo, por otros dominios (el de los árboles regulares, en este caso).
- Reemplazar la unificación sintáctica por otros mecanismos de resolución de restricciones formuladas en algún lenguaje (resolución de sistemas de igualdades y desigualdades en este caso).

.

Aunque Prolog II es reconocido habitualmente hoy en día como el primer lenguaje lógico con restricciones, el impulso definitivo a la programación lógica con restricciones (CLP) vino dado por la introducción, por parte de Jaffar y Lassez [78, 79], del esquema $CLP(X)$ como marco general para realizar CLP. Existen, a nuestro juicio, bastantes razones para el éxito de esta propuesta:

- La motivación que hay detrás de $CLP(X)$ es muy sólida, al menos para la comunidad de los programadores, y en cierto modo complementaria de la de Prolog II: el argumento para crear Prolog II es que Prolog está demasiado alejado de la *teoría*; el argumento de $CLP(X)$ es que Prolog está demasiado alejado de la *práctica*. Ello es debido al sometimiento de la programación lógica al universo de Herbrand, que obliga a una representación simbólica de los datos que resulta forzada, poco intuitiva, y lo que es realmente grave [3], impide utilizar mecanismos eficientes de cómputo que se conozcan para el dominio en cuestión. Esta motivación se ve reflejada en el hecho de que la introducción de $CLP(X)$ es simultánea a la aparición de $CLP(\mathcal{R})$ [70, 82] ([83, 84] son referencias actualizadas), una instancia del esquema cuyo dominio son los números reales, y que maneja como restricciones sistemas (lineales) de ecuaciones, inecuaciones y desigualdades, para cuya resolución utiliza métodos numéricos eficientes, como eliminación gaussiana o el método del simplex.

[1] Según comenta Colmerauer en [33], el primer intérprete para un incipiente Prolog, realizado por Roussel [133], incluía la resolución combinada de igualdades y desigualdades entre términos, algo típicamente asociado al contexto de la programación lógica con restricciones. Este aspecto desapareció en el Prolog que conoció (y conquistó) el mundo, y fue sustituido por versiones más pragmáticas de la igualdad y desigualdad, que simplemente verifican la unificabilidad o no unificabilidad de términos.

[2] Un árbol es regular (o racional) si sólo tiene un número finito de subárboles distintos.

[3] Al fin y al cabo, también es forzada y poco intuitiva la idea cartesiana de algebrizar la geometría, y nadie diría hoy en día que la idea era desafortunada.

- La adopción de un 'esquema' parametrizado por una estructura, que se convierte en un lenguaje concreto al adoptar una estructura concreta, supone un acierto formal. El camino estaba ya iniciado en [80] donde, aunque el dominio era teóricamente arbitrario, las condiciones impuestas a él lo hacían equivalente a un cociente del universo de Herbrand definido por una teoría ecuacional, que jugaba en este caso el papel de parámetro del esquema.

- En la propia concepción del marco teórico está la pretensión de que las propiedades semánticas de la programación lógica – en particular, existencia de modelo mínimo caracterizable como mínimo punto fijo [50, 100, 5] – deben preservarse en todas las instancias del esquema. La idea importante aquí es que la programación lógica tiene estas propiedades por el uso de cláusulas *definidas* ; el universo de Herbrand y la unificación tienen poco que ver con dichas propiedades. Imponiendo además condiciones adicionales sobre el dominio y el lenguaje de restricciones se pueden extender aún más características de la programación lógica, como son la existencia de una semántica lógica (que establece la equivalencia entre respuestas computadas y fórmulas demostrables en una teoría) y una semántica de la negación como fallo finito.

Los principales resultados teóricos acerca de $CLP(X)$ se encuentran en el trabajo original [78], aunque revisiones y mejoras de sus fundamentos aparecen en [55, 109, 81]. Introducciones sencillas a $CLP(X)$ son [96, 30, 54]. Un intento de extender el uso de restricciones a cláusulas generales aparece en [23].

En [73] se da una visión algo más general de CLP, que hace abstracción de la sintaxis (en $CLP(X)$ se asume que las estructuras de base son Σ-estructuras, siendo Σ una signatura de primer orden), y se considera una clase de estructuras en lugar de una sola. Aparte de ello, el trabajo es muy interesante por la sencillez de las demostraciones que propone (es de destacar, en particular, la elegancia de la demostración de la completitud de la resolución con restricciones).

Desde su aparición, el esquema $CLP(X)$ ha monopolizado prácticamente la idea de programación lógica con restricciones, de modo similar a lo que ocurre con Prolog y la programación lógica. Muchos trabajos se han dedicado desde entonces a diversos aspectos de CLP. Puede decirse que, en su conjunto, dichos trabajos abordan un espectro de problemas [81, 12] que se corresponde con los problemas que interesan en programación lógica. Algunas líneas significativas son:

- Diseño e implementación de diferentes sistemas (lenguajes): $CLP(\mathcal{R})$ [70, 82, 83, 84], CAL [135], CHIP [47, 48, 71], Prolog III [34], $CLP(\mathcal{H}/\mathcal{E})$ [3], {log} [49, 22].

- Negación constructiva [145].

- Programación concurrente con restricciones [136, 137].

1.2 Programación lógico funcional

La integración de la programación lógica y la programación funcional es quizás la combinación de paradigmas más estudiada y mejor entendida. Algunas recopilaciones de distintas propuestas pueden encontrarse en [10, 65, 2, 114, 66, 69].

El motivo de tal interés proviene de que cada uno de los dos paradigmas, que tienen en común su naturaleza declarativa con una sólida fundamentación matemática, presenta particularidades deseables para la programación de alto nivel: multidireccionalidad de los argumentos, indeterminismo, variable lógica, estructuras de datos incompletas –en el caso de la programación lógica –, evaluación determinista, funciones parciales, orden superior, tipos y polimorfismo – en el de la programación funcional –. En términos genéricos, el estilo lógico es adecuado para hacer abstracción del control en procesos de búsqueda y generación de datos, mientras que en el estilo funcional se expresan mejor problemas con un flujo de datos determinista, teniendo un aire más 'algorítmico'.

A pesar de que la programación de orden superior es un ingrediente fundamental de la programación funcional moderna, la mayor parte de las propuestas de integración FP + LP cubren únicamente aspectos de primer orden. La dificultad de incluir también aspectos de orden superior puede ser explicada por el hecho de que la semántica denotacional habitual de los lenguajes funcionales de OS, basada en dominios obtenidos como solución de ecuaciones recursivas, no es adecuada – como se muestra p. ej. en [63, 61] – desde un punto de vista lógico, en el que los cómputos quieren verse como deducciones. Más adelante comentaremos brevemente algunas propuestas que incluyen orden superior para la integración FP + LP. La programación funcional de primer orden, sin embargo, puede basarse en la lógica ecuacional y la reescritura, mucho más fácilmente combinable con la programación lógica relacional, basada en lógica de Horn y resolución.

En algunas de las propuestas de integración se da clara prioridad a la parte funcional, proporcionando ciertas capacidades lógicas a un lenguaje funcional previo. Así ocurre, p. ej., con los lenguajes funcionales con abstracciones de conjuntos [43, 38, 141], y de manera incluso más acusada con otros lenguajes funcionales que incorporan sólo parte de las características de la programación lógica, como pueden ser variables lógicas y alguna forma restringida de unificación [128, 85].

Un enfoque mucho más frecuente ha consistido en adoptar una postura más simétrica entre ambos paradigmas, mediante algún tipo de combinación, tanto en lo referente a la sintaxis, como a los principios operacionales.

En cuanto a la *sintaxis*, los ingredientes básicos a combinar son ecuaciones y cláusulas de Horn. Aunque hay una variedad enorme en la forma de concretar la sintaxis, un formalismo suficientemente general en el que caben la mayor parte de las propuestas es el de los *sistemas de reescritura condicionales* [88, 13, 45] o, alternativamente, *cláusulas de Horn ecuacionales* [126, 74].

Por lo que se refiere al *mecanismo operacional*, se suele adoptar alguno que generalice, por una parte el ajuste de patrones ('*matching*') y reescritura – principio operacional de la programación funcional – con la unificación y resolución – principio operacional de la programación lógica –. En general, esto requiere reemplazar la unificación por unificación semántica o *E-unificación* [56, 87, 140]. Sin embargo, la E-unificación es un problema demasiado difícil, y sus métodos generales demasiado ineficientes para ser considerada como base del mecanismo de cómputo de un lenguaje de programación. Por este motivo, la mayor parte de las propuestas adoptan alguna restricción sobre los programas de modo que se pueda utilizar algún procedimiento más eficiente de E-unificación. Este procedimiento es, en casi todos los casos, estrechamiento ('*narrowing*'). El estrechamiento, tal como fue propuesto en [142],

sigue siendo una regla demasiado ineficiente, más adecuada para la demostración automática que para un lenguaje de programación. Sucesivos refinamientos del estrechamiento, y muy en particular el estrechamiento básico ('*basic narrowing*') [77, 124, 111] y sus muchas variantes, persiguen el objetivo de reducir el espacio de búsqueda, a costa posiblemente de imponer restricciones adicionales a los sistemas de reescritura. Una de las restricciones consideradas como razonables para un lenguaje de programación es la *disciplina de constructoras* [125], por la que el conjunto de símbolos de función se divide en dos: símbolos de *constructora*, para los que no hay reglas, y símbolos de *función* propiamente dichos.

El número de variantes del estrechamiento que se han propuesto es casi tan elevado como el número de lenguajes propuestos. En [69] se citan hasta 18 tipos distintos de estrechamiento (más aún si se incluyen aquellos lenguajes que, como *K-LEAF* [99, 58] simulan estrechamiento mediante resolución). De los posibles criterios para clasificar tan abultada colección de lenguajes y mecanismos operacionales, nos interesa destacar un aspecto: muchos de los lenguajes asumen *terminación* de los sistemas de reescritura que constituyen los programas, para garantizar la completitud (de la variante) del estrechamiento que utilizan, normalmente derivada del estrechamiento básico. En general, en estos casos, la semántica operacional se corresponde con lo que sería *evaluación impaciente* en un lenguaje funcional, y la *semántica declarativa* hace referencia a (posiblemente algún cociente de) el universo de Herbrand de los términos (árboles) finitos.

Otros lenguajes eluden la condición de terminación, para poder capturar la idea de evaluación perezosa y los cómputos con estructuras de datos infinitos, característicos de muchos lenguajes funcionales modernos [75]. Como contrapartida, otras restricciones – como disciplina de constructoras, linealidad por la izquierda, no ambigüedad – son impuestas [58, 63, 118]. Un ejemplo de ello es el lenguaje *BABEL* [117, 114, 118], que puede encuadrarse en la categoría de los lenguajes lógico-funcionales al estilo de Reddy [131, 132], que utilizan sintaxis funcional y, lo que es más importante, *estrechamiento perezoso* como mecanismo de cómputo. El estrechamiento perezoso retrasa la evaluación de los argumentos de una función hasta que son realmente necesarios para calcular el resultado de la llamada a la función. Los mismos principios de evaluación perezosa, en este caso simulando estrechamiento perezoso mediante resolución, sigue *K-LEAF* [99, 58]. *K-LEAF* y *BABEL* comparten una semántica declarativa de modelo mínimo basada en dominios de Scott [138], y más en particular, en el universo de Herbrand infinitario, que consta de los árboles finitos o infinitos, total o parcialmente definidos, construidos por medio de un conjunto de constructoras, más un elemento $\perp$ indefinido añadido. Las interpretaciones dan significado a las funciones como funciones continuas (en el sentido de la teoría de dominios). Debido a la semántica, se distingue entre la igualdad 'verdadera' ($=$) en el dominio, que no es continua, y una *igualdad estricta o continua* $==$, que es la que puede aparecer en los lados derechos de las reglas. Tanto en *K-LEAF* como en *BABEL* se supone $==$ definida a través de una serie de reglas, que fuerzan a su interpretación como igualdad estricta. El lenguaje *SFL* [63, 64, 61] comparte muchas características con *BABEL*. Las principales diferencias estriban en que *SFL* incorpora variables lógicas de orden superior, y la igualdad estricta está tratada como una primitiva predefinida, y no como una función más definida a través de reglas. La semántica operacional está al cargo directo de $==$ mediante reglas de transformación específicas.

En cuanto a las propuestas de implementación de lenguajes lógico-funcionales, los enfoques más habituales seguidos se recogen en la siguiente clasificación:

- Traducción (o compilación) a otro lenguaje declarativo, usualmente Prolog. Lás técnicas utilizadas se pueden considerar descendientes de algunas propuestas más antiguas para hacer algún tipo de evaluación perezosa en Prolog [122, 123]. Implementaciones basadas en compilación a Prolog se describen en [28, 86, 103, 7, 8].

- Desarrollo de máquinas abstractas diseñadas al efecto. En este caso la variedad de propuestas es mayor (véanse [29, 69] como 'surveys' sobre el tema). Se pueden agrupar en dos grandes familias:

  - Extensión de máquinas para lenguajes lógicos – lo que en la práctica equivale a decir extensión de la máquina abstracta de Warren (WAM) [148, 1] – para incorporar capacidades funcionales. Ejemplos de ello son [19, 68, 119].
  - Extensión de máquinas para lenguajes funcionales [127] para incorporar capacidades lógicas (unificación y 'backtracking'). Ejemplos de ello son [91, 116, 102, 25, 149, 101].

La incorporación plena de capacidades de orden superior en los lenguajes lógico funcionales, de modo que se disponga no sólo de funciones de orden superior, sino también de variables lógicas de orden superior, ha sido tratada en muchas menos ocasiones.

El lenguaje λ-Prolog [112, 121] extiende Prolog con variables lógicas de orden superior, λ-expresiones y unificación de orden superior. El lenguaje está fundado en el fragmento de Horn de una lógica de orden superior, mucho más potente, que combina la lógica de predicados con con el λ-cálculo con tipos. El lenguaje hace énfasis en la componente lógica, en detrimento de los aspectos funcionales; en particular, no se contempla la posibilidad de definir funciones de modo recursivo.

En el lenguaje lógico-funcional de orden superior $IDEAL$ [20, 11] se pueden definir funciones mediante reglas condicionales y predicados mediante cláusulas definidas. Tanto para la semántica como para la implementación, $IDEAL$ se remite al lenguaje de primer orden *K-LEAF* [99, 58], a través de una traducción que sigue las ideas de Warren en [147].

Un tratamiento más completo del orden superior en programación lógico-funcional se da en el lenguaje SFL [62, 63, 64, 61], cuyos programas son sistemas de reescritura aplicativos condicionales, que siguen la disciplina de constructoras y admiten variables extra en las condiciones. A diferencia de $IDEAL$, SFL dispone de semánticas declarativa y operacional de orden superior, con resultados de corrección y completitud, compatibles además con una traducción a primer orden al estilo de Warren, que puede ser probada también correcta y completa [60, 61].

1.3 Programación lógico funcional con restricciones

Aunque la inclusión de restricciones como parte de un lenguaje de programación parece ligada de modo más natural al espíritu de la programación lógica que al de la programación funcional, existen propuestas para la combinación FP + CP, para obtener programación funcional con restricciones (CFP). Conocemos dos al respecto.

En [39] se propone una modificación del esquema de Höfeld-Smolka [73] como base para la programación funcional con restricciones, desarrollando un esquema $CFP(X)$ al estilo de

$CLP(X)$. Sin embargo, las funciones son indeterministas (sus valores son conjuntos), y el lenguaje es de primer orden, por lo que en definitiva puede considerarse que una función $CFP(X)$ es una variante sintáctica de un predicado de $CLP(X)$.

Otra propuesta que puede servir como base para CFP es el λ-cálculo con restricciones (*constrained λ-calculus*) de [37, 110]. Es una extensión del λ-cálculo que incorpora a la sintaxis la posibilidad del uso de restricciones, así como una regla de reducción que, en caso de que una restricción determine funcionalmente los valores de algunas variables, permite sustituir éstas por aquellos. Se imponen condiciones muy fuertes para garantizar la propiedad de Church-Rosser del sistema. Además, las funciones definidas son estrictas.

En cuanto a la combinación Programación lógica + Programación funcional + Programación con restricciones (LP + FP + CP), para obtener Programación lógico funcional con restricciones (CFLP), tampoco abundan las propuestas.

En [41, 42] se propone el paradigma DCP (*Definitional Constraint Programming*) para definir lenguajes lógico funcionales con restricciones. DCP es en realidad un esquema $DCP(X)$, donde la idea de esquema es la misma que la nuestra, heredada por tanto de $CLP(X)$: se obtiene un lenguaje al fijar un sistema de restricciones X determinado. Al igual que en [73], la noción de sistema de restricciones hace abstracción de la sintaxis. En términos generales, $DCP(X)$ se define como $CLP(FP(X))$, es decir, como una instancia de $CLP(X)$ donde la estructura de base se enriquece con nuevas funciones definidas por medio de un lenguaje funcional, y el lenguaje de restricciones se amplía con ecuaciones entre $FP(X)$-expresiones, a resolver por estrechamiento. En $DCP(X)$ no se permite el uso de restricciones en la definición de las funciones, con lo que la integración conseguida está bastante limitada. Por otra parte, las propiedades semánticas del esquema no están del todo aclaradas en los trabajos citados.

En [120] se presenta el lenguaje $CFLP-L$, definido como una extensión del λ-cálculo para incluir variables lógicas, elecciones no deterministas y restricciones. La extensión está inspirada en el λ-cálculo con restricciones de [37, 110], con quien comparte algunas de sus limitaciones, en particular la naturaleza estricta de las funciones. Como estructuras de base se consideran modelos iniciales de especificaciones algebraicas. Los modelos están restringidos a ser álgebras generadas por términos (*term generated*). Los autores alegan para ello que no tiene interés considerar dominios de base en los que hay elementos no representables como términos; pero uno de los aspectos importantes de las restricciones es su capacidad para representar implícitamente elementos del dominio, que no tienen por qué poderse representar como términos. Por ejemplo, la restricción $X * X = 2,\ X > 0$ representa implícitamente en $CLP(\mathcal{R})$ al número $\sqrt{2}$, que no es representable como término en el lenguaje. Asimismo, en Prolog II, la restricción $X = s(X)$ representa un árbol racional infinito, que no es representable como término. Como vemos, dos de los lenguajes con restricciones más representativos incumplen el requisito pedido en $CFLP - L$. En cuanto a la semántica del lenguaje, se proporciona una semántica de continuaciones (*continuation semantics*) que determina la semántica operacional. No se aporta semántica declarativa.

En [107] se presenta por vez primera nuestra propuesta de integración para CFLP a través del esquema $CFLP(X)$. Si bien los fundamentos de la semántica declarativa quedan allí establecidos con precisión, no ocurre lo mismo con los aspectos operacionales, que están limitados al caso, muy particular, de que los dominios de base sean cpo's planos y todas las

funciones, tanto primitivas como definidas, sean estrictas. El estudio del estrechamiento por restricciones para el caso general – dominios de Scott cualesquiera como soportes y funciones posiblemente no estrictas – se realiza en [105]. Sin embargo, en este trabajo no se aborda la cuestión de la combinación del estrechamiento por restricciones con algún mecanismo, específico de la estructura de base, de resolución con restricciones, con lo que la aplicabilidad práctica del esquema queda en entredicho. La extensión de $CFLP(X)$ para incluir dicha combinación, así como la aplicación al lenguaje $SFL_{\neq}$ de las propiedades generales del esquema, se presentan en [8]. Ese trabajo no contiene ninguna demostración; es esta memoria la que da respaldo a los resultados allí contenidos.

Terminamos este apartado citando el interés también creciente que está despertando el uso de restricciones en el campo de la reescritura, deducción ecuacional y, de modo más genérico, de la *demostración automática* [89, 26, 134]. Estas investigaciones tienen cierta relación con la amalgama LP + FP + CP, ya que la reescritura y la lógica ecuacional forman parte de los fundamentos de muchas propuestas para la combinación LP + FP. Sin embargo, el interés en esos trabajos se circunscribe a restricciones sobre dominios simbólicos (términos, habitualmente), como son igualdad, desigualdad, prioridad según algún orden de términos, etc., estando por tanto alejados de nuestra idea de que la programación con restricciones se refiere a computación sobre dominios predeterminados cualesquiera. Se puede, de todos modos, descubrir una relación más estrecha en algunos trabajos relativos a problemas de unificación en dominios combinados [90, 21].

2 El esquema $CFLP(X)$

2.1 Estructuras con restricciones

Por simplicidad, consideraremos estructuras sobre una signatura con un sólo género. Una *signatura* es un conjunto de símbolos Σ, cada uno de los cuales tiene una *aridad* asociada $n \geq 0$. El conjunto de símbolos Σ lo suponemos dividido como $\Sigma = F \cup \Pi$, siendo F y Π disjuntos. F es el conjunto de *símbolos de función primitiva* y Π el de *símbolos de predicado (o relación) primitivo*. Si F^n (Π^n resp.) es el conjunto de símbolos de F (Π resp.) de aridad n, tenemos $F = \bigcup_{n \in N} F^n$ y $\Pi = \bigcup_{n \in N} \Pi^n$.

Los símbolos de F y Π deben ser entendidos como símbolos de funciones (u operaciones) y predicados (o relaciones) con un significado prefijado y conocido de antemano, definidos sobre un cierto dominio también prefijado. Dichas funciones y predicados no requieren por tanto ser definidos, sino que su significado pretendido, y el dominio sobre el que actúan, queda expresado a través de la noción de estructura sobre la signatura Σ.

Definición 2.1 *(Estructuras con restricciones)*
Dada una signatura Σ*, una* Σ–estructura con restricciones $\mathcal{R}$ *es una terna*

$$\langle (D, \sqsubseteq_D), \{f^R\}_{f \in F}, \{p^R\}_{p \in \Pi} \rangle$$

en donde

- *El conjunto* D*, con el orden* $\sqsubseteq_D$*, es un dominio de Scott [138],*
- *Para cada* $f \in F^n$*,* f^R *es una función continua (en el sentido de teoría de dominios)* $f^R : D \times \ldots \times D \longrightarrow D$
- *Para cada* $p \in \Pi^n$*,* p^R *es una función continua (en el sentido de teoría de dominios)* $p^R : D \times \ldots \times D \longrightarrow Bool$*, siendo* $Bool$ *el dominio* $\{true, \bot_{bool}\}$.

Por comodidad suponemos que $\{true, \bot_{bool}\}$ es un subdominio de D, es decir, que $true \in D$ y $\bot_{bool} = \bot_D$.

Nótese que la continuidad de p^R significa que se tiene $p^R(x_1, \ldots, x_n) = true$ si y solo si $p^R(u_1, \ldots, u_n) = true$ para ciertas aproximaciones finitas $u_i \sqsubseteq_D x_i$. Expresamos esto también diciendo que p^R es una *relación continua*. Puesto que en el estudio de $CFLP(X)$ sólo haremos algunas referencias marginales al problema de la negación, podemos interpretar $\bot_{bool}$ como el valor booleano $false$, lo que contrasta con el enfoque trivalorado de [95] que utiliza tres valores lógicos $\{true, false, \bot_{bool}\}$ para distinguir la situación en que se sabe positivamente que un átomo b es falso del caso en que se desconozca si es cierto o falso.

A continuación definiremos el lenguaje de restricciones determinado por una signatura Σ. Previamente, indiquemos que la signatura Σ y un conjunto numerable de *variables* V permiten definir de la forma habitual los siguientes *dominios sintácticos*:

$t \in \mathcal{T}_F$	expresiones primitivas cerradas
$t \in \mathcal{T}_F(V)$	expresiones primitivas
$c \in \mathcal{B}_{\Pi,F}$	átomos primitivos cerrados, $c \equiv p(t_1, \ldots, t_n), p \in \Pi, t_i \in \mathcal{T}_F$
$c \in \mathcal{B}_{\Pi,F}(V)$	átomos primitivos, $c \equiv p(t_1, \ldots, t_n), p \in \Pi, t_i \in \mathcal{T}_F(V)$

Definición 2.2 *(Restricciones)*

Dada una signatura $\Sigma = F \cup \Pi$, el conjunto Con_Σ de Σ–restricciones primitivas es el cierre mediante conjunción y cuantificación existencial del conjunto de átomos $\mathcal{B}_{\Pi,F}(V)$.

Nótese que en una restricción pueden aparecer variables (de hecho, es lo natural), pero, por comodidad, en la notación Con_Σ hemos omitido la referencia al conjunto de variables V.

Puesto que no contemplamos negación ni cuantificación universal, toda restricción $\varphi \in Con_\Sigma$ admite una forma prenexa equivalente de la forma $\exists\overline{X}(c_1, \ldots, c_n)$, donde $c_i \in \mathcal{B}_{\Pi,F}(V)$ y la coma ',' representa la conjunción. Asumiremos a lo largo del trabajo que las restricciones están en esta forma prenexa. Aceptamos también que en Con_Σ está la restricción trivialmente cierta, que notamos por $true$, aunque suponga un abuso de notación, pues $true$ está ya utilizado para otros propósitos. En ocasiones nos será útil considerar una conjunción como el multiconjunto de las componentes que la integran.

Definición 2.3 *(Valoraciones)*

Sea $\mathcal{R}$ una Σ–estructura con restricciones. Una valoración es una aplicación $\alpha : V \longrightarrow D$.

Cada valoración α puede extenderse de modo natural a $\mathcal{T}_F(V)$ y $\mathcal{B}_{\Pi,F}(V)$. Dotamos al conjunto de valoraciones $\mathcal{VAL}$ del orden usual

$$\alpha_1 \sqsubseteq_{VAL} \alpha_2 \Leftrightarrow \alpha_1(X) \sqsubseteq_D \alpha_2(X), \text{para todo} X \in V$$

con el que $(\mathcal{VAL}, \sqsubseteq_{VAL})$ es un dominio de Scott, cuyos elementos finitos están caracterizados del siguiente modo: $\alpha \in \mathcal{VAL}$ es finita $\Leftrightarrow$ $\alpha(X)$ es finito, $\forall X \in V \wedge \alpha(X) = \bot_D, \forall X \in V$ excepto un número finito.

Definición 2.4 *(Satisfacción de restricciones)*

- *Una valoración α* satisface *(o es* solución *de) un átomo primitivo c (y escribimos $\alpha \models c$) si $\alpha(c) = true$.*
- *Una valoración α* satisface *(o es* solución *de) una restricción primitiva sin cuantificaciones $\varphi \equiv c_1, \ldots, c_n$ (y escribimos $\alpha \models \varphi$) si $\alpha \models c_i$, para todo i.*
- *Una valoración α* satisface *(o es* solución *de) una restricción primitiva $\varphi \equiv \exists\overline{X}\psi$ (y escribimos $\alpha \models \varphi$) si $\alpha' \models \psi$ para alguna valoración α' que coincida con α sobre $V - \overline{X}$.*
- *Una restricción primitiva φ es* satisfactible *si existe α tal que $\alpha \models \varphi$.*

Obsérvese que, aunque técnicamente suponemos que una valoración está definida sobre el conjunto V de todas las variables, con la definición anterior es claro que para tener $\alpha \models \varphi$ sólo es relevante el valor de α sobre las variables libres de φ.

Debemos asumir, para que el lenguaje que estemos definiendo tenga interés, que la satisfactibilidad de una restricción puede ser verificada de forma efectiva por algún mecanismo de *resolución de restricciones*. A pesar de ello, gran parte del estudio de las propiedades

del esquema $CFLP(X)$ no dependen de este hecho, por lo que de momento no necesitamos extendernos más sobre el tema. Más adelante analizaremos el papel que puede jugar en los cómputos un tal mecanismo de resolución de restricciones, así como algunas propiedades que debe cumplir para poder esperar un buen comportamiento del sistema de cómputo.

2.2 Definición de nuevas funciones. $CFLP$-programas

Dada una signatura $\Sigma = F \cup \Pi$, consideremos una nueva signatura Δ de modo que $\Delta \cap \Sigma = \emptyset$. Los símbolos de Δ deben ser entendidos como *símbolos de funciones no primitivas*, cuyo significado debe ser definido por el usuario como se indica a continuación. La nueva signatura agranda los dominios sintácticos a

$e \in \mathcal{T}_{F \cup \Delta}(V)$	Expresiones
$b \in \mathcal{B}_{\Pi, F \cup \Delta}(V)$	Atomos
$\varphi \in Con_{\Sigma \cup \Delta}$	Restricciones

Puesto que los únicos símbolos de predicado de los que disponemos son los primitivos de Π, todos los átomos de que consta una restricción $\varphi \in Con_{\Sigma \cup \Delta}$ son de la forma

$$p(e_1, \ldots, e_n), \quad p \in \Pi, \ e_i \in \mathcal{T}_{F \cup \Delta}(V)$$

Nótese la posibilidad de que los argumentos de un átomo tengan subexpresiones no primitivas.

Los símbolos no primitivos se definen mediante *reglas condicionales con restricciones*. En concreto

Definición 2.5 *(Reglas. Programas)*

- *Una* regla *para $f \in \Delta$ es de la forma*

$$f(X_1, \ldots, X_n) = e \Leftarrow \varphi$$

donde

1. *X_1,...,X_n son variables distintas*
2. *$e \in \mathcal{T}_{F \cup \Delta}(V)$*
3. *$\varphi \in Con_{\Sigma \cup \Delta}$*

- *Un* programa *para Δ es un conjunto finito de reglas para los símbolos de Δ.*

$f(X_1, \ldots, X_n)$ es la *cabeza* de la regla, e el *cuerpo* y φ la *restricción*. Nótese que en el cuerpo y la restricción de una regla pueden aparecer funciones no primitivas. Aceptaremos $f(X_1, \ldots, X_n) = e$ como abreviatura de

$$f(X_1, \ldots, X_n) = e \Leftarrow true$$

A falta por supuesto de dotar a las reglas de un programa de una semántica declarativa precisa, podemos de momento proporcionar la siguiente lectura lógica a una regla

$f(X_1, \ldots, X_n) = e \Leftarrow \varphi$: '$f(X_1, \ldots, X_n)$ es igual a (más precisamente, está aproximado por) e si se verifica la condición φ'.

No imponemos en principio ninguna condición sintáctica adicional sobre las reglas de un programa, como podrían ser 'inexistencia de variables libres en el cuerpo de una regla' o 'no ambigüedad del conjunto de reglas correspondientes a un mismo símbolo'. El motivo es que queremos estudiar con la máxima generalidad en qué condiciones podemos dar 'significado' a un programa. Como veremos, para ello sólo será requerida una condición semántica muy general de consistencia del programa.

Nótese también que, a diferencia de la presentación del esquema que se hace en [107, 105], no contemplamos explícitamente la posibilidad de definir predicados no primitivos. Ello no supone realmente pérdida de generalidad, pues un predicado p puede definirse mediante reglas de la forma

$$p(X_1, \ldots, X_n) = true \Leftarrow \varphi$$

para las que podríamos usar notación clausal

$$p(X_1, \ldots, X_n) : -\varphi$$

Existe además la posibilidad de incluir en la restricción de una regla condiciones de la forma

$$p(e_1, \ldots, e_n) == true$$

que podríamos abreviar a

$$p(e_1, \ldots, e_n)$$

Estamos asumiendo aquí que entre los símbolos de Π disponemos de algún tipo de igualdad $==$. De no ser así, definiríamos $==$ como igualdad estricta para el subdominio $Bool$, es decir: $(x == y) = true$ si y solo si $x = y = true$.

2.3 Algunas instancias de $CFLP(X)$

En este apartado estudiamos algunas instancias de $CFLP(X)$, poniendo de manifiesto cómo el esquema es una generalización de otros paradigmas de programación declarativa, que se convierten en casos particulares de aquél.

2.3.1 Programación lógica como instancia de $CFLP(X)$

Un programa lógico sobre una signatura de constructoras [4] $\mathcal{CS} = \bigcup \mathcal{CS}^n$ que defina un conjunto de predicados Δ puede entenderse como un programa en la instancia $CFLP(\mathcal{H}_{\mathcal{CS}})$ definida como sigue:

- Como signatura de base consideramos $\Sigma = \mathcal{CS} \cup \{==\}$ [5].

[4] En el contexto de la programación lógica relacional lo habitual es llamar 'símbolos de función' a lo que aquí llamamos 'constructoras', ya que no se contempla la posibilidad de más funciones que las libres para construir términos.

[5] En todas las instancias del esquema utilizaremos $==$ como símbolo de la igualdad, porque su interpretación nunca va a poder ser la igualdad $=$ 'de verdad', que no es continua en ningún dominio no trivial.

- El dominio $\mathcal{H}_{\mathcal{CS}_\perp}$ de la Σ-estructura de base es la extensión a cpo plano del universo de Herbrand $\mathcal{H}_{\mathcal{CS}}$ definido por $\mathcal{CS}$, al que añadimos un elemento indefinido $\perp$. El universo de Herbrand $\mathcal{H}_{\mathcal{CS}}$ es el conjunto de los términos sin variables – equivalentemente, árboles finitos – $\mathcal{T}_{\mathcal{CS}}$. El orden en $\mathcal{H}_{\mathcal{CS}_\perp}$ viene dado, como en toda extensión de un conjunto a cpo plano, por

$$a \sqsubseteq b \Leftrightarrow (a = \perp \vee a = b), \text{ para todos } a, b \in \mathcal{H}_{\mathcal{CS}_\perp}$$

- Los símbolos primitivos de $\Sigma = \mathcal{CS} \cup \{==\}$ están interpretados por las extensiones estrictas de las constructoras libres de $\mathcal{CS}$ y la igualdad [6] en $\mathcal{H}_{\mathcal{CS}}$, respectivamente.

- Las reglas del programa lógico, que serán de la forma

$$p(t_1, \ldots, t_n) :- b_1, \ldots, b_m.$$

  siendo $p \in \Delta$, t_i términos, b_j átomos, deben transformarse en la $CFLP$-regla

$$p(X_1, \ldots, X_n) = true \Leftarrow \begin{array}{l} X_1 == t_1, \ldots, X_n == t_n, \\ b_1 == true, \ldots, b_m == true \end{array}$$

  que considera al predicado p como una función 'booleana' que toma el valor $true$ para los argumentos en los que el predicado p se verifica, y el valor $\perp$ para el resto (incluido el $\perp$ añadido al dominio).

Obsérvese que la unificación está contemplada como un problema de resolución de restricciones: la unificación de un argumento X_i con el término t_i se expresa mediante la restricción $X_i == t_i$. El que de modo adicional el átomo b_j en el cuerpo de una regla se transforme en la restricción $b_j == true$ es menos relevante. Se debe simplemente al hecho ya comentado de que la sintaxis propuesta requiere que las condiciones de una regla sean restricciones expresadas mediante los predicados primitivos. Por comodidad, aceptamos en lo que sigue la sintaxis de 'programa lógico' como una abreviatura para la sintaxis de las $CFLP$-reglas correspondientes.

Introducimos a continuación un ejemplo concreto que nos servirá para comparar cómo algunos lenguajes declarativos se pueden considerar como instancias del esquema $CFLP(X)$.

Ejemplo 2.1

El ejemplo consiste simplemente en programar un autómata que reconozca el lenguaje $(ab)^*$. El autómata dispone de tres estados – s_a, s_b, s_{no} – que representan respectivamente '*se espera una a*', '*se espera una b*' y '*rechazar*'. El estado s_a es el inicial y el único aceptador. Las transiciones son las obvias.

Una forma natural de escribir un programa lógico para el autómata es la siguiente, si suponemos que la entrada está representada como una lista y los estados como constantes.

[6] La extensión estricta $==$ de la igualdad $=$ en $\mathcal{H}_{\mathcal{CS}}$ vendrá dada por: $(x == y) = true$ si y solo si x, y son iguales y distintos de $\perp$; en otro caso, $(x == y) = \perp$.

$$aceptar(L) : -inicial(S_0), recorrer(L, S_0, S_n), aceptador(S_n).$$

$$recorrer([\,], S, S).$$
$$recorrer([X \mid Xs], S, S_n) : -delta(X, S, S_1), recorrer(Xs, S_1, S_n).$$

$$delta(a, s_a, s_b). \qquad delta(b, s_a, s_{no}).$$
$$delta(a, s_b, s_{no}). \qquad delta(b, s_b, s_a).$$
$$delta(a, s_{no}, s_{no}). \qquad delta(b, s_{no}, s_{no}).$$

$$inicial(s_a).$$

$$aceptador(s_a).$$

Con las convenciones sintácticas indicadas más arriba, el programa anterior es un $CFLP$-programa.

Es interesante comentar que el hecho de considerar la programación lógica como instancia de $CFLP(X)$ nos proporciona de modo inmediato mayores recursos expresivos, ya que se pueden definir funciones, y no solamente 'predicados' (funciones con valor $true$). En el programa anterior es posiblemente más natural considerar los predicados $recorrer/3, delta/3$ como funciones $recorrer/2, delta/2$ definidas por las reglas

$$recorrer([\,], S) = S.$$
$$recorrer([X \mid Xs], S) = recorrer(Xs, delta(X, S)).$$

$$delta(a, s_a) = s_b. \qquad delta(b, s_a) = s_{no}.$$
$$delta(a, s_b) = s_{no}. \qquad delta(b, s_b) = s_a.$$
$$delta(a, s_{no}) = s_{no}. \qquad delta(b, s_{no}) = s_{no}.$$

y redefinir $aceptar/1$ como

$$aceptar(L) : -inicial(S_0), aceptador(recorrer(L, S_0)).$$

Nótese sin embargo que las funciones que se definan serán funciones estrictas (pues las condiciones $X_i == t_i$ que reemplazan a la unificación de X_i con t_i no se satisfacen si $X_i = \bot$), y que en definitiva esta posibilidad de definir funciones podría perfectamente considerarse a su vez como una facilidad sintáctica en el marco de la programación lógica relacional. De todos modos, similares ampliaciones de Prolog con funciones estrictas han sido objeto de algunas investigaciones [91] que intentan sacar provecho en la implementación del carácter funcional del lenguaje. Como detalle marginal indiquemos que la semántica declarativa del lenguaje obtenido queda bien establecida, al ser heredada de la que vamos a desarrollar en general para $CFLP(X)$.
$\diamond$

2.3.2 $CFLP(X)$ como generalización del esquema $CLP(X)$

El esquema $CLP(X)$ [78, 79, 81] es una generalización de la programación lógica, que se convierte en una instancia de aquél. Mostramos aquí, siguiendo pautas muy similares a las

del apartado anterior, cómo toda instancia de $CLP(X)$, definida de acuerdo con [78, 79], puede considerarse como instancia de $CFLP(X)$.

Asumimos pues una instancia $CLP(\mathcal{R})$, en la que está fijada una signatura de base $\Sigma = F \cup \Pi$ que consta, como en nuestro caso, de sendos conjuntos de símbolos de funciones primitivas F y predicados primitivos Π. Suele requerirse, aunque no es esencial, que en Π haya un símbolo de igualdad $==$. $\mathcal{R}$ es una Σ-estructura (en el sentido de CLP), que consta de un dominio D (el dominio de cómputo) e interpretaciones fijas para los símbolos de Σ como funciones y predicados (relaciones) sobre D.

La instancia $CLP(\mathcal{R})$ puede convertirse en una instancia $CFLP(\mathcal{R}_\perp)$, sin más que considerar la misma signatura Σ (añadiendo la constante $true$ si es preciso), y la Σ-estructura (en el sentido de $CFLP$) $CFLP(\mathcal{R}_\perp)$ que tiene por dominio $D_\perp$ la extensión a cpo plano de D y que interpreta los símbolos primitivos como las extensiones estrictas de las intepretaciones correspondientes en $\mathcal{R}$.

La correspondencia entre programas en ambos marcos es casi inmediata: una CLP-regla para un predicado definido tiene la forma

$$p(X_1, \ldots, X_n) : -b_1, \ldots, b_m \Box \varphi$$

siendo los b_i átomos encabezados por predicados definidos y φ una restricción primitiva. La correspondiente $CFLP$-regla será

$$p(X_1, \ldots, X_n) = true \Leftarrow b_1 == true, \ldots, b_m == true, \varphi$$

para la que podríamos aceptar, como anteriormente, la sintaxis de 'programa lógico'

$$p(X_1, \ldots, X_n) : -b_1, \ldots, b_m, \varphi$$

resultando prácticamente idéntica a la regla original.

Al igual que en el caso de la programación lógica, trasladar una instancia de CLP a $CFLP$ nos proporciona de manera 'gratuita' la posibilidad de definir funciones (que serán, como entonces, estrictas).

Un lenguaje interesante que puede obtenerse como instancia de $CLP(X)$, y por tanto de nuestro esquema $CFLP(X)$, es *PROLOG II* [31, 32, 33]. *PROLOG II* puede caracterizarse como la instancia $CLP(\mathcal{RT}_{\mathcal{CS}\cup\{==,\neq\}})$, en la que el dominio de cómputo es el conjunto $\mathcal{RT}$ de los árboles regulares sobre un conjunto de constructoras $\mathcal{CS}$, sobre el que se dispone de los predicados primitivos de igualdad y desigualdad, que son decidibles en este dominio [32, 108]. Nótese que en el dominio plano $\mathcal{RT}_\perp$ que consideramos al pasar al marco $CFLP$, todos los elementos son finitos (respecto al orden del dominio) aun cuando sean árboles infinitos, y que las operaciones primitivas son estrictas.

Modifiquemos, para aprovechar las posibilidades de *PROLOG II*, el ejemplo 2.1 del autómata – que, por cierto, es una ligera modificación de un ejemplo propuesto por Colmerauer [31]–. En lugar de utilizar constantes, representamos un estado S del autómata como un término estructurado $est(St, St', SiNo)$, donde St es la representación del estado al que se llega si la entrada es a, St' el correspondiente para entrada b, y $SiNo$ puede ser si o no, indicando si el estado es aceptador o no.

Con esta representación no es necesario hacer explícita mediante reglas la función de transición, pues toda la información necesaria está incluida en la representación de los estados (en la del inicial, p. ej.). Las siguientes reglas constituyen un programa *PROLOG II* para el autómata, y un programa en $CFLP$, si adoptamos las facilidades sintácticas del apartado 2.3.1.

$$
\begin{array}{l}
aceptar(L) : -inicial(S_0), recorrer(L, S_0, S_n), aceptador(S_n). \\
inicial(S_a) : - \; S_a \;\; == \;\; est(S_b, S_{no}, si), \\
\qquad\qquad\quad\;\; S_b \;\; == \;\; est(S_{no}, S_a, no), \\
\qquad\qquad\quad\;\; S_{no} \;\; == \;\; est(S_{no}, S_{no}, no). \\
recorrer([\,], S, S). \\
recorrer([a \mid Xs], est(S, \_, \_), S_n) : -recorrer(Xs, S, S_n). \\
recorrer([b \mid Xs], est(\_, S, \_), S_n) : -recorrer(Xs, S, S_n). \\
aceptador(est(\_, \_, si)).
\end{array}
$$

Nótese que S_a, S_b, etc en las reglas anteriores son variables, y que es el predicado *inicial* el que define, a través de las restricciones en el cuerpo de su regla, la representación para el estado inicial explicada arriba. Pero es necesario disponer de árboles regulares, pues las condiciones impuestas a S_a, S_b, S_{no} en esa cláusula admiten como solución árboles regulares infinitos. La unificación sintáctica habitual para términos fallaría por el 'occur-check'.

Para terminar con este apartado, recordemos que en nuestro esquema tenemos la posibilidad de programar en lo que en este caso podríamos llamar *PROLOG II con funciones*. Por ejemplo, parece más natural definir *recorrer* como función mediante las reglas

$$
\begin{array}{l}
recorrer([\,], S) = S. \\
recorrer([a \mid Xs], est(S, \_, \_)) = recorrer(Xs, S). \\
recorrer([b \mid Xs], est(\_, S, \_)) = recorrer(Xs, S).
\end{array}
$$

2.3.3 Programación lógico-funcional

Bajo el nombre de 'programación lógico-funcional' pueden encuadrarse muchos lenguajes (véase la sección 1.2) que resultan de combinar las características de ambos paradigmas. De entre ellos, destacamos *K-LEAF* [99, 58], *BABEL* [117, 114, 118] y *SFL* [63, 64, 61] como los más próximos a las ideas sobre las que reposa nuestro esquema $CFLP(X)$. Usaremos en concreto *SFL* [7] como lenguaje de referencia para el resto de este apartado. La noción de programación lógico-funcional (de primer orden y perezosa) que proporciona *SFL* puede ser considerada –como en las instancias anteriores – como programación sobre un dominio de árboles; pero en este caso la noción adecuada de árbol viene dada por los elementos del *universo de Herbrand infinitario* $\mathcal{H}$ [58, 118] determinado por un conjunto de constructoras, constituido por el conjunto de los árboles finitos o infinitos, total o parcialmente definidos, construidos con ayuda de las constructoras más un elemento $\perp$ añadido.

En *SFL*, la forma de definir árboles infinitos es en cierto modo complementaria de la de *Prolog II*: podemos definir funciones para este propósito, pero no predicados definidos

[7] Más exactamente, tomamos aquí como referencia el fragmento de primer orden de *SFL*. La programación de orden superior se considera en la sección 2.3.4, más adelante.

por un conjunto de ecuaciones. El motivo de ello es que la igualdad en el dominio $\mathcal{H}$ no es computable. Aunque más adelante trataremos con más detalle el problema de la igualdad, indiquemos aquí que una noción natural de igualdad que se puede considerar para ser utilizada en las condiciones de las reglas es la *igualdad estricta* $==$: $t == s$ si y solo si los árboles t y s son idénticos, finitos y totalmente definidos (no contienen $\perp$). Con esta igualdad las condiciones de, p. ej., la regla para *inicial* en el apartado 2.3.2 no admiten solución. De hecho, ningún conjunto de ecuaciones escritas con $==$ admite como soluciones árboles infinitos, y más en general, las soluciones de una restricción continua en este dominio no pueden ser *sólo* árboles infinitos.

Ahora bien, en lugar de intentar definir árboles infinitos por medio de restricciones, podemos definir funciones cuyos valores sean dichos árboles infinitos, obtenidos como límites de aproximaciones finitas. Por ejemplo, los árboles infinitos que en el caso de *Prolog II* representaban a los estados en nuestro ejemplo del autómata, pueden obtenerse ahora como valores de las funciones (no constructoras) constantes s_a, s_b, s_{no} definidas por las reglas

$$\begin{aligned} s_a &= est(s_b, s_{no}, si). \\ s_b &= est(s_{no}, s_a, no). \\ s_{no} &= est(s_{no}, s_{no}, no). \end{aligned}$$

Por aplicación de estas reglas se irían 'construyendo' aproximaciones cuyos límites serían los valores denotados por s_a, s_b, s_{no}, que resultarían ser los mismos árboles infinitos que se obtenían en el caso de *Prolog II*. Obsérvese que si las mismas reglas se utilizan en la extensión funcional de *Prolog II* descrita en 2.3.2, las funciones s_a, s_b, s_{no} quedarían indefinidas, debido a la naturaleza estricta que en ese caso tienen las constructoras, y que es heredada por las funciones definidas.

El resto de nuestro programa ejemplo podría quedar en SFL como sigue [8]

$$\begin{aligned} &aceptar(L) : -aceptador(recorrer(L, s_a)). \\ &aceptador(est(S, S_1, si)). \\ &recorrer([\,], S) = S. \\ &recorrer([a \mid Xs], est(S, S_1, Flag)) = recorrer(Xs, S). \\ &recorrer([b \mid Xs], est(S_1, S, Flag)) = recorrer(Xs, S). \end{aligned}$$

que podrían ser incluso $CFLP$-reglas si, como en anteriores ocasiones, aceptamos como facilidades sintácticas la notación de cláusulas para abreviar la definición de 'predicados' como funciones con valor $true$, y el uso de patrones en las cabezas de las reglas para expresar un problema de unificación. En este caso, sin embargo, la cuestión de la unificación requiere un análisis más cuidadoso, como discutimos a continuación.

Consideremos, por ejemplo, la segunda regla para $recorrer$. En las dos instancias estudiadas en 2.3.1 y 2.3.2, deberíamos considerar dicha regla como abreviatura de la $CFLP$-regla

$$recorrer(L, St) = recorrer(Xs, S) \Leftarrow L == [a \mid Xs], St == est(S, S_1, Flag).$$

[8] Las cláusulas estilo Prolog no se admiten directamente en la sintaxis de SFL, pero pueden considerarse como una facilidad sintáctica, exactamente igual que hacemos para $CFLP(X)$.

en la que la unificación de los argumentos L, St con los patrones $[a \mid Xs]$ y $est(S, S_1, Flag)$ está expresada a través de las ecuaciones

$$L == [a \mid Xs], St == est(S, S_1, Flag)$$

de la condición de la regla.

Pero recordemos que en SFL la igualdad $==$ debe interpretarse como igualdad *estricta* en el sentido de que

$$t == s \Leftrightarrow t, s \text{ son idénticos, finitos y totales}$$

y sin embargo, para la naturaleza perezosa del lenguaje es crucial el hecho de que la unificación pueda tener éxito incluso en presencia de argumentos que denoten objetos infinitos. Utilizando $==$ para expresar la unificación, resultaría en nuestro ejemplo que la regla indicada no es aplicable a la expresión $recorrer([a, b], s_a)$, pues la condición $s_a == est(S, S_1, Flag)$ no puede ser satisfecha al denotar s_a un árbol infinito. Puesto que las otras reglas de $recorrer$ tampoco serían aplicables, $recorrer([a, b], s_a)$ quedaría indefinido, y también $aceptar([a, b])$, en contra de lo pretendido.

Lo que nos vendría bien, en realidad, es expresar la unificación mediante la igualdad 'de verdad' $=$. Nuestra regla ejemplo quedaría traducida por

$$recorrer(L, St) = recorrer(Xs, S) \Leftarrow L = [a \mid Xs], St = est(S, S_1, Flag).$$

En este caso la condición $St = est(S, S_1, Flag)$ está por una parte indicando que St debe ser un árbol de la forma $est(S, S_1, Flag)$ para ciertos valores (posiblemente infinitos o parciales) de $S, S_1, Flag$, y por otra determinando de manera unívoca dichos valores $S, S_1, Flag$ (de los cuales sólo uno, S, es usado en el resto de la regla).

El problema es que, como ya hemos dicho, la igualdad $=$ en el universo de Herbrand infinitario no es continua (no es, de hecho, computable). No podemos resolver este problema – al convertir SFL en una instancia de $CFLP(X)$ – intentando dar a la igualdad $==$ un significado continuo más próximo a $=$, pues puede probarse que con la definición dada $==$ es la máxima aproximación continua a la igualdad en el dominio $\mathcal{H}$.

Para expresar la unificación de forma continua, necesitamos introducir nuevas operaciones predefinidas que desempeñen el doble papel que juega la unificación: selección de alternativas (condiciones de aplicabilidad de las reglas) y acceso a componentes de los argumentos. Estas operaciones predefinidas van a ser *funciones selectoras* de las componentes de árboles y *predicados reconocedores* del aspecto más externo de los árboles. Funciones y predicados de este estilo forman parte de la propia sintaxis de algún lenguaje lógico-funcional [141].

Para nuestra regla ejemplo – la segunda de $recorrer$ –, necesitaríamos

- Predicados reconocedores: is_cons, is_a, is_est, todos ellos de aridad 1, que se verifican cuando el argumento es un árbol cuya constructora raíz es la correspondiente al predicado.
- Funciones selectoras: $cons_1, cons_2, est_1, est_2, est_3$, también de aridad 1, que devuelven la componente del argumento indicada por el número, si es que el argumento está encabezado por la constructora correspondiente.

Antes de dar definiciones más precisas, indiquemos que nuestra regla ejemplo

$$recorrer([a \mid Xs], est(S, S_1, Flag)) = recorrer(Xs, S).$$

podría considerarse como abreviatura sintáctica de la $CFLP$-regla

$$recorrer(L, St) = recorrer(cons_2(L), est_1(St)) \Leftarrow \\ is\_cons(L), is\_a(cons_1(L)), is\_est(St).$$

Algunas observaciones:

- La aparición de constructoras en los patrones de la cabeza se corresponde con el uso de los reconocedores is_c en la restricción.
- Las variables de los patrones que se usen también en el lado derecho de la regla (Xs, S en el ejemplo) requieren el uso de las selectoras c_k. Quedan así cubiertos los dos papeles comentados de la unificación.
- Para las constructoras c constantes (de aridad 0) no hay inconveniente en usar la igualdad $==$ en lugar del reconocedor is_c. En el ejemplo, la condición $is\_a(cons_1(L))$ puede reemplazarse por $cons_1(L) == a$.
- Para que la unificación con los patrones de una regla pueda ser expresada tal como se ha esbozado, es importante que los patrones sean *lineales*, es decir, que no aparezcan variables repetidas en la cabeza de la regla. En efecto, si nuestra regla ejemplo fuese

  $$recorrer([a \mid Xs], est(S, S, Flag)) = recorrer(Xs, S).$$

  donde el patrón $est(S, S, Flag)$ pretendiese significar que el segundo argumento ha de ser un árbol de la forma $est(t_1, t_2, f)$ con $t_1 = t_2$ (pero posiblemente infinitos), no tendríamos forma de indicar este hecho mediante las operaciones continuas de que disponemos [9]. Esto no afecta a la validez de nuestra propuesta, pues la condición de linealidad de las cabezas es de hecho exigida a los programas SFL.

Continuando aún con nuestro ejemplo, ésta sería la 'traducción' a $CFLP$ del programa completo

$$s_a = est(s_b, s_{no}, si).$$
$$s_b = est(s_{no}, s_a, no).$$
$$s_{no} = est(s_{no}, s_{no}, no).$$
$$aceptar(L) : -aceptador(recorrer(L, s_a)).$$
$$aceptador(St) : -is\_est(St), is\_si(est_3(St)).$$
$$recorrer(L, S) = S \Leftarrow is\_nil(L).$$
$$recorrer(L, St) = recorrer(cons_2(L), est_1(St)) \Leftarrow \\ is\_cons(L), is\_a(cons_1(L)), is\_est(St).$$
$$recorrer(L, St) = recorrer(cons_2(L), est_2(St)) \Leftarrow \\ is\_cons(L), is\_b(cons_1(L)), is\_est(St).$$

[9] Si la condición que se pretende expresar es la igualdad estricta $t_1 == t_2$, no habría ningún inconveniente, y simplemente aceptaríamos la repetición de variables en la cabeza como una facilidad sintáctica más.

donde hemos usado la notación de cláusulas para definir predicados, cuyo significado ya hemos establecido para cualquier instancia del esquema.

En general, un programa SFL sobre una signatura de constructoras $\mathcal{CS} = \bigcup \mathcal{CS}^n$ puede entenderse como un programa en la instancia $CFLP(\mathcal{H}_{\mathcal{CS}})$ definida como sigue:

- Como signatura de base consideramos $\Sigma = F \cup \Pi$, siendo

$$F = \mathcal{CS} \cup \{c_k : c \in \mathcal{CS}^n, 1 \leq k \leq n\}$$

$$\Pi = \{==\} \cup \{is\_c : c \in \mathcal{CS}\}$$

  Como en anteriores ocasiones, suponemos que $\mathcal{CS}$ dispone de la constante $true$ para expresar el valor de los predicados definidos por el programa.

- El dominio de base es el universo de Herbrand infinitario $\mathcal{H}_{\mathcal{CS}}$ definido por $\mathcal{CS} \cup \{\bot\}$, es decir, el conjunto de los árboles finitos o infinitos, total o parcialmente definidos, construidos con ayuda de $\mathcal{CS} \cup \{\bot\}$. El orden en $\mathcal{H}_{\mathcal{CS}}$ es el menor orden $\sqsubseteq$ que verifica
  - $\bot \sqsubseteq t, \forall t \in \mathcal{H}_{\mathcal{CS}}$
  - $c \in \mathcal{CS}^n, t_1 \sqsubseteq s_1, \ldots, t_n \sqsubseteq s_n \Rightarrow c(t_1, \ldots, t_n) \sqsubseteq c(s_1, \ldots, s_n)$.

  Con este orden los elementos finitos del dominio son los árboles finitos, y los elementos totales son los árboles totalmente definidos, es decir, los que no contienen $\bot$ en sus hojas.

- Los símbolos de $\mathcal{CS}$ están interpretados como constructoras libres no estrictas, y los símbolos c_k como las funciones selectoras

$$c_k(t) = \begin{cases} t_k & \text{si } t = c(t_1, \ldots, t_n) \\ \bot & \text{e.o.c.} \end{cases}$$

- El símbolo $==$ se interpreta como la igualdad estricta

$$(t_1 == t_2) = \begin{cases} true & \text{si } t_1, t_2 \text{ son iguales, finitos y totales} \\ \bot_{bool} & \text{e.o.c.} \end{cases}$$

  y los símbolos is_c como los predicados reconocedores

$$is\_c(t) = \begin{cases} true & \text{si } t = c(t_1, \ldots, t_n) \\ \bot_{bool} & \text{e.o.c.} \end{cases}$$

  Es fácil comprobar que todas las operaciones primitivas son continuas.

- Las reglas del programa SFL se transforman en reglas para la instancia $CFLP(\mathcal{H}_{\mathcal{CS}})$ de acuerdo con las ideas apuntadas en el ejemplo anterior.

En la segunda parte de este trabajo (véase el capítulo 5) analizaremos con detalle el proceso de traducción. Es más, veremos que tiene una relación muy estrecha con la traducción

'no continua' que utilizaba igualdades $X = t$ para expresar la unificación del argumento X con el patrón t. Esto justificará ese uso restringido de la igualdad no continua $=$.

Digamos, para terminar, que toda la segunda parte de este trabajo será dedicada al estudio de una generalización de esta instancia, en la que consideraremos como restricciones primitivas sobre los árboles inifinitos no sólo la igualdad $==$ ya definida, sino también una restricción de desigualdad $=/=$. La condición $x =/= y$ expresará que x e y son inconsistentes en $\mathcal{H}$, es decir, que discrepan en alguna constructora en una misma posición. Utilizaremos esta instancia ampliada, llamémosla $CFLP(\mathcal{H}_{\neq})$, en algunos ejemplos.

2.3.4 Programación lógico-funcional de orden superior

El enfoque dado en [62, 63, 64, 60, 61] a la programación lógico-funcional de orden superior permite considerar ésta como un asunto de primer orden, tando desde el punto sintáctico como semántico, lo que nos va a permitir caracterizarla como una instancia de nuestro esquema.

Comencemos por especificar el dominio de cómputo que corresponde a un programa que utiliza un conjunto de símbolos de constructoras $\mathcal{CS} = \bigcup_{n\in N} \mathcal{CS}^n$ y define un conjunto de símbolos de funciones no libres $F = \bigcup_{n\in N} F^n$. Consideramos como dominio de base el *Universo de Herbrand $\mathcal{PH}_\Sigma$ de los patrones parciales* (posiblemente infinitos) determinado por $\Sigma = \mathcal{CS} \cup F$. $\mathcal{PH}_\Sigma$ puede ser definido como el universo de Herbrand infinitario (en el sentido de 2.3.3) $\mathcal{H}_{\Sigma'}$ determinado por la signatura

$$\Sigma' = \bigcup_{n\in N} \{c^k : c \in \mathcal{CS}^n, 0 \leq k \leq n\} \cup \bigcup_{n\in N} \{f^k : f \in F^n, 0 \leq k < n\}$$

en la que se distinguen con símbolos distintos los distintos 'grados' en que se pueden aplicar los símbolos de $\mathcal{CS}$ y F (total o parcialmente para el caso de $\mathcal{CS}$, sólo parcialmente en el de F).

Podríamos decir que el dominio $\mathcal{PH}_\Sigma$ da significado a los términos construidos y a las aplicaciones parciales de símbolos de F. Será la semántica declarativa del programa la que se encargará de dotar de significado a las aplicaciones totales de los símbolos de F (a través de las reglas que los definan). Observemos que aunque el dominio está determinado por $\mathcal{CS} \cup F$, y por tanto depende de una manera muy estrecha del programa, la situación no es absolutamente nueva, pues también en el caso de programas lógicos y lógico-funcionales de primer orden el dominio dependía del conjunto $\mathcal{CS}$ de constructoras utilizadas en él.

En $\mathcal{PH}_\Sigma$ (entendido como $\mathcal{H}_{\Sigma'}$) se dispone también de la igualdad continua $==$ y de las funciones selectoras c_k y predicados reconocedores is_c expuestos en 2.3.3. En particular, recordemos que

$$(t == s) = \begin{cases} true & \text{si } t, s \text{ son iguales, finitos y totales} \\ \bot & \text{en otro caso} \end{cases}$$

Esta definición da a la igualdad $==$ un carácter *intensional*. Para aclarar esta afirmación, recordemos que algunos elementos de $\mathcal{PH}_\Sigma$ pueden entenderse como aplicaciones parciales de los símbolos de constructora o función, y por consiguiente, representan funciones, aun cuando hayan sido convertidas en objetos de primer orden al identificar $\mathcal{PH}_\Sigma$ con $\mathcal{H}_{\Sigma'}$. Puede perfectamente suceder que dos elementos distintos $t, s \in \mathcal{H}_{\Sigma'}$ representen la misma función (en

el sentido matemático usual, o sea, extensional) y sin embargo, no se tendrá $(t == s) = true$. Eso sucederá por ejemplo con $f^1(a)$ y $g^1(b)$, si $a, b \in \mathcal{CS}^0$ y $f, g \in F^2$ están definidas del mismo modo, p. ej., como proyecciones sobre el segundo argumento

$$f(X, Y) = Y.$$
$$g(X, Y) = Y.$$

Tanto $f^1(a)$ como $g^1(b)$ representan la función identidad, pero de manera intensionalmente distinta.

En cuanto a los programas (que suponemos escritos en sintaxis de primer orden), estarán compuestos de reglas para los símbolos de F de la forma

$$f(t_1, \ldots, t_n) = e \Leftarrow \varphi$$

donde $f \in F^n$, t_i son términos lineales, e es una expresión, y φ es una colección de ecuaciones de la forma $e_1 == e_2$. Como ya es habitual para nosotros, aceptamos los patrones en la cabeza como una facilidad sintáctica para expresar la unificación, que podría eliminarse mediante el uso de las funciones selectoras y predicados reconocedores, de la misma manera que en 2.3.3.

Al escribir en sintaxis de primer orden una regla que utilice recursos de orden superior, aparecerán posiblemente, tanto en e como en φ, expresiones de la forma $@(e_1, e_2)$, para indicar la aplicación de e_1 a e_2. Debemos considerar @ como un símbolo de función definida ($@ \in \Delta^2$) pero $@ \notin F$, es decir, @ no sirve para construir patrones parciales (los elementos del dominio $\mathcal{PH}_\Sigma$). El símbolo @ viene definido por las siguientes reglas, que suponemos añadidas a todo programa.

$$@(c^k(X_1, \ldots, X_k), E) = c^{k+1}(X_1, \ldots, X_k, E).$$
$$\quad \% \text{ Para cada } c \in \mathcal{CS}^n, 0 \leq k < n$$
$$@(f^k(X_1, \ldots, X_k), E) = f^{k+1}(X_1, \ldots, X_k, E).$$
$$\quad \% \text{ Para cada } f \in F^n, 0 \leq k < n - 1$$
$$@(f^{n-1}(X_1, \ldots, X_{n-1}), E) = f(X_1, \ldots, X_{n-1}, E).$$
$$\quad \% \text{ Para cada } f \in F^n$$

Obsérvese que no hay regla para el caso $@(c^n(X_1, \ldots, X_n), E)$ con $c \in \mathcal{CS}^n$, y por tanto @ queda indefinida en ese caso, como es natural, ya que $c^n(X_1, \ldots, X_n)$ con $c \in \mathcal{CS}^n$ es un objeto puramente de primer orden, que no representa una función.

El significado que a @ le dan las dos primeras reglas es por completo independiente de las reglas del programa que estemos considerando. Podemos considerar que es la parte 'predefinida' de @. Esto ya no sucede con la tercera regla para @, y es por ello por lo que @ debe considerarse una función definida, y no predefinida.

En [61] se estudia con detalle todo lo relativo a esta concepción de primer orden de un lenguaje lógico funcional de orden superior, así como su relación con un enfoque puramente de orden superior.

2.3.5 Arboles sobre una estructura con restricciones

Es habitual en programación lógica con restricciones el definir una instancia por el procedimiento que sigue: una vez fijada una estructura inicial de interés (p. ej., los números reales

con las operaciones $+, *$ y las restricciones $=, <, >$), se considera como dominio de cómputo para la instancia el conjunto de los árboles (términos) finitos construidos con auxilio de un conjunto de constructoras libres añadido, y con elementos del dominio inicial en las hojas (p. ej. listas de números reales). Como resultado se obtienen lenguajes muy flexibles, en los que se combina la posibilidad de realizar cómputos simbólicos (usando los términos) con cómputos sobre el dominio concreto. Un ejemplo típico de ello es $CLP(\mathcal{R})$ [82, 84].

Nuestro propósito en este apartado es explicar cómo un proceso similar se puede realizar fácilmente en el marco de $CFLP(X)$, obteniendo como resultado instancias aún más expresivas, no sólo porque podremos definir funciones y predicados en los programas, sino porque el dominio a construir va a ser más amplio, al incluir árboles infinitos.

Sea entonces $\Sigma = F \cup \Pi$ una signatura inicial y

$$\mathcal{R} = \langle (D, \sqsubseteq_D), (f^D)_{f \in F}, (p^D)_{p \in \Pi} \rangle$$

una Σ-estructura con restricciones. Suponemos que $== \in \Pi$ y que $==^D$ es una aproximación continua de la igualdad en D, es decir

$$==^D : D \times D \to \{\text{true}, \bot_{\text{bool}}\} \text{ es continua y } (a ==^D b) = true \Rightarrow a = b$$

Consideremos ahora un conjunto de nuevos símbolos de constructora $\mathcal{CS} = \bigcup \mathcal{CS}^n$. El conjunto $\mathcal{CS}$, junto con el dominio D, define el nuevo dominio $\mathcal{T}_D$ cuyo soporte es el conjunto de los árboles, finitos o infinitos, con nodos ocupados por constructoras (no constantes) de $\mathcal{CS}$ y cuyas hojas son constructoras constantes de $\mathcal{CS}$ o elementos de D (en particular, $\bot$).

Formalmente, dicho soporte puede definirse como el universo de Herbrand infinitario (ver 2.3.3) determinado por el conjunto de constructoras $\mathcal{CS} \cup (D - \{\bot\})$, donde los elementos de $D - \{\bot\}$ son nuevas constantes para la construcción de dicho universo. Nótese que el dominio original D puede considerarse como un subconjunto del dominio extendido $\mathcal{T}_D$ sin más que identificar cada elemento $x \in D$ con el árbol en cuya raíz está la constante x, y el indefinido de D con el propio de la construcción del universo de Herbrand infinitario. Usaremos dicha identificación en lo que sigue.

El orden que consideramos en $\mathcal{T}_D$ no es sin embargo el habitual para el universo de Herbrand anterior, pues los elementos de D (vistos como árboles) serían incomparables entre sí, con lo que perderíamos la estructura de orden de D. Lo natural es considerar el menor orden $\sqsubseteq_{\mathcal{T}_D}$ que verifica

- $\bot \sqsubseteq_{\mathcal{T}_D} t, \forall t \in \mathcal{T}_D$
- $a, b \in D, a \sqsubseteq_D b \Rightarrow a \sqsubseteq_{\mathcal{T}_D} b$
- $c \in \mathcal{CS}^n, t_1 \sqsubseteq_{\mathcal{T}_D} s_1, \ldots, t_n \sqsubseteq_{\mathcal{T}_D} s_n \Rightarrow c(t_1, \ldots, t_n) \sqsubseteq_{\mathcal{T}_D} c(s_1, \ldots, s_n)$

Con este orden $\mathcal{T}_D$ es un dominio de Scott, que convertimos en una estructura con restricciones sobre la signatura $\Sigma' = F' \cup \Pi'$ dada por

$$F' = F \cup \mathcal{CS} \cup \{c_k : c \in \mathcal{CS}^n, 1 \leq k \leq n\}$$
$$\Pi' = \Pi \cup \{is_c : c \in \mathcal{CS}\}$$

interpretando los símbolos de Σ' como sigue:

- Un símbolo de la signatura original Σ se interpreta en $\mathcal{T}_D$ como la mínima extensión continua de la correspondiente interpretación en D. Es decir, si $f \in F^n$, definimos

$$\begin{array}{rccl} f^{\mathcal{T}_D}: & \mathcal{T}_D^n & \longrightarrow & \mathcal{T}_D \\ & (x_1,\ldots,x_n) & \longrightarrow & f^D(d(x_1),\ldots,d(x_n)) \end{array}$$

  donde

$$d(x) = \begin{cases} x \text{ si } x \in D \\ \bot \text{ e.o.c.} \end{cases}$$

  es decir, reemplazamos los árboles 'genuinos' por $\bot$. Nótese que $f^{\mathcal{T}_D}$ es en efecto una extensión continua de f^D, y además la menos definida posible, pues para todo $x \in \mathcal{T}_D$ debe verificarse

$$f^{\mathcal{T}_D}(x_1,\ldots,x,\ldots,x_n) \sqsupseteq f^{\mathcal{T}_D}(x_1,\ldots,\bot_{\mathcal{T}_D},\ldots,x_n)$$

  y disponemos de un solo elemento indefinido $\bot_{\mathcal{T}_D} = \bot_D$. Una definición análoga se tiene para los predicados $p \in \Pi$ que no sean el de igualdad $==$, del que hablamos más abajo.

- Los símbolos de constructora añadidos $\mathcal{CS}$ se interpretan como en el caso de 2.3.3, es decir, como constructoras libres perezosas. También de modo análogo se definen las funciones selectoras c_k y los predicados reconocedores is_c.

- La igualdad $==_{\mathcal{T}_D}$ en $\mathcal{T}_D$ requiere un tratamiento distinto del resto de funciones y predicados en $F \cup \Pi$. Ahora no nos interesa una extensión minimal de $==_D$, ya que queremos disponer de una igualdad específica de árboles. Por ejemplo, si $(a ==_D b) = true$, queremos que $(c(a) == c(b)) = true$ (siendo $c \in \mathcal{CS}^1$) y no $(c(a) == c(b)) = \bot_{bool}$, como resultaría de la definición para el resto de los predicados de Π. Definimos $==_{\mathcal{T}_D}$ como la máxima aproximación continua de la igualdad en $\mathcal{T}_D$ que preserva la igualdad $==_D$ predefinida en D. Puede comprobarse que $==_{\mathcal{T}_D}$ viene dada por: $t ==_{\mathcal{T}_D} s \Leftrightarrow t = s$, t es finito y total en $\mathcal{T}_D$ y toda hoja $a \in D$ de t verifica $a ==_D a$. Nótese que los elementos finitos y totales de $\mathcal{T}_D$ son los árboles finitos cuyas hojas son constantes de $\mathcal{CS}^0$ o elementos finitos y totales de D (lo que excluye a $\bot$). Nótese también que si la igualdad $==_D$ viene dada por

$$a ==_D b \Leftrightarrow a = b, a \text{ es finito y total en } D$$

  es decir, $==_D$ es la máxima restricción continua de $=$ en D, entonces la definición anterior se reduce a

$$t ==_D s \Leftrightarrow t = s, t \text{ es finito y total en } \mathcal{T}_D$$

  lo que en particular coincide con la noción de igualdad continua definida para el universo de Herbrand infinitario.

  Indiquemos también que, en algunas instancias construidas de este modo, un tratamiento similar al dado para la igualdad $==$ podría ser conveniente para otros predicados o funciones de la estructura inicial (por ejemplo, si ésta dispone también de una desigualdad $=/=$). Ello dependerá lógicamente del interés que tenga el dar significado no trivial a dicha operación o predicado en caso de ser aplicada a árboles.

Para terminar este apartado, veamos un ejemplo concreto de una instancia de este estilo.

Ejemplo 2.2 *(Arboles de números reales)*

Consideremos la estructura $\langle \mathcal{R}_\perp, +, -, *, <, >, == \rangle$ cuyo dominio es la extensión a dominio plano del conjunto de los números reales $\mathcal{R}$, con las operaciones indicadas interpretadas del modo natural. Si consideramos ahora el conjunto de constructoras $\mathcal{CS} = \{[\_ \mid \_], []\}$, el dominio $\mathcal{T}_D$ consta de las listas [10] (posiblemente infinitas o parciales) de números reales.

El sencillo programa siguiente ilustra las posibilidades del lenguaje obtenido.

$$
\begin{array}{l}
numfib = genfib(1,2). \\
genfib(M,N) = [M \mid genfib(N, M+N)]. \\
fib(X) = pertenece(X, numfib). \\
pertenece(X,[Y|Ys]) = false \Leftarrow X < Y. \\
pertenece(X,[Y|Ys]) = true \Leftarrow X == Y. \\
pertenece(X,[Y|Ys]) = pertenece(X,Ys) \Leftarrow X > Y.
\end{array}
$$

La función $fib/1$ reconoce si un número real X está o no en la sucesión de Fibonacci, representada como una lista infinita computada por la función constante $numfib/0$ (ésta a su vez se apoya en $genfib/2$, que es la que realmente construye la lista infinita de los números de Fibonacci a partir de dos semillas $M = 1, N = 2$). La pertenencia de un número a la lista infinita se reconoce mediante $pertenece/2$, que aprovecha para ello la monotonía de la lista generada por $numfib$. Nótese que tanto $genfib$ como $pertenece$ hacen uso de las operaciones y relaciones predefinidas sobre los números reales. La ventaja de tal posibilidad frente a la clásica representación de los números (naturales) como términos (mediante $cero/0$ y $suc/1$) nos parece obvia.
◇

[10] Pseudolistas más bien, si no se consideran géneros, ya que entonces [_ | _] se comporta como una constructora de pares; p. ej. [1 | 1] estaría en el dominio.

3 Semántica declarativa de $CFLP$-programas

3.1 Interpretaciones y modelos

Como ya sabemos, en cada instancia del esquema $CFLP(X)$ se asume la existencia de una signatura Σ (de símbolos primitivos) y una estructura con restricciones sobre Σ, ambas totalmente *prefijadas*, lo que en particular incluye a la interpretación de los símbolos de función y predicado primitivos. Lo que definimos a continuación es lo que entendemos por *interpretación* de los símbolos de función no primitiva $f \in \Delta$.

Definición 3.1 *(Interpretaciones) Una* interpretación *I asigna a cada símbolo de función $f \in \Delta^n$ una función continua $f^I : D^n \longrightarrow D$. El conjunto $\mathcal{INT}$ de las interpretaciones puede ser dotado del orden habitual*

$$I_1 \sqsubseteq_{\mathcal{INT}} I_2 \Leftrightarrow f^{I_1}(\overline{x}) \sqsubseteq_D f^{I_2}(\overline{x}) \text{ para todo } f \in \Delta^n, \overline{x} \in D^n$$

Con este orden $\mathcal{INT}$ es un dominio de Scott, cuyos elementos finitos pueden caracterizarse del siguiente modo: $I \in \mathcal{INT}$ es finita si y solo si

- $f^I(\overline{x})$ es finito (en D), para toda $f \in \Delta^n, \overline{x} \in D^n$,
- $f^I(\overline{x}) \neq \bot_D$ sólo para un conjunto finito de tuplas $(f, \overline{x}), f \in \Delta^n, \overline{x} \in D^n$.

Dada $I \in \mathcal{INT}$, toda valoración α puede extenderse a los conjuntos $\mathcal{T}_{F \cup \Delta}(V)$ y $\mathcal{B}_{\Pi, F \cup \Delta}(V)$. Escribimos $\alpha[\![e]\!]_I$ para indicar el valor de e bajo α e I. La noción de satisfactibilidad también puede ser extendida al caso de restricciones no primitivas $\varphi \in Con_{\Sigma \cup \Delta}$, y escribimos $\alpha \models^I \varphi$ para indicar que α satisface φ bajo la interpretación I.

Utilizaremos repetidas veces, a veces sin mencionarla, la siguiente propiedad.

Proposición 3.1 *(Continuidad de la evaluación)*
Para toda expresión $e \in \mathcal{T}_{F \cup \Delta}(V)$ $(\in Con_{\Sigma \cup \Delta}$ resp.), la función

$$ev_e : \mathcal{VAL} \times \mathcal{INT} \longrightarrow D \quad (\{true, \bot_{bool}\} \text{ resp.})$$

definida por $ev_e(\alpha, I) = \alpha[\![e]\!]_I$ es continua

Demostración:
Se obtiene sin dificultad por inducción sobre la estructura de e (véase, p. ej., [114]). □

La siguiente definición establece nuestra noción de *modelo* de un programa.

Definición 3.2 *(Modelos)*

- *Una interpretación I es* modelo de una regla $f(\overline{X}) = e \Leftarrow \varphi$ *si y solo si $f^I(\alpha(\overline{X})) \sqsupseteq \alpha[\![e]\!]_I$ para toda $\alpha \in \mathcal{VAL}$ tal que $\alpha \models^I \varphi$.*

- *Una interpretación I es* modelo de un programa $\mathcal{P}$ *si lo es de todas sus reglas. Un programa es* consistente *si tiene algún modelo.*

Nótese, aunque resulte trivial, que los modelos son extensiones persistentes de la estructura de base, ya que no se permiten reglas para los símbolos primitivos, cuyas interpretaciones están totalmente predeterminadas. Otras nociones de persistencia han sido utilizadas en alguna ocasión [40] como base de lo que debe entenderse por definir una función sobre una estructura.

Nótese también que con esta definición las reglas de un programa están consideradas como 'inecuaciones con restricciones' más que como 'ecuaciones con restricciones'. Ello corresponde de modo natural con la lectura operacional pretendida para las reglas, que es la de 'reglas de reescritura con restricciones', y que se pondrá de manifiesto en la definición del operador de 'consecuencias inmediatas' asociado a un programa y en la formulación precisa de la semántica operacional. Un ejemplo que veremos posteriormente ilustrará claramente la repercusión que tiene, en la semántica declarativa de $CFLP$-programas, la definición de modelo que hemos dado.

La siguiente notación será utilizada con frecuencia

Notación 3.1

Dadas $I \in \mathcal{INT}, f \in \Delta^n, \overline{x} \in D^n$, escribimos

$$C(I, f, \overline{x}) \equiv \{\alpha[\![e]\!]_I \mid f(\overline{X}) = e \Leftarrow \varphi \in \mathcal{P}, \alpha(\overline{X}) = \overline{x}, \alpha \models^I \varphi\}$$

La siguiente casi obvia proposición utiliza los conjuntos recién introducidos para caracterizar los modelos de un programa. Aprovechamos también la proposición para dar una definición adicional.

Proposición 3.2

$I \in \mathcal{INT}$ es un modelo de $\mathcal{P}$ si y solo si para cada $f \in \Delta^n, \overline{x} \in D^n$, existe $s = \sqcup C(I, f, \overline{x})$ y verifica $s \sqsubseteq f^I(\overline{x})$. Si $\sqcup C(I, f, \overline{x}) = f^I(\overline{x})$ para cada $f \in \Delta^n, \overline{x} \in D^n$, se dice que I es un modelo exacto *de $\mathcal{P}$.*

Demostración:
$\Rightarrow$) Puesto que I es un modelo de $\mathcal{P}$ se tiene $f^I(\overline{x}) \sqsupseteq z$ para todo $z \in C(I, f, \overline{x})$. Así pues $C(I, f, \overline{x})$ está acotado por $f^I(\overline{x})$, y por ser D un dominio de Scott, tiene supremo s, que obviamente será $\sqsubseteq f^I(\overline{x})$.

$\Leftarrow$) Es aún mas obvia. □

3.2 Existencia de modelo mínimo

La definición de programa no garantiza que todo programa sea consistente. Probaremos en este apartado que todo programa consistente tiene modelo mínimo. Daremos dos demostraciones alternativas de este hecho. La primera (más sencilla) simplemente prueba que el ínfimo

de los modelos de un programa es también modelo. La segunda proporciona más información, pues caracteriza al modelo mínimo como el menor punto fijo de un operador de 'consecuencias inmediatas' asociado al programa. Esta caracterización será clave para los resultados de completitud de la semántica operacional que se obtendrán en la sección 4.3.

Teorema 3.1 *(Existencia de modelo mínimo, versión 1)*

Sea $\mathcal{P}$ un programa consistente y $\mathcal{M}_\mathcal{P} = \{I \in \mathcal{INT} \mid I \text{ es modelo de } \mathcal{P}\}$. Entonces $I_\mathcal{P} \equiv \sqcap\mathcal{M}_\mathcal{P}$ es el modelo mínimo de $\mathcal{P}$.

Demostración:
En primer lugar, observemos que el ínfimo $\sqcap\mathcal{M}_\mathcal{P}$ existe en $\mathcal{INT}$, pues $\mathcal{M}_\mathcal{P}$ es no vacío por hipótesis [11].

Debemos probar que $I_\mathcal{P}$ es modelo de $\mathcal{P}$. Sea entonces $f(\overline{X}) = e \Leftarrow \varphi$ una regla de $\mathcal{P}$, y $\alpha \models^{I_\mathcal{P}} \varphi$. Para cualquier $I \in \mathcal{M}_\mathcal{P}$ se tendrá también $\alpha \models^{I} \varphi$, y por tanto

$$f^I(\alpha(\overline{X})) \sqsupseteq \alpha[\![e]\!]_I \sqsupseteq \alpha[\![e]\!]_{I_\mathcal{P}}$$

Como esto sucede para toda $I \in \mathcal{M}_\mathcal{P}$, tenemos finalmente $f^{I_\mathcal{P}}(\alpha(\overline{X})) \sqsupseteq \alpha[\![e]\!]_{I_\mathcal{P}}$, o sea, $I_\mathcal{P}$ es modelo de la regla. □

Introducimos ahora el operador de 'consecuencias inmediatas' asociado a un programa.

Definición 3.3 *(Operador $T_\mathcal{P}$)*

Dado un programa $\mathcal{P}$, definimos (parcialmente sobre $\mathcal{INT}$) el operador $T_\mathcal{P} : \mathcal{INT} \dashrightarrow \mathcal{INT}$ como sigue: dadas $I \in \mathcal{INT}, f \in \Delta^n, \overline{x} \in D^n$,

$$f^{T_\mathcal{P}(I)}(\overline{x}) = \sqcup C(I, f, \overline{x})$$

Debe entenderse que $T_\mathcal{P}(I)$ está definido si y solo si el supremo anterior existe para todos $f \in \Delta^n$, $\overline{x} \in D^n$.

Nótese que $T_\mathcal{P}(I)$ puede en efecto no estar definido para algunas interpretaciones. No vale la pena definir en esos casos $T_\mathcal{P}(I) = \bot_{\mathcal{INT}}$ pues a pesar de todo $T_\mathcal{P}$ podría no ser siquiera monótono, como ilustra el ejemplo que sigue a continuación. En consecuencia, no podemos sacar partido de los resultados clásicos de existencia de mínimo punto fijo para operadores continuos (veáse, p. ej., [5] para un uso típico de tales técnicas).

Ejemplo 3.1

El que $T_\mathcal{P}$ sea parcial puede ocurrir incluso con programas consistentes. Consideremos el programa $\mathcal{P}$ en $CFLP(H)$ dado por las dos reglas (incondicionales)

$$f(X) = s(g(X)).$$
$$f(X) = s(h(X)).$$

[11] En un dominio de Scott – en el que todo conjunto no vacío acotado (superiormente) tiene supremo – existe también el ínfimo de cualquier conjunto no vacío A. En efecto, el ínfimo no es sino el supremo del conjunto de las cotas inferiores de A, que es no vacío ($\bot$ es cota inferior) y está obviamente acotado superiormente.

El programa $\mathcal{P}$ es incluso consistente. Al no haber reglas para g, h, el modelo mínimo $I_\mathcal{P}$ viene dado por

$$g^{I_\mathcal{P}}(x) = h^{I_\mathcal{P}}(x) = \bot, \quad f^{I_\mathcal{P}}(x) = s(\bot), \quad \forall x$$

Es fácil verificar que $T_\mathcal{P}(I_\mathcal{P})$ está definido y vale $T_\mathcal{P}(I_\mathcal{P}) = I_\mathcal{P}$ (esto es además consecuencia de los resultados que veremos a continuación, por los que $I_\mathcal{P}$ es el menor punto punto fijo de $T_\mathcal{P}$).

Sin embargo, existen interpretaciones I más definidas que $I_\mathcal{P}$ – es decir, $I_\mathcal{P} \sqsubseteq_{INT} I$ – para las que $T_\mathcal{P}$ no está definida. Un ejemplo de ello es I dada por

$$f^I(x) = s(0), \quad g^I(x) = 0, \quad h^I(x) = s(0), \quad \forall x$$

que verifica $I_\mathcal{P} \sqsubseteq_{INT} I$. Para cualquier x que consideremos se tiene

$$C(I, f, x) = \{s(0), s(s(0))\}$$

por lo que $C(I, f, x)$ no tiene supremo, y por tanto $T_\mathcal{P}(I)$ no está definido. Es claro además que si convenimos $T_\mathcal{P}(I) = \bot_{INT}$ el operador $T_\mathcal{P}$ no sería monótono, pues $T_\mathcal{P}(I_\mathcal{P}) \sqsupset \bot_{INT}$.
$\diamond$

La definición del operador $T_\mathcal{P}$ requiere cierta justificación, pues para que $T_\mathcal{P}(I)$ sea una interpretación no basta con que $f^{T_\mathcal{P}(I)}$ esté definida para toda $f \in \Delta$, sino que se requiere su continuidad, que no se deduce trivialmente de la definición anterior.

Proposición 3.3

Dadas $I \in \mathcal{INT}, f \in \Delta$, si $T_\mathcal{P}(I)$ está definido, entonces $f^{T_\mathcal{P}(I)}$ es continua.

Demostración:
Basta probar

(i) $f^{T_\mathcal{P}(I)}$ es monótona

(ii) $f^{T_\mathcal{P}(I)}(\bigsqcup_{\overline{a} \in \overline{A}} \overline{a}) \sqsubseteq \bigsqcup_{\overline{a} \in \overline{A}} f^{T_\mathcal{P}(I)}(\overline{a})$

Para probar *(i)*, sean $\overline{a} \sqsubseteq \overline{b}$ y sea $\alpha_0[\![e]\!]_I \in C(I, f, \overline{a})$ con $f(\overline{X}) = e \Leftarrow \varphi \in \mathcal{P}$, $\alpha_0(\overline{X}) = \overline{a}$, y $\alpha_0 \models^I \varphi$. Consideremos α_1 tal que $\alpha_1(\overline{X}) = \overline{b}$ y α_1 coincide con α_0 en el resto de variables, con lo que $\alpha_0 \sqsubseteq_{\mathcal{VAL}} \alpha_1$. Por continuidad de la evaluación tendremos $\alpha_1 \models^I \varphi$ (por lo que $\alpha_1[\![e]\!]_I \in C(I, f, \overline{b})$) y además $\alpha_0[\![e]\!]_I \sqsubseteq \alpha_1[\![e]\!]_I$. Así pues, todo elemento de $C(I, f, \overline{a})$ está acotado por otro de $C(I, f, \overline{b})$, con lo que finalmente

$$f^{T_\mathcal{P}(I)}(\overline{a}) = \sqcup C(I, f, \overline{a}) \sqsubseteq \sqcup C(I, f, \overline{b}) = f^{T_\mathcal{P}(I)}(\overline{b})$$

lo que concluye (i).

Para ver *(ii)*, sea $\alpha[\![e]\!]_I \in C(I, f, \bigsqcup_{\overline{a} \in \overline{A}} \overline{a})$ con $f(\overline{X}) = e \Leftarrow \varphi \in \mathcal{P}$, $\alpha(\overline{X}) = \bigsqcup_{\overline{a} \in \overline{A}} \overline{a}$, y $\alpha \models^I \varphi$. Para cada $\overline{a} \in \overline{A}$, consideremos $\alpha_{\overline{a}}$ tal que $\alpha_{\overline{a}}(\overline{X}) = \overline{a}$ y $\alpha_{\overline{a}}$ coincide con α en el resto de variables, con lo que $\alpha = \bigsqcup_{\overline{a} \in \overline{A}} \alpha_{\overline{a}}$. Por continuidad de la evaluación y puesto que $\alpha \models^I \varphi$, existe $\overline{a_0}$ tal que $\alpha_{\overline{a}} \models^I \varphi$ para toda $\overline{a} \sqsupseteq \overline{a_0}$, y por tanto $\alpha_{\overline{a}}[\![e]\!]_I \in C(I, f, \overline{a})$ para

toda $\overline{a} \sqsupseteq \overline{a_0}$. Así pues, por la definición del operador $T_{\mathcal{P}}$, para cualquier $\overline{a} \sqsupseteq \overline{a_0}$ tenemos $\alpha_{\overline{a}}[\![e]\!]_I \sqsubseteq f^{T_{\mathcal{P}}(I)}(\overline{a})$, y en consecuencia

$$\alpha[\![e]\!]_I = \bigsqcup_{\overline{a} \in \overline{A}} \alpha_{\overline{a}}[\![e]\!]_I \sqsubseteq \bigsqcup_{\overline{a} \in \overline{A}} f^{T_{\mathcal{P}}(I)}(\overline{a})$$

Como esto sucede para todo $\alpha[\![e]\!]_I \in C(I, f, \bigsqcup_{\overline{a} \in \overline{A}} \overline{a})$, tenemos que en efecto

$$f^{T_{\mathcal{P}}(I)}(\bigsqcup_{\overline{a} \in \overline{A}} \overline{a}) \sqsubseteq \sqcup C(I, f, \bigsqcup_{\overline{a} \in \overline{A}} \overline{a}) \sqsubseteq \bigsqcup_{\overline{a} \in \overline{A}} f^{T_{\mathcal{P}}(I)}(\overline{a})$$

lo que concluye (ii). □

Los dos siguientes lemas establecen propiedades del operador $T_{\mathcal{P}}$. El primero de ellos afirma que $T_{\mathcal{P}}$ es monótono en cierto sentido, y el segundo caracteriza los modelos de un programa $\mathcal{P}$ en términos del operador $T_{\mathcal{P}}$.

Lema 3.1

$T_{\mathcal{P}}$ es monótono en el siguiente sentido: si $I_1 \sqsubseteq_{\mathcal{INT}} I_2$ y $T_{\mathcal{P}}(I_2)$ está definido, entonces $T_{\mathcal{P}}(I_1)$ también lo está y se tiene $T_{\mathcal{P}}(I_1) \sqsubseteq_{\mathcal{INT}} T_{\mathcal{P}}(I_2)$.

Demostración:
Es fácil verificar que, para cada $f \in \Delta^n, \overline{x} \in D^n$, todo elemento de $C(I_1, f, \overline{x})$ está acotado por otro de $C(I_2, f, \overline{x})$. Por lo tanto $C(I_1, f, \overline{x})$ está acotado (y por tanto tiene supremo) por $\sqcup C(I_2, f, \overline{x})$ (que existe pues $T_{\mathcal{P}}(I_2)$ está definido). Es más, se tiene

$$f^{T_{\mathcal{P}}(I_1)}(\overline{x}) = \sqcup C(I_1, f, \overline{x}) \sqsubseteq \sqcup C(I_2, f, \overline{x}) = f^{T_{\mathcal{P}}(I_2)}(\overline{x})$$

□

Hablaremos de 'la monotonía de $T_{\mathcal{P}}$', a pesar de su sentido restringido.

Lema 3.2

$I \in \mathcal{INT}$ es modelo de un programa $\mathcal{P}$ si y solo si $T_{\mathcal{P}}(I)$ está definido y $T_{\mathcal{P}}(I) \sqsubseteq I$. En este caso $T_{\mathcal{P}}(I)$ es también modelo de $\mathcal{P}$.

Demostración:
La primera parte del lema no es sino una reformulación de la proposición 3.2. Para la segunda, obsérvese que $T_{\mathcal{P}}(I) \sqsubseteq I$ implica, por la monotonía de $T_{\mathcal{P}}$, que $T_{\mathcal{P}}(T_{\mathcal{P}}(I))$ está también definido y $T_{\mathcal{P}}(T_{\mathcal{P}}(I)) \sqsubseteq T_{\mathcal{P}}(I)$. Por la primera parte del lema, $T_{\mathcal{P}}(I)$ es un modelo de $\mathcal{P}$. □

Utilizaremos la siguiente notación estándar para las sucesivas iteraciones del operador $T_{\mathcal{P}}$.

Notación 3.2

$$T_{\mathcal{P}} \uparrow 0 \equiv \bot_{\mathcal{INT}}, T_{\mathcal{P}} \uparrow (k+1) \equiv T_{\mathcal{P}}(T_{\mathcal{P}} \uparrow k), T_{\mathcal{P}} \uparrow \omega \equiv \bigsqcup_k T_{\mathcal{P}} \uparrow k$$

en caso de que los lados derechos estén definidos.

El resultado más importante de esta sección es el siguiente

Teorema 3.2 *(Existencia de modelo mínimo, versión 2)*

Un programa $\mathcal{P}$ es consistente si y solo si $T_\mathcal{P} \uparrow \omega$ existe. En este caso, además, $I_\mathcal{P} \equiv T_\mathcal{P} \uparrow \omega$ es el menor modelo de $\mathcal{P}$, el menor punto fijo de $T_\mathcal{P}$ y es un modelo exacto.

Demostración:
$\Rightarrow$) Sea I un modelo cualquiera de $\mathcal{P}$. Probaremos en primer lugar por inducción lo siguiente: Para todo $k \geq 0$, $T_\mathcal{P} \uparrow k$ está definido y verifica $T_\mathcal{P} \uparrow k \sqsubseteq I$. El caso $k = 0$ es trivial. Supongamos ahora que $T_\mathcal{P} \uparrow k \sqsubseteq I$. Puesto que $T_\mathcal{P}(I)$ está definido y $T_\mathcal{P}(I) \sqsubseteq I$ por el lema 3.2, concluimos por monotonía de $T_\mathcal{P}$ que $T_\mathcal{P} \uparrow k+1 = T_\mathcal{P}(T_\mathcal{P} \uparrow k)$ está también definido y verifica $T_\mathcal{P} \uparrow k+1 \sqsubseteq T_\mathcal{P}(I) \sqsubseteq I$, lo que completa la inducción.

Usando una vez más la monotonía de $T_\mathcal{P}$ es muy fácil probar que las iteraciones $T_\mathcal{P} \uparrow k$ forman una cadena acotada, como acabamos de ver, por I. Así pues, existe

$$I_\mathcal{P} = T_\mathcal{P} \uparrow \omega = \bigsqcup_k T_\mathcal{P} \uparrow k$$

que además verifica $I_\mathcal{P} \sqsubseteq I$.

$\Leftarrow$) Probaremos que $\mathcal{P}$ es consistente probando que precisamente un modelo de $\mathcal{P}$ viene dado por $I_\mathcal{P} = T_\mathcal{P} \uparrow \omega$. Sea $f(\overline{X}) = e \Leftarrow \varphi$ una regla de $\mathcal{P}$, y sea $\alpha \models^{I_\mathcal{P}} \varphi$. Por continuidad de la evaluación, existe k_0 tal que $\alpha \models^{T_\mathcal{P} \uparrow k} \varphi$ para todo $k \geq k_0$, y por la definición de $T_\mathcal{P}$ se tiene $\alpha[\![e]\!]_{T_\mathcal{P} \uparrow k} \sqsubseteq f^{T_\mathcal{P} \uparrow k+1}(\alpha(\overline{X})) \sqsubseteq f^{I_\mathcal{P}}(\alpha(\overline{X}))$. En consecuencia, y de nuevo por continuidad de la evaluación, se tiene

$$\alpha[\![e]\!]_{I_\mathcal{P}} \sqsubseteq \bigsqcup_{k \geq k_0} \alpha[\![e]\!]_{T_\mathcal{P} \uparrow k} \sqsubseteq f^{I_\mathcal{P}}(\alpha(\overline{X}))$$

esto es, $I_\mathcal{P}$ es modelo de la regla.

Que $I_\mathcal{P}$ es es el modelo mínimo de $\mathcal{P}$ es claro pues hemos probado que $I_\mathcal{P} \sqsubseteq I$ para todo modelo I de $\mathcal{P}$. $I_\mathcal{P}$ es también un punto fijo de $T_\mathcal{P}$ ya que el lema 3.2 garantiza que $T_\mathcal{P}(I_\mathcal{P}) \sqsubseteq I_\mathcal{P}$ y que $T_\mathcal{P}(I_\mathcal{P})$ es modelo de $\mathcal{P}$, por lo que también $I_\mathcal{P} \sqsubseteq T_\mathcal{P}(I_\mathcal{P})$. Pero entonces $I_\mathcal{P}$ es el menor punto fijo ya que el lema 3.2 garantiza también que todo punto fijo de $T_\mathcal{P}$ es modelo de $\mathcal{P}$.

Finalmente $f^{I_\mathcal{P}}(\overline{x}) = f^{T_\mathcal{P}(I_\mathcal{P})}(\overline{x}) = \sqcup C(I_\mathcal{P}, f, \overline{x})$, por lo que $I_\mathcal{P}$ es modelo exacto de $\mathcal{P}$. $\square$

3.3 Indecidibilidad de la consistencia de un programa

En seguida veremos que la consistencia de un programa es una propiedad, en general, indecidible [12]. A pesar de ello, el haber probado la existencia de modelo mínimo para programas consistentes tiene mucho interés, pues deja la posibilidad – en cada lenguaje diseñado como instancia del esquema – de fijar si se desea condiciones más restrictivas a los programas que garanticen su consistencia, sin peligro de que se pierda la existencia de modelo mínimo.

[12] Para estructuras muy simples puede no ser así.

Veamos un ejemplo, ciertamente artificial, de una instancia del esquema en la que es indecidible la consistencia de un programa.

Ejemplo 3.2 *(Indecidibilidad de la consistencia de un programa)*

Consideremos la signatura $\Sigma = F \cup \Pi$ con

$$F^0 = \{\mathbf{n} : n \in \mathbb{N}\} \quad \Pi^1 = \{\mathbf{para}\}$$

y la Σ-estructura $\mathcal{R}$ cuyo dominio es $\mathbb{N}_\perp$ y en la que interpretamos $\mathbf{n}$ como el número natural n y

$$\mathbf{para} : \mathbb{N}_\perp \to \{\text{true}, \perp_{\text{bool}}\}$$

está definida por

$$\mathbf{para}(x) = \begin{cases} \text{true} & \text{si } x \in \mathbb{N} \text{ y la máquina de Turing de} \\ & \text{código } x \text{ con entrada 0 para} \\ \perp_{\text{bool}} & \text{en otro caso} \end{cases}$$

Nótese que **para** es obviamente continua (y representa la función característica de un predicado recursivamente enumerable no recursivo).

Ahora, para cada número natural n consideramos el programa $Para_n$ que define la función de aridad 0 $para_n$ mediante las reglas

$$\begin{array}{lcll} para_n & = & 1 & \Leftarrow \mathbf{para}(\mathbf{n}) \\ para_n & = & 0 & \Leftarrow \mathbf{para}(\mathbf{n}) \end{array}$$

Se tiene que $Para_n$ es consistente si y solo si $\mathbf{para}(\mathbf{n}) = \perp$, es decir, si la máquina de Turing de código n con entrada 0 no para (en cuyo caso un modelo del programa es $\perp_{\mathcal{INT}}$). Decidir la consistencia de programas nos permitiría decidir el problema de parada.

Si bien la estructura elegida es muy artificial, parece claro que podemos exhibir ejemplos similares para otras más realistas en las que podamos representar los números naturales y simular máquinas de Turing. Lógicamente los detalles serían mucho más complicados que los del ejemplo presentado y no los vamos a abordar aquí. Desarrollos técnicos de ese estilo pueden encontrarse, por ejemplo, en [4]; véase también [5].
◇

La condición que introducimos a continuación permite garantizar la consistencia de un programa.

Definición 3.4 *(Programas no ambiguos)*

Un programa $\mathcal{P}$ se dice no ambiguo *si para cada par de variantes de reglas de $\mathcal{P}$ para un mismo símbolo $f \in \Delta$,*

$$f(\overline{X}) = e \Leftarrow \varphi \quad y \quad f(\overline{X}) = e' \Leftarrow \varphi'$$

tales que las variables que no aparezcan en la cabeza estén renombradas aparte, se verifica la siguiente condición:

$$\alpha[\![e]\!]_I = \alpha[\![e']\!]_I, \text{ para toda } I \in \mathcal{INT}, \alpha \in \mathcal{VAL} \text{ con } \alpha \models^I \varphi, \alpha \models^I \varphi'$$

El par de variantes puede ser incluso de la misma regla para f.

Esta condición implica (es equivalente, de hecho) que los conjuntos $C(I, f, \overline{x})$ son vacíos o unitarios, por lo que obviamente tienen supremo. Como consecuencia el operador $T_{\mathcal{P}}$ está totalmente definido, $T_{\mathcal{P}} \uparrow \omega$ existe y el programa en cuestión es consistente.

El siguiente ejemplo muestra que hay programas ambiguos que sin embargo son consistentes.

Ejemplo 3.3

Consideremos la signatura $\Sigma = F \cup \Pi$ con

$$F^0 = \{0, 1\} \quad F^2 = \{+, -\} \quad \Pi^2 = \{=\}$$

y la Σ-estructura $\mathcal{R}$ cuyo dominio es $\mathcal{Z}_\perp$ y en la que interpretamos los símbolos de Σ de manera natural.

Sea P el programa que define la función f mediante las reglas

$$\begin{array}{lcrcl} f(X) & = & 1 & \Leftarrow & X = 0. \\ f(X) & = & f(X-1) & \Leftarrow & true. \end{array}$$

El programa P es consistente (un modelo viene dado por I con $f^I(z) = 1, \forall z \in \mathcal{Z}_\perp$), pero sin embargo es ambiguo, pues para $J \in \mathcal{INT}$, $\alpha \in \mathcal{VAL}$ dadas por

$$f^J(z) = \begin{cases} 1 & \text{si } z = 0 \\ 0 & \text{si } z \in \mathcal{Z}, z \neq 0 \\ \perp & \text{si } z = \perp \end{cases} \qquad \alpha(X) = 0 \ \text{ para toda } X \in V$$

se tiene que α satisface las restricciones de ambas reglas, pero

$$\alpha[\![1]\!]_J = 1 \quad \alpha[\![f(X-1)]\!]_J = 0$$

$\diamond$

Debemos observar que la no ambigüedad es aún una condición indecidible, como prueba el ejemplo 3.4 más abajo. Sin embargo, es una condición más sencilla que la mera consistencia, susceptible de ser reemplazada por condiciones (dependientes ya de la instancia del esquema) más fuertes pero decidibles de no ambigüedad, como las propuestas en [117, 58, 63, 61] para algunos lenguajes lógico-funcionales. En algunos casos (véase el ejemplo 3.5 más abajo) la no ambigüedad de los programas se verifica de manera trivial.

Ejemplo 3.4 *(Indecidibilidad de la ambigüedad de un programa)*

Observemos simplemente que con la misma signatura, estructura y programa del ejemplo 3.2 se verifica

$$Para_n \text{ es consistente} \Leftrightarrow Para_n \text{ es no ambiguo}$$

$\diamond$

Ejemplo 3.5 *(Programas lógicos)*

Si consideramos $CFLP$-programas 'lógicos' en los que solamente se definan predicados mediante 'cláusulas' de la forma

$$f(\overline{X}) = true \Leftarrow \varphi$$

la condición de no ambigüedad resulta obvia, y por tanto los programas son consistentes. En este caso el teorema 3.2 queda particularizado a los teoremas análogos de existencia de modelo mínimo como mínimo punto fijo en CLP [78, 79, 73], que son a su vez generalizaciones del teorema clásico para LP [50].
◇

3.4 La noción de modelo. Discusión

Terminamos esta sección con un ejemplo, prometido en su momento, que ponga de manifiesto la naturaleza 'inecuacional' que a las $CFLP$-reglas impone la noción de modelo que hemos dado.

Ejemplo 3.6

Consideremos la misma signatura, estructura y programa del ejemplo 3.3. Recordemos que la función f viene definida por las reglas

$$\begin{array}{lcrcl} f(X) & = & 1 & \Leftarrow & X = 0. \\ f(X) & = & f(X-1) & \Leftarrow & true. \end{array}$$

Las sucesivas aproximaciones $T_{\mathcal{P}} \uparrow k$ al modelo mínimo $I_{\mathcal{P}} = T_{\mathcal{P}} \uparrow \omega$ vienen dadas por

$$f^{T_{\mathcal{P}} \uparrow k}(z) = \begin{cases} 1 & \text{si } 0 \leq z < k \\ \bot & \text{en otro caso} \end{cases}$$

y el modelo mínimo por

$$f^{I_{\mathcal{P}}}(z) = \begin{cases} 1 & \text{si } 0 \leq z \\ \bot & \text{en otro caso} \end{cases}$$

Obsérvese que las dos reglas de f son aplicables a $z = 0$, para el que las denotaciones de los lados derechos de ambas son, para $I_{\mathcal{P}}$, $f_{I_{\mathcal{P}}}(1) = 1$ y $f_{I_{\mathcal{P}}}(0-1) = \bot$.

Podríamos adoptar una noción alternativa de modelo en la que cada regla tuviera que ser satisfecha como ecuación condicional (es decir, reemplazando en la definición de modelo la condición $f^I(\alpha(\overline{X})) \sqsupseteq \alpha[\![e]\!]_I$ por la más restrictiva $f^I(\alpha(\overline{X})) = \alpha[\![e]\!]_I$). También en este caso el programa anterior tendría modelo mínimo $I_{\mathcal{P}}{}'$, que vendría dado por

$$f^{I_{\mathcal{P}}{}'}(z) = \begin{cases} 1 & \text{si } z \in \mathcal{Z} \\ \bot & \text{en otro caso} \end{cases}$$

y no coincidiría por tanto con $T_{\mathcal{P}} \uparrow \omega$.

La diferencia entre ambas situaciones queda más clara en este ejemplo al analizar las lecturas respectivas que deben darse a la segunda regla para f, cuyas instancias para sucesivos valores $X = 0, -1, -2, \ldots$ son

$$f(0) = f(-1) \quad f(-1) = f(-2) \quad f(-2) = f(-3) \quad \ldots$$

Con nuestra noción de modelo estas reglas deben leerse como

$$f(0) \sqsupseteq f(-1) \quad f(-1) \sqsupseteq f(-2) \quad f(-2) \sqsupseteq f(-3) \quad \ldots$$

y nada obliga a que $f(-1), f(-2), \ldots$ estén más definidas que $\bot$.

Por el contrario, con la visión ecuacional tendríamos realmente igualdades, y el valor $f(0) = 1$ proporcionado por la primera regla de f se transmitiría a $f(-1), f(-2), \ldots$

$\diamond$

Parece razonable que nuestra semántica declarativa se corresponda con un uso direccional (de izquierda a derecha) de las reglas de un programa, es decir, con una lectura operacional de las mismas como reglas de reescritura.

Hay que indicar de todos modos que la correspondencia no es absoluta, en el sentido de que nuestra noción de modelo asigna significado declarativo (modelo mínimo) a todo programa consistente, mientras que, como veremos en la próxima sección, no es de esperar que la reescritura sea completa para todos los programas consistentes. Nosotros probaremos la completitud para una clase de programas, que llamaremos *deterministas*, que incluye a los programas no ambiguos.

4 Semántica operacional de $CFLP$-programas

En esta sección presentamos la semántica operacional de los $CFLP$-programas. Como ya hemos anticipado varias veces, las reglas de un programa están concebidas para ser usadas como reglas de reescritura, y en eso van a consistir en definitiva los cómputos: en reescribir expresiones no primitivas $f(e_1, \ldots, e_n)$ (f no primitiva) mediante las reglas que definen tales funciones. Pero primeramente debemos decir *para qué* queremos hacer cómputos.

4.1 Objetivos y soluciones

Como en muchos otros lenguajes con capacidades lógicas, *ejecutar* un $CFLP$-programa es sinónimo de *resolver un objetivo*, donde un objetivo expresa una condición acerca de unas *variables*. Es decir,

Definición 4.1 *(Objetivos)*
Un objetivo *es una restricción* $\varphi \in Con_{\Sigma \cup \Delta}$.

Resolver un objetivo es determinar valores de las variables (relevantes) del objetivo que satisfagan la condición. Esos valores son las soluciones del objetivo. Las variables relevantes son, naturalmente, las variables libres del objetivo o variables *globales*. No esperamos de los cómputos que produzcan las soluciones concretas una a una, sino más bien que generen como respuestas otras condiciones que podamos considerar 'simples', cuyas soluciones sean (parte de las) soluciones del objetivo inicial. Así la condición final determina implícitamente un conjunto de soluciones. Este es un punto de vista habitual en lenguajes 'lógicos'. Por ejemplo, una respuesta de un cómputo Prolog podría ser la sustitución $\{X/[a \mid L]\}$, que expresa una condición sobre X — la condición $\exists L(X = [a \mid L])$ — verificada por infinitos valores concretos, p. ej., $X = [a]$, $X = [a, b, c]$, etc.

Nuestra siguiente definición establece con precisión cuáles son nuestras nociones de 'solución' y 'condición simple'. Recordemos antes que, aunque en $CFLP(X)$ sólo consideramos predicados primitivos, los argumentos de los átomos de un objetivo φ pueden tener partes no primitivas, y por tanto el que se satisfaga o no la condición expresada en un objetivo depende de la interpretación de los símbolos no primitivos. Definimos pues

Definición 4.2 *(Soluciones)*

- *Una* solución *para un objetivo* φ *en una interpretación* I *es una valoración* α *tal que* $\alpha \models^I \varphi$.
- φ *está* semánticamente terminado *para* α *si* α *es solución de* φ *en toda interpretación* I.

Añadimos la palabra 'semánticamente' al hablar de un objetivo terminado φ porque en φ pueden aparecer subexpresiones no primitivas (y por tanto en un sentido 'sintáctico' la evaluación de la parte no primitiva de φ no está terminada) cuyo valor, sin embargo, es irrelevante para el hecho de que α sea solución de φ. Nos interesa a continuación dar dos

caracterizaciones equivalentes – casi inmediatas – del carácter semánticamente terminado de un objetivo. Nos hace falta previamente la noción de *cáscara* de una expresión o restricción, para expresar su parte primitiva 'externa'.

Definición 4.3 *(Cáscaras)*
La cáscara $|\ e\ |$ *de una expresión* $e \in \mathcal{T}_{F\cup\Delta}(V)$ *o átomo* $b \in \mathcal{B}_{\Pi,F}(V)$ *se define inductivamente como sigue:*

- $|\ X\ | = X$, *si* X *es una variable*
- $|\ c(e_1,\ldots,e_n)\ | = c(|\ e_1\ |,\ldots,|\ e_n\ |)$, *si* $c \in F$ *(*$c \in \Pi$ *en el caso de átomos)*
- $|\ f(e_1,\ldots,e_n)\ | = \bot$, *si* $f \in \Delta$.

Observemos que $|\ e\ |$ es una expresión primitiva para la signatura $\Sigma = F\cup\{\bot\}\cup\Pi$, donde $\bot$ es un nuevo símbolo de constante que, una vez interpretado como $\bot_D$, nos permite hablar de $\alpha[\![|\ e\ |]\!]$. Nótese además que

$$\alpha[\![|\ e\ |]\!] = \alpha[\![|\ e\ |]\!]_{\bot_{INT}}$$

La noción de cáscara se extiende de manera obvia a restricciones, y tendrá sentido entonces la notación $\alpha \models |\ \varphi\ |$, con significado equivalente a $\alpha \models^{\bot_{INT}} \varphi$.

De los anteriores comentarios, junto con la continuidad de la evaluación (Prop. 3.1), se deduce de modo inmediato el siguiente resultado.

Proposición 4.1
Dados un objetivo φ *y una valoración* α, *son equivalentes*

- φ *está semánticamente terminado para* α.
- α *es solución de* φ *en* $\bot_{INT}$.
- α *es solución de* $|\ \varphi\ |$.

El siguiente ejemplo muestra por qué la noción de objetivo semánticamente terminado está asociada a una determinada solución.

Ejemplo 4.1

Asumiendo las constructoras $0/0$, $s/1$ y $c/2$, consideramos, en la instancia $CFLP(\mathcal{H}_{\neq})$ (véase el apartado 2.3.3), el programa que define una función f mediante la única regla

$$f(X) = X$$

El objetivo

$$\varphi \equiv \quad c(X, f(X)) =\!\!\!/= c(s(0), 0)$$

admite, entre otras, las soluciones (para el modelo mínimo $I_{\mathcal{P}}$ que interpreta f como la identidad) dadas por $\alpha_1(X) = 0, \alpha_2(X) = s(0)$. φ está semánticamente terminado para α_1 pues se tiene

$$\alpha_1(|\ \varphi\ |) \equiv \alpha_1(c(X,\bot) =\!\!\!= c(s(0),0)) \equiv c(0,\bot) =\!\!\!= c(s(0),0) \equiv true$$

es decir, $\alpha_1 \models\!\!\!|\ \varphi\ |$.

No ocurre lo mismo con α_2, pues se tiene

$$\alpha_2(|\ \varphi\ |) \equiv \alpha_2(c(X,\bot) =\!\!\!= c(s(0),0)) \equiv c(s(0),\bot) =\!\!\!= c(s(0),0) \equiv \bot$$

Para reconocer que α_2 es solución de φ en $I_{\mathcal{P}}$ debemos evaluar $f(X)$ usando la regla de f para obtener el nuevo objetivo

$$\varphi' \equiv \quad c(X,X) =\!\!\!= c(s(0),0)$$

que ahora sí estaría semánticamente terminado para α_2.

Aunque sea bastante más evidente, aprovechemos también este ejemplo para mostrar que la noción de solución depende de la interpretación que estemos considerando; por ejemplo, la valoración α_2 dada arriba no es solución de φ con respecto a la interpretación (que no modelo) I en la que $f^I(x) = 0, \forall x$.

$\diamond$

En general, estaremos interesados en el caso de que α sea solución de φ en los modelos de $\mathcal{P}$. Podremos omitir entonces la referencia a I, y suponer, si queremos, que I es el modelo mínimo $I_{\mathcal{P}}$ del programa $\mathcal{P}$. Eso no supone pérdida de generalidad, pues se tiene

Proposición 4.2

α es solución de φ en el modelo mínimo $I_{\mathcal{P}}$ si y solo si lo es en todo modelo I de $\mathcal{P}$.

Demostración:
Para la implicación no trivial, basta observar que si $\alpha \models^{I_{\mathcal{P}}} \varphi$ e I es modelo de $\mathcal{P}$ (y por tanto $I \sqsupseteq_{\mathcal{INT}} I_{\mathcal{P}}$), por continuidad de la evaluación se tiene también $\alpha \models^{I} \varphi$. $\square$

4.2 Estrechamiento por restricciones. Corrección

Necesitaremos en lo que sigue algunas notaciones y terminologías para hablar de subexpresiones y reemplazamientos. Usaremos, sin precisar más (ver, p. ej., [46]), la noción de *posición* en una expresión, y terminologías habituales, como 'posiciones más externas' verificando una cierta condición, prefijos de posiciones, etc. Asumimos también alguna forma bien definida de asignar posiciones en una restricción (considerando, p. ej., la coma ',' como un símbolo binario que asocia por la derecha).

Notación 4.1

Dada $e \in \mathcal{T}_{F \cup \Delta}(V)$,

- $Pos(e)$ *es el conjunto de posiciones de* e. *Análogamente para* $\varphi \in Con_{\Sigma \cup \Delta}$.
- *Si* $u \in Pos(e)$, $[e]_u$ *denota la subexpresión de* e *en la posición* u.
- $u \in Pos(e)$ *se dice* no primitiva *si* $[e]_u = f(e_1, \ldots, e_n)$, $f \in \Delta^n$. $NPos(e)$ *es el conjunto de posiciones no primitivas de* e.
- $u \in Pos(e)$ *se dice* crítica *si* u *es no primitiva y lo más externa posible.* $CPos(e)$ *es el conjunto de posiciones críticas de* e.
- *Si* $u \in Pos(e)$, $e[u \leftarrow r]$ *denota la expresión que resulta de reemplazar en* e *la subexpresión* $[e]_u$ *por* r.
- *Análogas definiciones para* $\varphi \in Con_{\Sigma \cup \Delta}$ *en lugar de* $e \in \mathcal{T}_{F \cup \Delta}(V)$.

El mecanismo de cómputo que proponemos inicialmente para resolver objetivos (será refinado posteriormente en la sección 4.4) consiste en la única regla

Definición 4.4 *(Estrechamiento por restricciones)*

- *La relación* $\leadsto$ *de* un paso de estrechamiento por restricciones *está definida como sigue:*

$$\varphi \leadsto_{u,R} \exists \overline{Y}(\varphi[u \leftarrow e\sigma], \psi\sigma)$$

si: *(i)* $[\varphi]_u = f(e_1, \ldots, e_n)$, *con* $f \in \Delta$
(ii) $f(X_1, \ldots, X_n) = e \Leftarrow \psi$ *es una variante de la regla* $R \in \mathcal{P}$ *(con variables* $\overline{Y}$ *que no aparezcan en* φ*)*
(iii) σ *es la sustitución* $\{X_i/e_i\}$

- *Un* cómputo *para un objetivo* φ_0 *es una secuencia, finita o infinita, de pasos de estrechamiento*

$$\varphi_0 \leadsto_{u_0,R_0} \varphi_1 \leadsto_{u_1,R_1} \varphi_2 \leadsto_{u_2,R_2} \cdots$$

de modo que, para cada i, *la variante de* R_i *es con variables que no estén en* $\varphi_0, \ldots, \varphi_i$.

- *Un cómputo finito* $\varphi_0 \leadsto_{u_0,R_0} \ldots \leadsto \varphi_n$ *está semánticamente terminado para* α *si* φ_n *lo está.*

Sintácticamente, la relación $\leadsto$ expresa solamente reescritura. La denominamos sin embargo 'estrechamiento' pues al efectuar $\varphi \leadsto \psi$ el conjunto de soluciones de ψ puede ser más reducido que el de φ, debido a las restricciones de la regla utilizada R, que han sido añadidas a φ.

Nótese que las variables (frescas) de la regla R con la que se estrechó se introducen cuantificadas existencialmente [13]. Compartimos la opinión, expresada por ejemplo en [87, 21],

[13] Según la definición anterior, entre las variables $\overline{Y}$ que se introducen existencialmente se encuentran las variables $X_1, \ldots, X_n$ de la variante de la regla $R \in \mathcal{P}$ utilizada. Puesto que, de nuevo por la definición de $\leadsto$, $X_1, \ldots, X_n$ van a desaparecer, al ser sustituidas por los parámetros correspondientes $e_1, \ldots, e_n$, su cuantificación resulta superflua, y puede ser omitida.

de que eso contribuye a aclarar el sentido lógico de un paso de cómputo. Como es típico en un sistema de resolución 'top-down', este sentido lógico es

Si $\varphi \leadsto \psi$, entonces $\psi \Rightarrow \varphi$

Este es, de hecho, el contenido del teorema de corrección que viene a continuación. Utilizaremos en su demostración el siguiente resultado, consecuencia sencilla del *lema de sustitución* estándar.

Lema 4.1

Si $\alpha(X_i) = \alpha[\![e_i]\!]_I$ *para* $i = 1, \ldots, n$, *entonces para toda expresión* e *se tiene* $\alpha[\![e]\!]_I = \alpha[\![e\sigma]\!]_I$, *donde* σ *es la sustitución* $\{X_i/e_i : i = 1, \ldots, n\}$. *Como consecuencia* $\alpha \models^I \varphi \Leftrightarrow \alpha \models^I \varphi\sigma$.

Teorema 4.1 *(Corrección del estrechamiento por restricciones)*

Si $\varphi \leadsto^* \psi$ *y* α *es solución de* ψ, *entonces* α *es también solución de* φ.

Demostración:
Obviamente basta probar el resultado para un solo paso

$$\varphi \leadsto_{u,R} \psi$$

siendo (i) $[\varphi]_u = f(e_1, \ldots, e_n)$, con $f \in \Delta$
(ii) $f(X_1, \ldots, X_n) = e \Leftarrow \varphi'$ es una variante (con variables $\overline{Y}$) de la regla R
(iii) σ es la sustitución $\{X_i/e_i\}$
(iv) $\psi \equiv \exists\overline{Y}(\varphi[u \leftarrow e\sigma],\ \varphi'\sigma)$

Sea I un modelo de $\mathcal{P}$ y $\alpha \models^I \psi$. Existe entonces α' (que coincide con α, excepto posiblemente sobre $\overline{Y}$) tal que

$$(1) \qquad \alpha' \models^I (\varphi[u \leftarrow e],\ \varphi')\sigma$$

Consideremos ahora la valoración α'' dada por

$$\alpha''(X) = \begin{cases} \alpha'[\![e_i]\!]_I & \text{si } X = X_i \\ \alpha'(X) & \text{en otro caso} \end{cases}$$

Puesto que en $(\varphi[u \leftarrow e],\ \varphi')\sigma$ no aparecen las variables X_i se tiene que también

$$(2) \qquad \alpha'' \models^I (\varphi[u \leftarrow e],\ \varphi')\sigma$$

Como además $\alpha''[\![e_i]\!]_I = \alpha'[\![e_i]\!]_I = \alpha''(X_i)$ tenemos, por (2) y el lema 4.1

$$(3) \qquad \alpha'' \models^I \varphi[u \leftarrow e],\ \varphi'$$

Ahora, como $\alpha'' \models^I \varphi'$ e I es modelo de la regla R se tiene $\alpha''[\![f(X_1, \ldots, X_n)]\!]_I \sqsupseteq \alpha''[\![e]\!]_I$, y de nuevo por el lema de sustitución $\alpha''[\![f(e_1, \ldots, e_n)]\!]_I \sqsupseteq \alpha''[\![e]\!]_I$, lo que junto con $\alpha'' \models^I \varphi[u \leftarrow e]$ garantiza que $\alpha'' \models^I \varphi$. Ahora, como α'' coincide con α excepto posiblemente sobre las variables $\overline{X}, \overline{Y}$, que no aparecen en φ, se tiene finalmente $\alpha \models^I \varphi$. □

4.3 Completitud del estrechamiento por restricciones

El objetivo de este apartado es obtener resultados que, informalmente, afirmen que toda solución de un objetivo queda capturada por un $\leadsto$-cómputo terminado en un sentido adecuado. Comencemos por discutir algunos aspectos importantes de los resultados que esperamos obtener.

En primer lugar, no podemos esperar en general que mediante $\leadsto$-pasos, aunque sean los 'más adecuados', consigamos eliminar todas las subexpresiones no primitivas de un objetivo *con soluciones*. Como consecuencia, la noción adecuada de cómputo terminado será la de cómputo que llega a un objetivo semánticamente terminado, según la definición 4.2. El siguiente ejemplo justifica nuestra afirmación.

Ejemplo 4.2

Consideremos la instancia $CFLP(\mathcal{H}_{\neq})$ presentada al final del apartado 2.3.3. Supongamos un conjunto de constructoras $\mathcal{CS}^0 = \{a, b\}$, $\mathcal{CS}^2 = \{c\}$, y una función no primitiva $f \in \Delta$ definida por la única regla $f(X) = f(X)$.

Consideremos el objetivo $\varphi \equiv c(f(X), a) =/= c(a, b)$. Es claro que φ admite como solución cualquier valoración, incluso aunque en el modelo mínimo de este programa se tenga $f(X) = \bot, \forall x \in \mathcal{H}$. Es más, φ está semánticamente terminado para cualquier α, pues se tiene

$$\alpha[\![\ c(f(X), a) =/= c(a, b)\ ]\!] = \alpha[\![c(\bot, a) =/= c(a, b)]\!] = true$$

Sin embargo, mediante estrechamiento no podemos hacer desaparecer de φ la expresión no primitiva $f(X)$. El único cómputo posible es

$$c(f(X), a) =/= c(a, b) \leadsto c(f(X), a) =/= c(a, b) \leadsto \ldots$$

$\diamond$

Una segunda cuestión importante relativa a los resultados de completitud es la siguiente: el operador $T_{\mathcal{P}}$, base de nuestra semántica declarativa de modelo mínimo, está definido en términos del *supremo* de una colección de valores obtenidos mediante aplicación de diferentes reglas del programa para una misma función no primitiva. Operacionalmente eso significa que diferentes $\leadsto$-pasos, alternativos para un mismo objetivo y posición, pueden contribuir 'parcialmente' a la obtención de la denotación final (o sea, en el modelo mínimo) de una subexpresión, sin que ninguno de ellos por separado sea suficiente. El siguiente ejemplo lo pone de manifiesto.

Ejemplo 4.3

Consideremos de nuevo la instancia $CFLP(\mathcal{H}_{\neq})$, y el mismo conjunto de constructoras del ejemplo anterior. Junto con la la función f definida como antes por $f(X) = f(X)$,

consideramos ahora otra g definida por las reglas

$$g(X) = c(a, f(X)).$$
$$g(X) = c(f(X), b).$$

Este programa es consistente (nótese también que es ambiguo), y en el modelo mínimo se tiene

$$f(x) = \bot, \quad g(x) = c(a,b), \quad \forall x \in \mathcal{H}$$

Así pues el objetivo $\varphi \equiv g(X) == c(a,b)$ admite como solución cualquier valoración. Sin embargo, no hay cómputos semánticamente terminados, pues los únicos cómputos posibles a partir de φ son los dos siguientes

$$g(X) == c(a,b) \leadsto c(a, f(X)) == c(a,b) \leadsto c(a, f(X)) == c(a,b) \leadsto \ldots$$
$$g(X) == c(a,b) \leadsto c(f(X), b) == c(a,b) \leadsto c(f(X), b) == c(a,b) \leadsto \ldots$$

y ninguno termina semánticamente, pues

$$\alpha[\![\mid c(a, f(X)) == c(a,b) \mid]\!] = \alpha[\![c(a,\bot) == c(a,b)]\!] = \bot$$

y análogamente con $c(f(X), b) == c(a,b)$.

$\diamond$

En consecuencia, debemos imponer a los programas condiciones más restrictivas que la mera consistencia para obtener completitud. En otro caso, deberíamos complementar la regla para $\leadsto$ con otra adicional para 'recolectar y hacer el supremo de valores parcialmente calculados por diferentes cómputos', lo que no parece tener mucho interés, al menos desde un punto de vista práctico.

Introducimos a continuación la propiedad que pedimos a los programas para garantizar la completitud del estrechamiento por restricciones.

Definición 4.5

Un programa $\mathcal{P}$ se dice determinista *si se verifica*

$$f^{T_{\mathcal{P}}(I)}\overline{x} = max\ C(I, f, \overline{x}) \qquad (convenimos\ max\ \emptyset = \bot)$$

para toda interpretación I tal que $T_{\mathcal{P}}(I)$ esté definido, y toda $f \in \Delta^n, \overline{x} \in D^n$.

Puede apreciarse informalmente que esta condición subsana los problemas indicados arriba, pues al reemplazar 'supremo' por 'máximo' en la definición de $T_{\mathcal{P}}$ se está indicando que para obtener el valor de una expresión no primitiva va a haber (al menos) una regla adecuada. Observemos, por si acaso la condición de determinismo parece excesivamente artificial y diseñada a propósito, que todo programa no ambiguo es determinista. Asumimos para el resto de la sección que los programas son consistentes y deterministas.

Podemos ya presentar un enunciado razonable para un resultado de completitud de $\leadsto$.

Teorema 4.2

Sea α una solución de φ. Entonces existe un cómputo $\varphi \leadsto^* \psi$ de modo que ψ está semánticamente terminado para α.

El resultado anterior – que resultará ser cierto como corolario de otros más fuertes que probaremos más adelante – no es suficientemente satisfactorio, pues no revela si para construir un cómputo terminado para un objetivo φ y solución α es indiferente la elección de la posición no primitiva en φ en la que hacer estrechamiento [14]. De no serlo, el espacio de búsqueda determinado por un objetivo podría ser vastísimo. El siguiente ejemplo muestra que de hecho la elección de la posición es relevante.

Ejemplo 4.4

Consideremos el programa (en $CFLP(\mathcal{H})$, ver 2.3.3)

$$\begin{array}{l} g(X) = 0. \\ f(X) = X \Leftarrow f(X) == f(X) \end{array} .$$

Es claro que en el modelo mínimo $T_{\mathcal{P}} \uparrow \omega$ se tiene

$$g^{T_{\mathcal{P}}\uparrow\omega}(x) = 0 \ (\forall x) \quad \text{y} \quad f^{T_{\mathcal{P}}\uparrow\omega}(x) = \bot \ (\forall x)$$

El objetivo $g(f(X)) == 0$ admite como solución (en $T_{\mathcal{P}} \uparrow \omega$) cualquier valoración α, y se tiene, de hecho, $\alpha \models^{T_{\mathcal{P}}\uparrow 1} g(f(X)) == 0$. Así pues, un cómputo aparentemente 'adecuado' consistiría en dar un paso de reducción para f y otro para g. Obtendríamos

$$\begin{array}{lll} & g(\underline{f}(X)) == 0 & (\varphi_0) \\ \leadsto & \underline{g}(X) == 0, f(X) == f(X) & (\varphi_1) \\ \leadsto & 0 == 0, f(X) == f(X) & (\varphi_2) \end{array}$$

y se tiene que φ_2 no tiene ninguna solución, lo que de hecho ya sucede para φ_1. Esto nos indica que no hay ningún cómputo terminado para φ_0 que comience por reducir $f(X)$. Por supuesto, es fácil indicar en este ejemplo un cómputo que termina (el único, en este caso)

$$\underline{g}(f(X)) == 0 \leadsto 0 == 0 \quad (\forall \alpha, \alpha \models 0 == 0)$$

Nótese que la subexpresión $f(X)$ no ha requerido ser evaluada. Más aún, como hemos visto antes, su evaluación supone perder todas las soluciones del objetivo inicial.

$\diamond$

Como conclusión adicional de este ejemplo, y ya pensando en caracterizar las *posiciones demandadas* a las que queremos restringir el uso de $\leadsto$, observamos que conocer el nivel

[14] Parece claro que, una vez elegida una posición, la elección de la regla del programa a utilizar *no* es indiferente.

de evaluación k (es decir, la iteración $T_{\mathcal{P}} \uparrow k$) necesario para capturar una solución, no es suficiente para reconocer dichas posiciones, ya que el valor de k queda fijado por el máximo de los niveles requeridos por cada una de las subexpresiones no primitivas. En el ejemplo anterior es g la responsable de que el nivel sea $T_{\mathcal{P}} \uparrow 1$. A f le bastaría el nivel $T_{\mathcal{P}} \uparrow 0$, pues no requiere evaluación.

Para poder expresar niveles de evaluación diferentes para distintas subexpresiones no primitivas, introducimos la noción de *etiquetado con interpretaciones*, que generaliza la idea de interpretación.

Definición 4.6 *(Etiquetado con interpretaciones)*

Sea $e \in \mathcal{T}_{F \cup \Delta}(V)$ no primitiva. Un etiquetado con interpretaciones *para e es una aplicación $\tau : NPos(e) \longrightarrow \mathcal{INT}$. Denotamos por $\mathcal{TINT}_e$ al conjunto de los etiquetados con interpretaciones para e. La definición se extiende sin dificultad al caso de restricciones $\varphi \in Con_{\Sigma \cup \Delta}$.*

Un etiquetado τ puede entenderse también como el resultado de reemplazar cada aparición en e de símbolos no primitivos $f \in \Delta$ por nuevos símbolos $f^{\tau(u)}$, siendo u la posición correspondiente a f. Hablaremos indistintamente del etiquetado con interpretaciones τ para e o de la *expresión etiquetada* e^{τ}.

De modo natural, podemos identificar (para una expresión e dada) una interpretación I con el etiquetado I definido por $I(u) = I, \forall u \in NPos(e)$. Hablaremos entonces del etiquetado *homogéneo* I. También de modo natural podemos extender al caso de etiquetados las nociones de 'orden entre interpretaciones', y 'valor de una expresión bajo una valoración y una interpretación'. Las siguientes definiciones se encargan de ello.

Definición 4.7

$$\tau_1 \sqsubseteq_{\mathcal{TINT}_e} \tau_2 \Leftrightarrow \tau_1(u) \sqsubseteq_{\mathcal{INT}} \tau_2(u), \forall u \in NPos(e)$$

Nótese que si $I_1 \sqsubseteq_{\mathcal{INT}} I_2$, también $I_1 \sqsubseteq_{\mathcal{TINT}} I_2$. Es fácil probar que $(\mathcal{TINT}_e, \sqsubseteq_{\mathcal{TINT}_e})$ es un dominio de Scott, cuyo mínimo es $\bot_{\mathcal{INT}}$.

Definición 4.8

El valor *de una expresión e bajo una valoración α y un etiquetado $\tau \in \mathcal{TINT}_e$, que denotamos indistintamente por $\alpha[\![e]\!]_\tau$ o $\alpha[\![e^\tau]\!]$, está definido inductivamente por*

- $\alpha[\![X]\!]_\tau = X$
- $\alpha[\![c(e_1, \ldots, e_n)]\!]_\tau = c(\alpha[\![e_1]\!]_\tau, \ldots, \alpha[\![e_n]\!]_\tau)$, *si* $c \in F$
- $\alpha[\![f(e_1, \ldots, e_n)]\!]_\tau = f^{\tau(u)}(\alpha[\![e_1]\!]_\tau, \ldots, \alpha[\![e_n]\!]_\tau)$, *si* $f \in \Delta, [e]_u = f(e_1, \ldots, e_n)$

Un par de observaciones: al considerar restricciones $\varphi \in Con_{\Sigma \cup \Delta}$ en lugar de expresiones, tiene sentido la notación $\alpha \models^\tau \varphi$ (o $\alpha \models \varphi^\tau$, indistintamente). Por otra parte, no es difícil

probar que la proposición 3.1 sobre continuidad de la evaluación sigue siendo cierta cuando consideramos $\mathcal{TINT}_e$ en lugar de $\mathcal{INT}$.

En lo que sigue consideraremos exclusivamente etiquetados que utilicen como interpretaciones las aproximaciones $T_\mathcal{P} \uparrow k$ al modelo mínimo $T_\mathcal{P} \uparrow \omega$. Para una expresión dada e (y análogamente para una restricción dada φ), notamos por $\mathcal{TINT}_e^n$ al conjunto de etiquetados que utilicen iteraciones $T_\mathcal{P} \uparrow k$, con $k \leq n$, y $\mathcal{TINT}_e^\omega = \bigcup_{n \in \mathbb{N}} \mathcal{TINT}_e^n$. Resulta fácil de probar lo siguiente:

Proposición 4.3

Dada una restricción $\varphi \in Con_{\Sigma \cup \Delta}$, se tiene

- $\bigsqcup_{\tau \in \mathcal{TINT}_\varphi^\omega} \tau = T_\mathcal{P} \uparrow \omega$
- $\alpha \models^{T_\mathcal{P} \uparrow \omega} \varphi \Leftrightarrow \alpha \models \varphi^\tau$, *para algún* $\tau \in \mathcal{TINT}_\varphi^\omega$

Este resultado generaliza lo que ya sabíamos para las iteraciones $T_\mathcal{P} \uparrow k$, con la ventaja de que los etiquetados permiten afinar más al expresar el grado de evaluación requerido para satisfacer un cierto objetivo φ, ya que permiten distinguir diferentes niveles k para distintas subexpresiones no primitivas en φ.

Ejemplo 4.5

Si consideramos el programa del ejemplo 4.4, algunos etiquetados para el objetivo φ dado por $g(f(X)) == 0$ podrían ser

$$\begin{array}{rl} \varphi^{\tau_0} \equiv & g^{T_\mathcal{P} \uparrow 0}(f^{T_\mathcal{P} \uparrow 0}(X)) == 0 \\ \varphi^{\tau_1} \equiv & g^{T_\mathcal{P} \uparrow 1}(f^{T_\mathcal{P} \uparrow 1}(X)) == 0 \\ \varphi^{\tau_2} \equiv & g^{T_\mathcal{P} \uparrow 1}(f^{T_\mathcal{P} \uparrow 0}(X)) == 0 \end{array}$$

Los dos primeros son etiquetados homogéneos que corresponden a las interpretaciones (de $\mathcal{INT}$) $T_\mathcal{P} \uparrow 0 (\equiv \bot_{\mathcal{INT}})$ y $T_\mathcal{P} \uparrow 1$. φ^{τ_0} expresa un nivel 'insuficiente' de evaluación, pues no captura ninguna solución α (en el sentido de que $\alpha \not\models \varphi^{\tau_0}$). φ^{τ_1} captura todas las soluciones – pues $\alpha \models \varphi^{\tau_1}, \forall \alpha$ – pero, como vimos en el ejemplo 4.4, en cierto sentido expresa un nivel 'excesivo' de evaluación para $f(X)$. El etiquetado φ^{τ_2} sí que aprovecha la posibilidad de indicar distintos niveles para distintas subexpresiones, y puede considerarse adecuado pues captura todas las soluciones ($\alpha \models \varphi^{\tau_2}, \forall \alpha$) expresando el nivel justo de evaluación que corresponde a cada subexpresión ($f(X)$ no requiere ser evaluada, y basta la aplicación de una regla de g a $g(f(X))$ para terminar).

En lo sucesivo, para simplificar la notación, escribiremos k en lugar de $T_\mathcal{P} \uparrow k$ al etiquetar una expresión.

$\diamond$

A continuación introducimos una terminología para expresar el hecho, ilustrado en el ejemplo anterior, de que un etiquetado de un objetivo capture una de sus soluciones.

Definición 4.9 *(Testigos)*

Sea α una solución de φ (o sea, $\alpha \models^{T_P \uparrow \omega} \varphi$). Se dice que un etiquetado $\tau \in \mathcal{TINT}^{\omega}_{\varphi}$ es un testigo *de φ para α si $\alpha \models^{\tau} \varphi$. Un testigo se dice* minimal *si lo es con relación a $\sqsubseteq_{\mathcal{TINT}^{\omega}_{\varphi}}$.*

Obsérvese que el orden $\sqsubseteq_{\mathcal{TINT}^{\omega}_{\varphi}}$ está bien fundado (dado un etiquetado τ, sólo hay un número finito de etiquetados menores), de lo que, junto con la proposición 4.3, podemos deducir

Proposición 4.4

Si α es solución de φ, existe algún testigo minimal de φ para α.

El siguiente ejemplo muestra que la noción de testigo depende de la solución considerada, y que una solución puede tener varios testigos minimales.

Ejemplo 4.6

Consideremos el programa (en $CFLP(\mathcal{H})$, ver 2.3.3)

$$\begin{array}{l} or(true, X) = true. \\ or(X, true) = true. \\ p(0) = true. \\ q(0) = true. \\ q(s(X)) = true \Leftarrow q(X) == true \end{array}$$

y el objetivo

$$\varphi \equiv \quad or(p(X), q(s(X))) == true$$

Dos soluciones de φ pueden ser α_1 y α_2, donde $\alpha_1(X) = 0$ y $\alpha_2(X) = s(0)$. Un testigo minimal para α_1 es

$$\varphi^{\tau_1} \equiv \quad or^1(p^1(X), q^0(s(X))) == true$$

que sin embargo no es testigo para α_2. Un testigo minimal para α_2 es

$$\varphi^{\tau_2} \equiv \quad or^1(p^0(X), q^3(s(X))) == true$$

que resulta ser también testigo, aunque no minimal, para α_1. Nótese además que φ^{τ_2} no es comparable (en el orden de etiquetados) con φ^{τ_1}. Con seguridad α_1 tendrá algún testigo minimal menor que φ^{τ_2}, y por tanto distinto de φ^{τ_1}. En nuestro caso el único es

$$\varphi^{\tau_3} \equiv \quad or^1(p^0(X), q^2(s(X))) == true$$

$\diamond$

Podemos imaginar que distintos testigos minimales para una misma solución corresponden a 'planes de evaluación' alternativos. Informalmente, para obtener el teorema de completitud razonamos como sigue: dado un objetivo φ y una solución α, consideramos un testigo

minimal φ^τ. Sea cual sea (indeterminismo 'don't care') la posición *demandada* – o sea, etiquetada con $T_\mathcal{P} \uparrow k, k > 0$ – que elijamos, la subexpresión correspondiente $f(e_1, \ldots, e_n)$ debe ser aún reducida de acuerdo con τ. Una de las reglas de f (indeterminismo 'don't know') nos proporciona el valor $f^{T_\mathcal{P}\uparrow k}(\alpha[\![e_1]\!]_\tau, \ldots, \alpha[\![e_n]\!]_\tau)$, de modo que los símbolos no primitivos posiblemente introducidos por el lado derecho de la regla requerirán ser evaluados hasta un nivel $\leq T_\mathcal{P} \uparrow k - 1$, es decir, el nuevo objetivo es *más sencillo* y estaremos 'más cerca' de terminar el cómputo. Reiterando el proceso, llegaremos a terminar el cómputo.

Queda todavía bastante camino hasta convertir esta exposición informal en una demostración precisa. Hemos remarcado en cursiva dos nociones que van a jugar un papel importante: la primera, 'posiciones demandadas', es clave para formular lo que entendemos por *estrechamiento perezoso con restricciones*, que es la regla de cómputo para la que vamos a probar completitud. La herramienta técnica más importante que utilizaremos para la demostración es un *orden* entre objetivos que refleje con precisión la idea de objetivo 'más sencillo'. Este orden debe ser una extensión de $\sqsubseteq_{\mathcal{TINT}}$ — que se aplica a etiquetados de una misma expresión e o restricción φ — a un orden $\prec$ definido sobre el conjunto de todas las expresiones etiquetadas (y análogamente sobre el conjunto de las restricciones u objetivos etiquetados). Como en estas definiciones sólo intervienen expresiones etiquetadas, para facilitar la lectura omitiremos por un tiempo el uso de los superíndices τ. En lo que sigue, notamos mediante $\mathcal{T}Exp$, $\mathcal{T}Con$ y $\mathcal{T}Exp^k$ a los conjuntos de las expresiones etiquetadas, las restricciones etiquetadas, y las expresiones etiquetadas con etiquetas $\leq k$.

El orden $\prec$ que vamos a considerar no es más que el clásico '*recursive path ordering* (RPO)' ([44]; véase también [46]). Previamente hemos de definir un orden $>$ de *prioridad* de símbolos etiquetados (entre los que incluimos los símbolos primitivos como etiquetados de sí mismos), en el que está basado $\prec$.

Definición 4.10 *(Prioridad de símbolos etiquetados)*

- $f^k > c, \forall f \in \Delta, k \in \mathbb{N}, c \in F \cup \Pi$
- $f^k > g^{k'}, \forall f, g \in \Delta, k, k' \in \mathbb{N}, k > k'$

Definimos ahora simultáneamente el orden $\prec$ entre expresiones etiquetadas y su extensión $\ll$ a multiconjuntos de expresiones etiquetadas.

Definición 4.11 *(Orden de expresiones etiquetadas)*

(i) El orden $\prec$ entre expresiones etiquetadas viene definido por

- $X \prec e, \forall X \in V, e \in \mathcal{T}Exp, e \notin V$
- $h(e_1, \ldots, e_n) \prec l(e_1', \ldots, e_m')$ *si y solo si se verifica alguna de las tres condiciones siguientes*
  1. $h(e_1, \ldots, e_n) \preceq e_j'$, *para algún* $j \in \{1, \ldots, m\}$
  2. $h < l$ *y* $e_i \prec l(e_1', \ldots, e_m')$, *para todo* $i \in \{1, \ldots, n\}$
  3. $h = l$ *y* $\{e_1, \ldots, e_n\} \ll \{e_1', \ldots, e_m'\}$, *siendo* $\ll$ *la extensión a multiconjuntos de* $\prec$.

(ii) La extensión a multiconjuntos de $\prec$ es el menor orden $\ll$ que verifica $S \cup \{e_1, \ldots, e_n\} \ll S \cup \{e\}$, para cada multiconjunto S de expresiones de $\mathcal{T}Exp$, cada $e, e_1, \ldots, e_n \in \mathcal{T}Exp$ con $n \geq 0$ y $e_i \prec e, \forall i = 1 \ldots n$.

Obsérvese que $\prec$ es realmente un orden RPO en el sentido de [44], pues en lo que se refiere a $\prec$ estamos tratando a las variables como nuevos símbolos de constante, asignándoles prioridad mínima.

El *orden de restricciones u objetivos etiquetados* que consideramos es de nuevo la extensión $\ll$ a multiconjuntos de $\prec$, lo que tiene sentido, pues las restricciones etiquetadas son multiconjuntos de átomos etiquetados, a los que se aplica el orden $\prec$ definido anteriormente. Debe entenderse que las posibles cuantificaciones que aparezcan son ignoradas a efectos del orden entre restricciones. Es decir

Definición 4.12 *(Orden de restricciones etiquetadas)*
Dadas $\exists\overline{U}\varphi, \exists\overline{V}\psi \in \mathcal{T}Con$, donde φ, ψ no tienen cuantificaciones, definimos

$$\exists\overline{U}\varphi \ll \exists\overline{V}\psi \Leftrightarrow S_\varphi \ll S_\psi$$

siendo S_φ, S_ψ los multiconjuntos de los átomos etiquetados de φ, ψ.

Recogemos en el siguiente lema algunas propiedades de los órdenes recién definidos.

Lema 4.2

(i) $\prec$ y $\ll$ son órdenes bien fundados.

(ii) Si $e_1 \prec e_2$, entonces $e[u \leftarrow e_1] \prec e[u \leftarrow e_2]$, para todas $e, e_1, e_2 \in \mathcal{T}Exp$

(iii) Si $e \prec e'$, siendo $e' \equiv f^k(X_1, \ldots, X_n)$, entonces $e\sigma \prec e'\sigma$, para toda sustitución $\sigma \equiv \{Y_1/e'_1, \ldots, Y_l/e'_l\}$ con $e'_i \in \mathcal{T}Exp, \{Y_1, \ldots, Y_l\} \subset \{X_1, \ldots, X_n\}$.

(iv) Si $e \in \mathcal{T}Exp^k$ y $k < k'$, entonces $e \prec f^{k'}(e_1, \ldots, e_n)$, para cualquier $f \in \Delta$, $e_1, \ldots, e_n \in \mathcal{T}Exp$.

(v) Si $t, t_1, \ldots, t_n \prec t'$ entonces

$$\{s[u \leftarrow t'], s_1, \ldots, s_m\} \gg \{s[u \leftarrow t], t_1, \ldots, t_n, s_1, \ldots, s_m\}$$

para $s_1, \ldots, s_m \in \mathcal{T}Exp$ cualesquiera.

Demostración:
Los apartados $(i), (ii)$ son bien conocidos ([44, 46]) para los órdenes RPO. En cuanto a $(iv), (v)$, son casi simples observaciones escritas para su uso futuro, y cuya prueba es casi inmediata. El apartado (iii) merece más atención, y lo probamos por inducción sobre la estructura de e.

- Sea $e \equiv X$. Si $X \notin \{Y_1, \ldots, Y_l\}$, entonces $e\sigma \equiv X \prec e'\sigma$, ya que $e'\sigma$ no es una variable. Si, por el contrario, $X \in \{Y_1, \ldots, Y_l\} \subset \{X_1, \ldots, X_n\}$, entonces $e\sigma$ es un subtérmino de $e'\sigma$, por lo que $e\sigma \prec e'\sigma$.

- Sea $e \equiv g^l(e_1, \ldots, e_m)$, y supongamos el resultado cierto para $e_1, \ldots, e_m$.

  Es claro que no puede verificarse $g^l \succ f^k$, pues en ese caso la única posibilidad para $e \prec f^k(X_1, \ldots, X_n)$ es que $e \prec X_i$ para algún i, lo que no puede suceder.

  Tampoco puede darse $g^l \equiv f^k$, pues entonces se tendría $m \equiv n, e \equiv f^k(e_1, \ldots, e_n)$, y la condición $e \prec e'$ implicaría $\{e_1, \ldots, e_n\} \ll \{X_1, \ldots, X_n\}$, que tampoco puede ser cierto.

  Así pues se tiene $g^l \prec f^k$, y puesto que $e \prec e'$, debe verificarse $e_i \prec e'$ para todo $i = 1, \ldots, n$. La hipótesis de inducción garantiza que $e_i\sigma \prec e'\sigma$, y finalmente

$$e\sigma \equiv g^l(e_1\sigma, \ldots, e_m\sigma) \prec f^k(X_1\sigma, \ldots, X_n\sigma) \equiv e'\sigma$$

  lo que completa la inducción.

□

El siguiente lema es el resultado técnico más importante de esta sección, por ser la clave para probar la completitud.

Lema 4.3 *(Reducción de la complejidad)*

Sea α una solución de φ, u una posición no primitiva en φ tal que $[\varphi]_u = f(e_1, \ldots, e_n)$ ($f \in \Delta^n$), y φ^τ un testigo minimal de φ para α tal que $[\varphi^\tau]_u = f^k(e_1^{\tau_1}, \ldots, e_n^{\tau_n})$, con $k > 0$. Entonces existe una regla $R \in \mathcal{P}$ para f para la que $\varphi \rightsquigarrow_{u,R} \psi$, y de modo que

(i) α es solución de ψ

(ii) ψ tiene un testigo minimal $\psi^{\tau'}$ para α tal que $\psi^{\tau'} \ll \varphi^\tau$.

Demostración:

En primer lugar, observemos que $\alpha[\![f^k(e_1^{\tau_1}, \ldots, e_n^{\tau_n})]\!]$ no puede ser $\bot$, pues en ese caso podríamos obtener un testigo $\ll \varphi^\tau$ (contra la hipótesis de minimalidad de φ^τ) sin más que reemplazar en la posición u de φ^τ el símbolo f^k por el de prioridad estrictamente más pequeña f^0.

Ahora, si tenemos en cuenta que

$$(\bot \neq)\alpha[\![f^k(e_1^{\tau_1}, \ldots, e_n^{\tau_n})]\!] = f^{T_\mathcal{P}\uparrow k}(\alpha[\![e_1^{\tau_1}]\!], \ldots, \alpha[\![e_n^{\tau_n}]\!])$$

podemos concluir, de la definición de $T_\mathcal{P} \uparrow k$ $(= T_\mathcal{P}(T_\mathcal{P} \uparrow k-1))$ y de la condición de determinismo del programa $\mathcal{P}$, que existe una regla $R \equiv \ f(X_1, \ldots, X_n) = e \Leftarrow \pi$ y una valoración α' tales que

$$\begin{array}{ll} (1) & \alpha'(X_i) = \alpha[\![e_i^{\tau_i}]\!], \text{ para todo } i = 1, \ldots, n \\ (2) & \alpha' \models^{T_\mathcal{P}\uparrow k-1} \pi \\ (3) & \alpha'[\![e]\!]_{T_\mathcal{P}\uparrow k-1} = \alpha[\![f^k(e_1^{\tau_1}, \ldots, e_n^{\tau_n})]\!] \end{array}$$

Podemos suponer además que las variables $\overline{Y}$ de R no aparecen en φ (considerando una variante si es preciso) y que α' coincide con α excepto posiblemente sobre las variables $\overline{Y}$ (ya que las condiciones pedidas a α' sólo afectan a estas últimas). En particular, α' coincide con α sobre las variables libres de φ, por lo que se tiene $\alpha' \models \varphi^\tau$, y además (1), (3) pueden ser reformulados como

$$(1') \quad \alpha'(X_i) = \alpha'[\![e^{\tau_i}]\!]$$
$$(3') \quad \alpha'[\![e]\!]_{T_\mathcal{P}\uparrow k-1} = \alpha'[\![f^k(e_1^{\tau_1}, \ldots, e_n^{\tau_n})]\!]$$

Si usamos la regla R para estrechar φ obtendremos

$$\varphi \rightsquigarrow_{u,R} \psi$$

siendo $\psi \equiv \exists\overline{Y}(\varphi[u \leftarrow e\sigma], \pi\sigma)$, con $\sigma = \{X_1/e_1, \ldots, X_n/e_n\}$. Para demostrar el lema basta probar que ψ tiene un testigo ψ^τ para α tal que $\psi^\tau \ll \varphi^\tau$, pues por definición de testigo se tendría $\alpha \models \psi^\tau$ (y por tanto $\alpha \models \psi$, es decir, se tiene *(i)*), y para *(ii)* bastará considerar un testigo minimal $\psi^{\tau'} \ll \psi^\tau$. El testigo de ψ buscado puede venir dado por

$$\psi^\tau \equiv \exists\overline{Y}(\varphi^\tau[u \leftarrow e^{(k-1)}\sigma^\tau], \pi^{(k-1)}\sigma^\tau)$$

donde $\sigma^\tau = \{X_1/e_1^{\tau_1}, \ldots, X_n/e_n^{\tau_n}\}$, y $e^{(k-1)}, \pi^{(k-1)}$ denotan los etiquetados homogéneos de e, π correspondientes a $T_\mathcal{P} \uparrow k-1$.

Que $\psi^\tau \ll \varphi^\tau$ se deduce del lema 4.2*(iv)*. En efecto: 4.2*(iv)* nos asegura que tanto $e^{(k-1)}$ como $\pi^{(k-1)}$ son $\prec f^k(X_1, \ldots, X_n)$. Aplicando *(iii)* del mismo lema, tenemos

$$e^{(k-1)}\sigma^\tau \prec f^k(e_1^{\tau_1}, \ldots, e_n^{\tau_n})$$

y lo mismo para $\pi^{(k-1)}\sigma^\tau$. Finalmente, el lema 4.2*(v)* nos permite concluir que $\psi^\tau \ll \varphi^\tau$.

Sólo resta probar que $\alpha \models \psi^\tau$, y para ello es suficiente – ya que α' coincide con α excepto posiblemente en $\overline{Y}$ – probar que

$$\alpha' \models \varphi^\tau[u \leftarrow e^{(k-1)}\sigma^\tau], \pi^{(k-1)}\sigma^\tau$$

Para ello observemos que, por la definición de σ^τ y por $(1')$,

$$\alpha'[\![X_i\sigma^\tau]\!] = \alpha'[\![e_i^{\tau_i}]\!] = \alpha'(X_i), \text{ para todo } i = 1, \ldots, n$$

y además, por el significado de $e^{(k-1)}$ y por $(3')$

$$\alpha'[\![e^{(k-1)}]\!] = \alpha'[\![e]\!]_{T_\mathcal{P}\uparrow k-1} = \alpha'[\![f^k(e_1^{\tau_1}, \ldots, e_n^{\tau_n})]\!]$$

Podemos deducir entonces, por el lema 4.1, que

$$\alpha[\![e^{(k-1)}\sigma^\tau]\!] = \alpha[\![e^{(k-1)}]\!] = \alpha[\![f^k(e_1^{\tau_1}, \ldots, e_n^{\tau_n})]\!]$$

Como consecuencia, y ya que $\alpha' \models \varphi^\tau$, se tendrá también $\alpha' \models \varphi^\tau[u \leftarrow e^{(k-1)}\sigma^\tau]$. Un razonamiento similar, algo más simple, nos permite afirmar que $\alpha' \models \pi^{(k-1)}\sigma^\tau$, utilizando en este caso la condición (2) de arriba. En definitiva, hemos obtenido

$$\alpha' \models \varphi^\tau[u \leftarrow e^{(k-1)}\sigma^\tau], \pi^{(k-1)}\sigma^\tau$$

lo que concluye la demostración.

□

Los pasos de estrechamiento dados conforme al lema anterior son pasos que denominamos *perezosos*. La noción de pereza admite sin embargo algunos matices que desarrollamos a continuación.

Definición 4.13 *(Posiciones demandadas)*
Sea α una solución de φ y $u \in NPos(\varphi)$.

- *Si φ^τ es un testigo minimal de φ para α, se dice que u* está demandada por *α, τ si $\tau(u) = T_\mathcal{P} \uparrow k, k > 0$.*
- *Se dice que u* está demandada por *α si lo está por algún testigo minimal para α.*
- *Se dice que u* está absolutamente demandada *si lo está por todas las soluciones de φ y todos sus testigos minimales.*

La existencia de posiciones absolutamente demandadas no está garantizada para todo objetivo, como muestra el siguiente ejemplo.

Ejemplo 4.7

Consideremos, en $CFLP(\mathcal{H}_{\neq})$, dos funciones no primitivas f, g definidas por las reglas

$$f(X) = a.$$
$$g(X) = b.$$

donde $a, b \in \mathcal{CS}^0$. Si consideramos el objetivo $\varphi \equiv c(f(X), g(X)) =/= c(b, a)$, donde $c \in \mathcal{CS}^2$, se tiene que φ (que admite cualquier valoración α como solución) no tiene posiciones absolutamente demandadas, pues para cualquier α dos testigos minimales para α son

$$c(f^1(X), g^0(X)) =/= c(b, a) \quad \text{y} \quad c(f^0(X), g^1(X)) =/= c(b, a)$$

con lo que ni $f(X)$ ni $g(X)$ están demandadas por *todos* los testigos minimales

◇

La noción de posición absolutamente demandada recuerda a la de 'redex necesario' (*needed redex*) de los sistemas ortogonales de reescritura [76]. A diferencia de éstos, como ya hemos visto, la existencia de posiciones absolutamente demandadas no está garantizada. En su lugar, habría que hablar de 'conjuntos completos de posiciones demandadas', a semejanza de los 'conjuntos de redexes necesarios' ('*sets of needed redexes*') de [139]. Desde un punto de vista más aplicado, la posible inexistencia de posiciones absolutamente demandadas implica que, en general, no hay estrategias completas para el estrechamiento con restricciones, si por

estrategia se entiende la selección de *una* posición no primitiva en el objetivo. Habría que considerar, en su lugar, una noción más general de estrategia, como en [53], que se refiera a la selección de un conjunto de posiciones.

Definamos a continuación lo que entendemos por cómputos perezosos.

Definición 4.14 *(Cómputos perezosos)*

Sea α una solución de φ_0 y $\varphi_0 \leadsto_{u_0,R_0} \varphi_1 \leadsto_{u_1,R_1} \ldots \leadsto_{u_{n-1},R_{n-1}} \varphi_n$ un cómputo que captura α, es decir, α es solución de φ_n (y por tanto de φ_i para $i = 0, \ldots, n$).

- *El cómputo es* localmente perezoso *para α si u_i está demandada por α, para todo $i = 0, \ldots, n-1$.*
- *El cómputo es* globalmente perezoso *para α si existen testigos minimales $\varphi_0^{\tau_0}, \ldots, \varphi_n^{\tau_n}$ de $\varphi_0, \ldots, \varphi_n$ tales que $\varphi_0^{\tau_0} \gg \varphi_1^{\tau_1} \gg \ldots \gg \varphi_n^{\tau_n}$ y u_i está demandada por $\alpha, \varphi_i^{\tau_i}$, para todo $i = 0, \ldots, n-1$.*

Es obvio que todo cómputo globalmente perezoso lo es localmente. Es también obvio que no existen cómputos globalmente perezosos infinitos.

El lema 4.3 nos permite concluir inmediatamente que todo cómputo perezoso (sea localmente o globalmente) puede ser continuado. Con más precisión tenemos

Teorema 4.3

- *Sea α una solución de φ_0 y $\varphi_0 \leadsto^* \varphi_n$ un cómputo localmente perezoso para α. Entonces, para toda posición u en φ_n demandada por α, se puede dar un paso $\varphi_n \leadsto_{u,R} \varphi_{n+1}$ tal que $\varphi_0 \leadsto^* \varphi_{n+1}$ es localmente perezoso para α.*
- *Sea α una solución de φ_0 y $\varphi_0 \leadsto \varphi_1 \leadsto \ldots \leadsto \varphi_n$ un cómputo globalmente perezoso para α con testigos minimales $\varphi_0^{\tau_0} \gg \varphi_1^{\tau_1} \gg \ldots \gg \varphi_n^{\tau_n}$. Entonces, para toda posición u en φ_n demandada por $\alpha, \varphi_n^{\tau_n}$, se puede dar un paso $\varphi_n \leadsto_{u,R} \varphi_{n+1}$ y elegir un testigo minimal $\varphi_{n+1}^{\tau_{n+1}}$ de φ_{n+1} para α de modo que $\varphi_0 \leadsto \ldots \leadsto \varphi_{n+1}$ es globalmente perezoso para α.*

El siguiente teorema resulta ahora casi inmediato

Teorema 4.4 *(Completitud del estrechamiento perezoso por restricciones)*

Sea α una solución de φ. Entonces existe un cómputo $\varphi \leadsto^* \psi$ globalmente perezoso y semánticamente terminado para α.

Algunos comentarios sobre los resultados anteriores. El lema 4.3 sobre reducción de la complejidad de objetivos es el que proporciona la información más importante, y donde se concentra la mayor parte del trabajo técnico que utiliza los órdenes definidos. Este lema es más aprovechado en el segundo apartado del teorema 4.3 que le sigue, que podríamos parafrasear así: *si la elección de las reglas es la adecuada, un cómputo globalmente perezoso*

termina, independientemente de las posiciones (demandadas) elegidas para reducir. El primer apartado de ese mismo teorema es más débil, y podríamos leerlo así: *si la elección de las reglas es la adecuada, un cómputo localmente perezoso se puede continuar, independientemente de las posiciones (demandadas) elegidas para reducir, y eventualmente terminar.* Recordemos (ver ejemplo 4.4) que esto no tiene por qué ser cierto si se reduce en posiciones no demandadas, en cuyo caso podemos perder la solución buscada, con independencia de la regla elegida para reducir.

El teorema de completitud 4.4, como suele ser habitual, tiene un enunciado más claro, pero es más débil en el sentido de que no refleja el indeterminismo 'don't care' de las posiciones elegidas.

Los siguientes ejemplos pretenden aclarar la diferencia entre cómputos local y globalmente perezosos.

Ejemplo 4.8

Consideremos el programa

$$\begin{array}{llll} (Or1) & or(true, Y) = true. & (F) & f(X) = g(X) \\ (Or2) & or(X, true) = true. & (G) & g(X) = true \end{array}$$

y el objetivo $\varphi_0 \equiv \quad or(f(X), g(Y)) == true$, para el que cualquier α es solución.

El siguiente cómputo es localmente, pero no globalmente perezoso.

$$\begin{array}{ll} & or(\underline{f}(X), g(Y)) == true \quad (\varphi_0) \\ \rightsquigarrow_{(F)} & or(\overline{g}(X), \underline{g}(Y)) == true \quad (\varphi_1) \\ \rightsquigarrow_{(G)} & \underline{or}(g(X), true) == true \quad (\varphi_2) \\ \rightsquigarrow_{(Or2)} & true == true, true = true \quad (\varphi_3) \end{array}$$

Las posiciones usadas están subrayadas. Son todas ellas demandadas (por cualquier solución), como indica la siguiente secuencia de testigos minimales:

$$\begin{array}{ll} or^1(f^2(X), g^0(Y)) == true & (\varphi_0^{\tau_0}) \\ or^1(g^0(X), g^1(Y)) == true & (\varphi_1^{\tau_1}) \\ or^1(g^0(X), true) == true & (\varphi_2^{\tau_2}) \\ true == true, true = true & (\varphi_3^{\tau_3}) \end{array}$$

Por tanto el cómputo es localmente perezoso. No es globalmente perezoso, pues no se verifica $\varphi_0^{\tau_0} \gg \varphi_1^{\tau_1}$, y para el único otro testigo minimal de φ_1, que vendría dado por $\varphi_1^{\tau_1'} \equiv \quad or^1(g^1(X), g^0(Y)) == true$, sí se tiene $\varphi_0^{\tau_0} \gg \varphi_1^{\tau_1'}$, pero la posición utilizada en el paso $\varphi_1 \rightsquigarrow \varphi_2$ no es demandada por $\varphi_1^{\tau_1'}$. Podríamos describir la situación diciendo que en el paso $\varphi_1 \rightsquigarrow \varphi_2$ hemos 'cambiado de testigo (o de plan)', por lo que en el cómputo global hemos hecho reducciones innecesarias. En efecto, las reducciones realizadas en $\varphi_1 \rightsquigarrow \varphi_2 \rightsquigarrow \varphi_3$ pueden efectuarse desde φ_0, sin que el paso $\varphi_0 \rightsquigarrow \varphi_1$ aporte nada. Dicho con más claridad, un cómputo globalmente perezoso sería

$$\begin{array}{lll} & or(f(X), \underline{g}(Y)) == true & (\varphi_0) \\ \leadsto_{(G)} & \underline{or}(f(X), true) == true & (\psi_1) \\ \leadsto_{(Or2)} & true == true, true = true & (\psi_2) \end{array}$$

al que correspondería la secuencia de testigos minimales

$$\begin{array}{ll} or^1(f^0(X), g^1(Y)) == true & (\varphi_0^{\tau_0'}) \\ or^1(f^0(X), true) == true & (\psi_1^{\tau_1}) \\ true == true, true = true & (\psi_2^{\tau_2}) \end{array}$$

para la que $\varphi^{\tau_0'} \gg \psi^{\tau_1} \gg \psi^{\tau_2}$.

Otro cómputo globalmente perezoso provendría de perseverar en el 'plan' correspondiente al paso $\varphi_0 \leadsto \varphi_1$

$$\begin{array}{lll} & or(\underline{f}(X), g(Y)) == true & (\varphi_0) \\ \leadsto_{(F)} & or(\underline{g}(X), g(Y)) == true & (\varphi_1) \\ \leadsto_{(G)} & \underline{or}(true, g(Y)) == true & (\varphi_2') \\ \leadsto_{(Or1)} & true == true, true = true & (\varphi_3') \end{array}$$

para el que una secuencia de testigos minimales decrecientes sería

$$\begin{array}{ll} or^1(f^2(X), g^0(Y)) == true & (\varphi_0^{\tau_0}) \\ or^1(g^1(X), g^0(Y)) == true & (\varphi_1^{\tau_1'}) \\ or^1(true, g^0(Y)) == true & (\varphi_2^{\tau_2'}) \\ true == true, true = true & (\varphi_3^{\tau_3'}) \end{array}$$

$\diamond$

En el ejemplo anterior todos los cómputos presentados (de hecho todos los posibles) terminan. En el siguiente vemos que puede haber cómputos localmente perezosos infinitos, en los que se cambia constantemente 'de plan' [15].

Ejemplo 4.9

Dado el programa

$$\begin{array}{ll} (F) & f(X) = g(f(X)). \\ (G1) & g(0) = 0. \\ (G2) & g(X) = h(X). \\ (H) & h(X) = 0. \end{array}$$

existe un cómputo localmente perezoso infinito para el objetivo $f(0) == 0$

$$\begin{array}{lll} & \underline{f}(0) == 0 & (\varphi_0) \\ \leadsto & g(\underline{f}(0)) == 0 & (\varphi_1) \\ \leadsto & g(g(\underline{f}(0))) == 0 & (\varphi_2) \\ \leadsto & \ldots & \end{array}$$

[15] También podríamos llamarlos entonces *locamente* perezosos.

Cada posición utilizada (subrayada) es demandada, como prueba la secuencia de testigos minimales

$$\begin{array}{ll} f^3(0) == 0 & (\varphi_0^{\tau_0}) \\ g^1(f^3(0)) == 0 & (\varphi_1^{\tau_1}) \\ g^1(g^1(f^3(0))) == 0 & (\varphi_2^{\tau_2}) \\ \ldots & \end{array}$$

El cómputo no es globalmente perezoso pues, por ejemplo, no se tiene $\varphi_0^{\tau_0} \gg \varphi_1^{\tau_1}$, y para el único otro testigo minimal de φ_1, que es $g^2(f^0(0)) == 0$ (y que sí resulta ser $\ll \varphi_0^{\tau_0}$), la posición en la que se reduce en el siguiente paso (la de $f(X)$) no está demandada. Es decir, hemos 'cambiado de plan'. Lo mismo pasa en el resto de los pasos del cómputo con la subexpresión $f(X)$ que aparece cada vez más internamente.

Nótese de todos modos que, aunque puedan ser infinitos, en los cómputos localmente perezosos nunca estamos perdidos del todo (si la elección de las reglas es adecuada), pues en cualquier paso el cómputo se puede continuar a uno globalmente perezoso (desde ese punto), y por tanto terminar.

$\diamond$

4.4 Combinación del estrechamiento con la resolución de restricciones

El estrechamiento por restricciones ($\leadsto$) tal como lo hemos definido no es suficiente en la práctica como regla de cómputo. Algunas de las objeciones que se pueden presentar serían:

- La regla para $\leadsto$ simplemente realiza reescritura, no contemplándose ninguna noción de resolución o simplificación de restricciones. Como consecuencia, las respuestas computadas (objetivos terminados) pueden consistir en una enorme e ininteligible colección de restricciones.
- La propia noción de cómputo semánticamente terminado es de escaso valor práctico. Por una parte, el que un objetivo esté terminado o no depende de la solución particular que consideremos, y además en un objetivo semánticamente terminado pueden aparecer subexpresiones no primitivas que, al menos desde un punto de vista 'sintáctico', estarían aún pendientes de ser evaluadas (y de hecho pueden estarlo para soluciones distintas de aquéllas para las que el objetivo esté terminado). Como es natural, cuando resolvemos en la práctica un objetivo no conocemos de antemano cuáles son sus soluciones, y precisamente lo que esperamos del cómputo es obtener una colección (posiblemente infinita, eso sí) de restricciones primitivas en algún modo de *forma resuelta*, cuyas soluciones sean las soluciones del objetivo original.
- También la noción de *paso perezoso*, basada en la de *posición demandada*, es dependiente de la solución particular bajo consideración. Desearíamos disponer de criterios efectivos para seleccionar posiciones en las que reducir.

El ejemplo que sigue muestra algunos de los problemas indicados.

Ejemplo 4.10

Consideraremos en este caso un programa de aspecto más 'real', para poder comparar mejor el tipo de cómputos y respuestas que obtenemos mediante $\leadsto$ con las que podríamos considerar razonables desde un punto de vista práctico.

Consideremos el conjunto de constructoras $\mathcal{CS}$ dado por

$$\mathcal{CS}^0 = \{true, false, 0, [\,]\},\ \mathcal{CS}^1 = \{s\},\ \mathcal{CS}^2 = \{[.|.]\}$$

En la instancia $CFLP(\mathcal{H}_{\neq})$ definida por $\mathcal{CS}$, consideremos el programa que define las funciones *card/1* y *elem/2* mediante las reglas

$$\begin{array}{lcll} elem(X,L) & = & false & \Leftarrow L = [\,] \quad (M_1) \\ elem(X,L) & = & true & \Leftarrow L = [Y \mid Ys], X == Y. \quad (M_2) \\ elem(X,L) & = & elem(X,Ys) & \Leftarrow L = [Y \mid Ys], X =/= Y. \quad (M_3) \end{array}$$

$$\begin{array}{lcll} card(L) & = & 0 & \Leftarrow L = [\,] \quad (S_1) \\ card(L) & = & card(Xs) & \Leftarrow L = [X \mid Xs], elem(X,Xs) == true. \quad (S_2) \\ card(L) & = & s(card(Xs)) & \Leftarrow L = [X \mid Xs], elem(X,Xs) == false. \quad (S_3) \end{array}$$

La función booleana *elem* devuelve *true* o *false* dependiendo de si su primer argumento es o no un elemento del segundo argumento, que debe ser una lista. La función *card* devuelve el número de elementos distintos de su argumento, que también debe ser una lista. Obsérvese que en las reglas anteriores hemos utilizado igualdades de la forma $X = t$ para expresar la unificación de un argumento X con un patrón lineal t. Como se comentó en el apartado 2.3.3, la igualdad $=$ no es continua en $CFLP(\mathcal{H}_{\neq})$, y por consiguiente el uso de restricciones del tipo $X =$ debe entenderse como una abreviatura sintáctica para evitar el uso explícito de las funciones selectoras c_k y los predicados reconocedores is_c, de lectura sin duda más engorrosa. En la $\leadsto$-derivación que analizamos a continuación, mantenemos dicha representación abreviada, sin que este hecho afecte de modo relevante a la discusión que sigue, ya que todos los $\leadsto$-pasos se dan en igualdades estrictas $==$ o desigualdades $=/=$. En cualquier caso, en el siguiente capítulo quedará justificado el uso restringido que hacemos de $=$ para expresar la unificación.

Un objetivo para este programa podría ser

$$\Leftarrow card([A,B]) =/= N \quad (\varphi_0)$$

Una solución α de φ_0 viene dada por

$$\alpha(A) = 0, \quad \alpha(B) = s(0), \quad \alpha(N) = s(0)$$

y un valor cualquiera para el resto de las variables. Una $\leadsto$–derivación a partir de φ_0 que capture α podría ser la siguiente, en la que hemos abreviado de manera obvia los símbolos

elem, card, true y *false*.

$$\begin{array}{ll} & \underline{ca}([A,B]) =/= N & (\varphi_0)\\ \rightsquigarrow_{(S_3)} & \exists X, Xs([A,B] = [X \mid Xs], \underline{el}(X, Xs) == f, \\ & s(ca(Xs)) =/= N) & (\varphi_1)\\ \rightsquigarrow_{(M_3)} & \exists X, Xs, Y, Ys([A,B] = [X \mid Xs], Xs = [Y \mid Ys], X =/= Y, \\ & \underline{el}(X, Ys) == f, s(ca(Xs)) =/= N) & (\varphi_2)\\ \rightsquigarrow_{(M_1)} & \exists X, Xs, Y, Ys([A,B] = [X \mid Xs], Xs = [Y \mid Ys], \\ & X =/= Y, Ys = [], f == f, s(\underline{ca}(Xs)) =/= N) & (\varphi_3) \end{array}$$

$$\begin{array}{ll} \rightsquigarrow_{(S_3)} & \exists X, Xs, Y, Ys, X', Xs'([A,B] = [X \mid Xs], Xs = [Y \mid Ys], \\ & X =/= Y, Ys = [], f == f, Xs = [X' \mid Xs'], \\ & \underline{el}(X', Xs') == f, s(s(ca(Xs'))) =/= N) & (\varphi_4)\\ \rightsquigarrow_{(M_1)} & \exists X, Xs, Y, Ys, X', Xs'([A,B] = [X \mid Xs], Xs = [Y \mid Ys], \\ & X =/= Y, Ys = [], f == f, Xs = [X' \mid Xs'], Xs' = [], \\ & f == f, s(s(ca(Xs'))) =/= N) & (\varphi_5) \end{array}$$

Podemos comprobar que φ_5 está semánticamente terminado para α, es decir, α es solución de $\mid \varphi_5 \mid$. En efecto, podemos redefinir α en las variables existenciales de φ_5 (que son X, Xs, Y, Ys, X' y Xs') como

$$\alpha(X) = 0,\ \alpha(Xs) = [s(0)],\ \alpha(Y) = \alpha(X') = s(0),\ \alpha(Ys) = \alpha(Xs') = []$$

en cuyo caso $\alpha(\mid \varphi_5 \mid)$ resulta ser

$$\begin{array}{l} [0, s(0)] = [0, s(0)], [s(0)] = [s(0)], 0 =/= s(0), [\,] = [\,], f == f, \\ [s(0)] = [s(0)], [\,] = [\,], f == f, s(s(\bot)) =/= s(0) \end{array}$$

que es cierta en $\mathcal{H}_{\neq}$.

A pesar del empeño que hemos puesto en ordenar en cada paso la restricción acumulada de la forma más legible, hay que aceptar que proporcionar a un usuario la restricción φ_5 como respuesta al objetivo inicial φ_0 no le animaría precisamente para usar nuestro sistema. Y si no estamos preocupados en potenciales usuarios, sino solamente en la posibilidad teórica de implementar el lenguaje $CFLP(\mathcal{H}_{\neq})$, hay alguna pregunta que responder, en particular: ¿cómo se reconocen los cómputos semánticamente terminados? Por ejemplo, en la derivación anterior φ_4 no está semánticamente terminado para ninguna solución, a diferencia de φ_5 que, como hemos visto, sí lo está para alguna. En ambos casos quedan aún expresiones no primitivas. ¿Cómo se sabe que además hay otras soluciones para las que tampoco φ_5 está semánticamente terminado, aunque las capture, como por ejemplo la que viene dada por $\alpha(A) = 0, \alpha(B) = s(0), \alpha(N) = s(s(s(0)))$?

$\diamond$

Para concluir esta discusión inicial, podemos decir que al considerar el estrechamiento por restricciones como única regla de cómputo para una instancia $CFLP(\mathcal{R})$ de nuestro esquema,

se está obviando el hecho de que un lenguaje con restricciones es interesante justamente por la existencia de algún mecanismo específico (dependiente de la estructura de base), efectivo y deseablemente eficiente, que permita determinar la satisfactibilidad de restricciones primitivas y obtener formas simplificadas (o resueltas) para éstas.

Nuestro objetivo ahora es precisar qué entendemos por un mecanismo de resolución de restricciones, cómo se combina con el estrechamiento por restricciones ($\leadsto$), y bajo qué condiciones podemos garantizar que esta semántica operacional refinada preserva las buenas propiedades – corrección y completitud – de que goza el estrechamiento por restricciones.

4.4.1 Resolución de restricciones

Dada una Σ-estructura $\mathcal{R}$, postulamos la existencia de una relación

$$\leadsto^{cs} \subset Con_\Sigma \times (Con_\Sigma \cup \{FAIL\})$$

de *resolución de restricciones primitivas*. Escribiremos $\varphi \leadsto^{cs} \psi$ y leeremos φ *se reduce a* ψ (en un paso). Podemos entender $FAIL$ como una restricción trivialmente falsa (que, por definición de $\leadsto^{cs}$, es irreducible). Operacionalmente, $\varphi \leadsto^{cs} FAIL$ indica un cómputo fallido y sólo puede producirse (salvo que $\leadsto^{cs}$ no sea correcta, ver definición más abajo) si φ es insatisfactible. Puesto que $\leadsto^{cs}$ determina de modo implícito un mecanismo de resolución de restricciones, nos referiremos a $\leadsto^{cs}$ indistintamente como 'relación de resolución de restricciones' o 'sistema de resolución de restricciones'.

Estaremos interesados en sistemas de resolución de restricciones que cumplan algunas propiedades naturales, como las que recoge la siguiente definición.

Definición 4.15

Un sistema de resolución de restricciones primitivas $\leadsto^{cs}$ *se dice*

(i) correcto *si para todo paso* $\varphi \leadsto^{cs} \psi$ *y* $\alpha \models \psi$, *se tiene* $\alpha \models \varphi$.

(ii) completo *si para toda* $\varphi \leadsto^{cs}$*-reducible y toda* $\alpha \models \varphi$, *existe* ψ *tal que* $\varphi \leadsto^{cs} \psi$ *y* $\alpha \models \psi$.

(iii) terminante *si no hay reducciones* $\varphi_0 \leadsto^{cs} \varphi_1 \leadsto^{cs} \varphi_2 \leadsto^{cs} \ldots$ *infinitas.*

(iv) semiterminante *si no hay reducciones* $\varphi_0 \leadsto^{cs} \varphi_1 \leadsto^{cs} \varphi_2 \leadsto^{cs} \ldots$ *infinitas, tales que* φ_i *sea satisfactible* $\forall i$.

(v) que admite formas resueltas *si* φ *es satisfactible, para toda* $\varphi \leadsto^{cs}$*-irreducible distinta de* $FAIL$. *Llamamos* forma resuelta *a cada una de estas* φ , *y si* $\varphi_0 \leadsto^{cs^*} \varphi$, *decimos que* φ *es una forma resuelta de* φ_0.

(vi) de decisión *si es correcto, completo, terminante y admite formas resueltas.*

(vii) de semidecisión *si es correcto, completo, semiterminante y admite formas resueltas.*

(viii) finitamente ramificado *si para cualquier* $\varphi \in Con_\Sigma$ *el conjunto* $\{\psi \in Con_\Sigma \mid \varphi \leadsto^{cs} \psi\}$ *es finito.*

Como consecuencia bastante inmediata de estas definiciones, obtenemos las propiedades que siguen, en las que usamos la notación $sol(\varphi) = \{\alpha \mid \alpha \models \varphi\}$.

Proposición 4.5

Sea $\leadsto^{cs}$ de semidecisión. Entonces:

(a) Si $\varphi \leadsto^{cs^*} FAIL$, entonces φ es insatisfactible.

(b) Si $\alpha \models \varphi$, existe una forma resuelta ψ de φ tal que $\alpha \models \psi$.

(c) Si $\leadsto^{cs}$ es de decisión se tiene el recíproco de (a). Si además es finitamente ramificado, toda restricción satisfactible φ tiene un número finito de formas resueltas $\psi_1, \ldots, \psi_n$, y se tiene $sol(\varphi) = sol(\psi_1) \cup \ldots \cup sol(\psi_n)$.

En lo sucesivo, suponemos que $\leadsto^{cs}$ es (al menos) de semidecisión y finitamente ramificado. El disponer de la relación $\leadsto^{cs}$ nos permite de momento dar una noción más efectiva de lo que entendemos por terminar un cómputo.

Definición 4.16 *(Objetivos terminados)*

Se dice que φ está terminado para *α si φ es una forma resuelta y α es solución de φ.*

Aunque por conveniencia para futuros resultados la definición sigue haciendo referencia a una determinada solución, obsérvese que toda forma resuelta es un objetivo terminado para sus soluciones, por lo que si $\leadsto^{cs}$ está definida de manera efectiva mediante reglas que permitan reconocer la $\leadsto^{cs}$-irreducibilidad, ya disponemos de un criterio efectivo de terminación de los cómputos.

La resolución de restricciones $\leadsto^{cs}$ en $\mathcal{R}$ está concebida para ser combinada con el estrechamiento por restricciones $\leadsto$ y obtener así un mecanismo de cómputo de carácter más práctico para la instancia $CFLP(\mathcal{R})$, que permita simplificar las restricciones durante el cómputo, descartar posibles cómputos sin solución, y proporcionar respuestas en forma simplificada. Pero observemos que hemos supuesto $\leadsto^{cs}$ definida sobre restricciones primitivas, mientras que en general en los $CFLP$-cómputos encontraremos restricciones con subexpresiones no primitivas, sin que ello requiera necesariamente la reducción por estrechamiento ($\leadsto$) de dichas subexpresiones, ya que podría violarse el carácter perezoso de los cómputos.

Ejemplo 4.11

Aunque no será hasta el próximo capítulo (ver 5.4) cuando presentaremos, mediante un conjunto de reglas, un sistema de resolución de restricciones primitivas para la instancia $CFLP(\mathcal{H}_{\neq})$, no sorprenderá que, dadas las constructoras $c/2, s/1, 0/0$, se pueda efectuar una $\leadsto^{cs}$-derivación

$$c(0, X) =/= c(s(0), Y) \leadsto^{cs^*} true$$

que reconozca que la desigualdad inicial es cierta con independencia de los valores de X e Y. Esta última afirmación seguiría siendo válida si la desigualdad inicial fuese $c(0, f(X)) =/= c(s(0), Y)$ (φ), donde f es una función definida, por lo que también sería de esperar que

$$c(0, f(X)) =/= c(s(0), Y) \leadsto^{cs^*} true$$

Sin embargo, desde un punto de vista meramente formal, una derivación como la anterior no es posible por medio de un resolvedor de restricciones $\rightsquigarrow^{cs}$ que, como se indica arriba, suponemos definido solamente para restricciones primitivas; no es éste el caso de la desigualdad $c(0, f(X)) =/= c(s(0), Y)$.

Obsérvese además que para conseguir reducir φ finalmente a *true* no podemos confiar en estrechar φ (mediante $\rightsquigarrow$ en la posición de $f(X)$) hasta obtener una restricción primitiva ψ, para posteriormente resolver $\psi \rightsquigarrow^{cs^*} true$; no solamente se haría trabajo superfluo (pues φ es cierta con independencia del valor de $f(X)$), sino que estrechar en $f(X)$ (el único uso posible de $\rightsquigarrow$ en este caso) puede suponer pérdida de soluciones de φ. Así sucedería, por ejemplo, si f viene definida por la regla $f(X) = X \Leftarrow f(X) == f(X)$. Tendríamos entonces

$$\varphi \rightsquigarrow c(0, X) =/= c(s(0), Y), f(X) == f(X) \quad (\psi)$$

con lo que pasamos de φ, que admite cualquier valoración como solución, a ψ, que no tiene soluciones.

La situación presentada aquí es muy similar a la del ejemplo 4.4, donde se mostraba por primera vez el peligro de estrechar en una posición no demandada. De todos modos, entre ambos ejemplos existe alguna diferencia de interés para la discusión que nos ocupa ahora: en 4.4 $f(X)$ aparecía como argumento no demandado en la expresión $g(f(X))$, y de hecho podíamos hacer desaparecer $f(X)$ estrechando en la posición de g. En nuestro caso actual, sólo las reglas de $\rightsquigarrow^{cs}$ podrían hacerse cargo de la eliminación de $f(X)$, si no fuese porque $\rightsquigarrow^{cs}$ está definido sólo sobre restricciones primitivas.

$\diamond$

Se desprende de todo lo anterior la necesidad de considerar una *extensión* de $\rightsquigarrow^{cs}$, llamémosla $\rightsquigarrow^{\widehat{cs}}$, que actúe también sobre restricciones no primitivas. La idea de tal extensión, es que permita dar pasos de resolución de restricciones (posiblemente no primitivas) que sean válidos con independencia de la interpretación que pueda darse a los símbolos no primitivos.

Es esta extensión $\rightsquigarrow^{\widehat{cs}}$ la que queremos combinar con el estrechamiento por restricciones $\rightsquigarrow$, para obtener un mecanismo de cómputo más realista. En la siguiente definición precisamos las propiedades que esperamos de $\rightsquigarrow^{\widehat{cs}}$ para ser una extensión adecuada de $\rightsquigarrow^{cs}$.

Definición 4.17

Sea $\rightsquigarrow^{cs} \subset Con_\Sigma \times (Con_\Sigma \cup \{FAIL\})$ *un sistema de semidecisión de restricciones primitivas. Decimos que* $\rightsquigarrow^{\widehat{cs}} \subset Con_{\Sigma\cup\Delta} \times (Con_{\Sigma\cup\Delta} \cup \{FAIL\})$ extiende con pereza *a* $\rightsquigarrow^{cs}$ *si verifica las siguientes propiedades:*

- $\rightsquigarrow^{\widehat{cs}}$*, restringido a* Con_Σ *(restricciones primitivas), coincide con* $\rightsquigarrow^{cs}$*.*
- Corrección*: Para toda* $I \in \mathcal{INT}$*, si* $\varphi \rightsquigarrow^{\widehat{cs}} \psi$ *y* $\alpha \models^I \psi$ *entonces* $\alpha \models^I \varphi$*.*
- Completitud*: Para toda* $I \in \mathcal{INT}$*, si* φ *es* $\rightsquigarrow^{\widehat{cs}}$*-reducible y* $\alpha \models^I \varphi$*, entonces* $\varphi \rightsquigarrow^{\widehat{cs}} \psi$ *y* $\alpha \models^I \psi$*, para alguna* ψ*.*

- Pereza: *Si* $\alpha \models\mid \varphi \mid$, *existe* ψ *primitiva tal que* $\varphi \leadsto^{\widehat{cs}^*} \psi$ *y* $\alpha \models\mid \psi \mid$.

La corrección y completitud se piden para interpretaciones cualesquiera porque entendemos $\leadsto^{\widehat{cs}}$ como un sistema independiente del programa, y por tanto independiente del significado que puedan tener los símbolos no primitivos. Nótese por otra parte que la corrección y la completitud de $\leadsto^{\widehat{cs}}$ implican en particular la corrección y completitud de $\leadsto^{cs}$.

Para analizar el significado y la necesidad de la condición de pereza, hagamos antes algunas consideraciones acerca de una forma obvia de conseguir, a partir de $\leadsto^{cs}$, una extensión $\leadsto^{\widehat{cs}}$ correcta y completa. Informalmente, para efectuar un paso $\varphi \leadsto^{\widehat{cs}} \psi$, siendo φ una restricción no primitiva, se realiza $\varphi' \leadsto^{\widehat{cs}} \psi'$, siendo φ' el resultado de reemplazar por variables nuevas las subexpresiones no primitivas más externas de φ; la restricción (posiblemente no primitiva) ψ se obtiene de efectuar el reemplazamiento inverso en ψ'.

Aplicando este mecanismo a nuestro ejemplo anterior ya obtendríamos, como deseábamos,

$$c(0, f(X)) =/= c(s(0), Y) \leadsto^{\widehat{cs}^*} true$$

supuesto que se tuviese (como parece razonable)

$$c(0, U) =/= c(s(0), Y) \leadsto^{cs^*} true$$

Pero por desgracia la corrección y completitud requeridas arriba para $\leadsto^{\widehat{cs}}$ no son suficientes – y por tanto tampoco el mecanismo descrito – para nuestros propósitos, al no garantizar que todas las soluciones de un objetivo queden cubiertas por un cómputo terminado. Dicho de otro modo, no queda asegurado que sea posible pasar de un cómputo semánticamente terminado a otro terminado. Veámoslo con un ejemplo.

Ejemplo 4.12

Sea $\mathcal{CS}^0 = \{0\}$, $\mathcal{CS}^1 = \{s\}$, y consideremos, en la instancia $CFLP(\mathcal{H}_{\neq})$ correspondiente a $\mathcal{CS}$, el programa que define la función constante $f/0$ por la regla

$$f = s(f)$$

Teniendo en cuenta que la función f denota el árbol infinito $s(s(s\ldots$, es claro que el objetivo φ dado por

$$X =/= f$$

admite como soluciones las infinitas valoraciones que dan a X los respectivos valores

$$0, s(0), s(s(0)), \ldots$$

Mediante un paso de estrechamiento

$$X =/= \underline{f} \leadsto X =/= s(f)$$

llegamos a un objetivo semánticamente terminado para la solución $\alpha(X) = 0$, pero sin embargo no terminado, pues $X =/= s(f)$ no es una restricción primitiva. Sin embargo, en

virtud del mecanismo propuesto más arriba para la extensión $\widehat{\leadsto^{cs}}$, $X =/= s(f)$ resultaría ser $\widehat{\leadsto^{cs}}$-irreducible, si se tiene en cuenta que la restricción primitiva $X =/= s(U)$ está en forma resuelta, de acuerdo con las reglas para $\mathcal{CS}$ que daremos en la sección 5.4. Así pues, la única forma de continuar el cómputo sería dar un nuevo paso de estrechamiento

$$X =/= s(\underline{f}) \leadsto X =/= s(s(f))$$

El nuevo objetivo está semánticamente terminado, pero una vez más no terminado, para las soluciones $\alpha(X) = 0$ y $\alpha(X) = s(0)$. Estaríamos en una situación similar a la de antes, en presencia de un objetivo no terminado y sin embargo $\widehat{\leadsto^{cs}}$-irreducible. Pasos de estrechamiento adicionales no resuelven la situación, por lo que concluimos que para el objetivo inicial φ, aun teniendo infinitas soluciones, no existen cómputos terminados. En resumen: se ha perdido la completitud (siempre asumiendo como extensión $\widehat{\leadsto^{cs}}$ la obtenida por el procedimiento de más arriba).

$\diamond$

La propiedad de pereza que pedimos a $\widehat{\leadsto^{cs}}$ está introducida precisamente para paliar este problema. La idea es que $\leadsto^{cs}$ permita pasar de objetivos semánticamente terminados a objetivos terminados, es decir, que se haga cargo de la eliminación de aquellas subexpresiones no primitivas cuya evaluación no sea necesaria.

Es fácil probar que, en el caso de que $\widehat{\leadsto^{cs}}$ sea terminante, la condición de pereza se puede expresar de un modo más simple.

Proposición 4.6

Sea $\widehat{\leadsto^{cs}}$ una extensión correcta, completa y terminante de $\leadsto^{cs}$. Entonces: $\widehat{\leadsto^{cs}}$ es perezosa si y solo si para toda φ no primitiva y $\widehat{\leadsto^{cs}}$-irreducible, se tiene que $\mid \varphi \mid$ es insatisfactible.

Ejemplo 4.13

Considerando el programa y objetivo del último ejemplo, veamos cómo debería comportarse la extensión $\widehat{\leadsto^{cs}}$ para ajustarse a la condición de pereza que acabamos de imponer.

Que el objetivo $\varphi \equiv \ X =/= s(f)$ esté semánticamente terminado para α dada por $\alpha(X) = 0$ quiere decir que $\alpha \models\mid \varphi \mid$, y por tanto $\mid \varphi \mid$ es satisfactible. Por la condición de pereza, φ debe ser $\widehat{\leadsto^{cs}}$-reducible (posiblemente en varios pasos) a una restricción primitiva que capture α. Las reglas que daremos en 5.4 para $\widehat{\leadsto^{cs}}$ en $CFLP(\mathcal{H}_{\neq})$ permitirán en efecto los dos siguientes $\widehat{\leadsto^{cs}}$-pasos alternativos para φ:

$$X =/= s(f) \widehat{\leadsto^{cs}} X = 0 \qquad \text{(Alternativa 1)}$$
$$X =/= s(f) \widehat{\leadsto^{cs}} X = s(U), U =/= f \qquad \text{(Alternativa 2)}$$

La primera alternativa es 'perezosa', en el sentido de que captura aquellas soluciones de φ para las que la evaluación de f no es necesaria, haciendo de hecho desaparecer f de φ.

Observemos que hemos llegado (en este caso en un solo paso, en general se necesitarán más) del objetivo semánticamente terminado φ a la forma resuelta $X = 0$, es decir, a un objetivo terminado.

La segunda alternativa cubre el resto de las infinitas soluciones de φ. Para continuar el cómputo por ella es preciso estrechar f para obtener

$$X = s(U), U =/= \underline{f} \leadsto^{\widehat{cs}} X = s(U), U =/= s(f)$$

Para este nuevo objetivo existen de nuevo dos $\leadsto^{\widehat{cs}}$-alternativas

$$X = s(U), U =/= s(f) \leadsto^{\widehat{cs}} X = s(U), U = 0$$
$$X = s(U), U =/= s(f) \leadsto^{\widehat{cs}} X = s(U), U = s(V), V =/= f$$

la primera de las cuales es ya primitiva y se puede $\leadsto^{cs}$-reducir a la forma resuelta $X = s(0)$. Reiterando el proceso se irían obteniendo el resto de las soluciones del objetivo.

$\diamond$

En cierto sentido, se puede afirmar que la distinción que hemos hecho entre $\leadsto^{cs}$ y $\leadsto^{\widehat{cs}}$ es meramente retórica, pues en definitiva es $\leadsto^{\widehat{cs}}$ (que incluye a $\leadsto^{cs}$), la que se combina con el estrechamiento $\leadsto$. No hay necesidad real de definirla como extensión de ningún sistema previo $\leadsto^{cs}$ [16]. Si lo hemos hecho así ha sido por motivos de claridad de la exposición, y para poner énfasis en lo que podría ser un proceso bastante natural de diseñar $\leadsto^{\widehat{cs}}$:

- En primer lugar se diseña $\leadsto^{cs}$, sin preocuparse en absoluto de que vaya a ser usada en combinación con $\leadsto$, ya que las propiedades que le pedimos dependen *exclusivamente* del dominio de base, independientemente de su uso en una instancia de $CFLP(X)$. De hecho, en muchos casos $\leadsto^{cs}$ puede ser un sistema conocido de antemano, cuya existencia sea precisamente la que justifique el interés de la instancia en cuestión.
- En segundo lugar, se examina qué modificaciones hacen falta en la extensión 'natural' de $\leadsto^{cs}$, para conseguir la condición de pereza.

Una vez clarificado el papel de la extensión $\leadsto^{\widehat{cs}}$ y las propiedades deseables para ella, incurriremos en un abuso más de notación y utilizaremos $\leadsto^{cs}$ en lugar de $\leadsto^{\widehat{cs}}$, aclarando cuando sea preciso si estamos ante una restricción primitiva o no.

4.4.2 Corrección y completitud

Ya estamos en condiciones de definir con precisión la semántica operacional refinada, que consiste simplemente en la combinación del estrechamiento por restricciones con la resolución

[16] En este caso, la primera condición de la definición 4.17 debería reemplazarse por: '$\leadsto^{\widehat{cs}}$, restringido a Con_Σ, es un sistema de semidecisión'. Como además la corrección y completitud de $\leadsto^{\widehat{cs}}$ implican la de su restricción a Con_Σ, la condición anterior queda reducida a la semiterminación y la existencia de formas resueltas.

de restricciones. Es decir, la relación que expresa *un paso de cómputo* viene dada por

$$\rightsquigarrow^{ncs} \equiv \rightsquigarrow \cup \rightsquigarrow^{cs}$$

para la que se tienen los siguientes resultados de corrección y completitud, análogos a los teoremas 4.1 y 4.4 para $\rightsquigarrow$.

Teorema 4.5 *(Corrección de $\rightsquigarrow^{ncs}$)*
Si $\varphi \rightsquigarrow^{ncs^*} \psi$ y α es solución de ψ, entonces α es también solución de φ.

Demostración:
Por una sencilla inducción sobre el número de pasos del cómputo $\varphi \rightsquigarrow^{ncs^*} \psi$. Para un paso $\varphi \rightsquigarrow^{ncs} \psi$ basta tener en cuenta la corrección de $\rightsquigarrow$ (teorema 4.1) y la propiedad de corrección pedida a $\rightsquigarrow^{cs}$. □

En el teorema siguiente se establece la completitud de la semántica operacional dada por la combinación $\rightsquigarrow^{ncs}$. En su enunciado, la noción de 'localmente perezoso' se refiere a los $\rightsquigarrow$-pasos de que conste un cómputo. No hay ningún inconveniente en aplicar tal noción a $\rightsquigarrow^{ncs}$-cómputos.

Teorema 4.6 *(Completitud de $\rightsquigarrow^{ncs}$)*
Sea α una solución de φ. Entonces existe un cómputo localmente perezoso $\varphi \rightsquigarrow^{ncs^*} \psi$ terminado para α.

Demostración:
Por el teorema 4.4 existe un cómputo $\varphi \rightsquigarrow^* \psi_0$ globalmente perezoso (por tanto localmente perezoso) y semánticamente terminado para α, es decir, $\alpha \models\mid \psi_0 \mid$. Por la condición de pereza, existe ψ_1 primitiva tal que $\psi_0 \rightsquigarrow^{cs^*} \psi_1$. Como $\rightsquigarrow^{cs}$, restringido al caso de restricciones primitivas, es un un sistema de semidecisión, se tiene, por la proposición 4.5, que existe una forma resuelta, ψ_2, para ψ_1, que tiene a α como solución. Eso quiere decir que hemos encontrado un cómputo terminado para α, que vendrá dado por

$$\varphi \rightsquigarrow^* \psi_0 \rightsquigarrow^{\widehat{cs}^*} \psi_1 \rightsquigarrow^{cs^*} \psi_2$$

□

Este teorema, aun siendo interesante, constituye un resultado no muy fuerte de completitud, pues no dice nada acerca de cómo intercalar pasos de estrechamiento $\rightsquigarrow$ con pasos de resolución de restricciones. El cómputo obtenido en la demostración tiene de hecho un aspecto bien poco 'combinado', pues consta de una serie de pasos de estrechamiento, seguidos de otros de resolución de restricciones. Eso no quiere decir, por supuesto, que no pueda haber otros cómputos de carácter más mixto que también terminen, pero lo cierto es que ni el teorema, ni su demostración dicen nada al respecto. Una parte importante de la segunda parte de este trabajo – en la que aplicaremos los resultados generales del esquema $CFLP(X)$ a la instancia $CFLP(\mathcal{H}_{\neq})$ – consistirá en obtener resultados de completitud más fuertes que éste. El

investigar condiciones sobre los sistemas de resolución de restricciones que permitan obtener, con carácter general, resultados de completitud más fuertes que el anterior, constituye sin duda un punto de interés para trabajos futuros.

Para terminar este apartado, revisamos el ejemplo que lo inició, intentando poner de manifiesto el hecho de que la semántica operacional combinada dada por $\leadsto^{ncs}$ proporciona cómputos mucho más razonables.

Ejemplo 4.14

Consideremos el mismo programa, objetivo y solución del ejemplo 4.10. Un posible $\leadsto^{ncs}$-cómputo sería el siguiente, en el que $\leadsto^{cs}$ es la relación definida por las reglas que se presentarán en la sección 5.4. Como las reglas usadas son muy razonables, abreviaremos algunos pasos en la forma $\leadsto^{cs^*}$.

$$
\begin{array}{ll}
 & \underline{ca}([A,B]) =/= N \qquad (\varphi_0)\\
\leadsto_{(S_3)} & \exists X, Xs([A,B] = [X \mid Xs], el(X,Xs) == f,\\
 & s(ca(Xs)) =/= N) \qquad (\varphi_1)\\
\leadsto^{cs} & \exists X, Xs(A = X, [B] = Xs, el(X,Xs) == f,\\
 & s(ca(Xs)) =/= N) \qquad (\varphi_2)\\
\leadsto^{cs^*} & \underline{el}(A,[B]) == f, s(ca([B])) =/= N \qquad (\varphi_3)\\
\leadsto_{(M_3)} & \exists Y, Ys([B] = [Y \mid Ys], A =/= Y,\\
 & el(A,Ys) == f, s(ca([B])) =/= N) \qquad (\varphi_4)\\
\leadsto^{cs^*} & A =/= B, \underline{el}(A,[]) == f, s(ca([B])) =/= N \qquad (\varphi_5)\\
\leadsto_{(M_1)} & A =/= B, [] = [], f == f, s(ca([B])) =/= N \qquad (\varphi_6)\\
\leadsto^{cs^*} & A =/= B, s(\underline{ca}([B])) =/= N \qquad (\varphi_7)\\
\leadsto_{(S_3)} & \exists X', Xs'(A =/= B, [B] = [X' \mid Xs'], el(X',Xs') == f,\\
 & s(s(ca(Xs'))) =/= N) \qquad (\varphi_8)\\
\leadsto^{cs^*} & A =/= B, \underline{el}(B,[]) == f, s(s(ca([])))) =/= N \qquad (\varphi_9)\\
\leadsto_{(M_1)} & A =/= B, [] = [], f == f, s(s(ca([])))) =/= N) \qquad (\varphi_{10})\\
\leadsto^{cs^*} & A =/= B, s(s(ca([])))) =/= N \qquad (\varphi_{11})\\
\leadsto^{cs} & \exists U(A =/= B, N = s(U), U =/= s(ca([])))) \qquad (\varphi_{12})\\
\leadsto^{cs} & \exists U(A =/= B, N = s(U), U = 0) \qquad (\varphi_{13})\\
\leadsto^{cs} & A =/= B, N = s(0) \qquad (\varphi_{14})
\end{array}
$$

La diferencia con el cómputo que inició este apartado es notoria. Ahora sí resulta claro que φ_{14} captura la solución α dada por $\alpha(A) = 0, \alpha(B) = s(0), \alpha(N) = s(0)$. Hemos detallado cuidadosamente los últimos pasos de resolución de restricciones para que se aprecie el paso desde φ_{10} (que ya estaba semánticamente terminado para α, pero contenía aún expresiones no primitivas), hasta φ_{14}, que es una restricción primitiva irreducible.

Es interesante observar que, en este ejemplo particular, una forma en principio más natural de terminar el cómputo a partir de φ_{11}, sería reducir $card([])$ a 0, para obtener el nuevo objetivo $A =/= B, s(s(0)) =/= N$, que ya estaría terminado, y además sería una respuesta más general que la obtenida en el cómputo mostrado. Debemos indicar en este punto que la posibilidad de haber procedido así está basada en la propiedad de que $card([\,])$ puede ser reducido a forma normal; pero esto no va suceder así en general, ya que hay expresiones que

denotan árboles infinitos. Por otra parte, tal propiedad, además de ser indecidible, depende por supuesto de las reglas del programa que definan las funciones, mientras que en nuestra concepción la resolución de restricciones $\leadsto^{cs}$ debe ser un mecanismo independiente de las reglas.

$\diamond$

Parte II

Estudio de $SFL_{\neq}$

5 El lenguaje $SFL_{\neq}$

En lo que resta de este trabajo nos dedicaremos al estudio detallado de $SFL_{\neq}$, un lenguaje en el que se combinan las características de un lenguaje lógico-funcional perezoso con el uso de desigualdades. El lenguaje $SFL_{\neq}$ está introducido en [8].

Son varios los objetivos que queremos cubrir al considerar este lenguaje concreto. En un lugar destacado se encuentra nuestro interés en comprobar si el esquema teórico $CFLP(X)$ que hemos investigado en la primera parte contribuye realmente al estudio y desarrollo de lenguajes de programación concretos. Como ya hemos visto, el esquema es desde luego bastante general, en el sentido de que muchos paradigmas de programación declarativa son expresables en él. Sería entonces demasiado esperar que, dada la relativa simplicidad con que hemos obtenido sus propiedades más importantes, todos los problemas (siquiera teóricos) relativos a un lenguaje quedasen automáticamente resueltos por el mero hecho de ser una instancia del esquema. Como veremos, no es así en el caso de $SFL_{\neq}$. Eso tiene sin embargo su interés, porque el proceso de ajuste (muy fino, en ocasiones) que tendremos que hacer para fijar $SFL_{\neq}$ como instancia de $CFLP(X)$ será cooperativo: a veces, la generalidad del esquema servirá para comprender mejor algunos aspectos que estaban ocultos por lo concreto del lenguaje; en otras ocasiones, la necesidad de abordar con métodos específicos problemas no resueltos por el esquema puede aportar luz a las limitaciones de éste último y sugerir formas de superarlas.

Como segundo objetivo importante está, por supuesto, el lenguaje en sí. La combinación de igualdades y desigualdades en un mismo sistema ha despertado interés en la literatura desde hace tiempo, y nuestro lenguaje es una contribución más en esa línea. Queremos además investigar el lenguaje de una forma bastante integral, desplazándonos de manera gradual desde la fundamentación matemática de sus propiedades teóricas, hasta los aspectos más concretos de su implementación.

5.1 Introducción

Muchas de las propuestas que se han hecho para la integración de la programación funcional y lógica (veáse la sección 1.2) utilizan *sistemas de reescritura condicionales* como programas y *estrechamiento* como mecanismo operacional para evaluación de expresiones o resolución de objetivos ecuacionales. En particular, los llamados lenguajes lógico-funcionales utilizan *estrechamiento perezoso* [131] como semántica operacional.

El estrechamiento es un mecanismo de cómputo que, al combinar unificación con reescritura, produce sustituciones – que pueden ser entendidas como ecuaciones $X = t$ – como respuestas. Lo que nos interesa de ello en este momento es destacar el hecho de que, como cualquier otro mecanismo que produzca sustituciones como respuestas (resolución, p. ej.), el estrechamiento puede necesitar obtener un número infinito de respuestas en algunos casos en los que el uso de información negativa podría reducir el conjunto de soluciones a un tamaño finito. Por ejemplo, la desigualdad $X \neq Y$ no puede ser reemplazada por ningún conjunto finito y equivalente de ecuaciones (salvo si el dominio sobre el que varían las variables es finito). Parece claro que desde el punto de vista de la capacidad expresiva es interesante utilizar desigualdades para expresar condiciones en los programas y, lo que es más importante,

permitir su aparición como parte de las respuestas (lo que obviamente exige mantenerlas durante los cómputos).

La idea de complementar la unificación – entendida como un proceso de resolución de ecuaciones sobre algún dominio de términos o árboles – con la resolución de desigualdades forma parte de la propia concepción de PROLOG II [31, 32, 33] – aceptado hoy día como el primer lenguaje de programación lógica con restricciones – y se mantiene en PROLOG III [34]. Otros lenguajes, aparecidos ya como instancias del esquema $CLP(X)$ [78, 79], incorporan también desigualdades, interpretadas sobre un cierto dominio de términos o árboles, a su lenguaje de restricciones. Ejemplos de ello son $CLP(FT)$ [143], cuyo dominio es el propio universo de Herbrand, o el lenguaje $\{\log\}$ [49, 22] cuyo dominio extiende el universo de Herbrand mediante una operación de construcción de conjuntos finitos. Por otra parte, sin estar expresamente relacionado con ningún lenguaje de programación, existe mucho trabajo teórico profundo sobre unificación y resolución de desigualdades (veáse, p. ej., [87, 35] como recopilaciones recientes).

Trabajos ya clásicos acerca de problemas ecuacionales [17] interpretados en distintas álgebras de árboles (finitos, racionales o infinitos) son [32, 97, 108, 36].

En [52] se da un procedimiento para resolver desigualdades (implícitamente cuantificadas existencialmente) interpretadas en un cociente del universo de Herbrand determinado por una teoría ecuacional E. El procedimiento, basado en estrechamiento, es completo si E viene dada por un sistema de reescritura convergente básico ('*ground convergent*'). El procedimiento, que requiere en general un costoso test de reducibilidad básica, puede hacerse más eficiente en caso de que el sistema esté basado en constructoras. Por utilizar estrechamiento como regla de cómputo, las soluciones obtenidas son sustituciones, no contémplandose la posibilidad de incluir desigualdades en las respuestas. Por otra parte, los sistemas de reescritura considerados son incondicionales.

Una situación similar se da en [15], donde se propone una ampliación del lenguage LPG [14] para permitir la aparición de desigualdades en las condiciones de las cláusulas que definen predicados. El uso de desigualdades no está contemplado en la definición de funciones, que se efectúa mediante reglas condicionales (sin variables extra). El procedimiento usado en este caso es una extensión de SLD-resolución, que produce sustituciones como respuestas.

En [129, 130] se adapta la propuesta de Stuckey [145] de negación constructiva en $CLP(X)$, dando como resultado un procedimiento, basado en estrechamiento, que resuelve constructivamente la negación con respecto a una teoría ecuacional que admite, a su vez, negaciones en las condiciones de las claúsulas. El procedimiento es correcto y completo para teorías normales canónicas, completamente definidas, con disciplina de constructoras y sin variables extra en las condiciones. El uso que se hace de la negación (desigualdades) es en este caso más expresivo que en los anteriores (y que en el nuestro) pues, como es natural en el contexto de la negación constructiva, incluye cuantificación universal. Un enfoque similar, basado en las ideas de negación constructiva, se realiza en [115], para un lenguaje lógico funcional con funciones estrictas.

Ninguna de las propuestas anteriores puede considerarse adecuada para nuestros propósitos. Aparte de los inconvenientes ya citados, ninguna de ellas sirve para expresar progra-

[17] Un *problema ecuacional* [36, 35] es una fórmula de primer orden cuyo único predicado es la igualdad.

mación lógico-funcional *perezosa*. El reflejo semántico de ello es que en los trabajos citados la igualdad y la desigualdad se interpretan como igualdad y desigualdad 'verdaderas' $(=, \neq)$ en el dominio de base, y en particular son una negación lógica de la otra. Como se desprende de algunas discusiones previas (ver 2.3.3), y en seguida volveremos a precisar, eso no es así en nuestro lenguaje $SFL_{\neq}$, cuya semántica requiere interpretaciones continuas para las operaciones predefinidas, incluyendo la igualdad y desigualdad que hayan de considerarse $(==, =/=)$. Entre otras consecuencias, eso supone que $==$ y $=/=$ no son negación una de otra.

Pasemos ya a presentar el lenguaje $SFL_{\neq}$.

5.2 Sintaxis de $SFL_{\neq}$. $SFL_{\neq}$-programas

Partimos de una signatura de primer orden, constituida por un conjunto de símbolos de constructora $\mathcal{CS} = \bigcup \mathcal{CS}^n$, junto con un conjunto de símbolos de función $\Delta = \bigcup \Delta^n$. $\mathcal{CS}^n$ y Δ^n son los conjuntos de símbolos (de constructora y función respectivamente) de aridad n. Suponemos que la constructora *true* forma parte de $\mathcal{CS}^0$, y que $\mathcal{CS}^n$ es no vacío para alguna aridad $n > 0$, es decir, hay al menos una constructora no constante [18]. Usaremos $c, d, \ldots$ como símbolos de $\mathcal{CS}$, y $f, g, \ldots$ para símbolos de Δ. Estos símbolos, en compañía de un conjunto numerable de variables V, determinan, como en otras ocasiones, los conjuntos $Term(\equiv \mathcal{T}_{\mathcal{CS}}(V))$ de *términos* (en los que aparecen variables y constructoras) y $Exp(\equiv \mathcal{T}_{\mathcal{CS} \cup \Delta}(V))$ de *expresiones* (en las que pueden aparecer también símbolos de función). Nótese que los términos son, para una signatura de constructoras, las expresiones primitivas que hemos considerado anteriormente para signaturas más generales.

Definición 5.1 *($SFL_{\neq}$-reglas y programas)*

(a) Una $SFL_{\neq}$-regla *para definir un símbolo de función* $f \in \Delta$ *tiene la forma*

$$f(t_1, \ldots, t_n) = e \Leftarrow Ig, Desig$$

donde

- $t_1, \ldots, t_n$ *son términos de* $Term$*, y la tupla* $\langle t_1, \ldots, t_n \rangle$ *es lineal (es decir, no hay una variable en ella que aparezca más de una vez). A* $f(t_1, \ldots, t_n)$ *se le llama* cabeza *de la regla y a* $t_1, \ldots, t_n$*,* patrones *de la regla.*
- e *es una expresión de* Exp*, que llamamos* cuerpo *de la regla.*
- Ig *es de la forma* $l_1 == r_1, \ldots, l_n == r_n$*, siendo* $l_1, r_1, \ldots, l_n, r_n$ *expresiones de* Exp*.*
- $Desig$ *es de la forma* $l'_1 =/= r'_1, \ldots, l'_m =/= r'_m$*, siendo* $l'_1, r'_1, \ldots, l'_m, r'_m$ *expresiones de* Exp*. A* $Ig, Desig$ *le llamamos* restricción o condición *de la regla.*

[18] Esta condición es necesaria para que el universo de Herbrand sea infinito, ya que los métodos que estudiaremos no son válidos para el caso de universos finitos. Si, por motivos prácticos, se consideran distintos géneros, cada uno de ellos debe cumplir la condición pedida al conjunto de constructoras.

(ii) *Un* $SFL_{\neq}$-programa *es un conjunto finito de reglas para los símbolos de* Δ.

(iii) *Un* $SFL_{\neq}$-objetivo inicial *es una restricción* $Ig, Desig$ *similar a la condición de una regla.*

Como ya hicimos en el caso general de $CFLP(X)$, no imponemos en la propia definición condiciones sintácticas de *no ambigüedad*. Preferimos establecer condiciones semánticas – que podremos expresar cuando dispongamos de una semántica declarativa – sin que ello impida que luego se adopten condiciones efectivas suficientes para garantizar aquéllas.

En una primera explicación informal del sentido de estas reglas, digamos que operacionalmente serán utilizadas como reglas de reescritura condicionales. El símbolo $==$ (*igualdad estricta*) en una condición $l == r$ expresa que l y r pueden ser reducidos a términos unificables. El símbolo $=/=$ (*desigualdad o inconsistencia*) en una condición $l =/= r$ expresa que l y r pueden ser reducidos hasta un punto en el que se pueda detectar la aparición de símbolos de constructora diferentes a ambos lados. El motivo de que no usemos los símbolos $=$ y $\neq$, y de que no hablemos sin más de igualdad y desigualdad, es que, como veremos en seguida, $==$ y $=/=$ no van a ser interpretados como la igualdad $=$ y desigualdad $\neq$ 'de verdad' en el dominio de cómputo que vamos a fijar.

Veamos antes un par de ejemplos sencillos de $SFL_{\neq}$-programas. El primero, ya utilizado en 4.10 para otros propósitos, nos muestra el uso de las desigualdades en programas y respuestas. El segundo, tomado de [103], pone de manifiesto que $SFL_{\neq}$ es una extensión del (fragmento de primer orden del) lenguaje lógico-funcional perezoso SFL [63, 64, 61], y como tal puede manejar objetos infinitos. Aunque SFL no es lo específico de nuestra aportación, no está de más recordarlo con un ejemplo.

Ejemplo 5.1

Consideramos los conjuntos de símbolos $\mathcal{CS}, \Delta$ dados por

$$\begin{array}{lll} \mathcal{CS}^0 = \{true, false, 0, [\,]\} & \mathcal{CS}^1 = \{s\} & \mathcal{CS}^2 = \{[.|.]\} \\ \Delta^1 = \{card/1\} & \Delta^2 = \{elem/2\} & \end{array}$$

Las funciones *card/1* y *elem/2* están definidas mediante las reglas

$$\begin{array}{lcll} elem(X,[\,]) & = & false. & (M_1) \\ elem(X,[Y \mid Ys]) & = & true \Leftarrow X == Y. & (M_2) \\ elem(X,[Y \mid Ys]) & = & elem(X,Ys) \Leftarrow X =/= Y. & (M_3) \end{array}$$

$$\begin{array}{lcll} card([\,]) & = & 0. & (S_1) \\ card([X \mid Xs]) & = & card(Xs) \Leftarrow elem(X,Xs) == true. & (S_2) \\ card([X \mid Xs]) & = & s(card(Xs)) \Leftarrow elem(X,Xs) == false. & (S_3) \end{array}$$

Un objetivo para este programa podría ser $card([X,Y]) == N$, para el que esperaríamos como respuestas $X == Y, N == s(0)$ y $X =/= Y, N == s(s(0))$. Nótese de paso cómo al escribir objetivos de la forma $e == N$ se fuerza la reducción de e hasta conseguir un término, que se recoge en N.

$\diamond$

Ejemplo 5.2

Consideramos los conjuntos de símbolos $\mathcal{CS}, \Delta$ dados por

$$\begin{array}{lll} \mathcal{CS}^0 = \{true, false, 0, [\,]\} & \mathcal{CS}^1 = \{s\} & \mathcal{CS}^2 = \{[.|.]\} \\ \Delta^1 = \{nats\_desde/1\} & \Delta^2 = \{menor\_ig/2, corta/2\} & \end{array}$$

Las funciones de Δ están definidas mediante las reglas

$$\begin{array}{lcll} menor\_ig(0,Y) & = & true. & (M_1) \\ menor\_ig(s(X),0) & = & false. & (M_2) \\ menor\_ig(s(X),s(Y)) & = & menor\_ig(X,Y). & (M_3) \end{array}$$

$$\begin{array}{lcll} corta(N,[\,]) & = & [\,]. & (C_1) \\ corta(N,[X|Xs]) & = & [X] \Leftarrow menor\_ig(N,X) == true. & (C_2) \\ corta(N,[X|Xs]) & = & [X|corta(N,Xs)] \Leftarrow menor\_ig(N,X) == false. & (C_3) \end{array}$$

$$nats\_desde(N) = [N|nats\_desde(s(N))]. \quad (N)$$

y su significado es claro:

$menor\_ig(n,m)$ determina si el número natural n – representado mediante las constructoras 0 y s – es menor o igual que m.

$corta(n,l)$ devuelve un segmento inicial de la lista l (posiblemente infinita), cortada en el primer elemento mayor o igual que n.

$nats\_desde(n)$ genera la lista infinita de los naturales, a partir de n.

Un objetivo sería $corta(N, nats\_desde(0)) == [0, s(0)]$, para el que esperamos la respuesta $N = s(0)$.

$\diamond$

5.3 $SFL_{\neq}$ como instancia de $CFLP(X)$

Para caracterizar $SFL_{\neq}$ como instancia de nuestro esquema, debemos precisar cuál es el dominio de cómputo que se considera, y cómo se interpretan las funciones y predicados primitivos de los que se disponga.

Fijado el conjunto de constructoras $\mathcal{CS}$, y dado un nuevo símbolo $\perp \notin \mathcal{CS}$, el dominio de base es el universo de Herbrand infinitario $\mathcal{H}$ definido por $\mathcal{CS} \cup \{\perp\}$, es decir, el conjunto de los árboles finitos o infinitos, total o parcialmente definidos, construidos con ayuda de $\mathcal{CS} \cup \{\perp\}$.

El orden en $\mathcal{H}$ es el menor orden $\sqsubseteq$ que verifica

- $\perp \sqsubseteq t, \forall t \in \mathcal{H}$, es decir $\perp$ es el elemento mínimo, y representa el árbol totalmente indefinido

- $c \in \mathcal{CS}^n, t_1 \sqsubseteq s_1, \ldots, t_n \sqsubseteq s_n \Rightarrow c(t_1, \ldots, t_n) \sqsubseteq c(s_1, \ldots, s_n)$.

Con este orden los elementos finitos del dominio son los árboles finitos, y los elementos totales son los árboles totalmente definidos, es decir, los que no contienen $\perp$ en sus hojas.

El repertorio de símbolos de función y predicado primitivos consta al menos de $\mathcal{CS}$ como símbolos de función primitiva, y $==, =/=$ como símbolos de predicado primitivo. A ellos habrá que añadir las operaciones que sean necesarias para expresar la unificación con los patrones de las reglas. Posponiendo de momento este asunto, podemos ya fijar la interpretación de las anteriores. Sólo en la definición usaremos el superíndice que distingue un símbolo de su interpretación.

- Los símbolos de $\mathcal{CS}$ están interpretados como constructoras libres no estrictas, es decir: si $c \in \mathcal{CS}^n$
$$c^{\mathcal{H}}(x_1, \ldots, x_n) = c(x_1, \ldots, x_n), \quad \forall \overline{x} \in \mathcal{H}^n$$

- El símbolo $==$ se interpreta como la *igualdad estricta* o *continua* definida por
$$(x_1 ==^{\mathcal{H}} x_2) = \begin{cases} true & \text{si } x_1, x_2 \text{ son iguales, finitos y totales} \\ \perp_{bool} & \text{e.o.c.} \end{cases}$$

- El símbolo $=/=$ se interpreta como la *desigualdad continua* o *inconsistencia* definida por
$$(x_1 =/=^{\mathcal{H}} x_2) = \begin{cases} true & \text{si } x_1, x_2 \text{ son inconsistentes en } \mathcal{H} \\ \perp_{bool} & \text{e.o.c.} \end{cases}$$

  Inconsistencia de x_1 y x_2 en un dominio quiere decir inexistencia de cota superior común (para el orden del dominio). Por el orden definido en $\mathcal{H}$, esto equivale a que para al menos una posición, x_1 y x_2 difieren en alguna constructora. Eso puede suceder aunque x_1 o x_2 sean parciales o infinitos. Por ejemplo, $c(\perp, a)$ y $c(\perp, d(\perp))$ son inconsistentes pues hay un conflicto de constructoras entre a y $d(\perp)$. Así mismo sucede con $c(\perp, a)$ y $c(\perp, d(d(d(\cdots))))$. En ambos casos la desigualdad $=/=$ tendría valor $true$. No pasa lo mismo con $c(\perp, a)$ y $c(b, \perp)$, que son ambos aproximaciones de la cota superior común $c(b, a)$; en este caso, a pesar de que son árboles distintos (o sea, $\neq$ es $true$), la desigualdad $=/=$ no se verifica (queda igual a $\perp$).

  Es fácil comprobar que estas operaciones son continuas. Es más, no es difícil probar que $==, =/=$ son las máximas (con relación al orden $\sqsubseteq$ de las funciones de $\mathcal{H}^n$ en $\mathcal{H}$) aproximaciones continuas a la igualdad $=$ y desigualdad $\neq$ 'de verdad' en $\mathcal{H}$. Aclaremos algo esta afirmación. Con nuestra visión de los predicados como funciones bivaluadas sobre el dominio $\{true, \perp\}$, la igualdad $=$ y su negación $\neq$ habrían de entenderse como las funciones
$$(x_1 = x_2) = \begin{cases} true & \text{si } x_1, x_2 \text{ son el mismo árbol de } \mathcal{H} \\ \perp_{bool} & \text{e.o.c.} \end{cases}$$
$$(x_1 \neq x_2) = \begin{cases} true & \text{si } x_1, x_2 \text{ no son el mismo árbol de } \mathcal{H} \\ \perp_{bool} & \text{e.o.c.} \end{cases}$$

Nótese que $\neq$ es por supuesto la negación de $=$, en el sentido de que

$$(x_1 \neq x_2) \text{ es } true \Leftrightarrow (x_1 = x_2) \text{ no es } true$$

Lo que ocurre con $=$ y $\neq$ es que no son computables, lo que se refleja en el hecho de que no son continuas, ni siquiera monótonas. En efecto, al verificarse que $\bot = \bot$ es $true$, la monotonía exigiría que $x = y$ fuese también $true$ para árboles x, y cualesquiera (pues $\bot$ es el mínimo elemento), cosa obviamente falsa. De modo similar, como $\bot \neq x$ es $true$ para cualquier árbol $x \in \mathcal{H}$ distinto de $\bot$, por monotonía se tendría que $x \neq x$ debería ser $true$.

Que $==$ y $=/=$ son aproximaciones (como funciones) de $=$ y $=/=$ es claro, así como la lectura lógica de este hecho:

$$x == y \Rightarrow x = y$$
$$x =/= y \Rightarrow x \neq y$$

El que $==$ y $=/=$ sean máximas quiere decir que no podemos conseguir capturar más casos de $=$ y $\neq$, y preservar al tiempo la continuidad.

Nótese también que $=/=$ no es la negación de $==$, pues, por ejemplo, ni $\bot == \bot$ ni $\bot =/= \bot$ son $true$.

Nuestro siguiente paso es esclarecer el asunto de la unificación con los patrones de la cabeza de una $SFL_{\neq}$-regla. Son totalmente pertinentes aquí, y no las vamos a repetir, la discusión y motivaciones que dimos en 2.3.3 acerca del uso de predicados *reconocedores* is_c y funciones *selectoras* c_k para expresar el doble papel de la unificación con los patrones de una $SFL_{\neq}$-regla: imponer condiciones de aplicabilidad a la regla, y expresar acceso a las componentes de los argumentos. Recordemos sus definiciones: Para cada $c \in \mathcal{CS}^n, k \in \{1, \ldots, n\}, t \in \mathcal{H}$

$$c_k(t) = \begin{cases} t_k & \text{si } t = c(t_1, \ldots, t_n) \\ \bot & \text{e.o.c.} \end{cases}$$
$$is\_c(t) = \begin{cases} true & \text{si } t = c(t_1, \ldots, t_n) \\ \bot_{bool} & \text{e.o.c.} \end{cases}$$

Es muy fácil comprobar que estas operaciones son continuas.

Recordemos también con un ejemplo cómo sería la traducción de una regla. La regla (S_2) del ejemplo 4.10 quedaría así (usamos *cons* como nombre alternativo del constructor de listas):

$$card(L) \;=\; card(cons_2(L)) \Leftarrow is\_cons(L), elem(cons_1(L), cons_2(L)) == true$$

Si consideramos la signatura primitiva Σ que resulta de añadir las nuevas operaciones is_c, c_k a las que ya teníamos $\mathcal{CS} \cup \{==, =/=\}$, es claro que estamos ante una instancia del esquema $CFLP(X)$, digamos $CFLP(\mathcal{H}_\Sigma)$, y por tanto el lenguaje correspondiente heredaría todas sus propiedades. En particular, si consideramos a $SFL_{\neq}$ como una variante sintáctica de $CFLP(\mathcal{H}_\Sigma)$, ya habríamos dotado a nuestro lenguaje de una semántica declarativa precisa.

¿Y en cuanto a la semántica operacional? Sin duda, los resultados de corrección y completitud obtenidos para el estrechamiento con restricciones ($\leadsto$) en 4.3 serían de inmediata aplicación a $CFLP(\mathcal{H}_\Sigma)$, y también los relativos a la combinación de $\leadsto$ con un resolvedor de restricciones $\leadsto^{cs}$. Ahí reside el problema: no parece muy natural dedicarse a diseñar un sistema de resolución de restricciones que utiliza algunas operaciones primitivas elegidas muy artificialmente para ajustarnos al esquema.

Cambiemos un poco nuestro punto de vista: si nuestro problema es la unificación, que tradicionalmente se contempla como un problema de resolución de ecuaciones, ¿por qué no intentar seguir la tradición? Al fin y al cabo, al pedir que un argumento a unifique con un patrón t, estamos pidiendo que se verifique $a = t$ para ciertos valores de las variables que intervienen. Así pues, parece al menos intuitivamente correcto, considerar una $SFL_{\neq}$-regla

$$f(t_1, \ldots, t_n) = e \Leftarrow Ig, Desig$$

como una abreviatura sintáctica de la regla

$$f(X_1, \ldots, X_n) = e \Leftarrow X_1 = t_1, \ldots, X_n = t_n, Ig, Desig$$

donde $X_1, \ldots, X_n$ son variables nuevas.

Las reglas de esta forma están al menos en la *sintaxis* de CFLP. De hecho, si consideramos como signatura primitiva $\Sigma_= \equiv \mathcal{CS} \cup \{=, ==, =/=\}$, podríamos hablar de una 'instancia sintáctica' del esquema, digamos $CFLP(\mathcal{H}_{\Sigma_=})$, en la que los predicados primitivos $=, ==, =/=$ están interpretados como igualdad, igualdad estricta e inconsistencia. Aunque no podemos aplicar sin más los resultados para el esquema, ya que $=$ no es continua, nótese que las nociones de interpretación, modelo, operador $T_\mathcal{P}$, etc, tienen perfecto sentido. El utilizar $=$ en la sintaxis no sólo simplifica la traducción; más importante es el hecho de que nos permitirá proponer un resolvedor de restricciones sencillo, natural e intuitivo.

Tenemos entonces, para un $SFL_{\neq}$-programa, dos lecturas alternativas:

- Una como programa en $CFLP(\mathcal{H}_\Sigma)$, con propiedades bien establecidas, pero muy artificial.
- Otra como programa en $CFLP(\mathcal{H}_{\Sigma_=})$, más natural, pero de la que no tenemos resultados previos.

Obviamente, nuestro objetivo es ahora probar que ambas lecturas son equivalentes [19].

Fijemos algunas notaciones que usaremos hasta que aclaremos la relación entre ambos tipos de programas. Asumiendo un conjunto prefijado de constructoras $\mathcal{CS}$ y otro de símbolos de función Δ, utilizaremos $\Sigma_=$, Σ y $\widehat{\Sigma}$ para notar, respectivamente: la signatura que usa $=$ para expresar la unificación, la que lo hace mediante las funciones selectoras c_k y los predicados reconocedores is_c, y la mezcla de ambas. Es decir

$$\begin{array}{lcl} \Sigma_= & \equiv & \mathcal{CS} \cup \{=, ==, =/=\} \cup \Delta \\ \Sigma & \equiv & \mathcal{CS} \cup \{c_k \mid c \in \mathcal{CS}^n, 1 \leq k \leq n\} \cup \{is_c \mid c \in \mathcal{CS}^n\} \cup \{==, =/=\} \cup \Delta \\ \widehat{\Sigma} & \equiv & \Sigma_= \cup \Sigma \end{array}$$

[19] Mediante $CFLP(\mathcal{H}_{\neq})$, notación que ya hemos utilizado en alguna ocasión, nos referimos indistintamente a las instancias equivalentes $CFLP(\mathcal{H}_\Sigma)$ y $CFLP(\mathcal{H}_{\Sigma_=})$. Si escribimos $\mathcal{H}_{\neq}$ es para resaltar el hecho de que el dominio es un universo de Herbrand, y que la operación primitiva más reseñable es la desigualdad.

Utilizaremos $\Sigma_{=}$, Σ y $\widehat{\Sigma}$ como cualificadores cuando hablemos de reglas, programas, etc. En particular, la clase de restricciones correspondiente a cada una de esas signaturas vendrá notada respectivamente por $Con_{\Sigma_{=}}$, Con_{Σ} y $Con_{\widehat{\Sigma}}$.

En la siguiente definición indicamos con precisión cómo 'desmaquillar' una $SFL_{\neq}$-regla para eliminar los patrones de la cabeza de la regla, y convertirla así a la sintaxis de $CFLP$. Damos los dos niveles de traducción a dicha sintaxis: el primero, inmediato, traduce a la signatura 'impura' $\Sigma_{=}$; el segundo nivel, a la signatura $CFLP$-pura Σ, requiere unas transformaciones algo más complicadas. La justificación de la correción de tal definición, y de la equivalencia entre ambas traducciones vendrá a continuación.

Definición 5.2 *(Traducción de $SFL_{\neq}$-reglas)*
Dada una $SFL_{\neq}$-regla $R_0 \equiv \quad f(t_1,\ldots,t_n) = e_0 \Leftarrow \varphi_0$, llamamos

(a) $\Sigma_{=}$-traducción *de R_0 a la $\Sigma_{=}$-regla*

$$R_1 \equiv \quad f(X_1,\ldots,X_n) = e_0 \Leftarrow X_1 = t_1,\ldots,X_n = t_n,\varphi_0$$

siendo $X_1,\ldots,X_n$ variables que no están en R_0.

(b) Σ-traducción (o $CFLP$-traducción) *de R_0 a una Σ-regla R_2 que provenga de aplicar reiteradamente a partir de la regla R_1 definida en* (a) *las transformaciones siguientes:*

$$\begin{array}{ll}(T_1) & f(\overline{X}) = e \Leftarrow l = V,\varphi \;\longmapsto\; f(\overline{X}) = (e \Leftarrow \varphi)\sigma \\ & \text{siendo } \sigma \equiv \{V/l\} \\ (T_2) & f(\overline{X}) = e \Leftarrow l = c(t_1,\ldots,t_n),\varphi \\ & \longmapsto f(\overline{X}) = e \Leftarrow is_c(l), c_1(l) = t_1,\ldots,c_n(l) = t_n,\varphi\end{array}$$

La $\Sigma_{=}$-traducción y Σ-traducción de un $SFL_{\neq}$-programa se obtienen traduciendo cada una de sus reglas.

Nótese que las transformaciones T_1 y T_2 están definidas en general sobre $\widehat{\Sigma}$-reglas, que son las que se obtienen en los pasos intermedios. El objeto de tales transformaciones ya ha sido comentado en otras ocasiones: sustituir las condiciones $X = t$ por sucesivas condiciones is_c que reflejen las distintas constructoras que aparecen en t, al tiempo que se van obteniendo las variables de t como accesos a componentes de X (es decir composiciones de la forma $c_i(d_j(\cdots(X)\cdots))$). Debemos probar que a partir de R_1 toda secuencia de aplicaciones de las transformaciones anteriores termina, y además en una regla R_2 que no usa $=$, o sea, una Σ-regla. Es más, no es difícil probar que tal regla es única (salvo el orden de aparición de los átomos en la restricción de la regla), pero eso no nos es necesario.

Lema 5.1

Sean R_0 y R_1 como en la definición 5.2. Entonces

(i) $\longmapsto$ es terminante a partir de R_1.

(ii) Si $R_1 \longmapsto^* R$ y R tiene alguna $=$-ecuación, entonces R es $\longmapsto$-reducible.

Demostración:
Es fácil probar, por inducción sobre el número de transformaciones, algunas propiedades acerca de una regla R tal que $R_1 \longmapsto^* R$. Se tiene que R es de la forma

$$f(X_1,\ldots,X_n) = e \Leftarrow l_1 = r_1,\ldots,l_m = r_m,\varphi$$

donde $X_1,\ldots,X_n$, son las mismas de la cabeza de R_1, $m \geq 0$, φ no tiene $=$-ecuaciones, $\langle r_1,\ldots,r_m\rangle$ es una tupla lineal de subtérminos de $t_1,\ldots,t_n$ (los patrones de la regla original R_0, lo que en particular supone que $var(\langle r_1,\ldots,r_m\rangle) \cap \{X_1,\ldots,X_n\} = \emptyset$), y cada l_i es una de las variables X_j, o un acceso a ella de la forma $c_k(d_h(\cdots(X_j)\cdots))$.

Pero entonces, si $m > 0$ (o sea, si R contiene una ecuación $l_i = r_i$), como r_i es un $\Sigma_=$-término por ser subtérmino de un t_j, se tiene que (T_1) o (T_2) son aplicables a R (no sería así si r_i pudiese empezar por una selectora c_k). Esto prueba (ii).

Por otra parte, si $R \longmapsto_{T_2} R'$, el tamaño total de los lados derechos de las $=$-ecuaciones de R' es menor que el de las de R. Esto también sucede para (T_1), pues la sustitución σ aplicada no afecta a las $=$-ecuaciones de R (por ser los lados derechos lineales en su conjunto, y con variables separadas de los lados izquierdos), por lo que concluimos que no puede haber $\longmapsto$-reducciones infinitas a partir de R_1, lo que prueba (i). □

Para formular la equivalencia de las traducciones, introducimos la siguiente notación.

Notación 5.1

Dada una $\hat{\Sigma}$-regla $R \equiv \quad f(X_1,\ldots,X_n) = e \Leftarrow \varphi$ notaremos por $S(R,I,\overline{x})$ al conjunto $\{\alpha[\![e]\!]_I \mid \alpha(\overline{X}) = \overline{x}, \alpha \models^I \varphi\}$.

El siguiente lema expresa técnicamente la equivalencia regla a regla de las dos traducciones.

Lema 5.2

Sea R_0 una $SFL_{\neq}$-regla, y R_1, R_2 la $\Sigma_=$-traducción y Σ-traducción, respectivamente, de R_0. Entonces: $\forall \overline{x} \in \mathcal{H}^n, I \in \mathcal{INT}, S(R_1,I,\overline{x}) = S(R_2,I,\overline{x})$.

Demostración:
Basta obviamente probar que cada paso de la transformación $\longmapsto$ preserva el conjunto S, es decir, que si $R \longmapsto R'$, siendo R alcanzable desde R_1, entonces $S(R,I,\overline{x}) = S(R',I,\overline{x})$. Para ello razonamos por separado acerca de T_1 y T_2.

- $R \longmapsto_{T_1} R'$, es decir, R es de la forma $f(\overline{X}) = e \Leftarrow l = V, \varphi$ y R' es $f(\overline{X}) = (e \Leftarrow \varphi)\sigma$, siendo $\sigma \equiv \{V/l\}$. Dadas $I, \overline{x}$, sea $a \in S(R,I,\overline{x})$, lo que por definición significa que $a = \alpha[\![e]\!]_I$ para cierta α tal que $\alpha(\overline{X}) = \overline{x}$ y $\alpha \models^I l = V, \varphi$. En particular, debe ser $\alpha(V) = \alpha[\![l]\!]_I$, y por el lema de sustitución 4.1 podemos concluir que $\alpha \models^I \varphi\sigma$ y $\alpha[\![e\sigma]\!]_I = \alpha[\![e]\!]_I = a$, es decir, $a \in S(R',I,\overline{x})$. Ya tenemos pues $S(R,I,\overline{x}) \subset S(R',I,\overline{x})$.
  Para la otra inclusión, sea $a \in S(R',I,\overline{x})$ con α tal que $\alpha(\overline{X}) = \overline{x}$, $\alpha \models^I \varphi\sigma$ y $a = \alpha[\![e\sigma]\!]_I$. Basta definir α' de modo que coincida con α excepto en la variable V (que no aparece en R'), para la que definimos $\alpha'(V) = \alpha[\![l]\!]_I$. Es fácil probar, como antes, que $\alpha' \models^I \varphi$ y $\alpha'[\![e]\!]_I = \alpha'[\![e\sigma]\!]_I = \alpha[\![e\sigma]\!]_I = a$, es decir, $a \in S(R,I,\overline{x})$.

- $R \longmapsto_{T_2} R'$: es claro, por la semántica de los predicados is_c y las funciones c_k, que las restricciones respectivas de R y R' tienen exactamente las mismas soluciones. Dado que el cuerpo de R y R' es también el mismo, se tiene $S(R, I, \overline{x}) = S(R', I, \overline{x})$.

□

El siguiente corolario muestra con mucha mayor claridad que todo $\Sigma_=$-programa que provenga de traducir un $SFL_{\neq}$-programa, tiene un Σ-programa equivalente. Queda claro pues que la unificación con patrones lineales encubre operaciones continuas, que pueden ser expresadas explícitamente en la sintaxis mediante las funciones c_k y los predicados is_c, o bien mediante la utilización, más cómoda y legible, de la igualdad $=$, que no es continua, pero de la que se está haciendo un 'uso continuo'. La demostración, a pesar de lo aparatoso de la notación que interviene, es muy sencilla.

Corolario 5.1 *(Equivalencia de $\Sigma_=$-programas y Σ-programas)*

Sea P_0 un $SFL_{\neq}$-programa, P y P' su $\Sigma_=$-traducción y Σ-traducción respectivamente. Entonces

(i) $T_P = T_{P'}$

(ii) P y P' tienen los mismos modelos.

(iii) El modelo mínimo de P y P' viene dado por $T_{\mathcal{P}} \uparrow \omega (= T_{P'} \uparrow \omega)$

Demostración:
Dadas $f \in \Delta^n, \overline{x} \in \mathcal{H}^n, I \in \mathcal{INT}$, consideremos los conjuntos $C_{\mathcal{P}}(I, f, \overline{x})$ y $C_{\mathcal{P}'}(I, f, \overline{x})$ (definidos en 3.1) correspondientes a los programas P y P' respectivamente. Si $\{R_1, \ldots, R_k\}$ y $\{R_1', \ldots, R_k'\}$ son los conjuntos de reglas para f de P y P', se tiene

$$\begin{array}{lcl} C_{\mathcal{P}}(I, f, \overline{x}) & = & \bigcup_{i=1}^{i=k} S(R_i, I, \overline{x}) \\ C_{\mathcal{P}'}(I, f, \overline{x}) & = & \bigcup_{i=1}^{i=k} S(R_i', I, \overline{x}) \end{array}$$

siendo los conjuntos S los introducidos más arriba. Pero por el resultado anterior, tenemos $S(R_i, I, \overline{x}) = S(R_i', I, \overline{x}), \forall i = 1 \ldots k$, y por tanto

$$(1) \quad C_{\mathcal{P}}(I, f, \overline{x}) = C_{\mathcal{P}'}(I, f, \overline{x})$$

Ahora ya es fácil continuar la demostración.

(i) Si recordamos que por definición $f^{T_{\mathcal{P}}(I)}(\overline{x}) \simeq \sqcup C_{\mathcal{P}}(I, f, \overline{x})$, y análogamente para P', es obvio, por (1), que $T_{\mathcal{P}} = T_{\mathcal{P}'}$.

(ii) Se deduce de (1) y de la Prop. 3.2 que caracteriza los modelos de un programa en términos de los conjuntos C. Hay que observar que dicha proposición no requiere la continuidad de las funciones primitivas usadas, sino sólo que el dominio base sea un dominio de Scott.

(iii) En virtud del Teor. 3.2, el modelo mínimo de P' viene dado por $I_{\mathcal{P}'} \equiv T_{P'} \uparrow \omega$. Pero de acuerdo con *(ii)*, $I_{\mathcal{P}'}$ es también el modelo mínimo de P, y según *(i)*, $T_{P'} \uparrow \omega = T_{\mathcal{P}} \uparrow \omega$, como queríamos. Obsérvese que la demostración del Teor. 3.2 sí usa la continuidad de las

funciones primitivas (utiliza la continuidad de la evaluación), y por eso ha sido importante obtener previamente la igualdad de los operadores. □

En lo que sigue, asumimos que la signatura $\Sigma_=$ es la utilizada para los programas, restricciones, etc. Por supuesto, el uso que se haga de la igualdad no continua $=$ está restringido a ser un uso 'continuo'. En cuanto a los programas, eso quiere decir que consideramos exclusivamente $\Sigma_=$-programas que sean $\Sigma_=$-traducciones de $SFL_{\neq}$-programas. Obsérvese que, de acuerdo con la definición 5.2, las $=$-ecuaciones que pueden aparecer en la restricción de la regla de un programa se ajustan a una sintaxis muy sencilla y rígida

$$X_1 = t_1, \ldots, X_n = t_n$$

donde $t_1, \ldots, t_n$ son los patrones de la cabeza de una $SFL_{\neq}$-regla (y por tanto, $\langle t_1, \ldots, t_n \rangle$ es una tupla lineal de términos primitivos) y $X_1, \ldots, X_n$ son variables nuevas introducidas en la $\Sigma_=$-traducción. En general, durante los cómputos se van a ir generando restricciones en las que, aun cuando se haga un uso 'continuo' de $=$, el aspecto de las $=$-ecuaciones no va a ser tan simple. Nuestro objetivo inmediato es definir con precisión la clase de $\Sigma_=$-restricciones (o sea, la subclase de $Con_{\Sigma_=}$) que vamos a permitir (llamaremos *aceptables* a dichas $\Sigma_=$-restricciones). La definición se apoya en la siguiente noción auxiliar.

Definición 5.3 *(Ciclos de variables)*

Una restricción $\varphi \in Con_{\Sigma_=}$ contiene ciclos de variables *si existen ecuaciones de* φ, $l_1 = r_1, \ldots, l_n = r_n$, *y variables* $X_1, \ldots, X_n$ *de modo que* $X_1 \in var(l_1) \cap var(r_n)$, *y* $X_i \in var(r_{i-1}) \cap var(l_i)$, *para* $1 < i \leq n$.

Una forma algo más legible de indicar que hay un ciclo de variables es

$$\begin{array}{ccccccccccc} X_1 & & X_2 & X_2 & & X_3 & & X_n & & X_1 \\ l_1 & = & r_1 & , \; l_2 & = & r_2 & , \ldots , & l_n & = & r_n \end{array}$$

Obsérvese que cualquier permutación cíclica de las ecuaciones de arriba sirve para manifestar también la existencia de un ciclo de variables.

Nótese, por otra parte, que una ecuación $X = t$ (o $t = X$) en la que $X \in var(t)$ es un ciclo de variables. Por tanto, la inexistencia de ciclos de variables en todas las restricciones que aparezcan durante un cómputo será una forma de garantizar que se puede realizar unificación sin necesidad de 'occur-check', como sucederá en las reglas que daremos para resolver $=$-ecuaciones. La noción de ciclo de variables 'prevé' la posibilidad de que en el futuro se generen ecuaciones del tipo anterior, como sucedería por ejemplo en $X = s(Y), Y = s(X)$ (que contiene un ciclo de variables), si se sustituyese X por $s(Y)$, para obtener $Y = s(s(Y))$.

Definición 5.4 *(Restricciones aceptables)*

(a) Una restricción $\varphi \in Con_{\Sigma_=}$ *de la forma* $\exists \overline{U}(e_1 = t_1, \ldots, e_n = t_n)$ *es* aceptable *si verifica:*

(i) $t_1, \ldots, t_n$ *son términos y* $\langle t_1, \ldots, t_n \rangle$ *es lineal.*

(ii) Todas las variables de $\langle t_1, \ldots, t_n \rangle$ *están en* $\overline{U}$.

(ii) φ no contiene ciclos de variables.

(b) Una restricción $\varphi \in Con_{\Sigma_=}$ de la forma $\exists\overline{U}(Unif, \psi)$, donde $Unif$ consta de las ecuaciones $=$ de φ, es aceptable *si $Unif$ lo es.*

Las condiciones impuestas a las $=$-ecuaciones en las restricciones aceptables son una generalización de las verificadas por las $=$-ecuaciones de la $\Sigma_=$-traducción de una $SFL_{\neq}$-regla. Es de notar la asimetría de las $=$-ecuaciones, cuyos lados derechos van a contener patrones lineales con los que se quiere unificar, mientras que los izquierdos pueden ser expresiones no primitivas.

Es obvio que todo objetivo inicial de un cómputo, al no contener $=$-ecuaciones, es aceptable. De hecho, en adelante aceptamos cualquier restricción aceptable como objetivo inicial.

De las restricciones aceptables deberemos probar algunas cosas: que durante los cómputos sólo se van a generar restricciones aceptables, por una parte, así como que hacen un uso legítimo de la igualdad $=$. Ahora bien, puesto que los cómputos que nos interesan son $\leadsto^{ncs}$-cómputos, posponemos dichas demostraciones hasta después de definir la relación $\leadsto^{cs}$ de resolución de restricciones (aceptables) que vamos a considerar.

5.4 Resolución de restricciones en $SFL_{\neq}$

Las siguientes reglas definen la relación $\leadsto^{cs}$ que proponemos para $SFL_{\neq}$. Las reglas se aplican a restricciones aceptables de $Con_{\Sigma_=}$.

A) Reglas para restricciones primitivas.

En las siguientes reglas t, s, t_i, s_i son términos $\in \mathcal{T}_F(V)$.

- **Reglas para $=$**

$=_1$) $\exists\overline{U}(c(t_1,\ldots,t_n) = c(s_1,\ldots,s_n), \varphi) \leadsto^{cs} \exists\overline{U}(t_1 = s_1, \ldots, t_n = s_n, \varphi)$
si $c \in CS^n$.
$=_2$) $\exists\overline{U}(c(t_1,\ldots,t_n) = d(s_1,\ldots,s_m), \varphi) \leadsto^{cs} FAIL$
si $c \not\equiv d$.
$=_3$) $\exists X, \overline{U}(X = t, \varphi) \leadsto^{cs} \exists\overline{U}(\varphi\sigma)$
si t no es una variable, $\sigma \equiv \{X/t\}$.
$=_4$) $\exists\overline{U}(X = t, \varphi) \leadsto^{cs} \exists\overline{U}(X = t, \varphi\sigma)$
si t no es una variable, $X \notin \overline{U}$, X aparece en φ, $\sigma \equiv \{X/t\}$.
$=_5$) $\exists X, \overline{U}(t = X, \varphi) \leadsto^{cs} \exists\overline{U}(\varphi\sigma)$
donde $\sigma \equiv \{X/t\}$.

La primera regla es una simple de regla de descomposición, mientras la segunda indica un conflicto de constructoras. Nótese que en la regla $=_3$ la variable X de la ecuación $X = t$ está cuantificada existencialmente, y por eso puede ser eliminada tras efectuar la sustitución X/t. En la regla $=_4$, por el contrario, la variable X es libre [20], por lo que

[20] Puesto que en ningún paso de un $\leadsto^{ncs}$-cómputo se introducen variables libres, se puede afirmar incluso que X es una variable libre del objetivo inicial (o variable *global*).

debemos dejar la ecuación $X = t$ como testigo de la condición impuesta a X. La regla $=_5$ es también, como $=_3$, una regla de eliminación de variables existenciales. Puesto que en una restricción aceptable las variables de los lados derechos de las =-ecuaciones son todas existenciales, no es necesaria otra regla complementaria a $=_5$ en la que se contemple la posibilidad de que X sea libre. Obsérseve además que en $=_3$, $=_4$ y $=_5$ no hace falta realizar 'occur check', debido a la inexistencia de ciclos de variables en las restricciones aceptables.

- **Reglas para ==**

$==_1$) $\exists\overline{U}(c(t_1,\ldots,t_n) == c(s_1,\ldots,s_n),\varphi) \rightsquigarrow^{cs} \exists\overline{U}(t_1 == s_1,\ldots,t_n == s_n,\varphi)$
si $c \in CS^n$.
$==_2$) $\exists\overline{U}(c(t_1,\ldots,t_n) == d(s_1,\ldots,s_m),\varphi) \rightsquigarrow^{cs} FAIL$
si $c \not\equiv d$.
$==_3$) $\exists\overline{U}(Unif, X == t,\varphi) \rightsquigarrow^{cs} \exists\overline{U}(Unif, X == t,\varphi\sigma)$
si X aparece en φ, X no aparece en t, $\sigma \equiv \{X/t\}$.
$Unif$ es el conjunto de ecuaciones $=$.
$==_4$) $\exists\overline{U}(X == t,\varphi) \rightsquigarrow^{cs} FAIL$
si $X \not\equiv t$, X aparece en t.
$==_5$) $\exists\overline{U}(t == X,\varphi) \rightsquigarrow^{cs} \exists\overline{U}(X == t,\varphi)$
si t no es una variable.

Las reglas $==_1$ y $==_2$ son, como en el caso de $=$, reglas de descomposición y conflicto. La regla $==_3$ es una regla de 'ligadura' (similar a $=_4$) para el caso de ecuaciones $X == t$. Nótese que en este caso sí es preciso, para efectuar la sustitución X/t, verificar que X no aparezca en t (para el caso contrario se dispone de la regla de fallo $==_4$). Nótese también que la sustitución indicada no se propaga a las =-ecuaciones de la restricción. Así se garantiza la aceptabilidad de la restricción resultante, que de otro modo podría perderse, como muestra el siguiente ejemplo: si en la restricción aceptable

$$\exists U, V(X = U, X = V, U == s(V))$$

se aplica globlamente la sustitución $U/s(V)$, se obtiene la restricción

$$\exists U, V(X = s(V), X = V, U == s(V))$$

que ya no es aceptable, al fallar la linealidad de la tupla de los lados derechos de las =-ecuaciones.

- **Reglas para =/=**

$=/=_1$) $\exists\overline{U}(c(t_1,\ldots,t_n) =/= c(s_1,\ldots,s_n),\varphi) \rightsquigarrow^{cs} \exists\overline{U}(t_i =/= s_i,\varphi)$
si $c \in CS^n, n > 0$.
% Una alternativa para cada i con $1 \leq i \leq n$
$=/=_2$) $\exists\overline{U}(c(t_1,\ldots,t_n) =/= d(s_1,\ldots,s_m),\varphi) \rightsquigarrow^{cs} \exists\overline{U}(\varphi)$
si $c \not\equiv d$.
$=/=_3$) $\exists\overline{U}(t =/= t,\varphi) \rightsquigarrow^{cs} FAIL$
$=/=_4$) $\exists\overline{U}(t =/= X,\varphi) \rightsquigarrow^{cs} \exists\overline{U}(X =/= t,\varphi)$
si t no es una variable.

Llamamos la atención sobre el hecho de que no hay regla para desigualdades de la forma $X =/= t$, lo que quiere decir que tales desigualdades van a formar parte de las formas resueltas. No debe sorprender tal hecho, que responde precisamente a nuestra intención al considerar el lenguaje $SFL_{\neq}$. Y

Para aplicar los resultados obtenidos en el capítulo anterior, deberemos probar que $\leadsto^{cs}$ constituye un sistema de semidecisión (ver 4.15). Como ya comentamos en 4.4.1, la corrección y completitud (por otra parte fáciles de probar) de $\leadsto^{cs}$ quedan implicadas por la corrección y completitud de la extensión de $\leadsto^{cs}$ al caso no primitivo que vamos a definir a continuación; pueden esperar, por tanto. Lo que debemos probar específicamente de $\leadsto^{cs}$ es su terminación o semiterminación, y la existencia de formas resueltas (ver 4.15), ya que esa condición no se le pide a la extensión al caso no primitivo $\leadsto^{\widehat{cs}}$. Examinaremos estas condiciones más tarde, junto con el resto de las propiedades de $\leadsto^{cs}$. De momento, nos interesa dejar claro que sólo se va a tener terminación para la clase de $\Sigma_{=}$-restricciones aceptables. Para $\Sigma_{=}$-restricciones en general, no se da la terminación de $\leadsto^{cs}$.

Ejemplo 5.3

La falta de '*occur check*' en las reglas $=_3, =_4, =_5$ permite la existencia de reducciones infinitas, como en

$$X = s(X), Y == s(X) \leadsto^{cs}_{=_4} X = s(X), Y == s(s(X)) \leadsto^{cs}_{=_4} \ldots$$

Es decir, $\leadsto^{cs}$ no es terminante en general.
$\diamond$

Esto no supone una limitación respecto a la teoría general, pues sólo la clase de $\Sigma_{=}$-restricciones 'continuas', para las que probaremos terminación, representan restricciones formulables en $CFLP(\mathcal{H}_{\neq})$.

Definimos a continuación la extensión de $\leadsto^{cs}$ al caso de restricciones no primitivas.

B) Reglas para restricciones no primitivas.

En las siguientes reglas, si no se indica otra cosa, t, s, t_i, s_i son expresiones $\in \mathcal{T}_{F \cup \Delta}(V)$.

Diremos que una variable X es *producida* en una restricción φ, si hay en φ una ecuación $f(e_1, \ldots, e_n) = X$, siendo $f \in \Delta$.

- **Reglas para $=$**

  Al considerar restricciones que involucren a expresiones no primitivas es cuando se va a hacer más patente el uso asimétrico que de $=$ se hace en las restricciones aceptables. Recordemos que sólo a los lados izquierdos pueden aparecer expresiones no primitivas. Las reglas extendidas para $=$ son las siguientes:

  $=_1, =_2, =_3, =_4$: idénticas al caso primitivo.

$=_5$) $\exists X, \overline{U}(t = X, \varphi) \rightsquigarrow^{cs} \exists \overline{U}(\varphi\sigma)$
si t es un término o X no aparece en φ, y $\sigma \equiv \{X/t\}$.
$=_6$) $\exists X, \overline{U}(l = X, \varphi) \rightsquigarrow^{cs} \exists \overline{U}, \overline{V}(t_1 = V_1, \ldots, t_k = V_k, \varphi\sigma)$
si X aparece en φ, y l no es término pero $l \equiv c(l_1, \ldots, l_n)$,
$\sigma \equiv \{X/sk(l)\}$,
$sk(l)$ es el resultado de reemplazar en l las subexpresiones
no primitivas más externas $t_1, \ldots, t_k$ por variables nuevas
$V_1, \ldots, V_k$.

La antigua regla $=_5$ del caso primitivo se desdobla ahora en las reglas $=_5$ y $=_6$. En presencia de una ecuación $l = X$, la nueva regla $=_5$ se comporta como la antigua si l es primitivo; si no lo es, se permite también la eliminación de la variable (existencial, con seguridad) X, si ésta no aparece en el resto de la restricción. En la regla $=_6$, en que se supone una ecuación de la forma $c(l_1, \ldots, l_n) = X$, la sustitución σ indicada realiza una imitación de la 'parte primitiva' del lado izquierdo, que viene dada por su *esqueleto* $sk(c(l_1, \ldots, l_n))$.

Es importante observar que, en el caso de ecuaciones del tipo $f(e_1, \ldots, e_n) = X$, no se realiza eliminación de X, si aparece más veces en el resto de la restricción. A una variable X que aparezca en una ecuación de tal tipo le llamamos *variable producida*. La idea es que, al continuar el cómputo, o bien la variable X desaparece del resto de la restricción (en cuyo caso se podría eliminar X mediante $=_5$), o bien se descubre que $f(e_1, \ldots, e_n)$ requiere estrechamiento, tras el cual posiblemente pueda aplicarse la regla $=_6$. De este modo, el resultado de la evaluación de $f(e_1, \ldots, e_n)$ es 'compartido' por todas las apariciones de X, y se ha evitado la realización de 'copias' de la expresión no primitiva $f(e_1, \ldots, e_n)$. Esta última observación es extensiva a todas las reglas de $\rightsquigarrow^{cs}$, en las todas las sustituciones que se consideran son de variables por términos primitivos [21]. Haremos una discusión más completa de estas cuestiones en el apartado 5.6.1.

Como comentario final, digamos que una alternativa a la regla $=_6$ es la siguiente, más sencilla de formular, pero que puede requerir muchos pasos para llegar a una ecuación $f(\ldots) = X$.

$=_6{}'$) $\exists X, \overline{U}(l = X, \varphi) \rightsquigarrow^{cs} \exists \overline{U}, \overline{V}(l_1 = V_1, \ldots, l_n = V_n, \varphi\sigma)$
si X aparece en φ, y l no es término pero $l \equiv c(l_1, \ldots, l_n)$,
$\sigma \equiv \{X/c(V_1, \ldots, V_n)\}$, $V_1, \ldots, V_n$ variables nuevas.

- **Reglas para $==$**

$==_1, ==_2, ==_5$: idénticas al caso primitivo.

$==_3$) $\exists \overline{U}(Unif, X == t, \varphi) \rightsquigarrow^{cs} \exists \overline{U}(Unif, X == t, \varphi\sigma)$
si t es un término, X aparece en φ, X no aparece en t, $\sigma \equiv \{X/t\}$.
$Unif$ es el conjunto de ecuaciones $=$.

[21] Excepto en $=_5$, en el caso $f(e_1, \ldots, e_n) = X$, cuando X no aparece más veces. Pero el efecto de $=_5$ en tal caso es simplemente eliminar la ecuación.

$==_4$) $\exists\overline{U}(Unif, X == r, \varphi)$
$\leadsto^{cs} \exists\overline{U},\overline{V}(Unif, X == sk(r), (V_1 == e_1, \ldots, V_k == e_k, \varphi)\sigma)$
r es no primitivo y de la forma $c(r_1, \ldots, r_n)$,
X no aparece en $\mid r \mid$, $\sigma \equiv \{X/sk(r)\}$.
$sk(l)$ es el resultado de reemplazar en r las subexpresiones
no primitivas más externas $e_1, \ldots, e_k$ por variables nuevas $V_1, \ldots, V_k$.
$Unif$ es el conjunto de ecuaciones $=$.

$==_6$) $\exists\overline{U}(X == t, \varphi) \leadsto^{cs} FAIL$
si $X \not\equiv t$, X aparece en $\mid t \mid$.

De modo similar a lo que hicimos en $=_6$ y $=_6{}'$, podemos también considerar, como alternativa a las reglas $==_3, ==_4$ recién propuestas, otra pareja de reglas más sencillas de formular, pero que requieren en general mucho más trabajo para efectuar las sustituciones de $==_3, ==_4$, y en sacar al exterior la parte no primitiva del lado derecho, en el caso de $==_4$.

$==_3{}'$) $\exists\overline{U}(Unif, X == Y, \varphi) \leadsto^{cs} \exists\overline{U}(Unif, X == Y, \varphi\sigma)$
si $X \not\equiv Y$, X aparece en φ, $\sigma \equiv \{X/Y\}$.
$Unif$ es el conjunto de ecuaciones $=$.
$==_4{}'$) $\exists\overline{U}(Unif, X == r, \varphi)$
$\leadsto^{cs} \exists\overline{U},\overline{V}(Unif, X == c(V_1, \ldots, V_n), (V_1 == r_1, \ldots, V_n == r_n, \varphi)\sigma)$
si r es de la forma $c(r_1, \ldots, r_n)$, X no aparece en $\mid r \mid$,
X aparece en φ o r es no primitivo,
$\sigma \equiv \{X/c(V_1, \ldots, V_n)\}$, $V_1, \ldots, V_n$ variables nuevas.
$Unif$ es el conjunto de ecuaciones $=$.

- **Reglas para $=\!\!/\!\!=$**

$=\!\!/\!\!=_1, =\!\!/\!\!=_2, =\!\!/\!\!=_3, =\!\!/\!\!=_4$: idénticas al caso primitivo.

$=\!\!/\!\!=_5$) $\exists\overline{U}(X =\!\!/\!\!= c(e_1, \ldots, e_n), \varphi) \leadsto^{cs} \exists\overline{U},\overline{V}(X = d(V_1, \ldots, V_m), \varphi\sigma)$
si $c(e_1, \ldots, e_n)$ no es un término o tiene variables producidas,
$c \neq d$, $V_1, \ldots, V_m$ variables nuevas
$\sigma \equiv \{X/d(V_1, \ldots, V_m)\}$.
$=\!\!/\!\!=_6$) $\exists\overline{U}(X =\!\!/\!\!= c(e_1, \ldots, e_n), \varphi) \leadsto^{cs} \exists\overline{U},\overline{V}(X = c(V_1, \ldots, V_n), (V_i =\!\!/\!\!= e_i, \varphi)\sigma)$
si $c(e_1, \ldots, e_n)$ no es un término o tiene variables producidas,
$V_1, \ldots, V_n$ nuevas variables.
$\sigma \equiv \{X/c(V_1, \ldots, V_n)\}$.
% Una alternativa para cada i con $1 \leq i \leq n$

Las reglas $=/=_5$ y $=/=_6$ se han añadido para que la extensión de $\leadsto^{cs}$ que estamos definiendo sea perezosa (ver definición 4.17). El ejemplo 4.13 sirve como ilustración de este hecho. Nótese además que en $=\!\!/\!\!=_5$ y $=\!\!/\!\!=_6$, el papel de una variable producida se equipara al de una expresión no primitiva.

Con esto queda completada la definición de la extensión $\rightsquigarrow^{cs}$. Es rutinario verificar que, para restricciones (aceptables) primitivas, las nuevas reglas de $\rightsquigarrow^{cs}$ se comportan como las antiguas.

Comenzamos ahora a probar muchas propiedades que tenemos pendientes de verificar. En primer lugar, veamos que la aceptabilidad de las restricciones es una propiedad que se preserva durante los cómputos.

Proposición 5.1

Si $\varphi \in Con_{\Sigma_=}$ es aceptable y $\varphi \rightsquigarrow^{ncs} \psi$, entonces ψ es aceptable.

Demostración:
Veamos que la aceptabilidad se preserva tanto si el paso es de estrechamiento $\rightsquigarrow$, como si lo es de resolución de restricciones $\rightsquigarrow^{cs}$.

A) Hagamos explícito en qué consiste un paso de estrechamiento mediante una $SFL_{\neq}$-regla, teniendo en cuenta que debemos considerar, para hacer el estrechamiento, su $\Sigma_=$-traducción: si $[\varphi]_u \equiv f(e_1, \ldots, e_n)$, un paso de estrechamiento en la posición u mediante una $SFL_{\neq}$-regla

$$R \equiv \ f(t_1, \ldots, t_n) = e \Leftarrow \psi$$

(cuyas variables $\overline{Y}$ no aparezcan en φ), resulta de la forma

$$\varphi \rightsquigarrow \exists \overline{Y}(\varphi[u \leftarrow e], e_1 = t_1, \ldots, e_n = t_n, \psi)$$

Si $\varphi \rightsquigarrow \psi$, es claro que ψ es aceptable, por la linealidad de los patrones $t_1, \ldots, t_n$, y por usarse una variante de la regla R con variables nuevas, todas ellas cuantificadas existencialmente.

B) Para el caso $\varphi \rightsquigarrow^{cs} \psi$, consideramos por separado cada regla para $\rightsquigarrow^{cs}$. Obviamente se pueden omitir las reglas de fallo y aquellas que no modifiquen el conjunto de $=$-ecuaciones, con lo que nos quedan $\{=_1, =_3, =_4, =_5, =_6, =/=_5, =/=_6\}$. Podemos suponer además que las restricciones sólo tienen $=$-ecuaciones (más la desigualdad pertinente, en el caso de $=/=_5$, $=/=_6$). Puesto que en muchas ocasiones deberemos probar que una cierta ψ es aceptable, usaremos las siguientes abreviaturas para las propiedades que deben cumplir las $=$-ecuaciones de una restricción aceptable:

T: Los lados derechos son términos
L: La tupla de los lados derechos es lineal
E: Las variables de los lados derechos son existenciales
C: No hay ciclos de variables

Podremos anotar estas abreviaturas en la forma **T**-φ, **C**-ψ, etc. Abreviaremos también la notación $X \in var(e)$ a $X \in e$.

- $(=_1)$ $\exists \overline{U}(c(t_1, \ldots, t_n) = c(s_1, \ldots, s_n), \varphi) \rightsquigarrow^{cs} \exists \overline{U}(t_1 = s_1, \ldots, t_n = s_n, \varphi)$
  Es obvio que **T**,**L**,**E** quedan preservadas. En cuanto a **C**, basta tener en cuenta que para

todo ciclo que utilice una de las ecuaciones $t_i = s_i$, podemos obtener otro reemplazando esta ecuación por $c(t_1, \ldots, t_n) = c(s_1, \ldots, s_n)$, por lo que si ψ tuviese un ciclo, φ también lo tendría.

- $(=_3)$ (φ) $\exists X, \overline{U}(X = t, \varphi_0) \rightsquigarrow^{cs} \exists \overline{U}(\varphi_0\sigma))$ (ψ), siendo $\sigma \equiv \{X/t\}$.
  T-ψ y **E**-ψ son obvias ya que t es un término por **T**-φ, y sus variables están en $\overline{U}$. Ahora, supongamos $\varphi_0 \equiv e_1 = t_1, \ldots, e_n = t_n$. Por **L**-$\varphi$, t es lineal y no comparte variables con $t_1, \ldots, t_n$; también por **L**-φ, X aparece a lo sumo una vez en $t_1, \ldots, t_n$. De ahí obtenemos que $\langle t_1\sigma, \ldots, t_n\sigma\rangle$ es lineal, es decir, **L**-ψ. Para probar **C**-ψ, veamos que de un ciclo en ψ podríamos extraer un ciclo en φ, en contra de **C**-φ.

  En efecto, supongamos que tenemos en $\varphi_0\sigma$ un ciclo de variables en la forma

  $$\begin{array}{ccccccc} X_1 & & X_2 & & X_k & & X_{k+1} \\ l_1\sigma & = & r_1\sigma & \ldots, & l_k\sigma & = & r_k\sigma \end{array}$$

  donde $X_{k+1} \equiv X_1$. Distinguimos dos casos

  - Supongamos $X_1 \notin l_1$. Entonces, puesto que $X_1 \in l_1\sigma$, se tiene $X \in l_1$ y $X_1 \in t$. De esto último, y por la linealidad de φ, deducimos que $X_1 \notin r_k$. Sea ahora m el más pequeño tal que $X_{m+1} \notin r_m$ (que existe pues $X_{k+1} \equiv X_1 \notin r_k$). Como $X_{m+1} \in r_m\sigma$, se tiene que $X \in r_m$ (y $X_{m+1} \in t$). Además, para todo $1 \leq j < m$ se tiene: $X_{j+1} \in r_j$ (por ser m el más pequeño para el que falla esta condición), y de nuevo por la linealidad de φ, $X_{j+1} \notin t$; pero como $X_{j+1} \in l_{j+1}\sigma$, deducimos $X_{j+1} \in l_{j+1}$. Pero entonces tenemos en φ el ciclo

    $$\begin{array}{ccccccc} X & & X_2 & & X_m & & X \\ l_1 & = & r_1 & \ldots, & l_m & = & r_m \end{array}$$

    El caso $X_1 \notin l_1$ se generaliza fácilmente a $X_i \notin l_i$ para algún $1 \leq i \leq n$, si se tiene en cuenta que cualquier permutación cíclica de un ciclo de variables lo es también. Así pues, el siguiente caso complementa el primero
  - Supongamos $X_i \in l_i$ para todo $1 \leq i \leq n$, y sea m tal que $X_{m+1} \notin r_m$ (de no existir, ya tenemos en φ el ciclo $\begin{array}{ccccccc} X_1 & & X_2 & & X_k & & X_1 \\ l_1 & = & r_1 & \ldots, & l_k & = & r_k \end{array}$). Como $X_{m+1} \in r_m\sigma$, tenemos $X \in r_m$ y $X_{m+1} \in t$. Por linealidad de φ se tiene $X \notin r_j$, y de aquí $X_{j+1} \in r_j$, para todo $j \neq m$. Tenemos entonces en φ el ciclo

    $$\begin{array}{ccccccccccccc} X_1 & & X_2 & & X_m & & X & X & & X_{m+1} & & X_k & & X_1 \\ l_1 & = & r_1 & \ldots, & l_m & = & r_m & , X & = & t & \ldots, & l_k & = & r_k \end{array}$$

- $(=_4)$ (φ) $\exists \overline{U}(X = t, \varphi_0) \rightsquigarrow^{cs} \exists \overline{U}(X = t, \varphi_0\sigma)$ (ψ), con $X \notin \overline{U}$ y $\sigma \equiv \{X/t\}$.
  Para **T**-ψ,**L**-ψ,**E**-ψ, basta observar que por **E**-φ, y puesto que $X \notin \overline{U}$, X no aparece en los lados derechos de las $=$-ecuaciones de φ_0, que por tanto no quedan afectados por σ. Para **C**-ψ vale la demostración de $=_3$ (que incluso podría simplificarse, teniendo en cuenta la observación anterior).

- $(=_5)$ (φ) $\exists X, \overline{U}(e = X, \varphi_0) \rightsquigarrow^{cs} \exists \overline{U}(\varphi_0\sigma)$ (ψ), con $\sigma \equiv \{X/e\}$.
  Para **T**-ψ,**L**-ψ,**E**-ψ observemos que, como antes, X no aparece en los lados derechos de las $=$-ecuaciones de φ_0, en este caso debido a **L**-φ. Probamos **C**-ψ por un razonamiento similar (algo más sencillo) al caso de $=_3$. Supongamos que tenemos en $\varphi_0\sigma$ un ciclo de variables en la forma
  $$\begin{array}{ccccccc} X_1 & & X_2 & & X_k & & X_{k+1} \\ l_1\sigma & = & r_1\sigma & , \ldots , & l_k\sigma & = & r_k\sigma \end{array}$$
  donde $X_{k+1} \equiv X_1$. Debe tenerse $X_{i+1} \in r_i$, para todo $1 \leq i \leq k$, ya que $X \notin r_i$ por la linealidad de φ. Supongamos que existe i tal que $X_i \notin l_i$ (en otro caso se tiene en φ el ciclo $\begin{array}{ccccccc} X_1 & & X_2 & & X_k & & X_{k+1} \\ l_1 & = & r_1 & , \ldots , & l_k & = & r_k \end{array}$). Por una observación hecha anteriormente podemos considerar, sin pérdida de generalidad, que $i = 1$, y por tanto $X_1 \notin l_1$. Se tendrá pues $X \in l_1$ y $X_1 \in e$. Ahora:

  - Si $X_i \in l_i$ para todo $1 < i \leq k$, tenemos el ciclo
    $$\begin{array}{ccccccccccc} X_1 & & X & & X & & X_2 & & X_k & & X_1 \\ e & = & X & , & l_1 & = & r_1 & , \ldots , & l_k & = & r_k \end{array}$$
  - En otro caso, sea m el menor tal que $1 < m \leq k$ y $X_m \notin l_m$, con lo que $X \in l_m$, $X_m \in e$. Tenemos entonces el ciclo
    $$\begin{array}{ccccccccccc} X_m & & X & & X & & X_2 & & X_{m-1} & & X_m \\ e & = & X & , & l_1 & = & r_1 & , \ldots , & l_{m-1} & = & r_{m-1} \end{array}$$

- $(=_6)$ (φ) $\exists X, \overline{U}(l = X, \varphi) \rightsquigarrow^{cs}$ $\exists \overline{U}, \overline{V}(t_1 = V_1, \ldots, t_k = V_k, \varphi\sigma)$ (ψ)
  en donde interesa destacar que las variables $V_1, \ldots, V_k$ son nuevas (por lo que las ecuaciones $t_1 = V_1, \ldots, t_k = V_k$ no pueden afectar a **T**, **L**, **E**, **C**), y $\sigma \equiv \{X/sk(l)\}$, por lo que es como si estuviéramos en presencia de la ecuación $sk(l) = X$, a la que aplicásemos a continuación $=_5$, para la que ya hemos probado que preserva la aceptabilidad.

- $(=\!\!\!/=_5)$,$(=\!\!\!/=_6)$ preservan claramente **T**, **L**, **E** y **C**.

□

El siguiente resultado establece que las restricciones aceptables de $Con_{\Sigma_=}$ realmente lo son para nuestros propósitos, es decir, pueden reemplazarse por restricciones equivalentes de Con_Σ, que son las '$CFLP$-puras'. Junto con el corolario 5.1, es una justificación del uso legítimo que hacemos de $=$. Nos basaremos en las mismas transformaciones que hicimos entonces, pero formuladas para restricciones y no para programas. La demostración de que las cosas funcionan bien, aun siendo similar a aquélla, no se puede obviar, pues el uso que de $=$ hacen las restricciones aceptables es bastante más general que en el caso de la traducción de un programa.

Teorema 5.1

Para toda $\varphi \in Con_{\Sigma_=}$ aceptable, existe $\psi \in Con_\Sigma$ equivalente a φ, en el sentido de que

$$\alpha \models^I \varphi \Leftrightarrow \alpha \models^I \psi, \text{ para todas } \alpha, I$$

Demostración:
Obtendremos ψ aplicando unas reglas de transformación a φ, cuyo objetivo es ir eliminando las ecuaciones $=$ de φ. En los pasos intermedios podrán aparecer simultáneamente tanto el símbolo $=$ como c_k e is_c. Por este motivo, las reglas están definidas sobre restricciones de $Con_{\widehat{\Sigma}}$ (en las que recordemos se permite dicho uso combinado). Son las dos siguientes:

$$\begin{array}{ll}(T_1) & \exists V, \overline{U}(e = V, \varphi) \longmapsto \exists \overline{U}(\varphi\sigma) \\ & \text{siendo } \sigma \equiv \{V/e\} \\ (T_2) & \exists \overline{U}(e = c(t_1, \ldots, t_n), \varphi) \longmapsto \exists \overline{U}(is_c(e), c_1(e) = t_1, \ldots, c_n(e) = t_n, \varphi)\end{array}$$

La relación $\longmapsto$ definida por $(T_1), (T_2)$ tiene las siguientes propiedades:

(1) $\longmapsto$ preserva soluciones, es decir: si $\varphi \longmapsto \psi$, entonces $\forall \alpha, I(\alpha \models^I \varphi \Leftrightarrow \alpha \models^I \psi)$.

En efecto, si $\varphi \longmapsto_{T_1} \psi$, se tiene que ψ es lógicamente equivalente a φ (téngase en cuenta que $=$ se interpreta como la igualdad verdadera, por lo que σ reemplaza iguales por iguales; además la ecuación $e = V$ se puede eliminar, porque V está cuantificada existencialmente). El caso de $\varphi \longmapsto_{T_2} \psi$ es inmediato teniendo en cuenta las definiciones de is_c y c_k.

(2) $\longmapsto$ preserva la aceptabilidad [22], es decir: si φ es aceptable y $\varphi \longmapsto \psi$ entonces ψ es aceptable.

En efecto, para el caso de la transformación (T_1), sirve la misma demostración que dimos en el teorema anterior para la regla $=_5$ de $\rightsquigarrow^{cs}$ (que no depende de que e sea un término o no en la ecuación $e = X$). En cuanto a la regla (T_2), es obvio que no introduce nuevos símbolos en los lados derechos de las $=$-ecuaciones, ni afecta a la linealidad del conjunto de los lados derechos. En cuanto a ciclos de variables, basta observar que cualquier ciclo que se pudiese construir en ψ con ayuda de una ecuación $c_i(e) = t_i$ puede reproducirse en φ considerando para el ciclo la ecuación $e = c(t_1, \ldots, t_n)$ en lugar de $c_i(e) = t_i$.

(3) $\longmapsto$ es terminante para restricciones aceptables, es decir: si $\varphi \in Con_{\Sigma_=}$ es aceptable, no existe ninguna $\longmapsto$-reducción infinita $\varphi \longmapsto \varphi_1 \longmapsto \varphi_2 \longmapsto \ldots$.

En efecto, de acuerdo con el punto anterior, todas las restricciones de la $\longmapsto$-reducción son aceptables. Para ellas, la regla (T_1) hace decrecer el número (N) de variables distintas de la restricción (ya que V no aparece en e, para una ecuación $e = V$). Por su parte, la regla (T_2) hace decrecer el tamaño total (T) de los lados derechos de $=$-ecuaciones, mientras deja constante el número de variables distintas de la restricción. Así pues, cada $\longmapsto$-paso hace decrecer la restricción en el orden lexicográfico sobre los pares (N, T), que es claramente bien fundado.

(4) Si φ es aceptable y $\longmapsto$-irreducible, φ no tiene $=$-ecuaciones (y por tanto $\varphi \in Con_{\Sigma}$).

Es obvio que a toda restricción aceptable con alguna ecuación $l = r$ se le puede aplicar (T_1) o (T_2), ya que r es un término, y en caso de ser variable, está cuantificada existencialmente.

[22] La noción de aceptabilidad se extiende sin problemas al caso de restricciones de $Con_{\widehat{\Sigma}}$, sin más que precisar que en la definición 5.4 al hablar de *términos* se entiende que en ellos sólo aparecen las constructoras de $\mathcal{CS}$, no los símbolos is_c, c_k.

El teorema se sigue ahora fácilmente: si $\varphi \in Con_{\Sigma_=}$, por (3) existe ψ tal que $\varphi \longmapsto \psi$ y ψ es $\longmapsto$-ireducible. Por (2) ψ es aceptable, y por (4) $\psi \in Con_\Sigma$. Finalmente, por (1), φ y ψ tienen las mismas soluciones. □

A diferencia del caso de programas, la transformación de $\varphi \in Con_{\Sigma_=}$ en una restricción $\psi \in Con_\Sigma$ no es única; eso sí, todas las posibles traducciones son equivalentes, a la vista de lo probado.

Veamos a continuación que $\rightsquigarrow^{cs}$ define un sistema de resolución de restricciones (aceptables) que verifica las condiciones requeridas en la sección 4.4.

Comencemos por examinar la estructura de las restricciones primitivas $\rightsquigarrow^{cs}$-irreducibles.

Proposición 5.2

Una restricción primitiva φ (distinta de $FAIL$) es $\rightsquigarrow^{cs}$-irreducible si y solo si

$$\varphi \equiv \quad \exists \overline{U}(Unif, Ig, Desig, Fin)$$

donde

- $Unif \equiv \quad X_1 = t_1, \ldots, X_n = t_n \quad (n \geq 0)$
- $Ig \equiv \quad Y_1 == s_1, \ldots, Y_m == s_m \quad (m \geq 0)$
- $Desig \equiv \quad Z_1 =/= r_1, \ldots, Z_k =/= r_k \quad (k \geq 0)$
- $Fin \equiv \quad U_1 == U_1, \ldots, U_l == U_l \quad (l \geq 0)$

y se verifica

- *Cada X_i aparece una sola vez en φ (en particular $X_i \notin \overline{U}$)*
- *Cada t_i es un término no variable con $var(t_i) \subset \overline{U}$*
- *La tupla $\langle t_1, \ldots, t_n \rangle$ es lineal*
- *Cada Y_i aparece una sola vez en $Ig, Desig, Fin$*
- *Cada s_i es un término*
- *Cada r_i es un término distinto de Z_i*

Demostración:
Aunque tediosa, es inmediata en ambos sentidos, por inspección directa de las reglas □

Como consecuencia de esta caracterización concluimos a continuación que $\rightsquigarrow^{cs}$ admite formas resueltas (ver Def. 4.15). El resultado siguiente proporciona incluso más información.

Proposición 5.3

Sea $\exists \overline{U} \varphi$ primitiva y $\rightsquigarrow^{cs}$-irreducible (distinta de $FAIL$). Entonces

(i) φ tiene soluciones finitas y totales.

(ii) $\exists\overline{U}\varphi$ es satisfactible y tiene soluciones finitas y totales.

Demostración:

(i) Sea $\varphi \equiv Unif, Ig, Desig, Fin$. Por un resultado conocido [97], $Ig, Desig$ tiene soluciones en el universo de Herbrand finitario [23], es decir, lo que en nuestro caso son soluciones finitas y totales. Eso quiere decir que también $Ig, Desig, Fin$ tiene soluciones finitas y totales; sea α una de ellas.

Sea $X = t$ una ecuación de $Unif$. Como X no aparece en $t, Ig, Desig, Fin$, podemos redefinir α en X como $\alpha(X) = \alpha[\![t]\!]$, con lo que α así redefinida es solución de $X = t, Ig, Desig, Fin$. Basta finalmente repetir el razonamiento para cada ecuación de $Unif$. Como comentario marginal, nótese que no hemos hecho uso de la linealidad de los lados derechos de $Unif$.

(ii) Es obvia a partir de (i), ya que toda solución de φ lo es de $\exists\overline{U}\varphi$.

□

Nótese que la existencia de soluciones finitas y totales de $\exists\overline{U}\varphi$, una vez que se sabe tiene alguna, estaba de hecho garantizada, por argumentos de continuidad: en efecto, como por el teorema 5.1 $\exists\overline{U}\varphi$ es una restricción continua, si α es solución de $\exists\overline{U}\varphi$, tiene una aproximación finita α' que lo es también. Si $\alpha'(X)$ es parcial para alguna X, basta reemplazar las apariciones de $\perp$ en $\alpha'(X)$ por un valor finito y total. La nueva α'' (finita y total) mayora a α' y es por tanto solución de $\exists\overline{U}\varphi$. Este razonamiento no sirve sin embargo para (i), pues al ignorar el prefijo existencial, no tenemos que φ sea continua.

A continuación probamos que $\rightsquigarrow^{cs}$, para restricciones (aceptables) primitivas, es terminante. Probamos de hecho algo más.

Lema 5.3

Sea R_0 el conjunto de reglas de $\rightsquigarrow^{cs}$ (para el caso general, no primitivo) exceptuando $=\!/\!=_5$, $=\!/\!=_6$, y sea $\rightsquigarrow^{cs}_{R_0}$ la subrelación de $\rightsquigarrow^{cs}$ que resulta. Entonces $\rightsquigarrow^{cs}_{R_0}$ es terminante.

Demostración:
Puesto que toda $\rightsquigarrow^{cs}$-reducción que use una regla de fallo es terminante, nos referimos en lo que sigue al resto de las reglas.

Consideraremos una combinación lexicográfica de varios órdenes, basados en distintas funciones para medir el 'tamaño' de una restricción. Para las definiciones que siguen, debe ignorarse el prefijo existencial que pudiera tener una restricción φ. Las funciones que usamos son:

[23] La existencia de soluciones está garantizada si el universo de Herbrand es infinito. En otro caso, no tiene por qué ser así. Si, por ejemplo, sólo se dispone de las constantes a, b, la restricción $X =\!/\!= a, X =\!/\!= b$ (que sería $\rightsquigarrow^{cs}$-irreducible) no admite soluciones.

- $|\ \varphi\ |_1$= Número total, en las $=$-ecuaciones, de posiciones no primitivas internas (o sea, dentro de expresiones $c(e_1, \ldots, e_n)$.
- $|\ \varphi\ |_2$= Número de variables distintas de φ.
- $|\ \varphi\ |_3$= Número de variables no $=$-resueltas de φ.
  Entendemos aquí por *variable $=$-resuelta en φ* una variable X cuya única aparición en φ es en una ecuación $X = t$.
- $|\ \varphi\ |_4$= Número de variables no $==$-resueltas de φ.
  Entendemos aquí por *variable $==$-resuelta en φ* una variable X que aparece en una ecuación $X == t$, pero no en t ni en el resto de las igualdades $==$ ni desigualdades $=\!\!/\!\!=$ de φ.
- $|\ \varphi\ |_5$= Tamaño de φ, es decir, el número de (apariciones de) símbolos de variable, constructora o función que hay en φ.
- $|\ \varphi\ |_6$= Número de condiciones de la forma $l == X$ o $l =\!\!/\!\!= X$ de φ.

Todas esas funciones inducen órdenes obviamente bien fundados. Y ahora tenemos en cuenta que

(1) $=_6$ hace disminuir $|\ \varphi\ |_1$.

(2) $=_3, =_5$ hacen disminuir $|\ \varphi\ |_2$, sin aumentar $|\ \varphi\ |_1$.

(3) $=_4$ hace disminuir $|\ \varphi\ |_3$, sin aumentar $|\ \varphi\ |_1, |\ \varphi\ |_2$.

(4) $==_3$ hace disminuir $|\ \varphi\ |_4$, sin aumentar $|\ \varphi\ |_1, |\ \varphi\ |_2, |\ \varphi\ |_3$.

(5) $=_1, ==_1, =\!\!/\!\!=_1, =\!\!/\!\!=_2$ hacen disminuir $|\ \varphi\ |_5$, sin aumentar $|\ \varphi\ |_1, |\ \varphi\ |_2, |\ \varphi\ |_3, |\ \varphi\ |_4$.

(6) $==_5, =\!\!/\!\!=_4$ hacen disminuir $|\ \varphi\ |_6$, sin aumentar $|\ \varphi\ |_1, |\ \varphi\ |_2, |\ \varphi\ |_3, |\ \varphi\ |_4, |\ \varphi\ |_5$.

En consecuencia, en cada $\leadsto^{cs}_{R_0}$-paso se reduce la tupla

$$\langle |\ \varphi\ |_1, |\ \varphi\ |_2, |\ \varphi\ |_3, |\ \varphi\ |_4, |\ \varphi\ |_5 \rangle$$

si consideramos para ella el orden lexicográfico, que está bien fundado. Así pues, no puede haber $\leadsto^{cs}_{R_0}$-reducciones infinitas. □

Corolario 5.2

$\leadsto^{cs}$ es terminante, para restricciones (aceptables) primitivas.

Aunque la terminación de $\leadsto^{cs}$ incluso para restricciones no primitivas no sea necesaria para los resultados que vamos a obtener, no cabe duda que sería interesante. No disponemos de una demostración completa de ello, pero creemos muy razonable que así sea.

Conjetura: $\leadsto^{cs}$ es terminante para restricciones aceptables posiblemente no primitivas.

Los anteriores resultados, junto con la corrección y completitud que probaremos en seguida (corolario 5.3) para el caso más general de restricciones posiblemente no primitivas, nos indican que $\rightsquigarrow^{cs}$ constituye un sistema de decisión (ver definición 4.15) para las restricciones (aceptables) primitivas.

En lo que sigue, $\rightsquigarrow^{cs}$ se refiere al caso general, en que las restricciones pueden ser no primitivas.

El siguiente lema, del que se sigue inmediatamente la correción y completitud (en el sentido de la definición 4.15) de $\rightsquigarrow^{cs}$, establece con precisión cómo $\rightsquigarrow^{cs}$ propaga soluciones. Utilizaremos las siguientes notaciones:

- $sol(\varphi, I) \equiv \{\alpha \mid \alpha \text{ es solución de } \varphi \text{ en } I\}$ $\quad (sol(FAIL, I) = \emptyset)$.
- $\varphi \rightsquigarrow^{cs}_{R} \psi$ indica que el paso se ha dado con la regla R, o con una regla de R, si R es un conjunto de reglas.
- $R_{=}, R_{==}, R_{\neq}$ denotan los conjuntos de $\rightsquigarrow^{cs}$-reglas para $=, ==$ y $=\!\!/\!\!=$ respectivamente.

Lema 5.4

(i) Sea $R = R_{=} \cup R_{==} \cup \{=\!\!/\!\!=_2, =\!\!/\!\!=_3, =\!\!/\!\!=_4\}$. *Si* $\varphi \rightsquigarrow^{cs}_{R} \psi$, *entonces* $sol(\varphi, I) = sol(\psi, I)$, *para toda* $I \in \mathcal{INT}$.

(ii) Si $=\!\!/\!\!=_1$ *es aplicable a* φ, *y* $\varphi \rightsquigarrow^{cs}_{r_i} \psi_i$ $\quad (1 \leq i \leq n)$, *siendo* $r_1, \ldots, r_n$ *las distintas alternativas de* $=\!\!/\!\!=_1$, *se tiene:* $sol(\varphi, I) = \bigcup_i sol(\psi_i, I)$

(iii) Similar a (ii) para $=\!\!/\!\!=_5, =\!\!/\!\!=_6$

Demostración:
No es difícil, procediendo regla por regla. Tiene interés sin embargo observar que las reglas $=_3, =_5, =_6$ preservan soluciones debido a que las variables eliminadas están cuantificadas existencialmente. También es importante la cuantificación existencial de las variables nuevas introducidas por las reglas $=\!\!/\!\!=_5, =\!\!/\!\!=_6$. □

Corolario 5.3

$\rightsquigarrow^{cs}$ *es correcta y completa.*

El lema previo es más informativo que este corolario, pues delimita juegos completos de $\rightsquigarrow^{cs}$-reglas, la mayor parte unitarios. La importancia a efectos prácticos es obvia: dada φ es probable que puedan darse muchos pasos distintos $\varphi \rightsquigarrow^{cs} \psi$, ya que muchas $\rightsquigarrow^{cs}$-reglas pueden ser aplicables a φ. Por la mera completitud sólo sabemos que entre todos esos pasos (como alternativas indeterministas) quedan capturadas todas las soluciones de φ. Usando el teorema previo, sabemos que si se aplica una de las reglas de *(i)*, se preservan las soluciones, y por tanto no es necesario probar otra alternativa, en caso de fallo o de búsqueda de soluciones alternativas. Si se aplica una de las instancias de $=\!/\!=_1$, hay que considerar todas sus alternativas. Otro tanto ocurre para $=\!\!/\!\!=_5, =\!\!/\!\!=_6$.

Lema 5.5

$\rightsquigarrow^{cs}$ es perezosa (ver definición 4.17), es decir: Si $\alpha \models\mid \varphi \mid$, existe ψ primitiva tal que $\varphi \rightsquigarrow^{cs^*} \psi$ y $\alpha \models \psi$

Demostración:
Podemos suponer que φ es no primitiva, pues en otro caso el resultado es trivial. Recuérdese además que la condición $\alpha \models\mid \varphi \mid$ es equivalente a $\alpha \models^{\perp_{INT}} \varphi$. Sean $R_0, \rightsquigarrow^{cs}_{R_0}$ como en el lema 5.3 (es decir, el resultado de excluir $=/=_5, =/=_6$ de $\rightsquigarrow^{cs}$). Sea ψ $\rightsquigarrow^{cs}_{R_0}$-irreducible y tal que $\alpha \models^{\perp_{INT}} \psi$ (o equivalentemente $\alpha \models\mid \psi \mid$). Tal ψ existe pues $\rightsquigarrow^{cs}_{R_0}$ es terminante (lema 5.3), y completa (como consecuencia del lema 5.4) para cualquier interpretación (en particular, para $\perp_{INT}$).. Como ψ es $\rightsquigarrow^{cs}_{R_0}$-irreducible y $\mid \psi \mid$ es satisfactible, es fácil ver que tiene la forma

$$\exists \overline{U}, \overline{X}(Prim, e_1 = X_1, \ldots, e_n = X_n, Y_1 =/= r_1, \ldots, Y_m =/= r_m)$$

donde $Prim$ es primitiva, $e_1, \ldots, e_n$ son de la forma $f(\_, \ldots, \_)$ (f no primitiva), y para todo i, X_i no aparece en $Prim$ ni es ninguna de las Y_j (pues $\alpha \models\mid e_i = X_i \mid$ implica que $\alpha(X_i) = \perp$, mientras que $\alpha \models\mid Y_j =/= r_j \mid$ implica $\alpha(Y_j) \neq \perp$). Nuestro objetivo ahora es eliminar la parte no primitiva que queda, que son las expresiones e_i, más las expresiones no primitivas que puedan aparecer en $r_1, \ldots, r_m$. Nótese que la forma de eliminar e_i es por eliminación de la ecuación $e_i = X_i$ (regla $=_5$), para lo que hace falta conseguir que la variable producida X_i desaparezca del resto de la restricción. Podemos conseguir esto, y al tiempo la eliminación de la parte no primitiva de $r_1, \ldots, r_m$, mediante aplicación reiterada de $=/=_5, =/=_6$ (en cada paso, la que corresponda para capturar α). Nótese que este último proceso termina, pues cada aplicación de $=/=_5$ elimina una desigualdad, y cada aplicación de $=/=_6$ disminuye la profundidad máxima a que se encuentra una expresión no primitiva o variable producida en el lado derecho de la desigualdad en cuestión. Al final del proceso, las desigualdades que resulten deben ser primitivas, y no pueden contener a una variable X_i en sus lados derechos, por lo que en efecto se pueden eliminar las ecuaciones $e_i = X_i$. □

Para terminar esta sección, veamos un resultado útil acerca de restricciones no primitivas, pero $\rightsquigarrow^{cs}$-irreducibles.

Proposición 5.4

Si φ es una restricción no primitiva $\rightsquigarrow^{cs}$-irreducible, entonces tiene una subexpresión e de la forma $f(e_1, \ldots, e_n)$ que verifica alguna de las condiciones siguientes:

- *φ tiene una ecuación de la forma $e = c(t_1, \ldots, t_n)$.*
- *φ tiene una ecuación de la forma $e == r$ (o $l == e$).*
- *φ tiene una desigualdad de la forma $e =/= r$ (o $l =/= e$).*

- *φ tiene una ecuación de la forma $e = X$, y alguna otra condición atómica de la forma*
  - $X = c(t_1, \ldots, t_n)$
  - $l == r$, *con* $X \in var(\mid l \mid, \mid r \mid)$
  - $X =/= r$

En cualquiera de las situaciones indicadas, e está absolutamente demandada en φ.

Demostración:
La prueba es rutinaria, por análisis de las distintas posibilidades. Veamos parte de los razonamientos necesarios.

Sea φ no primitiva y $\rightsquigarrow^{cs}$-irreducible. Podemos descartar entonces la posibilidad de que en φ haya condiciones de la forma $c(l_1, \ldots, l_n)$ *op* $d(r_1, \ldots, r_m)$, siendo *op* uno de los símbolos $=, ==, =/=$, pues una regla de descomposición o fallo se podría aplicar. Analicemos dónde pueden aparecer expresiones no primitivas en φ. Sea entonces u una posición crítica en φ, y sea $e \equiv [\varphi]_u$, (e debe tener la forma $f(e_1, \ldots, e_n)$). Si hay una condición de la forma $e == r$ o $e =/= r$ (o al revés), ya está. Suponemos pues que no hay condiciones de estos tipos. Ahora, si e está en una desigualdad $X =/= c(r_1, \ldots, r_n)$, las reglas $=/=_5, =/=_6$ son aplicables. Si es al revés, puede aplicarse $=/=_4$. Suponiendo que tampoco este tipo de condición aparece, sólo queda ya la posibilidad de que e esté en una condición $l = X$. Si e es interna a l, $=_6$ es aplicable. De no haber ninguna en esta situación, debe ser $e \equiv l$, es decir, hay una ecuación $e = X$. La prueba de que X verifica una de las condiciones del enunciado se haría de modo similar, y la omitimos. □

Esta propiedad es interesante pues, por una parte, proporciona un repertorio (no exhaustivo) de condiciones que aseguran que una posición no primitiva está demandada por cualquier solución y testigo minimal de ella, y por otra, establece que si no podemos dar un $\rightsquigarrow^{cs}$-paso, con seguridad hay una posición de este estilo, en la que por tanto vamos a poder hacer estrechamiento, con garantías (si usamos la regla del programa adecuada, pero esa es otra historia) de acercarnos a una respuesta terminada. Es importante observar que nuestro razonamiento no está condicionado por ninguna solución dada de antemano.

5.5 Resultados de completitud

A la vista de las propiedades que hemos probado para $\rightsquigarrow^{cs}$ en el apartado anterior, y teniendo en cuenta los resultados de la sección 4.4, en particular el teorema 4.6, ya podemos afirmar que $\rightsquigarrow^{ncs} = \rightsquigarrow \cup \rightsquigarrow^{cs}$ constituye una semántica operacional correcta y completa para $SFL_{\neq}$-programas. Es decir, tenemos

Teorema 5.2 *(Completitud de $\rightsquigarrow^{ncs}$)*

Sea α una solución de una restricción aceptable φ. Entonces existe un cómputo localmente perezoso $\varphi \rightsquigarrow^{ncs^*} \psi$ terminado para α.

Sin embargo, ya fue comentada la debilidad del teorema 4.6, porque no establece en qué manera se pueden intercalar $\rightsquigarrow$-pasos y $\rightsquigarrow^{cs}$-pasos. Nuestro objetivo para esta sección es

obtener un resultado de completitud más fuerte, que muestre la indiferencia de dar en cada momento un tipo de paso u otro (mientras sean perezosos), y limite por tanto el indeterminismo 'don't know' a la elección de la regla del programa a usar (en el caso de pasos de estrechamiento), y a la elección de la regla de resolución de restricciones a usar, dentro de un juego de reglas completas (ver lema 5.4).

Para poder obtener un resultado de completitud más potente que el dado por el teorema anterior, nos conviene en primer lugar dar una definición de lo que entendemos por un $\rightsquigarrow^{ncs}$-cómputo globalmente perezoso. Recordemos que, en efecto, esa era la noción adecuada para probar la completitud del estrechamiento por restricciones $\rightsquigarrow$. La idea es que en un cómputo globalmente perezoso, cada paso te acerca realmente a la solución.

El siguiente ejemplo explica por qué no extendemos sin más la definición que ya tenemos para $\rightsquigarrow$-cómputos, olvidándonos sin más de los posibles $\rightsquigarrow^{cs}$-pasos intercalados.

Ejemplo 5.4 *(Existencia de $\rightsquigarrow^{cs}$-pasos no perezosos)*

Dado el programa con una sola regla $f(X) = s(0)$, el objetivo $[f(0), 0] =/= [0, s(0)]$ admite un cómputo $\varphi_0 \rightsquigarrow^{ncs} \varphi_1 \rightsquigarrow^{ncs} \varphi_2 \rightsquigarrow^{ncs} \varphi_3$ dado por

$$[f(0), 0] =/= [0, s(0)] \rightsquigarrow^{cs} f(0) =/= 0 \rightsquigarrow s(0) =/= 0 \rightsquigarrow^{cs} true$$

El cómputo no es perezoso, en el sentido de que $[f^0(0), 0] =/= [0, s(0)]$ es el único testigo minimal de φ_0, y sin embargo posteriormente $f(0)$ ha sido estrechada. Sin embargo, fijándonos sólo en los pasos de estrechamiento, el cómputo sería perezoso, pues el único testigo minimal de φ_1 es $f^1(0) =/= 0$, y por tanto el paso $\varphi_1 \rightsquigarrow \varphi_2$ es perezoso.

Lo que va mal aquí es el paso $\varphi_0 \rightsquigarrow^{cs} \varphi_1$, que informalmente no es perezoso. Pueden darse ejemplos de cómputos infinitos debidos a este hecho.
$\diamond$

Concluimos de este ejemplo que los $\rightsquigarrow^{cs}$-pasos de un cómputo deben ser tenidos en cuenta a la hora de hablar de pereza global. Antes de definir lo que entendemos por $\rightsquigarrow^{ncs}$-cómputos globlamente perezosos, necesitamos algunos tecnicismos acerca de cómo se transmiten a través de los $\rightsquigarrow^{cs}$-pasos los etiquetados (o árboles de interpretaciones) introducidos en la sección 4.3. Recordemos que en estos etiquetados cada símbolo de función estaba etiquetado en la forma $f^{T_{\mathcal{P}} \uparrow k}$ (abreviadamente f^k). Distintas apariciones de un mismo símbolo f podían tener distintos etiquetados. Un etiquetado y una valoración determinaban por completo el valor de una expresión, y por tanto, la satisfacción de una restricción. Escribiremos $e^\tau, \ldots, \varphi^\tau, \psi^\tau, \ldots$ para referirnos a expresiones y restricciones etiquetadas, y utilizaremos las notaciones habituales $\alpha[\![e^\tau]\!]$ (valor de la expresión etiquetada e^τ bajo α) y $\alpha \models \varphi^\tau$ (α es solución de la restricción etiquetada φ^τ).

Observemos en primer lugar que las reglas de $\rightsquigarrow^{cs}$ sirven para cualquier conjunto de símbolos no primitivos. En particular, sirven para el conjunto de símbolos etiquetados $\{f^n \mid f \in \Delta, n \in \mathbb{N}\}$. Por tanto tiene perfecto sentido hablar de $\varphi^\tau \rightsquigarrow^{cs} \psi^\tau$.

Nótese además que φ es $\rightsquigarrow^{cs}$-reducible si y solo si φ^τ, lo es, siendo τ un etiquetado de φ, ya que si $\varphi \rightsquigarrow^{cs}_R \psi$, entonces $\varphi^\tau \rightsquigarrow^{cs}_R \psi^\tau$, donde ψ^τ está definido del modo natural, 'traspasando'

las etiquetas de los símbolos no primitivos de φ a las correspondientes apariciones de esos mismos símbolos en ψ.

Fácilmente se prueba el siguiente resultado.

Lema 5.6

- Correción*: Si* $\varphi^\tau \rightsquigarrow^{cs} \psi^\tau$ *y* $\alpha \models \psi^\tau$, *entonces* $\alpha \models \varphi^\tau$.
- Completitud*: Si* $\alpha \models \varphi^\tau$ *y* φ^τ *reducible, existe* $\varphi^\tau \rightsquigarrow^{cs} \psi^\tau$ *tal que* $\alpha \models \psi^\tau$. *Es más, si* φ^τ *es minimal, uno de tales* ψ^τ *es minimal.*

Ya podemos dar la definición que andábamos buscando.

Definición 5.5 *($\rightsquigarrow^{ncs}$-cómputos perezosos)*
Sea φ_0 *aceptable y* α *solución de* φ_0. *Un cómputo*

$$\varphi_0 \rightsquigarrow^{ncs} \varphi_1 \rightsquigarrow^{ncs} \varphi_2 \rightsquigarrow^{ncs} \ldots$$

se dice perezoso para α *si existen* $\varphi_0^{\tau_0}, \varphi_1^{\tau_1}, \varphi_2^{\tau_2}, \ldots$, *testigos minimales para* α *de* $\varphi_0, \varphi_1, \varphi_2, \ldots$, *de modo que*

(i) Si $\varphi_i \rightsquigarrow^{ncs} \varphi_{i+1}$ *es* $\varphi_i \rightsquigarrow^{cs} \varphi_{i+1}$, *entonces* $\varphi_i^{\tau_i} \rightsquigarrow^{cs} \varphi_{i+1}^{\tau_{i+1}}$.

(ii) Si $\varphi_i \rightsquigarrow^{ncs} \varphi_{i+1}$ *es* $\varphi_i \rightsquigarrow \varphi_{i+1}$, *entonces* $\varphi_i^{\tau_i} \gg \varphi_{i+1}^{\tau_{i+1}}$.

Nótese que todo $\rightsquigarrow$-cómputo globalmente perezoso (ver la definición 4.14) es perezoso como $\rightsquigarrow^{ncs}$-cómputo.

A modo de ejemplo, comprobemos que el cómputo del ejemplo 5.4, que informalmente detectábamos como no perezoso, realmente no lo es de acuerdo con esta definición. En efecto, los únicos testigos minimales de los objetivos φ_0 y φ_1 son $[f^0(0), 0] =\!\!\!/\!\!= [0, s(0)]$ y $f^1(0) =\!\!\!/\!\!= 0$ respectivamente, y se tiene

$$[f^0(0), 0] =\!\!\!/\!\!= [0, s(0)] \rightsquigarrow^{cs} f^0(0) =\!\!\!/\!\!= 0$$

pero no

$$[f^0(0), 0] =\!\!\!/\!\!= [0, s(0)] \rightsquigarrow^{cs} f^1(0) =\!\!\!/\!\!= 0$$

por lo que la condición (i) de la definición anterior no se satisface.

Consideramos a partir de aquí soluciones α finitas, porque vamos a necesitar considerar una valoración α como una sustitución (entendiendo $\perp$ como un nuevo símbolo de constructora). Nos hará falta más adelante el siguiente resultado técnico.

Lema 5.7

Sea $\exists\overline{U}\varphi^\tau$ *un etiquetado de una restricción aceptable* $\exists\overline{U}\varphi$, *y* α *finita tal que* $\alpha \models \exists\overline{U}\varphi^\tau$. *Entonces existe* α' *finita tal que* α' *coincide con* α, *excepto posiblemente en* $\overline{U}$, *y* $\alpha' \models \varphi^\tau$.

Demostración:
Haremos inducción sobre la longitud l de la transformación que fue introducida en el teorema 5.1 para pasar de restricciones de $Con_{\Sigma_=}$ (que usan $=$, como implícitamente se supone de φ) a restricciones de Con_Σ (que utilizan selectoras y reconocedores). El considerar además etiquetados no afecta la validez de aquellas transformaciones.

- $(l = 0)$: entonces φ no tiene $=$, es decir, $\varphi \in Con_\Sigma$. Pero en este caso, φ^τ es una restricción continua y por tanto, de tener soluciones (y las tiene, pues $\exists\overline{U}\varphi^\tau$ es satisfactible) debe tener alguna finita α', que puede incluso hacerse coincidir con α fuera de $\overline{U}$.

- $(l \to l+1)$: El caso de $\exists\overline{U}\varphi_0^{\tau_0} \longmapsto_{T_2} \varphi_1^{\tau_1}$ no ofrece ningún problema, pues T_2 preserva soluciones y cuantificaciones.

  Supongamos ahora que

  $$\exists\overline{U}, V(e^\tau = V, \psi^\tau) \longmapsto_{T_1} \exists\overline{U}(\psi^\tau\sigma)$$

  siendo $\sigma \equiv \{V/e^\tau\}$, y sea α finita con $\alpha \models \exists\overline{U}, V(e^\tau = V, \psi^\tau)$. Como T_1 preserva soluciones, se tiene $\alpha \models \exists\overline{U}(\psi^\tau\sigma)$. Por la hipótesis de inducción, existe α'' finita, que coincide con α excepto posiblemente en $\overline{U}$, y tal que $\alpha'' \models \psi^\tau\sigma$. Nótese que α'' no compromete a V, pues V no está en $\psi^\tau\sigma$. Si consideramos ahora α' de modo que coincida con α'' salvo en V, donde definimos $\alpha'(V) = \alpha''[\![e^\tau]\!]$, tenemos que α' es finita ($\alpha'(V)$ es finito por serlo α'' y ser e^τ un etiquetado con niveles finitos) y además $\alpha' \models e^\tau = V, \psi^\tau$, lo que concluye la demostración.

□

El siguiente teorema establece que todo cómputo perezoso que no esté ya terminado se puede continuar.

Teorema 5.3

Sea φ_0 aceptable, α solución finita de φ_0, $\varphi_0 \rightsquigarrow^{ncs^*} \varphi_n$ perezoso para α, con testigo final $\varphi_n^{\tau_n}$. Entonces

(i) Si φ_n es $\rightsquigarrow^{cs}$-reducible, existe φ_{n+1} tal que $\varphi_n \rightsquigarrow^{cs} \varphi_{n+1}$ y $\varphi_0 \rightsquigarrow^{ncs^*} \varphi_{n+1}$ es perezoso.

(ii) Si u es una posición demandada (para α) en $\varphi_n^{\tau_n}$, entonces existe φ_{n+1} tal que $\varphi_n \rightsquigarrow_u \varphi_{n+1}$ y $\varphi_0 \rightsquigarrow^{ncs^*} \varphi_{n+1}$ es perezoso.

(iii) Si φ_n es $\rightsquigarrow^{cs}$-irreducible y $\varphi_n^{\tau_n}$ no tiene posiciones demandadas para α, entonces φ_n está terminado para α.

Demostración:

(i) Basta aplicar la completitud de $\rightsquigarrow^{cs}$ (lema 5.3).

(ii) Basta aplicar el lema de reducción de la complejidad 4.3.

(iii) Puesto que φ_n es $\rightsquigarrow^{cs}$-irreducible, sólo falta ver que φ_n es primitiva. Y en efecto es así, pues si φ_n fuese no primitiva, como $\varphi_n^{\tau_n}$ no tiene posiciones demandadas para α, se tendría $\alpha \models\mid \varphi_n \mid$, y por la condición de pereza de $\rightsquigarrow^{cs}$, φ_n sería $\rightsquigarrow^{cs}$-reducible, contra la hipótesis.

□

Este resultado es muy satisfactorio desde el punto de vista de cómo intercalar los $\rightsquigarrow^{cs}$-pasos y los $\rightsquigarrow$-pasos (en posiciones demandadas) en un cómputo. Los tres apartados, en su conjunto, indican que si un cómputo no está terminado, uno de los dos tipos de pasos se ha de poder dar, manteniendo la pereza. Además, en caso de que ambos tipos de pasos sean posibles, es indiferente cuál de ellos decidamos hacer, pues en ambos casos puede darse un paso perezoso.

Ahora bien, no serviría de mucho la noción de cómputo perezoso que hemos fijado, ni por tanto el teorema anterior, si no probamos que los cómputos perezosos nos acercan de verdad a la respuesta. El siguiente teorema, que es el análogo al lema 4.3 que vimos para $\rightsquigarrow$-cómputos, responde a esta cuestión, y constituye sin duda el resultado más importante de esta sección; todo lo demás se pueden considerar preparativos para él.

Teorema 5.4

Sea φ_0 aceptable, α solución finita de φ_0. Entonces no existen cómputos perezosos infinitos para α a partir de φ_0.

Demostración:
Supongamos que existe un cómputo perezoso infinito para α

$$\exists\overline{U}_0\varphi_0 \rightsquigarrow^{ncs} \exists\overline{U}_1\varphi_1 \rightsquigarrow^{ncs} \exists\overline{U}_2\varphi_2 \rightsquigarrow^{ncs} \ldots$$

con testigos minimales $\exists\overline{U}_0\varphi_0^{\tau_0}, \exists\overline{U}_1\varphi_1^{\tau_1}, \exists\overline{U}_2\varphi_2^{\tau_2}, \ldots$ Sea α' tal que α' coincide con α en las variables libres de $\exists\overline{U}_0\varphi_0$ (y por tanto también en las de $\exists\overline{U}_i\varphi_i$, para todo i), y que además verifique $\alpha' \models \varphi_i^{\tau_i}$ (tal α' se puede construir, teniendo en cuenta el lema 5.7, redefiniendo de modo adecuado α sobre las nuevas variables existenciales introducidas en cada paso).

Definiremos a continuación un orden bien fundado $\succ$ entre expresiones etiquetadas, y probaremos que

$$\varphi_i^{\tau_i} \succ \varphi_{i+1}^{\tau_{i+1}}, \forall i$$

con lo que habremos llegado a una contradicción.

El orden $\succ$ va a ser una combinación lexicográfica de varios órdenes auxiliares definidos sobre distintos 'tamaños' asociados a una restricción etiquetada. Estos 'tamaños', que notamos por $|-|_0$, $|-|_1$, $|-|_2$ y $|-|_3$, están definidos como sigue:

- $|\exists\overline{U}\varphi^{\tau}|_0 = S_\varphi\alpha'$, siendo S_φ el multiconjunto de los miembros de todas las condiciones $=, ==, =/=$ que aparecen en la parte no $=$-resuelta de φ^{τ}. La parte $=$-resuelta de φ^{τ}

está formada por las ecuaciones $X = t$ tales que X no aparece más veces en φ^τ. $S_\varphi\alpha'$ se entiende como el resultado de aplicar α' a cada elemento de S_φ.

El considerar el multiconjunto S_φ, en lugar de la propia parte $=$-resuelta de φ^τ, es debido a la regla $=/=_4$ (ver más abajo). El considerar la parte $=$-resuelta de φ^τ en lugar de la propia φ^τ es debido fundamentalmente a la regla $=/=_5$, aunque ya puestos se usa en otras (ver más abajo).

El orden auxiliar $\succ_0$ que se considera sobre estos multiconjuntos es el orden $\gg$ definido en la sección 4.3 (véase Def. 4.12).

- $\mid \exists\overline{U}\varphi^\tau \mid_1 =$ Número de variables distintas de φ^τ.
- $\mid \exists\overline{U}\varphi^\tau \mid_2 =$ Número de $==$-ecuaciones no resueltas de φ^τ. Una ecuación $X == t$ está resuelta si t es un término, y X no aparece en t ni en el resto de las igualdades $==$ ni desigualdades $=/=$ de φ^τ.
- $\mid \exists\overline{U}\varphi^\tau \mid_3 =$ Número de condiciones de la forma $l == X$ o $l =/= X$ de φ^τ.

  Los órdenes $\succ_1, \succ_2, \succ_3$, asociados a $\mid - \mid_1, \mid - \mid_2, \mid - \mid_3$ son todos el orden $>$ de los naturales.

Finalmente, definimos $\succ$ como

$$\varphi^\tau \succ \psi^\tau \Leftrightarrow \langle \mid \varphi^\tau \mid_0, \mid \varphi^\tau \mid_1, \mid \varphi^\tau \mid_2, \mid \varphi^\tau \mid_3 \rangle \succ_l \langle \mid \psi^\tau \mid_0, \mid \psi^\tau \mid_1, \mid \psi^\tau \mid_2, \mid \psi^\tau \mid_3 \rangle$$

siendo $\succ_l$ el orden lexicográfico correspondiente a $\succ_0, \succ_1, \succ_2, \succ_3$. Es bien fundado por serlo todos estos.

Nos resta probar: $\varphi_i^{\tau_i} \succ \varphi_{i+1}^{\tau_{i+1}}, \forall i$.

- Si $\varphi_i \rightsquigarrow_{u,R} \varphi_{i+1}$, observemos que la misma posición u y regla R sirven para hacer $\varphi_i\alpha' \rightsquigarrow_{u,R} \varphi_{i+1}\alpha'$, y que se puede reproducir el razonamiento utilizado en el lema 4.3 para probar $\varphi_i^{\tau_i}\alpha' \gg \varphi_{i+1}^{\tau_{i+1}}\alpha'$, e incluso, con la terminología introducida arriba, $S_\varphi\alpha' \succ_0 S_\psi\alpha'$, y por consiguiente $\varphi_i^{\tau_i} \succ \varphi_{i+1}^{\tau_{i+1}}$.

  A continuación analizamos $\rightsquigarrow^{cs}$ regla por regla (excepto las de fallo, que no influyen para este resultado). Como para el orden que consideramos las cuantificaciones existenciales son irrelevantes, las omitimos. Asimismo, para facilitar la lectura, no indicamos explícitamente en la notación los etiquetados para expresiones y restricciones; es decir, $l, r, e, \ldots$ serán expresiones etiquetadas, $\varphi, \psi, \ldots$ restricciones etiquetadas.

- $\varphi_i \rightsquigarrow^{cs}_{=_1} \varphi_{i+1}$: Claramente $\mid \varphi_i \mid_0$ decrece.

  Para varias de las reglas que siguen, hay que observar que si α' es solución de $X = t$, siendo t un término, y $\sigma \equiv \{X/t\}$, entonces $X\alpha' \equiv t\alpha'$ y $\sigma\alpha' \equiv \alpha'$.

- $\varphi_i \rightsquigarrow^{cs}_{=_3} \varphi_{i+1}$: Decrece $\mid \varphi_i \mid_1$, mientras $\mid \varphi_i \mid_0$ queda $\succeq$.
- $\varphi_i \rightsquigarrow^{cs}_{=_4} \varphi_{i+1}$: $\mid \varphi_i \mid_0$ decrece, pues una ecuación ha pasado a ser resuelta.
- $\varphi_i \rightsquigarrow^{cs}_{=_5} \varphi_{i+1}$: Igual que $=_3$.

- $\varphi_i \rightsquigarrow^{cs}_{=_6} \varphi_{i+1}$: Decrece $\mid \varphi_i \mid_0$, ya que t_i es subexpresión de l y $V_i\alpha'$ es subtérmino de $X\alpha'$, pues ha de ser $X\alpha' \equiv sk(l)\alpha'$.
- $\varphi_i \rightsquigarrow^{cs}_{==_1} \varphi_{i+1}$: Igual que $=_1$.
- $\varphi_i \rightsquigarrow^{cs}_{==_3} \varphi_{i+1}$: $\mid \varphi_i \mid_0, \mid \varphi_i \mid_1$ quedan igual, pero $\mid \varphi_i \mid_2$ decrece.
- $\varphi_i \rightsquigarrow^{cs}_{==_4} \varphi_{i+1}$: Como el caso de $=_6$.
- $\varphi_i \rightsquigarrow^{cs}_{==_5} \varphi_{i+1}$: $\mid \varphi_i \mid_0, \mid \varphi_i \mid_1$ quedan igual, $\mid \varphi_i \mid_2$ queda $\geq$, pero $\mid \varphi_i \mid_3$ decrece.
- $\varphi_i \rightsquigarrow^{cs}_{\neq_1} \varphi_{i+1}$: Como el caso de $=_1$.
- $\varphi_i \rightsquigarrow^{cs}_{\neq_2} \varphi_{i+1}$: Decrece $\mid \varphi_i \mid_0$, pues eliminamos una desigualdad, y el resto queda igual.
- $\varphi_i \rightsquigarrow^{cs}_{\neq_4} \varphi_{i+1}$: $\mid \varphi_i \mid_0, \mid \varphi_i \mid_1, \mid \varphi_i \mid_2$ quedan igual, pero $\mid \varphi_i \mid_3$ decrece. Nótese que $\mid \varphi_i \mid_0$ queda igual porque hemos considerado los multiconjuntos S_φ.
- $\varphi_i \rightsquigarrow^{cs}_{\neq_5} \varphi_{i+1}$: Decrece $\mid \varphi_i \mid_0$, pues hemos reemplazado una desigualdad $X =/= c(e_1, \ldots, e_n)$ por una $=$-ecuación resuelta.
- $\varphi_i \rightsquigarrow^{cs}_{\neq_6} \varphi_{i+1}$: Decrece $\mid \varphi_i \mid_0$, pues hemos reemplazado una desigualdad $X =/= c(e_1, \ldots, e_n)$ por una $=$-ecuación resuelta, más una desigualdad $e_i =/= V_i$ tal que $e_i\alpha'$ es subtérmino de $c(e_1, \ldots, e_n)\alpha'$ y $V_i\alpha'$ lo es de $X\alpha'$.

□

Los dos últimos teoremas tienen como consecuencia, ya casi inmediata, el siguiente teorema de completitud.

Teorema 5.5

Sea φ_0 una restricción aceptable y α una solución finita de φ_0. Entonces existe un cómputo $\varphi_0 \rightsquigarrow^{ncs^*} \varphi_n$ perezoso para α y terminado.

Como en otras ocasiones, el enunciado de completitud resulta muy claro, pero pierde mucha información con relación a los dos teoremas que le preceden. Como consecuencia de éstos últimos, podemos destacar el alto grado de indeterminismo 'don't care' que ofrece $\rightsquigarrow^{ncs}$ como mecanismo de cómputo para resolver objetivos, con garantías de llegar a una forma resuelta (si las elecciones 'don't know' son las adecuadas). En efecto:

1. En cada paso, podemos elegir indistintamente $\rightsquigarrow^{cs}$ o $\rightsquigarrow$ para continuar (teorema 5.3).
2. Para aplicar $\rightsquigarrow^{cs}$, podemos elegir indistintamente cualquier juego completo de $\rightsquigarrow^{cs}$-reglas de los determinados en el lema 5.4. Si el conjunto no es unitario, la selección de la regla es 'don't know'.
3. Para aplicar $\rightsquigarrow$, podemos elegir indistintamente cualquier posición absolutamente demandada (ver teorema 5.3). En la proposición 5.4 se dan criterios efectivos para detectar posiciones absolutamente demandadas, y se asegura que alguno de ellos se cumple en caso de que ninguna regla de $\rightsquigarrow^{cs}$ sea aplicable. La selección de la regla del programa P a usar es 'don't know'.

4. Según el teorema 5.4, el proceso termina si las elecciones 'don't know' son las adecuadas.

Así pues se dispone de una gran flexibilidad a la hora de elegir un control determinado para efectuar los cómputos.

5.6 Algunas variantes para la resolución de restricciones

Discutimos en este apartado alternativas para algunas de las características del sistema de resolución de restricciones que hemos estudiado en apartados anteriores.

5.6.1 Compartición frente a reemplazamiento

De las reglas de resolución de restricciones ($\rightsquigarrow^{cs}$), son las correspondientes a la unificación ($=$) las que contribuyen principalmente al carácter perezoso de la semántica operacional. Más concretamente, son interesantes a este respecto las reglas que se refieren a ecuaciones de la forma $l = X$. Cuando l es un término primitivo, la ecuación es eliminada tras generar y propagar la sustitución X/l. De esto se encarga la regla $=_5$, que está también al cargo de eliminar la ecuación $l = X$ si la variable X no aparece en ningún otro sitio, incluso si l es no primitiva. A pesar de su aparente inocuidad, este último tipo de eliminaciones es clave para el carácter perezoso de los cómputos, al hacer desaparecer las expresiones no primitivas cuyo valor es irrelevante (y por tanto su evaluación innecesaria) para la satisfactibilidad de la restricción en cuestión.

Cuando en la ecuación $l = X$, l es de la forma $c(e_1, \ldots, e_n)$ pero no primitiva , la regla $=_6$ se encarga de eliminar también la ecuación, tras efectuar una 'ligadura parcial' de la variable X a la parte primitiva más externa de l, sacando al tiempo hacia el exterior la parte no primitiva de l, en forma de ecuaciones $f(e_1, \ldots, e_n) = U$.

En cuanto a las ecuaciones de la forma $f(e_1, \ldots, e_n) = X$, donde X aparece más veces en el resto de la restricción, no hay ninguna regla específica que las trate. Puede decirse que están a la espera de que

- la variable producida X se reemplace por otro término (por aplicación de la regla $=_3$),
- la variable X desaparezca del resto de la restricción (por aplicación de las reglas $=/=_5$, $=/=_6$ a una ecuación $Y =/= r[X]$, o de la regla $=_5$ a $l[X] = Y$, con Y eliminable por no aparecer más), o
- se descubra que $f(e_1, \ldots, e_n)$ está demandada y debe ser reducida mediante $\rightsquigarrow$ (por aparecer la variable X en alguna condición $X == r, X =/= r$ o simétrica). El estrechamiento de $f(e_1, \ldots, e_n)$ (en varios pasos quizás) reducirá esta expresión a forma normal de cabeza (i.e. a una variable o a algo de la forma $c(e_1, \ldots, e_n)$), por lo que de la ecuación $f(e_1, \ldots, e_n) = X$ se pasará a otra de alguno de los tipos examinados antes [24].

[24] Como en muchas otras ocasiones, estamos suponiendo que la restricción en cuestión es satisfactible, y que las reglas del programa usadas para estrechar son las adecuadas. En otro caso, puede no llegarse a ninguna forma normal de cabeza para $f(e_1, \ldots, e_n)$.

Llamaremos *suspensión* a la expresión $f(e_1,\ldots,e_n)$ de una ecuación $f(e_1,\ldots,e_n) = X$ (y por extensión, llamaremos también 'suspensión' a toda la ecuación), para indicar que su evaluación queda pospuesta o suspendida hasta que se pruebe como realmente necesaria.

Es importante observar que en este planteamiento todas las sustituciones que se generan son de variables por términos, y que el resultado de estrechar en una suspensión $f(e_1,\ldots,e_n) = X$ se transmite a todas las apariciones de su variable producida X, a través de las ligaduras que produzca la posterior aplicación de la regla $=_5$ o $=_6$, según corresponda. Este fenómeno se corresponde muy directamente con un requisito pedido habitualmente a un mecanismo de evaluación (de expresiones funcionales) para ser considerado perezoso: evitar que se repita la evaluación de las apariciones múltiples que pudiera tener una expresión funcional (no primitiva, en nuestro lenguaje), cuando ha sido pasada como *parámetro actual* en la aplicación de una regla del programa cuyo lado derecho sea no lineal (es decir, con múltiples apariciones de un parámetro formal). En nuestro formalismo, evitamos tales repeticiones, porque el propio paso de parámetros (que queda expresado en el tratamiento que se da a las ecuaciones $l = X$) queda suspendido en caso de que l sea una expresión funcional, y en caso de ser reducida, el resultado es compartido por todas las apariciones de X.

Discutiremos aquí una alternativa a nuestro conjunto de reglas para la unificación, en la que se suprime la distinción entre variables producidas y no producidas. En esta otra definición de $\rightsquigarrow^{cs}$ consideramos, para el caso de restricciones con expresiones no primitivas, exactamente el mismo juego de $\rightsquigarrow^{cs}$-reglas para las $=$-ecuaciones que fue propuesto para el caso de restricciones primitivas. La diferencia reside en que las antiguas reglas $=_5, =_6$ para el caso no primitivo quedan reemplazadas por la regla $=_5$

$$\exists X, \overline{U}(t = X, \varphi) \rightsquigarrow^{cs} \exists \overline{U}(\varphi\sigma), \quad \text{donde} \quad \sigma \equiv \{X/t\}$$

que efectúa el paso de parámetros, con independencia de cual sea la estructura de t. Como cambio secundario adicional, en las reglas de $\rightsquigarrow^{cs}$ se suprime toda mención a 'variables producidas' (reglas $=\!\!\!/\!=_5, =\!\!\!/\!=_6$).

No es difícil probar que el nuevo sistema de resolución de restricciones $\rightsquigarrow^{cs}$ así definido sigue siendo correcto (debido al carácter existencial de las variables en los lados derechos de las $=$-ecuaciones), completo ($=_5$ preserva soluciones) y perezoso (la demostración requiere pocos cambios respecto a la dada para las reglas originales).

Es innegable que el conjunto de reglas que resulta es además más claro y sencillo. Como contrapartida, la semántica operacional no respeta el criterio de no reevaluación de las múltiples copias que de una expresión no primitiva e puede hacer la regla $=_5$ al aplicarse a una ecuación $e = X$ (una copia por cada aparición adicional de X). El hecho tiene su importancia, al menos de cara a una posible implementación, pues el número de posiciones no primitivas en una restricción puede crecer exponencialmente al aplicar la nueva regla $=_5$ (frente a un no crecimiento, incluso posible decrecimiento de tal número, con las reglas originales). Este problema suele ser abordado [103, 86] como una mera cuestión técnica a resolver mediante una adecuada representación de las expresiones funcionales que intervengan en los cómputos, para garantizar la compartición ('*sharing*') por parte de expresiones homólogas (copias) del resultado de la evaluación de una cualquiera de ellas. Por nuestra parte, aun considerando válido tal enfoque (que en cierto sentido adoptaremos en próximas secciones), juzgamos interesante el que la semántica operacional, en su nivel más formal, se ajuste más estrechamente a la subsiguiente implementación.

Existe otra razón, probablemente de mayor peso, para haber adoptado el conjunto de reglas original. Es que los resultados de completitud obtenidos en la sección 5.5 para la semántica operacional combinada $\leadsto^{ncs}$ se demuestran con mucha mayor facilidad si se evita la realización de copias de expresiones no primitivas. En particular, se simplifica en gran medida la obtención del orden bien fundado que nos permite probar en el teorema 5.4 que no existen $\leadsto^{ncs}$-cómputos perezosos infinitos. Lejos de parecernos un motivo vergonzante, pensamos que la mayor simplicidad de las pruebas, añadidas a la mejor adecuación a lo que se pretende realizar en la práctica, son fuertes avales para justificar nuestra elección. Otros autores [87, 21] han dado opiniones similares acerca de las ventajas de evitar el reemplazamiento impaciente ('*eager replacement*') de variables. Un caso bien conocido es el del conjunto de reglas para AC-unificación dado por Stickel en [144]; la dificultad de la prueba de terminación, que no fue dada hasta varios años más tarde por Fages [51], es achacada habitualmente [87, 21] al reemplazamiento impaciente de variables.

Indiquemos que, si bien no proporcionamos prueba de ello, nos parece una conjetura más que razonable que el teorema 5.4 sobre inexistencia de $\leadsto^{ncs}$- cómputos globalmente perezosos infinitos sea también válido para el sistema de reglas alternativo que hemos descrito aquí. El teorema 5.3 no ofrece mayor dificultad. En cuanto al teorema de completitud 'débil' 5.2, por supuesto que sigue siendo cierto, ya que es un resultado general para $CFLP(X)$.

Para terminar comentemos que, cuando estudiemos la implementación de la semántica operacional, adoptaremos inicialmente la regla $=_5$ que realiza copias. Lo haremos en interés de la claridad de la exposición, pues ciertamente resulta más simple que las reglas que consideran suspensiones. Pero nótese que la introducción posterior de las suspensiones para obtener compartición estará sólidamente justificada por nuestros resultados.

5.6.2 Propagación global de las sustituciones

Ya vimos en su momento (véase la sección 5.4) el peligro de la propagación global de una sustitución $\sigma \equiv \{X/t\}$ correspondiente a una ecuación $X == t$ en la aplicación de $==_3$, y análogamente en $==_4$. Por ese motivo en dichas reglas la sustitución generada no se propaga más que al resto de las $==$-ecuaciones y a las $=/=$-desigualdades. Aunque existen otras soluciones, que en seguida analizaremos brevemente, hemos querido hacerlo así en el sistema de restricciones estudiado con detalle, porque así se obtiene la notable propiedad de que cualquier orden de aplicación de las reglas, incluyendo la de estrechamiento (en posiciones demandadas), es válido. Es decir, para cualquier control se tiene completitud.

Pensando en una posible implementación, sin embargo, restringir la aplicación de una sustitución a un ámbito local puede resultar más difícil que aplicarla globalmente. Ello ciertamente es así si, como haremos, usamos Prolog como lenguaje soporte para la implementación, en cuyo caso lo razonable y eficiente es usar la propia unificación de Prolog (de efecto global) para realizar y propagar las sustituciones generadas por las semántica operacional implementada. Así pues, como alternativa práctica, proponemos un cambio en las reglas para $==$, para aplicar las sustituciones globalmente, a costa de imponer condiciones adicionales.

La nueva regla $==_3$ sería

$==_3)$ $\exists\overline{U}(X == t, \varphi) \rightsquigarrow^{cs} \exists\overline{U}(X == t, \varphi\sigma)$
si t es un término, X aparece en φ, X no aparece en t, $\sigma \equiv \{X/t\}$.
y X, t no contiene variables producibles

Para $==_4$ tendríamos un cambio similar, pidiendo en este caso que no haya variables producibles en $X, sk(r)$.

Entendemos por *variables producibles* aquellas que aparecen en el lado derecho de una $=$-ecuación no resuelta. Con más precisión:

U es producible en φ si φ tiene una ecuación $l = r$ tal que $U \in var(r)$ y , o bien l no es una variable, o bien l es una variable que aparece más veces en φ.

Es claro que, con la condición impuesta, al aplicar $==_3$ no pueden generarse ciclos de variables en las $=$-ecuaciones no resueltas (pues las variables involucradas en $\{X/t\}$ no están en sus lados derechos); por lo que se refiere a las $=$-ecuaciones resueltas, es obvio que no pueden formar parte de ningún ciclo, pues sus lados izquierdos sólo aparecen una vez en toda la restricción.

Así pues, la propiedad (**C**) más esquiva de las restricciones aceptables se sigue preservando. También es obvio que se preserva la propiedad (**T**) de que sólo haya términos primitivos en los lados derechos de las $=$-ecuaciones. En cuanto a la linealidad de los lados derechos (**L**) y el que las variables de los lados derechos sean existenciales (**E**), pueden perderse en las $=$-ecuaciones resueltas. Por ejemplo, una aplicación legítima de la nueva $==_3$ sería

$\exists U, V(X = c(U),\ U = c(V),\ U == d(Y, Y))$
$\rightsquigarrow^{cs}_{==_3} \exists U, V(X = c(d(Y, Y)),\ d(Y, Y) = c(V),\ U == d(Y, Y))$

en la que hemos pasado de una restricción aceptable a otra en la que (**L**,**E**) fallan.

Pero, de nuevo en general, las propiedades (**L**,**E**) sí que se preservan en las $=$-ecuaciones no resueltas, pues sus lados derechos no son tocados. Y esto es lo importante para garantizar que las reglas de $\rightsquigarrow^{cs}$ siguan funcionando bien, ya que una $=$-ecuación resuelta no puede dejar de estarlo, y sólo pueden ser afectadas de nuevas instanciaciones de sus lados derechos que, ni pueden ser incorrectas, ni conducir a problemas de no terminación por la generación de ciclos. Para una justificación precisa de la validez del nuevo sistema de reglas, que no vamos a completar aquí, deberíamos pues relajar las condiciones de aceptabilidad, pidiendo (**L**,**E**) sólo a la parte no resuelta de una restricción, y probar resultados similares a los que ya obtuvimos.

Con las nuevas reglas hemos conseguido que toda sustitución que se genere durante el cómputo se aplique globalmente, lo que ya sabemos que reporta ventajas para una implementación, especialmente en Prolog. Pero ¿qué ocurre con las condiciones de aplicación de las reglas? No nos referimos ahora a que son aplicables en menos ocasiones, con lo que se pierde flexibilidad en el control, sino a la verificación de que se cumplen tales condiciones, es decir, que la sustitución a efectuar no involucra, ni en el dominio ni el el rango, una variable producible. Por supuesto, detectar tal situación es posible, pero claramente a costa de sofisticar la representación que se vaya a hacer de los términos (para distinguir variables

producibles) y/o efectuar recorridos costosos para reconocerlas. Probablemente con ello se contrarrestarían las ventajas conseguidas por las nuevas reglas.

Existe una forma de preservar las ventajas de la propagación global de sustituciones, al tiempo que se evita el inconveniente de tener que verificar si una ecuación contiene o no variables producibles: imponer restricciones más severas al orden en que se resuelven las condiciones de un objetivo, para asegurar que en el momento en que una ecuación estricta es considerada para ser resuelta, ya no tenga variables producibles. Un criterio muy sencillo para esto es el siguiente: *al hacer estrechamiento mediante una regla del programa, resolver por completo el problema de unificación correspondiente a la regla, antes de pasar al resto de las condiciones (==-ecuaciones y =/=-desigualdades).* Nótese que para resolver el problema de unificación pueden ser necesarios nuevos estrechamientos, a los que habría que aplicar de nuevo este criterio, etc. Este carácter recursivo del criterio anterior, lejos de ser un problema, resulta muy natural para una implementación Prolog, y de hecho va a ser adoptado por nosotros en el siguiente capítulo.

No queremos que, de la discusión anterior, se saque la conclusión de que consideramos la solución finalmente propuesta como la mejor posible. Simplemente pretendemos haber justificado que es una solución realista y fácil de llevar a la práctica.

5.6.3 Limitación a soluciones finitas y totales

La diferencia entre el significado de las igualdades = y == hace que dos ecuaciones resueltas $X = t$ y $X == t$ proporcionen información diferente. La segunda impone que X sea finito y total, mientras que la primera acepta como soluciones valores de X finitos o infinitos, totales o parciales. Eso quiere decir en particular que no podemos usar simplemente sustituciones para representar esas ecuaciones resueltas, pues a ambas les correspondería la sustitución $\{X/t\}$, y no podríamos distinguirlas. Relacionado con esto está el hecho de que las ecuaciones de la forma $X == X$ no se puedan eliminar, ya que no son trivialmente satisfactibles: imponen a X la condición de ser finito y total. Algo similar ocurre con las desigualdades del tipo $X =/= t$, cuando X aparece en t: tampoco son trivialmente satisfactibles, pues algunos valores infinitos no la verifican.

Ahora bien, si restringimos nuestro interés a las soluciones finitas y totales del objetivo inicial (lo que tiene sentido por la proposición 5.3), podemos ser más liberales en los tres aspectos indicados: podemos representar tanto $X = t$ como $X == t$ mediante la sustitución $\{X/t\}$, así como eliminar las ecuaciones $X == X$ y las desigualdades $X =/= e$, cuando X aparece en $| e |$. Estamos asumiendo aquí que, como propusimos en el apartado anterior, al tratar una ==-ecuación o una =/=-desigualdad, estamos seguros de que sus variables no son producibles. En otro caso podemos cometer una incorreción, como sucedería en

$$\exists X(f(0) = X, X == X) \leadsto^{cs} \exists X(f(0) = X) \leadsto^{cs} true$$

en caso de que $f(0)$ fuese parcial o infinito.

Al decir que podemos representar tanto $X = t$ como $X == t$ mediante la sustitución $\{X/t\}$ nos referimos a que en el momento en que una de dichas ecuaciones pasa a ser resuelta, y la sustitución $\{X/t\}$ es aplicada al resto de la restricción, la propia sustitución sirve como testigo de la condición que debe cumplir X, y por tanto no es necesario conservar la ecuación

para dicho propósito. Nótese que esto no afecta a la naturaleza perezosa del lenguaje, pues las = y ==-ecuaciones se resuelven de forma distinta, cada una de acuerdo a su semántica. Es sólo el tratamiento de las formas resueltas el que se hace igual.

Una consecuencia interesante de adoptar esta nueva variante es que una respuesta consta ahora de una sustitución idempotente $\sigma = \{X_1/t_1, \ldots, X_n/t_n\}$ más una conjunción de desigualdades $Y_1 =/= s_1, \ldots, Y_m =/= s_m$ tales que en ellas no aparecen las variables X_i, ni las variables Y_j aparecen en s_j.

También estos criterios serán aplicados en el próximo capítulo.

6 Implementación

En este capítulo utilizaremos los resultados de la sección anterior para proponer una implementación de $SFL_{\neq}$. Pretendemos hacerlo descendiendo de una manera gradual desde la semántica operacional abstracta proporcionada por $\leadsto^{ncs}$ hasta una implementación de 'bajo nivel' razonablemente eficiente. Para todos los pasos de este refinamiento gradual, utilizaremos Prolog como lenguaje común. Podemos pensar inicialmente en Prolog como lenguaje de especificación abstracto, y al final como el lenguaje de bajo nivel, de cuyos recursos se aprovecha sin escrúpulos nuestra implementación [25]. Otros formalismos, por supuesto, pueden ser utilizados para una especificación. En [16, 17], por ejemplo, utilizamos el formalismo de las 'álgebras dinámicas' [67], para intentar especificar implementaciones del estrechamiento basadas en máquinas abstractas [91, 102]. Nuestro interés ahora está en obtener de modo inmediato una implementación a partir de la especificación.

6.1 Una especificación de la semántica operacional

La especificación que propondremos está muy relacionada con las que se presentan en [103] para el lenguaje SFL. La especificación consiste realmente en la descripción de un proceso $\mathcal{PT}$ de traducción a Prolog, que convierte cada $SFL_{\neq}$-programa P en un conjunto de cláusulas $\mathcal{PT}(P)$, y cada objetivo G en un objetivo Prolog $\mathcal{PT}(G)$. La idea es que el SLD-espacio de búsqueda para el objetivo $\mathcal{PT}(G)$ determinado por las cláusulas $\mathcal{PT}(P)$ especifica $\leadsto^{ncs}$-cómputos que conducen a las soluciones de G. No es necesario suponer en principio que el régimen de control de Prolog (selección del objetivo más a la izquierda + búsqueda en profundidad y por la izquierda) sea el utilizado para la construcción del árbol ni para su recorrido. Allá donde haya alguna restricción especial en ese sentido, se indicará.

Hay algunos otros aspectos de la especificación que debemos comentar previamente a su presentación:

- La especificación se adecúa a las variantes de $\leadsto^{ncs}$ que se indicaron al final de la sección anterior:
  - 'Reemplazamiento' en lugar de 'compartición' (sólo de modo temporal).
  - Prioridad de las condiciones de unificación sobre las igualdades y desigualdades de la misma regla. Y como resultado de ésta
  - Efecto global de todas las sustituciones aplicadas.
  - Limitación a las soluciones finitas y totales.
  - Una respuesta consta de una sustitución más un conjunto de desigualdades de la forma $X =/= t$
- Las sustituciones que se apliquen durante el cómputo, y en consecuencia la sustitución de una respuesta, no se hacen explícitas en la especificación. Quedan subsumidas por la unificación de Prolog.

[25] Es bien conocido el prestigio de Prolog como lenguaje para escribir especificaciones ejecutables; menos citada, pero cada vez más extendida, está la idea de Prolog como '*ensamblador glorificado*' (la terminología es de Mario Rodríguez Artalejo).

- Resulta conveniente para la especificación concebir un conjunto de desigualdades asociadas a una misma variable X, como serían $X =/= t_1, \ldots, X =/= t_n$, como una expresión en sí misma, que leeríamos como 'X *tal que* $X =/= t_1, \ldots, X =/= t_n$', y a la que denominamos *variable restringida*. Otra forma de expresar esto mismo es decir que, en la representación que se vaya a adoptar para las expresiones y las restricciones de los cómputos, asumimos que una variable X está asociada, en cada una de sus apariciones, con la conjunción $X =/= t_1, \ldots, X =/= t_n$ de las desigualdades ya resueltas para ella. Con esta lectura, una respuesta está constituida por una sustitución más un conjunto de variables restringidas.

  Una idea similar a la de variable restringida se utiliza en [93, 94], donde se denomina *entorno* al conjunto de las asociaciones de variables con sus correspondientes restricciones de desigualdad.

- Para la especificación se debe partir de una determinada representación (como términos Prolog) de las expresiones del lenguaje fuente. Por lo que se refiere a las expresiones que aparecen en un programa o en el objetivo inicial, consideramos la representación natural, o sea, la identidad: $R[e] \equiv e$, $R[\varphi] \equiv \varphi$.

  Sin embargo, la aparición de variables restringidas durante el cómputo introduce un problema de representación cuya solución no es obvia (ni por supuesto única). Y puesto que una variable restringida puede ser argumento de una expresión más compleja, el problema de representación puede afectar en principio a todas las expresiones generadas durante el cómputo (*expresiones de cómputo*).

  Para no mezclar el problema de la representación con la especificación de la semántica operacional, basamos ésta en una representación de las expresiones de cómputo sólo parcialmente determinada. Asumimos simplemente la existencia de algunos procedimientos (pendientes de definir) que nos permitan realizar algunas operaciones básicas: reconocer la clase de expresión representada, acceder a componentes, etc...

  A pesar de lo anterior, en los comentarios que acompañan a la especificación se hablará a menudo de '*...la expresión E...*' en lugar de '*...la expresión representada por E...*'. Sólo en algunos casos nos ha parecido interesante hacer la distinción entre ambas.

Presentamos ya la especificación. En las cláusulas que siguen, cuando aparezcan $c, d, f, \ldots$ representando símbolos concretos de constructora o función, debe entenderse que hay una cláusula para cada símbolo. Damos en primer lugar una relación de los procedimientos dependientes de la representación, junto con una explicación informal de su significado pretendido. Deben entenderse como predicados extralógicos, en el mismo estilo de los $var/1, = /2, = ../2$ de Prolog.

- $es\_var(E)$: E representa una variable, restringida o no.
- $es\_c\_ap(E, c(E_1, \ldots, E_n))$: E representa una expresión $c(e_1, \ldots, e_n)$, donde cada E_i representa a e_i.
- $es\_f\_ap(E, f(E_1, \ldots, E_n))$: E representa una expresión $f(e_1, \ldots, e_n)$, donde cada E_i representa a e_i.

- $misma\_var(X,Y)$: X e Y representan a la misma variable.
- $compatibles\_vars(X,Y)$: X e Y representan a dos variables RX y RY que no están forzadas a ser diferentes, es decir, $RX =/= RY$ no está entre las restricciones de RX (y viceversa).
- $restricciones(X,C)$: C representa $t_1,\ldots,t_n$ si X es una variable con restricción asociada $X =/= t_1,\ldots,X =/= t_n$. Si X no está restringida, C verifica el predicado $es\_vacia(C)$.
- $inserta\_restricciones(C,X)$: Si C representa $t_1,\ldots,t_n$, añade a X las restricciones $X =/= t_1,\ldots,X =/= t_n$.
- $inserta\_restriccion(T,X)$: Añade a X la restricción $X =/= T$.
- $sustituye(X,T)$: efectúa la sustitución X/T. Suponemos que, como en el caso de $X = T$ en Prolog, [26] el efecto es global, es decir, afecta a todas las apariciones de X.

Comenzamos por especificar cómo se resuelve un conjunto de ==-ecuaciones y desigualdades, que es lo que constituye un objetivo inicial. Utilizaremos repetidas veces la noción de *forma normal de cabeza (fnc)*: una expresión e está en forma normal de cabeza si es una variable o tiene la forma $c(e_1,\ldots,e_n)$, siendo c una constructora.

```
/**********
resuelve(C): resuelve el conjunto de ==-ecuaciones y desigualdades C.
*********/

resuelve((Cond,Rest)) :- resuelve(Cond), resuelve(Rest).
resuelve(L == R) :- iguales(L,R).
resuelve(L =/= R) :- distintos(L,R).

/**********
iguales(L,R): resuelve la igualdad estricta L == R.
distintos(L,R): resuelve la desigualdad L =/= R.
*********/

iguales(L,R) :- fnc(L,HL), fnc(R,HR), iguales_fnc(HL,HR).

distintos(L,R) :- fnc(L,HL), fnc(R,HR), distintos_fnc(HL,HR).
```

Las reglas para *iguales* y *distintos* prevén la posibilidad de que alguno de sus dos miembros no estén en forma normal de cabeza (fnc), es decir, no sean una variable (posiblemente

[26] Obsérvese que, aunque el efecto que se pretende es similar, $sustituye(X,T)$ no puede reemplazarse por $X = T$ (unificación sintáctica de Prolog), ya que la variable de cómputo X puede estar representada mediante un término Prolog que no sea sintácticamente unificable con T).

restringida) o una expresión de la forma $c(e_1, \ldots, e_n)$, sino que sean de la forma $f(e_1, \ldots, e_n)$, con $f \in \Delta$. Sabemos entonces que ésa es una posición absolutamente demandada en la que es legítimo hacer estrechamiento (tantos pasos como sea preciso para obtener una forma normal de cabeza). El predicado fnc, que se especificará más adelante, se encarga de ello. Veamos primero cómo continúa la resolución de las condiciones.

```
/**********
iguales_fnc(L,R): resuelve L == R, donde L,R están en fnc.
*********/

iguales_fnc(X,H) :- es_var(X), !, ig_var(X,H).
iguales_fnc(H,X) :- es_var(X), !, ig_var(X,H).
iguales_fnc(L,R) :-
  es_c_ap(L, c(L1,...,Ln)),
  es_c_ap(R, c(R1,...,Rn)),
  iguales(L1,R1),...,iguales(Ln,Rn).
```

La tercera cláusula realiza descomposición, como indica la regla $==_1$ de $\leadsto^{cs}$. La regla de fallo por conflicto de constructoras ($==_2$) queda expresada implícitamente por el fallo de Prolog, al no haber claúsula aplicable a tal situación. Las dos primeras cláusulas remiten sin más a ig_var, para los casos en que uno de los dos miembros de la $==$-ecuación sea una variable. Esto cubre todos los casos posibles para $iguales\_fnc$. Llamamos la atención – por ser la primera vez que aparece uno de los predicados extralógicos no especificados – sobre el uso de $es\_var(X)$ para expresar la condición de que X es (representa, mejor) una variable. No podemos usar en su lugar el predicado predefinido de Prolog var, pues una variable de un $SFL_{\neq}$-cómputo podría no estar representada por una variable Prolog.

```
/**********
ig_var(X,H): resuelve X == H, donde X es una variable, y H está en fnc.
*********/

ig_var(X,Y) :-
  es_var(Y), misma_var(X,Y), !.
ig_var(X,Y) :-
  es_var(Y), !,
  compatibles_vars(X,Y),
  restricciones(X,C), inserta_restricciones(C,Y), sustituye(X,Y).
ig_var(X,E) :-
  es_c_ap(E, c(E1...,En)),
  no_aparece(X, c(E1,...,En)),
  pasa_a_resuelta(X, c(T1,...,Tn)),
  iguales(T1,E1),...,iguales(Tn,En).
```

La primera cláusula está eliminando una ecuación del tipo $X == X$. La segunda cláusula de ig_var corresponde a la regla $==_3{}'$ de $\leadsto^{cs}$, es decir, al caso de una ecuación $X == Y$ donde X e Y son variables, posiblemente con restricciones $X =/= t_1, \ldots, X =/= t_n$ e

$Y =\!\!/\!= s_1, \ldots, Y =\!\!/\!= s_m$, respectivamente. El procedimiento $compatibles\_vars(X,Y)$ se encarga de comprobar que $X =\!\!/\!= Y$ no sea una de ellas, pues en ese caso la ecuación $X == Y$ no es resoluble. En otro caso, añadimos a Y las restriciones $Y =\!\!/\!= t_1, \ldots, Y =\!\!/\!= t_n$ (para seguir el principio de que la variable Y debe estar asociada con sus restricciones $Y =\!\!/\!= t$ correspondientes) y ligamos X a Y.

La tercera cláusula se encarga del caso $X == c(e_1, \ldots, e_n)$, realizando la imitación indicada en $==_4{}'$. Como dijimos entonces, esta imitación puede ser muy trabajosa, y será reemplazada más adelante por otra más eficiente que refleje las reglas $==_3$ y $==_4$. En la imitación se genera una ecuación $X == c(T_1, \ldots, T_n)$ que queda en forma resuelta (y por tanto no se conserva) al aplicar la sustitución $X/c(T_1, \ldots, T_n)$. Tenemos un procedimiento específico, $pasa\_a\_resuelta$, para este paso a forma resuelta, porque hay algún trabajo adicional que hacer con las restricciones que pudiera tener X.

/**********
$pasa\_a\_resuelta(X,T)$: Efectúa la sustitución $\{X/T\}$ y propaga el vínculo a las restricciones asociadas a X.

$propaga(T,C)$: Si C representa $X =\!\!/\!= t_1, \ldots, X =\!\!/\!= t_n$, $propaga(T,C)$ resuelve las desigualdades $T =\!\!/\!= t_1, \ldots, T =\!\!/\!= t_n$.
*********/

$pasa\_a\_resuelta(X,T) :-$
  $restricciones(X,C), sustituye(X,T), propaga(T,C).$

$propaga(T,C) :- es\_vacia(C).$
$propaga(T,C) :- selecciona(S,C,Resto), resuelve(T =\!\!/\!= S), propaga(T,Resto).$

El sentido de $pasa\_a\_resuelta(X,T)$ sólo puede entenderse desde el punto de vista de las 'variables restringidas'. La idea es que las restricciones $X =\!\!/\!= t$ que se han ido generando, no están guardadas en ningún almacén global (del estilo de un argumento adicional, digamos *Restricciones*, que se arrastra de procedimiento a procedimiento a lo largo de todo el cómputo) sino que aparecen asociadas a la propia variable, de un modo aún no especificado. En el momento de aplicar la sustitución $\{X/T\}$, una restricción asociada $X =\!\!/\!= s$ ha de ser 'despertada', pues se ha convertido en $T =\!\!/\!= s$, que no estará resuelta en general.

El procedimiento auxiliar *propaga* simplemente resuelve una a una las desigualdades activadas de esta manera.

Con el predicado $no\_aparece$, que se encarga del 'occur check', queda completada la especificación de la resolución de ecuaciones estrictas.

/**********
$no\_aparece(X,E)$: verifica que X no aparece en la cáscara de E.
*********/

$no\_aparece(X,Y) :- es\_var(Y), !, not\ \ misma\_var(X,Y).$

$no\_aparece(X, E) :-$
  $es\_c\_ap(E, c(E_1, \ldots, E_n), !,$
  $no\_aparece(X, E_1), \ldots, no\_aparece(X, E_n).$
$no\_aparece(X, E).$ % E es no primitiva

Veamos ahora cómo especificar la resolución de desigualdades $l =/= r$, cuando en ambos miembros hay ya una forma normal de cabeza.

```
/**********
distintos_fnc(L,R): resuelve L =/= R, donde L,R están en fnc.
*********/
```

$distintos\_fnc(X, H) :- es\_var(X), !, nig\_var(X, H).$
$distintos\_fnc(H, Y) :- es\_var(Y), !, nig\_var(Y, H).$
$distintos\_fnc(L, R) :-$
  $es\_c\_ap(L, c(L_1, \ldots, L_n)),$
  $es\_c\_ap(R, d(R_1, \ldots, Rm).$
$distintos\_fnc(L, R) :-$
  $es\_c\_ap(L, c(L_1, \ldots, L_n)),$
  $es\_c\_ap(R, c(R_1, \ldots, R_n),$
  $(distintos(L_1, R_1)$
  ;
  ..... % Elección indeterminista
  ;
  $distintos(L_n, R_n)).$

Las dos primeras cláusulas remiten a nig_var para el caso de que uno de los dos miembros sea una variable. Las dos siguientes expresan directamente las reglas $=/=_1$ y $=/=_2$.

```
/**********
nig_var(X,H): resuelve X =/= H, donde X es una variable, y H está en fnc.
********/
```

$nig\_var(X, Y) :-$
  $es\_var(Y), !, not\ \ misma\_var(X, Y),$
  $inserta\_restriccion(Y, X),$
  $inserta\_restriccion(X, Y).$
$nig\_var(X, T) :-$
  $es\_termino(T), !, inserta\_restriccion(T, X).$

$nig\_var(X, E) :-$
  $es\_c\_ap(E, c(E_1, \ldots, E_n)),$
  $(unifica(X, d(U_1, \ldots, U_m))$ % Una alternativa para cada $d \not\equiv c$
  ;
  $unifica(X, c(U_1, \ldots, U_n)),$
  $distintos(U_i, E_i)).$ % Una alternativa para cada $i = 1 \cdots n$

La primera cláusula corresponde a una desigualdad $X =/= Y$ entre variables. Cubre, por una parte, la regla de fallo $=/=_3$, al pedir que X e Y no sean la misma variable. Si en efecto no lo son, recuérdese que no hay $\rightsquigarrow^{cs}$-regla para $X =/= Y$ porque se considera resuelta. Sin embargo, siguiendo una vez más la filosofía de las 'variables restringidas', nuestra especificación debe tomarse el trabajo de anotar la desigualdad $X =/= Y$ como nueva restricción de la X, y recíprocamente. Análogamente sucede con la tercera cláusula. Obsérvese que todas las restricciones asociadas a una variable son de la forma $X =/= t$, donde t es un término. El predicado $es\_termino(T)$ puede añadirse a la lista de predicados no especificados, o bien ser programado utilizando los ya existentes.

$es\_termino(X) :- es\_var(X), !.$
$es\_termino(T) :-$
  $es\_c\_ap(T, c(T_1, \ldots, T_n)),$
  $es\_termino(T_1), \ldots, es\_termino(T_n).$

La última cláusula – muchas en realidad, recogidas en un esquema – recoge las dos alternativas indeterministas ('don't know') expresadas en las reglas $=/=_5$ y $=/=_6$ de $\rightsquigarrow^{cs}$. A su vez cada una de ellas se desdobla en varias, correspondientes a distintas constructoras en el caso de $=/=_5$, y a distintos argumentos en el de $=/=_6$.

Veamos a continuación como se resuelven ecuaciones no estrictas $e = t$, es decir, cómo se realiza la unificación.

/**********
$unifica(E, T)$: Realiza la unificación de la expresión E y el término lineal T.
*********/

$unifica(E, T) :- var(T), !, T = E.$
$unifica(E, T) :- fnc(E, H), unifica\_fnc(H, T).$

La primera cláusula expresa el paso de parámetros *sin* compartición, es decir la regla $=_5$ en la versión de restricciones primitivas. En la segunda, puesto que T ha de ser de la forma $c(t_1, \ldots, t_n)$, se solicita una forma normal de cabeza de E.

/**********
$unifica\_fnc(H, T)$: Realiza la unificación de la forma normal de cabeza H y el término lineal T.
*********/

$unifica\_fnc(E, T) :-$
  $es\_var(E), !,$
  $pasa\_a\_resuelta(E, T).$
$unifica\_fnc(E, c(T_1, \ldots, T_n)) :-$
  $es\_c\_ap(E, c(E_1, \ldots, E_n)),$
  $unifica(E_1, T_1), \ldots, unifica(E_n, T_n).$

La primera cláusula se hace cargo de las reglas $=_3$ y $=_4$ de $\rightsquigarrow^{cs}$. Establece que para unificar una variable X con un término t, simplemente la pasamos a forma resuelta, como ya fue descrito para el caso de $==$-ecuaciones. Recordemos en este punto que el tratamiento que damos a las ecuaciones estrictas y no estrictas, una vez que están resueltas, es el mismo.

En cuanto a la segunda, expresa la descomposición de $=_1$.

Sólo resta, para completar nuestra especificación, indicar en qué consiste el procedimiento fnc para obtener una [27] forma normal de cabeza.

/**********
$fnc(E, H)$: Devuelve en H una forma normal de cabeza de E.
*********/

$fnc(E, H) : -es\_var(E), !, H = E.$
$fnc(E, H) : -es\_c\_ap(E, c(E_1, \ldots, E_n)), !, H = c(E_1, \ldots, E_n).$
$fnc(E, H) : -es\_f\_ap(E, f(E_1, \ldots, E_n)), !, \#f(E_1, \ldots, E_n, H).$

El símbolo $=$ de la segunda cláusula es la unificación de Prolog. El predicado $\#f$ que aparece en la tercera cláusula corresponde a un símbolo de función no primitiva $f \in \Delta^n$ (un predicado para cada símbolo, por supuesto), y va a servir para expresar el estrechamiento realizado mediante las reglas del $SFL_{\neq}$-programa de que disponga f. Nótese que la aridad de $\#f$ es $n+1$ si f tiene aridad n. El argumento adicional, colocado convencionalmente en último lugar, sirve para recoger el resultado del estrechamiento.

/**********
$\#f(E_1, \ldots, E_n, H)$: Reduce $f(E_1, \ldots, E_n)$ a forma normal de cabeza, y devuelve el resultado en H.
*********/

% Para cada $SFL_{\neq}$-regla $f(t_1, \ldots, t_n) = e \Leftarrow cond$

$\#f(E_1, \ldots, E_n, H) : -$
  $unifica(E_1, t_1), \ldots, unifica(E_n, t_n),$
  $resuelve(cond),$
  $fnc(e, H).$

La alternativa entre las distintas reglas para una misma f es indeterminista (don't know'). En esta cláusula debe entenderse que la unificación tiene prioridad (debe resolverse antes) sobre la resolución de la restricción *cond*.

Como se puede ver, esta cláusula coincide prácticamente con lo que hemos denominado la $\Sigma_=$-traducción de la $SFL_{\neq}$-regla de partida. La mayor diferencia estriba en el literal $fnc(e, H)$, que es el responsable de que se realicen posiblemente más pasos de estrechamiento, si es que e es a su vez de la forma $g(\ldots)$.

Veamos cómo quedarían en algún ejemplo concreto las cláusulas para los predicados $\#f$.

[27] Decimos '*una*' porque, si hace falta estrechamiento para ello, distintas derivaciones nos pueden conducir a distintas formas normales de cabeza (para valores distintos de las variables, por supuesto).

Ejemplo 6.1

Si consideramos el $SFL_{\neq}$-programa del ejemplo 4.10, las cláusulas para los predicados $\#elem$ y $\#card$ quedarían así

$$
\begin{array}{l}
\#elem(X, L, H) : - \\
\qquad unifica(L, [\,]), fnc(false, H). \\
\#elem(X, L, H) : - \\
\qquad unifica(L, [Y \mid Ys]), resuelve(X == Y), fnc(true, H). \\
\#elem(X, L, H) : - \\
\qquad unifica(L, [Y \mid Ys]), resuelve(X =\!\!= Y), fnc(elem(X, Ys), H).
\end{array}
$$

$$
\begin{array}{l}
\#card(L, H) : - \\
\qquad unifica(L, [\,]), fnc(0, H). \\
\#card(L, H) : - \\
\qquad unifica(L, [X \mid Xs]), resuelve(elem(X, Xs) == true), fnc(card(Xs), H). \\
\#card(L, H) : - \\
\qquad unifica(L, [X \mid Xs]), resuelve(elem(X, Xs) == false), fnc(s(card(Xs)), H)
\end{array}
$$

A la vista de estas cláusulas resulta obvia la conveniencia y facilidad de algunas optmizaciones que provienen de la posibilidad de evaluar parcialmente algunos de los predicados. Por ejemplo

- *resuelve* puede ser eliminado por completo, desdoblándolo en las llamadas a *iguales* y *distintos* en que consistiría su ejecución. Si esto se hace también con el objetivo inicial, *resuelve* puede ser eliminado por completo.
- La llamada a fnc de la traducción de una $SFL_{\neq}$-regla también puede evaluarse parcialmente, resultando así eliminada de acuerdo a los casos, según la forma del lado derecho r de la regla en cuestión.
  - Si r es de la forma $c(e_1, \ldots, e_n)$, se elimina $fnc(r, H)$ y se reemplaza H por r en la cabeza de la cláusula para $\#f$.
  - Si r es de la forma $g(e_1, \ldots, e_m)$, se reemplaza $fnc(r, H)$ por $\#g(e_1, \ldots, e_n, H)$.

Con estos cambios, las reglas de nuestro ejemplo quedarían

$$
\begin{array}{l}
\#elem(X, L, false) : - \\
\qquad unifica(L, [\,]). \\
\#elem(X, L, true) : - \\
\qquad unifica(L, [Y \mid Ys]), iguales(X, Y). \\
\#elem(X, L, H) : - \\
\qquad unifica(L, [Y \mid Ys]), distintos(X, Y), \#elem(X, Ys, H).
\end{array}
$$

$$
\begin{array}{l}
\#card(L,0):- \\
\qquad unifica(L,[\,]). \\
\#card(L,H):- \\
\qquad unifica(L,[X \mid Xs]), iguales(elem(X,Xs),true), \#card(Xs,H). \\
\#card(L,s(card(Xs))):- \\
\qquad unifica(L,[X \mid Xs]), iguales(elem(X,Xs),false).
\end{array}
$$

$\diamond$

6.2 Representación de expresiones y restricciones

El punto más importante es el de la representación de las variables, que debe ser adecuada para poder implementar los predicados pendientes de especificar de la sección anterior, entre los que destacamos como más importantes

- $sustituye(X,T)$: realiza el vínculo X/T, con efecto global similar al de $X = T$ de Prolog.
- $inserta\_restricciones(T,X)$: añade la restricción $X =/= T$ a X.

Las variables libres de restricciones no ofrecen mayor problema, pues pueden ser representadas por variables Prolog. En cuanto a una variable X con restricción asociada $X =/= t_1, \ldots, X =/= t_n$, es natural utilizar una representación como término Prolog compuesto de la forma $neq(RX,C)$, donde RX es una variable Prolog sobre la que realizaremos posibles futuras instanciaciones de X, y C recolecta las restricciones, p. ej., en forma de lista Prolog $[t_1, \ldots, t_n]$. Veamos en un pequeño ejemplo a qué consecuencias nos lleva esta representación.

Ejemplo 6.2

El término Prolog $neq(RX,[0,s(Y)])$ representaría a una variable X con restricciones asociadas $X =/= 0, X =/= s(Y)$, donde Y sería una variable libre de restricciones. Si más adelante debemos unificar X con $s(0)$, es decir, tenemos una llamada a

$$unifica(neq(RX,[0,s(Y)]),s(0))$$

nos encontramos con (saltando llamadas intermedias)

$$
\begin{array}{ll}
sustituye(neq(RX,[0,s(Y)]),s(0)), & \%X/s(0) \\
resuelve(neq(RX,[0,s(Y)]) =/= 0), & \%s(0) =/= 0 \\
resuelve(neq(RX,[0,s(Y)]) =/= s(Y)) & \%s(0) =/= s(Y)
\end{array}
$$

La ligadura ($sustituye(..)$) la establecemos mediante unificación Prolog $RX = s(0)$, con lo que las desigualdades pendientes por resolver son

$$\begin{array}{ll} resuelve(neq(s(0),[0,s(Y)]) =/= 0), & \%s(0) =/= 0 \\ resuelve(neq(s(0),[0,s(Y)]) =/= s(Y)) & \%s(0) =/= s(Y) \end{array}$$

Nótese que el término Prolog $neq(s(0),[0,s(Y)])$ representa ahora a $s(0)$. Por tanto, en ocasiones va a ser necesario desreferenciar términos Prolog de la forma $neq(\_,\_)$ para llegar al término representado. Nótese también que el efecto de la sustitución ha sido global. De las dos desigualdades pendientes de resolver, la primera debe tener éxito sin ligaduras adicionales, y la segunda debe instanciar Y al término Prolog $neq(RY,[0])$.

Si más adelante nos encontramos con $Y =/= s(0)$, debemos añadir esta restricción a las que ya tiene Y. En términos de nuestra especificación, se llamará a

$$inserta\_restriccion(s(0), neq(RY,[0]))$$

Su efecto debe ser instanciar (la representación de) Y para incorporar la nueva restricción, y el único "hueco" del que disponemos es la variable RY, que podría por ejemplo instanciarse a $neq(RY_1,[s(0),0])$, con lo que Y sería ahora $neq(neq(RY_1,[s(0),0],[0])$, y de nuevo sería necesaria una desreferenciación para llegar a su auténtico valor. Sin profundizar ahora en la discusión, indiquemos que no nos parece ésta la solución más oportuna. En su lugar, para añadir un nuevo término como restricción a $neq(R,C)$ preferimos modificar C, lo cual sugiere utilizar una lista incompleta para C.

En nuestro ejemplo, la X original sería $neq(RX,[0,s(Y)|LX])$, el primer estado para Y sería $neq(RY,[0|LY])$, y el segundo $neq(RY,[0,s(0)|LY_1])$.

$\diamond$

Vamos ya a especificar con precisión los predicados pendientes de definir, conforme a estas ideas acerca de la representación de variables. La especificación se convierte ya en una implementación Prolog.

$es\_var(X) :- var(X), !.$
$es\_var(neq(R,C)) :- es\_var(R).$

$restricciones(X,L) :- var(X), !.$
$restricciones(neq(R,C),C) :- var(R), !.$
$restricciones(neq(R,\_),C) :- restricciones(R,C).$

$es\_vacia(L) :- var(L).$

$sustituye(X,T) :- var(X), !, X = T.$
$sustituye(neq(R,C),T) :- liga(R,T).$

$inserta\_restriccion(T, X) : -var(X), !, X = neq(RX, [T|L]).$
$inserta\_restriccion(T, neq(R, C)) : -var(R), !, inserta(T, C).$
$inserta\_restriccion(T, neq(R, \_)) : -inserta\_restriccion(T, R).$

$inserta(T, L) : -var(L), !, L = [T|L_1].$
$inserta(T, [C|Cs]) : -inserta(T, Cs).$

$inserta(T, C)$ podría programarse para controlar que T no aparezca ya en C, pero no está clara su conveniencia, pues sería una operación costosa.

$misma\_var(X, Y) : -nombre\_var(X, NX), nombre\_var(Y, NY), NX == NY.$

$nombre\_var(X, NX) : -var(X), !, NX = X.$
$nombre\_var(neq(X, C), NX) : -nombre\_var(X, NX).$

$compatibles\_vars(X, Y) : -$
$\quad restricciones(Y, C), not\ \ var\_miembro(X, C).$

$var\_miembro(X, C) : -var(C), !, fail.$ % C es vacía
$var\_miembro(X, [T|Ts]) : -es\_var(T), misma\_var(X, T), !.$
$var\_miembro(X, [\_|Ts]) : -var\_miembro(X, Ts).$

$inserta\_restricciones(C, Y) : -es\_vacia(C), !.$
$inserta\_restricciones(C, Y) : -var(Y), !, Y = neq(Y1, C).$
$inserta\_restricciones(C, Y) : -restricciones(Y, CY), inserta\_l(C, CY).$

$inserta\_l(C, L) : -var(L), !, L = C.$
$inserta\_l(C, [T|Ts]) : -inserta\_l(C, Ts).$

$selecciona(S, L, Rest) : -var(L), !, fail.$
$selecciona(S, [S|Ts], Ts).$

La representación de expresiones no variables la hacemos de forma natural. Queda definida a través de los predicados es_c_ap y es_f_ap.

$es\_c\_ap(E, c(E_1, \ldots, E_n)) : -nonvar(E), !, E = c(E_1, \ldots, E_n).$
$es\_c\_ap(neq(R, C), c(E_1, \ldots, E_n)) : -es\_c\_ap(R, c(E_1, \ldots, E_n)).$

$es\_f\_ap(E, f(E_1, \ldots, E_n)) : -nonvar(E), !, E = f(E_1, \ldots, E_n).$

No es necesaria una segunda cláusula para es_f_ap, pues sólo se realizan ligaduras de variables restringidas a términos. Al disponer ya de definiciones explícitas para es_var, es_c_ap, es_f_ap, pueden eliminarse estos predicados, por evaluación parcial, a lo largo de la especificación. Por poner un ejemplo, las cláusulas para fnc quedarían así:

$fnc(E,H) : -var(E),!,H = E.$
$fnc(neq(R,C),H) : -var(R),!,H = neq(R,C).$
$fnc(neq(R,C),H) : -!, fnc(R,H).$
$fnc(c(E_1,\ldots,E_n),H) : -!, H = c(E_1,\ldots,E_n).$
$fnc(f(E_1,\ldots,E_n),H) : -\#f(E_1,\ldots,E_n,H).$

Con esto queda completada la especificación. Sin embargo el tratamiento de las variables restringidas encierra una trampa en principio muy desagradable, como muestra el siguiente ejemplo.

Ejemplo 6.3

De acuerdo con nuestro código, un objetivo $X =\!\!= Y$ debe resolverse mediante

$es\_var(Y),!,not\ \ misma\_var(X,Y),$
$inserta\_restricciones(Y,X),$
$inserta\_restricciones(X,Y).$

que en definitiva se convertirá en

$X = neq(RX,[Y|LX]),$
$Y = neq(RY,[X|LY])$

que es un problema de unificación que debería fallar por el 'occur-check'. Sin embargo, Prolog lo resuelve creando ligaduras cíclicas.

◇

Nótese que la existencia de esas ligaduras cíclicas no afecta a los aspectos fundamentales de la representación de variables restringidas por términos $neq(R,C)$, que recordemos son:

- R es una variable Prolog susceptible de posterior instanciación
- C es una lista Prolog incompleta cuyos elementos son representaciones de términos.

Nótese también que la especificación presentada hasta ahora no necesita absolutamente ninguna modificación debida a estas ligaduras cíclicas; dicho de otro modo, el código Prolog que constituye la especificación es seguro frente a este problema.

Proponemos pues aprovecharnos de la falta de 'occur-check' de Prolog, y aceptar las ligaduras cíclicas como parte de nuestra representación. De hecho, haciendo abstracción de que estamos utilizando Prolog como lenguaje objeto, esas ligaduras expresan referencias circulares que aparecen de modo natural en nuestra representación de las variables restringidas. En cierto sentido, podríamos decir que estamos programando en un subconjunto de Prolog II soportado por Prolog.

El ejemplo de la desigualdad $X =/= Y$ puede parecer forzado, pues no es estrictamente necesario (aunque nos parece conveniente) anotar esa restricción tanto para X como para Y. Hemos elegido este ejemplo por ser el caso más sencillo, pero situaciones similares que también darían lugar a ligaduras cíclicas pueden ocurrir, como sería en el caso de $X =/= s(Y), Y =/= s(X)$.

A continuación vamos a proponer varios refinamientos de la especificación. Algunos de ellos se refieren a aspectos que hemos dejado pendientes – como es el caso de la 'compartición' frente al 'reemplazamiento' –, mientras que otros son optimizaciones para aumentar la eficiencia de la implementación.

6.3 Refinamientos y optimizaciones de la especificación

6.3.1 Igualdad estricta

La resolución de una ecuación $X == e$, de acuerdo con nuestra especificación, puede hacer mucho trabajo innecesario (por repetido). Consideremos, por ejemplo, la ecuación

$$X == c(c(Y, f(X)), c(a, c(U, V)))$$

donde c, a son constructoras y f es una función no primitiva. Por la tercera cláusula de *ig_var*, hay que recorrer toda la cáscara del lado derecho para verificar que X no está en ella, (la aparición de X en $f(X)$ no forma parte de la cáscara), para posteriormente hacer una imitación que consiste en realizar la sustitución $X/c(U_1, U_2)$ (U_1, U_2 nuevas) y dejar planteadas las nuevas ecuaciones

$$U_1 == c(Y, f(c(U_1, U_2))), U_2 == c(a, c(U, V))$$

Para U_1, U_2 deben hacerse ahora sendos recorridos de las correspondientes cáscaras, claramente innecesarios, pues si X no aparecía en la cáscara original, tampoco pueden hacerlo las variables nuevas U_1, U_2 (nótese sin embargo que aparecen, porque así le ocurría a X, dentro de una subexpresión no primitiva).

Esto puede mejorarse obviamente, aprovechando el recorrido del lado derecho (que debe hacerse para el 'occur check') para construir un *esqueleto* del mismo, que sería como la cáscara, pero poniendo variables nuevas (en lugar de $\perp$) donde encontremos una subexpresión no primitiva. Ese esqueleto serviría para hacer una imitación 'más profunda'. Esta explicación no es más que una nueva justificación de la utilidad de la $\rightsquigarrow^{cs}$- regla $==_4$ (ver sección 5.4). El esqueleto descrito de una expresión e no es otra cosa que lo que allí llamamos $sk(e)$.

En nuestro ejemplo, de la ecuación original pasaríamos, en una sola imitación, a generar la sustitución $X/c(c(Y, U_1), c(a, c(U, V)))$, y obtendríamos la nueva ecuación pendiente de resolver $U_1 == f(c(c(Y, U_1), c(a, c(U, V))))$.

Proponemos pues la siguiente modificación para la cláusula correspondiente (la tercera) de *ig_var*, y la definición completa de *no_aparece*/4.

```
ig_var(X,E) :-                % E es de la forma c(e_1,...,e_n)
  no_aparece(X,E,SkE,Cont),
  pasa_a_resuelta(X,SkE),
  cont_ig_var(Cont).
```

En el tercer argumento de *no_aparece* se va a construir el esqueleto del lado derecho de la ecuación. El último argumento es una 'continuación' en la que se van a recolectar (en forma de diferencia de listas) las nuevas ecuaciones pendientes de resolver. De la resolución de éstas se encarga *cont_ig_var*.

```
no_aparece(X,Y,Y,L/L) :-  es_var(Y),!,not misma_var(X,Y).
no_aparece(X,E,c(Sk_1,...,Sk_n),L_0/L_n) :-
  es_c_ap(E,c(E_1,...,E_n)),!,
  no_aparece(X,E_1,Sk_1,L_0/L_1),
  ...,
  no_aparece(X,E_n,Sk_n,L_{n-1}/L_n).
no_aparece(X,E,U,[U == E|L]/L).     % E debe ser de la forma f(e_1,...,e_n)

cont_ig_var([]/[]) :-!.
cont_ig_var([E_1 == E_2|L_1]/L_2) :-
  iguales(E_1,E_2),
  cont_ig_var(L_1/L_2).
```

Nótese que este mecanismo sirve también si el lado derecho es un término primitivo t. En este caso, supuesto que X no aparezca en t, *no_aparece*$(X,t,Sk,L/L)$ tendrá éxito devolviendo en Sk el propio t y dejando vacía la lista de continuación. Es decir, estamos implementando realmente una mezcla de $==_4$ y $==_3$ (excepto el caso en que el lado derecho sea otra variable, que está tratado aparte).

Digamos, para terminar, que hay otra optimización natural de la igualdad, cuya idea básica es bastante obvia. Si, por ejemplo, debemos resolver la ecuación $s(Y) == f(X)$, todas aquellas reducciones de $f(X)$ que conduzcan a una forma normal de cabeza que no comience por la constructora s van a ser inútiles. La especificación actual permite esa situación, pues la evaluación a forma normal de cabeza de ambos miembros se hace de forma totalmente independiente. Podemos mejorar el rendimiento 'orientando' el cómputo de $f(X)$. Nuestra implementación real tiene en cuenta esta posibilidad. Desde un enfoque distinto, basado en un análisis estático y una transformación de programas, se persigue en [92] un objetivo similar, pero más ambicioso, al intentar capturar el máximo posible de 'orientación' de la evaluación de las funciones.

6.3.2 Desigualdad

Como en el caso de la igualdad, la especificación propuesta para la desigualdad puede suponer mucho trabajo innecesario. Consideremos, por ejemplo, una desigualdad $X =/= c(t, f(Y))$, siendo t un término muy grande. Después de recorrer toda la expresión para descubrir que no es un término, una primera solución para la desigualdad (regla $=/=_5$) propone

$$unifica(X, d(\_, \ldots, \_))$$

que no ofrece problemas. Pero la segunda alternativa realizará

$$unifica(X, c(U_1, U_2))$$

y después

$$distintos(c(U_1, U_2), c(t, f(Y))$$

una de cuyas alternativas será

$$distintos(U_1, t)$$

para la que hay que controlar de nuevo si t es o no un término.

Aparte de esto, ya fue comentado en el apartado 5.6.3 que, al estar interesados sólo en soluciones finitas, se puede resolver directamente con éxito una desigualdad $X =/= e$, en caso de X aparezca en $\mid e \mid$. Podemos efectuar este 'occur check' aprovechando el recorrido que debe hacerse de e para saber si es un término.

Como consecuencia de la discusión anterior, proponemos usar para las desigualdades una variante de $no\_aparece(X, E)$ – digamos $no\_aparece1$ – que, además de comprobar que X no ocurra en la cáscara de E, descubra si E es un término o no, y construya una réplica abreviada de la estructura de E – digamos $sk1(E)$ – definida como sigue:

$sk1(X) =$ var
$sk1(t) =$ term, si t es un término no variable
$sk1(c(e_1, \ldots, e_n)) =$ fnc$(sk1(e_1), \ldots, sk1(e_n))$, si alguna e_i no es un término
$sk1(f(e_1, \ldots, e_n)) =$ redex

Nótese que $var, term, fnc, redex$ son nuevas constructoras.

Por ejemplo,

$$sk1(c(d(Y, a), d(Y, f(X)))) = \text{fnc}(\text{term}, \text{fnc}(\text{var}, \text{redex}))$$

y la llamada

$$no\_aparece_1(X, c(d(Y, a), d(Y, f(X))), Sk, Term)$$

tiene éxito con

$Sk =$ fnc(term, fnc(var, redex))
$Term =$ noterm

Ese esqueleto $sk1(E)$ sirve para continuar rápidamente la resolución de la desigualdad.

La nueva especificación para nig_var y $no\_aparece1$ sería la siguiente:

$nig\_var(X,E) :-$
  $no\_aparece1(X,E,SkE,Term),!,$
  $atajo\_nig\_var(X,E,SkE).$
$nig\_var(X,E).$ % X aparece en la cáscara de E

$no\_aparece1(X,Y,var,Term) :-$
  $es\_var(Y),!,not\ \ misma\_var(X,Y).$
$no\_aparece1(X,E,SkE,Term) :-$
  $es\_c\_ap(E,c(E_1,\ldots,E_n)),!,$
  $no\_aparece1(X,E_1,Sk_1,Term),$
  $\ldots,$
  $no\_aparece1(X,E_n,Sk_n,Term),$
  $(Term == noterm,!,SkE = fnc(Sk_1,\ldots,Sk_n)$
  ;
  $SkE = term).$
$no\_aparece1(X,E,redex,noterm).$ % E debe ser de la forma $f(e_1,\ldots,e_n)$

$atajo\_nig\_var(X,Y,var) :-$ % Y es con seguridad una variable distinta a X
  $inserta\_restricciones(X,Y),$
  $inserta\_restricciones(Y,X).$
$atajo\_nig\_var(X,T,term) :-$
  $inserta\_restricciones(T,X).$
$atajo\_nig\_var(X,c(T_1,\ldots,T_n),fnc(Sk_1,\ldots,Sk_n)) :-$
  $unifica(X,d(U_1,\ldots,Ym)).\quad atajo\_nig\_var(X,c(T_1,\ldots,T_n),fnc(Sk_1,\ldots,Sk_n)) :-$
  $unifica(X,c(U_1,\ldots,U_n),$
  $(atajo\_nig\_var(U_1,T_1,Sk_1)$
  ;
  $\ldots\ldots\ldots\ldots\ldots\ldots$
  ;
  $atajo\_nig\_var(U_n,T_n,Sk_n)).$
$atajo\_nig\_var(X,E,redex) :- distintos(X,E).$

6.3.3 Un control más equitativo para la unificación

Dada una regla $f(t_1,\ldots,t_n) = e \Leftarrow \varphi$, y una llamada $f(e_1,\ldots,e_n)$, contemplamos la unificación de los argumentos e_i con los patrones t_i como un proceso global

$$unifica([e_1,\ldots,e_n],[t_1,\ldots,t_n])$$

en el que primeramente realizamos la parte de unificación que no requiere estrechamiento, y después la parte que requiere estrechamiento. Estamos autorizados a hacerlo así, porque nuestros resultados sobre la semántica combinada $\leadsto^{ncs}$ garantizan completitud para cualquier forma de intercalar los $\leadsto^{cs}$-pasos con los $\leadsto$-pasos. Este control tiene mejores propiedades de terminación, pues permite detectar más situaciones de fallo.

Un ejemplo obvio sería el siguiente: si la función f está definida por las dos reglas $f(0) = s(0)$ y $f(s(X)) = f(X)$, la resolución del problema de unificación

$$f(X) = 0, X = 0, X = s(0)$$

no termina si se da prioridad al la ecuación $f(X) = 0$, pues hay infinitas reducciones alternativas, de $f(X)$ a $s(0)$, que producen infinitos intentos, todos fallidos, de resolver $f(X) = 0$. Sin embargo, si se deja pendiente $f(X) = 0$ y se empieza por lo 'fácil', que es progresar con $\leadsto^{cs}$ mientras se pueda, se falla rápidamente: al resolver $X = 0$, se realiza la sustitución $X/0$, con lo que resulta $0 = s(0)$, y se falla.

Nótese que las propiedades de terminación de este control son estrictamente mejores que las del anterior, es decir: no se producen situaciones de no terminación que no se produjeran con el otro control. Ello es debido a que $\leadsto^{cs}_{=}$ es terminante. Nótese también que se sigue respetando la condición que impusimos para poder aplicar de modo global las sustituciones. Esta condición consistía en dar prioridad al problema de unificación introducido por una regla, frente al resto de las condiciones del lado derecho.

A continuación especificamos este nuevo control para la unificación. Se utiliza un procedimiento, $deref/2$, para efectuar la 'desreferenciación' de cada argumento. Su único objeto consiste en hacer algo más claro el código que resulta.

$deref(X, Y) :- var(X), !, Y = X.$
$deref(neq(R, C), Y) :- var(R), !, Y = neq(R, C).$
$deref(neq(R, C), Y) :- !, deref(R, Y).$
$deref(X, X).$

$unifica([E_1, \ldots, E_n], [T_1, \ldots, T_n]) :-$
  $deref(E_1, DE_1), unifica(DE_1, T_1, Q_1/Q_2),$
  $deref(E_2, DE_2), unifica(DE_2, T2, Q_2/Q_3),$
  .....,
  $deref(E_n, DE_n), unifica(DE_n, T_n, Qn/R),$
  $cont\_unifica(Q_1/R).$

El procedimiento auxiliar *unifica/3* se encarga de realizar aquella parte de la unificación que no requiere estrechamiento, recolectando además el trabajo que se ha dejado pendiente. Las listas (en forma de diferencia de listas) Qi/Qj tienen como elementos parejas (e, t), donde e es de la forma $g(e_1, \ldots, e_m)$ y t es un término lineal no variable. Estas parejas indican los problemas de unificación que se han dejado pendientes, y que se resuelven al final estrechando las expresiones e (de esto último se encarga el procedimiento *cont_unifica/1*).

$unifica(E, T, Q/Q) :- (var(T); var(E)), !, T = E.$
$unifica(neq(R, C), T, Q/Q) :- !, R = T, propaga(T, C).$
$unifica(c(E_1, \ldots, E_n), T, Q_0/Qn) :-$
  $!, T = c(T_1, \ldots, T_n),$
  $deref(E_1, DE_1), unifica(DE_1, T_1, Q_0/Q_1),$
  .....,
  $deref(E_n, DE_n), unifica(DE_n, T_n, Q_{n-1}/Qn).$

$unifica(E, T, [(E,T)|Q]/Q).$

$cont\_unifica([]/[]).$
$cont\_unifica([(E, c(T_1, \ldots, T_n)|Q]/R) :-$
  $fnc(E, c(U_1, \ldots, U_n)),$
  $unifica([U_1, \ldots, U_n], [T_1, , , .T_n]),$
  $cont\_unifica(Q/R).$

$cont\_unifica(QDif)$ resuelve por completo cada problema (e, t) que esté en $QDif$ antes de pasar al siguiente. Puede hacerse más equitativo, pasando los problemas pendientes generados por la unificación de e y t a la cola $QDif$, como refleja la siguiente definición (alternativa a la anterior) de *cont_unifica/1*.

$cont\_unifica([]/[]).$
$cont\_unifica([(E, c(T_1, \ldots, T_n)|Q]/R) :-$
  $fnc(E, c(U_1, \ldots, U_n)),$
  $unifica(c(U_1, \ldots, U_n), c(T_1, \ldots, T_n), R/S),$
  $cont\_unifica(Q/S).$

Ideas similares pueden usarse para la resolución de una secuencia de igualdades y desigualdades, contemplada como un proceso global.

Estas reglas de unificación sirven para problemas de unificación entre expresiones cualesquiera y términos, siempre que no sea preciso el 'occur-check'. En particular, sirven para el caso en que los términos sean lineales, que es nuestro caso, pero no es la única situación.

6.3.4 Compartición

Por simplicidad, hemos considerado hasta ahora en nuestra especificación que el paso de parámetros al efectuar estrechamiento mediante una regla del programa se ha efectuado con *copia* en lugar de *compartición*. Esta cuestión fue discutida en el apartado 5.6.1. Tanto por motivos prácticos (evitar la evaluación, por separado, de cada una de las copias de una expresión no primitiva) como teóricos (hemos probado la completitud de $\leadsto^{ncs}$ para el sistema $\leadsto^{cs}$ que realiza compartición), debemos reemplazar la 'copia' por la 'compartición'. Adoptaremos para ello una técnica descrita en [28], y que ha sido utilizada también en [86, 103].

La idea principal es cambiar la representación de las expresiones de la forma $f(e_1, \ldots, e_n)$, considerando en su lugar

$$f(e_1, \ldots, e_n, R, S)$$

donde R y S son variables Prolog, inicialmente sin ligaduras, que pretenden representar un *resultado* y un *estado* de evaluación. S quedará sin ligar mientras la expresión $f(e_1, \ldots, e_n)$ no sea evaluada a forma normal de cabeza, pasando a tomar el valor *on* cuando lo sea, momento en que R quedará instanciada con la forma normal de cabeza calculada [28]. Esta

[28] No podemos eliminar S y utilizar el estado (variable/no variable) de R para saber si $f(e_1, \ldots, e_n)$ ya fue evaluada, porque una variable es también una posible forma normal de cabeza.

representación será útil si todas las copias de $f(e_1,\ldots,e_n)$ que se hayan creado tienen esa misma representación, con las *mismas* variables R,S, pues entonces la evaluación de una cualquiera de las copias se transmite, a través de R,S, al resto.

Al término Prolog $f(e_1,\ldots,e_n,R,S)$ le denominamos *suspensión* o *forma suspendida* de $f(e_1,\ldots,e_n)$. Más en general, toda $SFL_{\neq}$-expresión debe ser representada mediante su forma suspendida, que es el resultado de reemplazar cada subexpresión $f(e_1,\ldots,e_n)$ por su suspensión. Con más precisión, definimos inductivamente la *forma suspendida de una* $SFL_{\neq}$*-expresión* como

- $fs(X) := X$, si $X \in Var$
- $fs(c(e_1,\ldots,e_m)) := c(fs(e_1),\ldots,fs(e_m))$, si $c \in CS^m$
- $fs(f(e_1,\ldots,e_n)) := f(fs(e_1),\ldots,fs(e_n),R,S)$, donde R,S son nuevas variables Prolog, si $f \in \Delta^n$

La *forma suspendida de una condición* es

$$fs(l_1 == r_1,\ldots,l'_1 =\!\!= r'_1,\ldots) := fs(l_1) == fs(r_1),\ldots,fs(l'_1) =\!\!= fs(r'_1),\ldots$$

La introducción de las formas suspendidas requiere algunos cambios en la especificación. El más relevante (el único, de hecho) afecta a las cláusulas para $fnc(E,H)$, que es el procedimiento encargado de obtener para una expresión E una forma normal de cabeza H. Hay que tener ahora la precaución de comprobar, en caso de que E sea una suspensión $f(e_1,\ldots,e_n,R,S)$, si ya ha sido evaluada o no, comprobando para ello el estado de S. Las cláusulas para fnc quedan así:

$fnc(E,H) :- var(E),!,H = E.$
$fnc(neq(R,C),H) :- var(R),!,H = neq(R,C).$
$fnc(neq(R,C),H) :- !,fnc(R,H).$
$fnc(c(E_1,\ldots,E_n),H) :- !,H = c(E_1,\ldots,E_n).$
$fnc(f(E_1,\ldots,E_n,R,S),H) :- S == on,!,fnc(R,H).$
$fnc(f(E_1,\ldots,E_n,R,S),H) :- \#f(E_1,\ldots,E_n,H),R = H,S = on.$

Para el resto de la especificación simplemente hay que tener en cuenta, de forma bastante obvia, esta nueva representación.

Debe tenerse en cuenta que la nueva representación afecta también a las reglas para los predicados $\#f$, así como a los objetivos. Veamos un ejemplo.

Ejemplo 6.4

Si consideramos el $SFL_{\neq}$-programa del ejemplo 6.1, las cláusulas para los predicados $\#elem$ y $\#card$ quedarían así:

$$
\begin{array}{l}
\#elem(X, L, false) : - \\
\qquad unifica(L, [\,]). \\
\#elem(X, L, true) : - \\
\qquad unifica(L, [Y \mid Ys]), iguales(X, Y). \\
\#elem(X, L, H) : - \\
\qquad unifica(L, [Y \mid Ys]), distintos(X, Y), \#elem(X, Ys, H).
\end{array}
$$

$$
\begin{array}{l}
\#card(L, 0) : - \\
\qquad unifica(L, [\,]). \\
\#card(L, H) : - \\
\qquad unifica(L, [X \mid Xs]), iguales(elem(X, Xs, R, S), true), \#card(Xs, H). \\
\#card(L, s(card(Xs, R, S))) : - \\
\qquad unifica(L, [X \mid Xs]), iguales(elem(X, Xs, R', S'), false).
\end{array}
$$

$\diamond$

6.4 Otras cuestiones de implementación

Una implementación de $SFL_{\neq}$, siguiendo las ideas expuestas hasta ahora (ver también [8]), ha sido realizada por miembros del equipo de trabajo sobre Programación Declarativa que existe en el Departamento de Informática y Automática de la Universidad Complutense de Madrid . Se ha integrado la resolución de desigualdades en un entorno previamente existente, *BabLog* [7, 6], que ya soportaba programación funcional y lógica de orden superior. El nuevo entorno resultante se ha denominado $FLPD$ [29]. $FLPD$ está realizado en su totalidad en BIMprolog (v 3.0), y puede ejecutarse en estaciones SUNsparc bajo SUNOS 4.0 o superior.

$FLPD$ ha heredado de sus antepasados muchas características añadidas a lo que tiene de específico – la combinación de desigualdades con el estrechamiento perezoso – que hacen de él un entorno potente y versátil para realizar programación declarativa. Describimos a continuación algunos de sus aspectos más notables.

- El lenguaje (del que $SFL_{\neq}$ es un subconjunto) es fuertemente tipado. El sistema de tipos es el habitual [113] en lenguajes funcionales como Miranda. Se encuentra descrito en [7, 6].
- La compilación de un programa consiste en su traducción a un programa Prolog 'equivalente', siguiendo las líneas expuestas en esta sección (en lo que se refiere a primer orden).
- El lenguaje incluye funciones de orden superior, incluso variables lógicas de orden superior. En la traducción de un programa P se genera un programa P' de primer orden, de tal forma que el orden superior se simula a través de una función de aplicación @.

[29] *F*unctional *L*ogic *P*rogramming with *D*isequalities.

Las reglas para @ son una adaptación de las presentadas en [60, 61] (véase también lo expuesto en el apartado 2.3.4 de esta tesis). Los cómputos utilizan el programa traducido de primer orden P', pero las respuestas se traducen en sentido inverso, para ser mostradas con sintaxis de orden superior. El sistema soporta variables lógicas de orden superior, pero las desigualdades deben afectar sólo a expresiones de primer orden.

- A diferencia de la que hemos utilizado en este trabajo, y debido a las capacidades de orden superior del sistema, la sintaxis del lenguaje es 'de orden superior', con funciones currificadas. Por este motivo, no se ha podido usar el lector de términos propio de Prolog para la lectura de los programas. Se dispone de un compilador para el lenguaje, que se encarga por completo de todas las fases de la lectura y análisis de los programas, realizando en particular detección y notificación detallada de los errores detectados. Una descripción de la sintaxis puede encontrarse en [57] (donde se describe un entorno previo para una versión no perezosa de Babel de orden superior [91]). Una presentación del diseño del compilador puede encontrarse en [104, 106].

- La traducción a Prolog de un programa puede hacerse de acuerdo a varios modos de compilación, según el régimen de control que se desee obtener para el estrechamiento perezoso. Las traducciones obtenidas son optimizaciones de las expuestas en [103, 8]. Los tres modos posibles corresponden a

  - Régimen *guiado por la demanda*, basado en unos *árboles definitorios* que se construyen de acuerdo a un análisis global de todas las reglas de un programa que corresponden a una misma función (en particular, un análisis de los patrones de las cabezas de las reglas).
  - Régimen *ingenuo*, en el que la traducción de cada regla es independiente de la de las demás. Es el que hemos presentado en este trabajo.
  - Régimen *mixto*, en que el usuario puede decidir, para cada función, cuál de los dos anteriores quiere usar.

- El entorno $FLPD$ proporciona algunas utilidades de carácter general (listados, ayudas, estadísticas, ...) controladas por un intérprete de comandos [7]. En la actualidad se está desarrollando un sistema de depuración [6], basado en el modelo de 'cajas' [24] habitual en los sistemas de traza de Prolog.

7 Conclusiones y trabajo futuro

Hemos desarrollado el esquema $CFLP(X)$ que sirve de marco para la integración de tres paradigmas importantes de programación declarativa: programación funcional, programación lógica y programación con restricciones. El sentido genérico que hemos dado, a la hora de concebir el esquema, al uso de restricciones en programación ha sido el de 'computación sobre un estructura dada'. La clase de estructuras a considerar ha estado determinada desde un principio por la naturaleza perezosa que se ha pretendido dotar a la componente funcional de la amalgama. La noción de 'estructura continua' que hemos adoptado, con dominios de Scott como soporte y operaciones primitivas continuas, parece adecuada para modelizar lenguajes con la capacidad de realizar cómputos que involucren objetos infinitos definidos como límites de aproximaciones finitas. Tal elección se ve avalada por la generalidad de los resultados obtenidos y la relativa simplicidad con que han sido probados. También parece natural y suficientemente general la idea de programa como conjunto de reglas condicionales para la definición de nuevas funciones y predicados. La generalidad del esquema así definido resulta ser grande. En particular, dentro de él pueden ser expresados los paradigmas que pretende extender. Es más: proporciona, sin esfuerzo adicional, una extensión funcional a todo lenguaje caracterizado como instancia del esquema, aun cuando inicialmente sólo tuviera carácter lógico; así sucede, por ejemplo, en el caso de Prolog (puro) o de las instancias de $CLP(X)$.

Se ha desarrollado una semántica declarativa que parte de la lectura de las reglas de un programa como 'inecuaciones condicionales'. Se ha dotado a los programas de una semántica de modelo mínimo, cuya existencia se ha probado, así como su caracterización como mínimo punto fijo de un operador 'de consecuencias inmediatas' asociado. Estos resultados se han probado bajo hipótesis lo menos restrictivas posibles (consistencia de un programa), generalizando y mejorando así resultados previos para lenguajes con una semántica declarativa similar.

Se ha propuesto un mecanismo de cómputo – estrechamiento por restricciones – que define una semántica operacional probada como correcta y completa (para programas no ambiguos en un sentido muy liberal) con relación a la semántica declarativa. Para probar la completitud, se ha dado una caracterización semántica de lo que debe entenderse por 'cómputos perezosos'. Posteriormente, en el ánimo de obtener una semántica operacional más práctica, se han proporcionado condiciones muy generales que debe cumplir un sistema de resolución de restricciones para poder ser combinado con el estrechamiento por restricciones, preservando corrección y completitud para la semántica operacional combinada.

Contemplando, ahora desde otro punto de vista, los resultados obtenidos, pensamos que se ha cumplido nuestro objetivo inicial de definir el esquema $CFLP(X)$ como una extensión conceptual 'suave' de la programación lógico funcional perezosa. El abordar los problemas desde un punto de vista más abstracto – en particular el estar liberados de los tecnicismos que apareja la unificación – nos ha permitido en ocasiones distinguir lo importante de lo accesorio, y nos ha conducido a resultados que, a pesar de ser más generales, han sido probados de forma más sencilla.

En una segunda parte del trabajo, hemos propuesto un lenguaje – $SFL_{\neq}$ –, que aumenta la expresividad de un lenguaje lógico funcional perezoso mediante el uso de desigualdades,

tanto en programas como en respuestas. El lenguaje ha sido caracterizado como una instancia del esquema $CFLP(X)$, heredando así todas sus propiedades. En el transcurso de esta caracterización ha sido necesario clarificar el significado de la unificación como operación continua, o, equivalentemente, el uso continuo que se puede hacer de la igualdad (no continua) $=$. Hemos propuesto un sistema de resolución de restricciones $SFL_{\neq}$ que, en combinación con el estrechamiento por restricciones, constituye un mecanismo de cómputo correcto y completo. A los resultados heredados del esquema, hemos podido añadir un resultado más potente de completitud. La implementación del lenguaje se ha planteado como un proceso de compilación de $SFL_{\neq}$-programas a Prolog. El proceso de traducción se ha descrito a partir de una especificación (parcialmente) independiente de la representación de las expresiones y restricciones. Una cuidadosa elección de ésta convierte a la especificación en un programa Prolog ejecutable. La claridad y simplicidad de la implementación obtenida, la posibilidad de aprovechar la gran eficiencia de los sistemas Prolog existentes, y la mayor facilidad para incorporar nuevas mejoras, justifican en nuestra opinión la validez de tal enfoque.

El trabajo desarrollado deja aún mucho espacio para futuras investigaciones, aparte de las cuestiones técnicas – más o menos difíciles, pero muy localizadas – que hemos ido señalando durante la exposición. Seguidamente indicamos las posibles líneas de continuación del trabajo que nos parecen más interesantes. De algunas de ellas éramos conscientes al iniciar el trabajo, mientras otras han surgido durante el desarrollo del mismo.

- Una cuestión de apariencia muy técnica, pero en nuestra opinión de gran interés, es la determinación de propiedades generales de los sistemas de resolución de restricciones, para garantizar resultados de completitud más fuertes que los obtenidos por nosotros en la sección 4.4. Un análisis crítico de lo que ocurre en el caso particular de $SFL_{\neq}$, para el que hemos obtenido tal tipo de resultados, puede contribuir a aclarar la cuestión.

- En relación con lo anterior, la falta de resultados de completitud para soluciones no básicas parece achacable a la existencia de una estructura *fija* a la que se refieren todas la cuestiones semánticas. Tiene sentido intentar paliar el problema asumiendo la existencia de una teoría existencialmente completa (*satisfaction complete*) asociada a la estructura de las restricciones. Esa visión ha sido fructífera [78, 79, 109] en el caso de la programación lógica con restricciones, y probablemente lo sería también en el nuestro. De modo alternativo – o mejor, complementario – podría reemplazarse la consideración de una estructura prefijada, por la de una clase (categoría) de estructuras. Las dos líneas apuntadas aquí contribuirían no sólo a obtener mejores resultados de completitud, sino a dotar al esquema de semánticas lógica y algebraica, a añadir a la de modelo mínimo y mínimo punto fijo ya desarrollada.

- Introducción de orden superior en el esquema. Podría intentarse una 'tenaza' similar a la de otros trabajos [64, 61] ya realizados para el caso de la programación lógico funcional de orden superior, y que consistiría en:

  - Dotar a un $CFLP$-lenguaje de orden superior de semánticas declarativa y operacional de orden superior.

  - Considerar una 'traducción' a primer orden (nuestro $CFLP(X)$), de la que se heredarían las semánticas declarativa y operacional.
  - Establecer la relación (deseablemente, equivalencia) entre ambas.

- Aumentar la expresividad de $SFL_{\neq}$ mediante la posibilidad de algún tipo (posiblemente restringido) de cuantificación universal. El uso simultáneo de cuantificación universal y desigualdad nos introduciría en una problemática similar a la de la negación constructiva [27, 145], que podría incluso ser abordada desde el marco más general del esquema $CFLP(X)$.

- Estudiar otras instancias interesantes del esquema. En particular, los lenguajes que manejan restricciones conjuntistas [49, 22] despiertan gran interés en la actualidad. La posibilidad de considerar incluso conjuntos infinitos, dada la naturaleza perezosa de nuestro esquema, resulta ciertamente sugerente. Para las implementaciones, podríamos seguir el enfoque de traducción a Prolog seguido en [103] y en esta misma memoria, o emprender el diseño de máquinas abstractas al estilo de [93, 94].

8 Trabajos publicados

Los números que se indican se refieren a la Bibliografía general.

[8] P. Arenas Sánchez, A. Gil Luezas, F.J. López Fraguas: *Combining Lazy Narrowing with Disequality Constraints*, Procs. Int. Symp. on Programming Language Implementation and Logic Programming (PLILP'94), Springer LNCS 844, 1994, 385-399. Extended version as Tech. Report DIA 94/2, 1994.

[16] E. Börger, F.J. López Fraguas, M. Rodríguez Artalejo: *Towards a Mathematical Specification of Narrowing Machines*, Tech. Rep. DIA 94/5, 1994.

[17] E. Börger, F.J. López Fraguas, M. Rodríguez Artalejo: *A model for mathematical analysis of functional logic languages and their implementations*, Procs. IFIP'94, Volume I, IFIP Transactions A-51,North-Holland, 1994, 410-415.

[57] A. Gil Luezas, T. Hortalá González, F.J. López Fraguas: *Babel, Manual de Utilización*, Tech. Report DIA 92/15, 1992.

[92] H. Kuchen, F.J. López Fraguas: *Result Directed Computing in a Functional Logic Language*, Procs. Second Int. Workshop on Functional/Logic Programming (A. Mück ed.), Munchen, 1993, 14-34. Also as Tech. Report 92-21, RWTH Aachen, 1992.

[93] H. Kuchen, F.J. López Fraguas, J.J. Moreno Navarro, M. Rodríguez Artalejo: *Implementing a Lazy Functional Logic Language with Disequality Constraints*, Procs. Joint Int. Conf and Symp. on Logic Programming, MIT Press, 1992, 207-221.

[94] H. Kuchen, F.J. López Fraguas, J.J. Moreno Navarro, M. Rodríguez Artalejo: *Implementing Disequality in a Lazy Functional Logic Language* , Tech. Report 92-20, RWTH Aachen, 1992.

[103] R. Loogen, F. J. López Fraguas, M. Rodríguez Artalejo: *A Demand Driven Computation Strategy for Lazy Narrowing*, Int. Symp. on Programming Language Implementation and Logic Programming (PLILP'93), Springer LNCS, 1993, 184-200.

[104] F.J. López Fraguas: *Diseño de un Compilador de Babel en Prolog*, Trabajo de Investigación de Tercer ciclo, DIA-UCM, 1990.

[105] F.J. López Fraguas: *A General Scheme for Constraint Functional Logic Programming*, Procs. Int. Conf. on Algebraic and Logic Programming (ALP'92), Springer LNCS 632, 1992, 213-217.

[106] F.J. López Fraguas, R.M. Pinero Fernández: *Building a Babel Compiler with Logic Programming Techniques*, Workshop on the Integration of Functional and Logic Programming, Granada, Spain, 1990.

[107] F.J. López Fraguas, M. Rodríguez Artalejo: *An Approach to Constraint Functional Logic Programming*, Tech. Rep. DIA 91/4, 1991.

9 Bibliografía

Referencias

[1] H. Aït-Kaci: *Warren's Abstract Machine: a rationale reconstruction*, MIT Press, 1991.

[2] H. Aït-Kaci, R. Nasr: *Integrating Logic and Functional Programming*, Lisp and Symbolic Computation, 2,1989,51-89.

[3] M. Alpuente Fresnedo: *El lenguaje $CLP(\mathcal{H}/\mathcal{E})$: Una Aproximación Basada en Restricciones a la Integración de la Programación Lógica y Ecuacional*, Tesis doctoral, UPV, Valencia, 1991.

[4] H. Andreka, I. Nemeti: *The generalized completeness of Horn predicate logic as a programming language*, Acta Cybernetica 4, 1978, 3-10.

[5] K. R. Apt: *Logic Programming*, in van Leeuwen (ed.) Handbook of Theoretical Computer Science, Vol. B, Elsevier Science Pub. 1990, 495-574.

[6] P. Arenas Sánchez: *Un modelo de depuración para el estrechamiento perezoso*, Trabajo de Investigación de Tercer Ciclo, DIA UCM, 1994.

[7] P. Arenas Sánchez, A. Gil Luezas: *BabLog, Manual de Utilización*, Tech. Report (in preparation).

[8] P. Arenas Sánchez, A. Gil Luezas, F.J. López Fraguas: *Combining Lazy Narrowing with Disequality Constraints*, Procs. Int. Symp. on Programming Language Implementation and Logic Programming (PLILP'94), Springer LNCS 844, 1994, 385-399. Extended version as Tech. Report DIA 94/2, 1994.

[9] Barendregt: *λ-calculus and Functional Programming*, in van Leeuwen (ed.) *Handbook of Theoretical Computer Science*, Vol. B, Elsevier Science Pub, 1990, 321-363.

[10] M. Bellia, G. Levi: *The Relation between Logic and Functional Languages*, Journal of Logic Programming, Vol.3, 1986, 217-236.

[11] M. Bellia, P.G. Bosco, G. Levi, C. Moiso, C. Palamidessi: *A two-level approach to logic and functional programming*, Procs. PARLE'87, Springer LNCS, 1987, 374-393.

[12] F. Benhamou, A.Colmerauer (Eds): *Constraint Logic Programming: Selected Research*, MIT Press 1993.

[13] J.A. Bergstra, J.W. Klop: *Conditional Rewrite Rules: Confluence and Termination*, Journal of Computer and System Science, 32, 1986, 323-362.

[14] D. Bert, R. Echahed: *Design and implementation of a generic, logic and functional programming language*, Procs. First European Symp. on Programming ESOP'86, Springer LNCS 213, 1986, 119-132.

[15] D. Bert, R. Echahed: *Integrating Disequations in the Algebraic and Logic Programming Language LPG*, ICLP'94 Workshop on Integration of Declarative Paradigms, 1994, 76-93.

[16] E. Börger, F.J. López Fraguas, M. Rodríguez Artalejo: *Towards a Mathematical Specification of Narrowing Machines*, Tech. Rep. DIA 94/5, 1994.

[17] E. Börger, F.J. López Fraguas, M. Rodríguez Artalejo: *A model for mathematical analysis of functional logic languages and their implementations*, Procs. IFIP'94, Volume I, IFIP Transactions A-51,North-Holland, 1994, 410-415.

[18] A. Borning: *The Programming Languages Aspects of ThingLab, a Constraint-Oriented Simulation Laboratory*, ACM Transactions on Programming Languages and Systems, 3(4), 1981, 252-387.

[19] P.G. Bosco, C. Cecchi, C. Moiso: *An extension of WAM for K-LEAF*, 6th Int. Conference on Logic Programming, Lisboa, MIT Press, 1989, 318-333.

[20] P.G. Bosco, E. Giovannetti: *IDEAL: An Ideal Deductive Applicativa Language*, Proc. IEEE Inter. Symposium on Logic Programming, Salt Lake City, 1986,89-94.

[21] A. Boudet: *Combining Unification Algorithms*, J. of Symbolic Computation, 16, 1993, 597-626.

[22] P. Bruscoli, A. Dovier,E. Pontelli, G. Rossi: *Compiling Intensional Sets in CLP* , Procs. 11th. Int. conf. on Logic Programming ICLP'94, MIT Press 1994, 647-661.

[23] H-J. Bürckert: *A Resolution Principle for Clauses with Constraints*, Proc. 10th Conference on Automated Deduction CADE'90, Springer LNCS 449, 178-192, 1990.

[24] L. Byrd: *Understanding the control flow of Prolog programs*, Procs. Workshop on Logic Programming, Debrecen, 1990.

[25] M. M. T. Chakravarty, H. C. R. Lock: *The Implementation of Lazy Narrowing*, Symposium on Programming Language Implementation and Logic Programming 1991, Springer LNCS 528, 1991, 123–134.

[26] R. Caferra, N. Zabel: *A method for simoultaneous search for refutations and models by equational constraint solving*,J. of Symbolic Computation, 13(6), 1992,613-642.

[27] D. Chan: *Constructive Negation based on Completed Database*, Proc. 5th International Conference on Logic Programming, MIT Press, 1988, 111-125.

[28] P.H. Cheong: *Compiling lazy narrowing into Prolog*, Technical Report 25, LIENS, 1990, to appear in: Journal of New Generation Computing.

[29] P.H. Cheong and L. Fribourg: *A survey of the implementations of narrowing*, in: J. Darlington and R. Dietrich (eds.) Declarative Programming. Workshops in Computing, Springer Verlag & BCS, 1992, 177-187.

[30] J. Cohen *Constraint Logic Programming Languages*, Communications of the ACM, 33, 52-68, July 1990.

[31] A. Colmerauer: *Prolog and infinite trees*, in: K.L. Clark, S.A. Tarnlund (eds.) *Logic Programming*, Academic Press, 1982, 231-251.

[32] A. Colmerauer: *Equations and inequations on finite and infinite trees*, Procs. FGCS'84, 1984, 85-99.

[33] A. Colmerauer: *Thoretical Model of Prolog II*, in: M. van Canegnem, D.H.D. Warren (eds.) *Logic Programming and its Applications*, Ablex Pub. Co., 1986,3-31.

[34] A. Colmerauer: *Opening the Prolog III Universe*, BYTE Magazine, August 1987.

[35] H. Comon: *Disunification: A Survey*, in : J.L.Lassez,G. Plotkin (eds.) *Computational Logic, Essays in Honor of Alan Robinson*, The MIT Press, 1991,322-359.

[36] H. Comon, P. Lescanne: *Equational problems and disunification*, J. of Symbolic Computation, 7, 1989, 371-425.

[37] J.N. Crossley, L. Mandel and M. Wirsing: *Untyped Constrained Lambda Calculus*, Technical Report 9318, Institut fur Informatic, Ludwig-Maximilians-Univ., Munchen, September 1993.

[38] J. Darlington, Y. K. Guo: *Narrowing and Unification in Functional Programming - An Evaluation Mechanism for Absolute Set Abstraction*, Proc. of the Conference on Rewriting Techniques and Applications, Springer LNCS 355, 1989, 92-108.

[39] J. Darlington, Y. K. Guo: *Constraint Functional Programming*, Technical Report, Imperial College, November 1989.

[40] J. Darlington, Y. K. Guo: *Constraint Equational Deduction*, Technical Report, Imperial College, March 1990.

[41] J. Darlington, Y. K. Guo, H. Pull: *A New Perspective on the Integration of Functional and Logic Languages*, Technical Report, Imperial College, Sep. 1991.

[42] J. Darlington, Y. K. Guo: *A New Perspective on Integrating Functions and Logic Languages*, Proceeding of the 3rd Conference on Fifth Generation Computer Systems, Tokyo, 682-693, 1992.

[43] J. Darlington, H. Pull: *The Unification of Functional and Logic Languages*, in D. DeGroot, G. Lindstrom *Logic Programming, Functions, Relations and Equations*, Prentice Hall,1986, 37-70.

[44] N. Dershowitz: *Orderings for term-rewriting systems*, Theoretical Computer Science, 17(3),1982, 279-301.

[45] N. Dershowitz, M. Okada: *A rationale for conditional equational rewriting*, Theoret. Comput. Sci. 75(1/2), 1990, 11–137.

[46] N. Dershowitz, J.P. Jouannaud: *Rewrite Systems*, in J. van Leeuwen (ed.) Handbook of Theoretical Computer Science, Vol. B, Elsevier Science Pub. 1990, 234-320.

[47] M. Dincbas, P. Van Hentenryck, H. Simonis, A. Aggoun: *The Constraint Logic Programming Languages CHIP*, Proccedings of the 2nd. International Conference on Fifth Generation Computer Systems, 1988,249-264.

[48] M. Dincbas, H. Simonis, P. Van Hentenryck: *Solving Large Combinatorial Problems in Logic Programming*, Journal of Logic Programming 8 (1/2), 75-93, 1990

[49] A. Dovier, G.F. Rossi: *Embedding extensional finite sets in CLP*, Procs. Int. Logic Progr. Symp., The MIT Press, 1993.

[50] M. H. van Emden, R. A. Kowalski: *The Semantics of Predicate Logic as a Programming Language*, Journal ACM, Vol 23 (4), 1976, 733-742.

[51] F. Fages: *Associative-Commutative Unification*, Procs. 7th International conference on Automated deduction (CADE'84), Springer LNCS 170, 1984, 194-208.

[52] M. Fernández: *Narrowing Based Procedures for Equational Disunification*, Applicable Algebra in Engineering, Communication and Computing 3,1992,1-26.

[53] D. de Frutos Escrig, M.I. Fernández Camacho: *On Narrowing Strategies for Partial Non-strict Functions*, Procs. TAPSOFT'91, Springer LNCS 494, 416-437.

[54] T. Früwirth. A. Herold, V. Kchenhoff, P.Lim, M. Wallace: *Constraint Logic Programming - An Informal Introduction*, in: Logic Programming in Action, Springer LNCS 636, 1992, 3-35.

[55] M. Gabrielli, G. Levi: *Modeling Answer Constraints in Constraint Logic Programs*, Proc. 8th International Conference on Logic Programming, 1991, 238-252.

[56] J. Gallier, W. Snyder: *Complete Sets of Transformations for General E-unification*, Theoretical Computer Science 67, 1989, 203-260.

[57] A. Gil Luezas, T. Hortalá González, F.J. López Fraguas: *Babel, Manual de Utilización*, Tech. Report DIA 92/15, 1992.

[58] E. Giovannetti, G. Levi, C. Moiso, C. Palamidessi: *Kernel LEAF: A Logic plus Functional Language*, Journal of Computer and System Sciences, Vol. 42, No. 2, Academic Press 1991, 139–185.

[59] E. Giovannetti, C. Moiso: *A Completeness result for E-unification Algorithms Based on Conditional Narrowing*, Proc. Workshop on Foundations of Logic and Functional Programming, Springer LNCS 306, 1986, 157-167.

[60] J.C. González-Moreno: *A correctness proof for Warren's HO into FO translation*, Procs. 8th Italian Conf. on Logic Programming GULP'93, 1993, 569-583.

[61] J.C. González-Moreno: *Programación Lógico Funcional de Orden Superior con Combinadores*, Tesis doctoral, DIA-UCM, Madrid 1994.

[62] J.C. González-Moreno, M.T. Hortalá-González, M. Rodríguez-Artalejo: *A Functional Logic Languaje with Higer Order Logic Variables*, Technical Report DIA 90/6, October 1990.

[63] J.C. González-Moreno, M.T. Hortalá-González, M. Rodríguez-Artalejo: *Denotational versus Declarative Semantics for Functional Programming*, Procs. Computer Science Logic CSL'91, Springer LNCS 626, 1992, 134–148.

[64] J.C. González-Moreno, M.T. Hortalá-González, M. Rodríguez-Artalejo: *On the Completeness of Narrowing as the Operational Semantics of Functional Logic Programming*, Procs. Computer Science Logic CSL'92, Springer LNCS 702, 1993, 216-230.

[65] D. de Groot , G. Lindstrom (eds): *Logic Programming: Functions, Relations and Ecuations*, Prentice Hall, 1986.

[66] Y.K. Guo, H. Lock: *A Classification Scheme for Declarative Programming Languages*, GMD-Studien 182, 1990.

[67] Y. Gurevich: *Evolving Algebras: A Tutorial Introduction*, EATCS Bulletin 43, 1991, 1-57.

[68] M. Hanus: *Compiling Logic Programs with Equality*, Workshop on Progr. Language Implementation and Logic Progr. (PLILP), Springer LNCS 456, 1990, 387–401.

[69] M. Hanus: *The Integration of Functions into Logic: From Theory to Practice*, to appear in J. of Logic Progr. Also available as Tech. Rep. MPI-I-94,201, January, 1994.

[70] N.C. Heintze, J. Jaffar, C.S. Lim, S. Michaylov, P.J. Stickey, R. Yap, C.N. Yee *The CLP Programmer's Manual*, Tech. Rep., Dep. Comp. Science, Monash Univ., 1986.

[71] P. van Hentenryck: *Constraint Satisfaction in Logic Programming*, MIT Press, 1989.

[72] P. van Hentenryck (Ed): Special Issue on *Constraint Logic Programming*, Journal of Logic Programming, 16, 3&4, 1993.

[73] M. Höhfeld, G. Smolka: *Definite Relations over Constraint Languages*, LILOG Report 53, IBM Germany, 1988 (to appear in J. of Logic Progr.).

[74] S. Hölldobler: *Foundations of Equational Logic Programming*, Springer LNCS 353, 1989.

[75] P. Hudak: *Conception, Evolution and Application of Functional Programming Languages*, ACM Computing Surveys, 21 (3), 1989, 359-411.

[76] G. Huet, J. J. Levy: *Call by Need Computations in Non-Ambiguous Linear Term Rewriting Systems*, Report 359, INRIA, 1979. Also appeared as *Computations in Ortogonal Rewriting Systems*, in: J.-L. Lassez, G. Plotkin (eds.) Computational Logic: Essays in Honour of Alan Robinson, MIT Press, 1991.

[77] J.-M. Hullot: *Canonical Forms and Unification*, 5th Conference on Automated Deduction, Springer LNCS 87, 1980, 318-334.

[78] J. Jaffar, J.L. Lassez: *Constraint Logic Programming*, Tecnical Report 86/73, Department of Computer Science, Monash University, June, 1986.

[79] J. Jaffar, J.L. Lassez: *Constraint Logic Programming*, Proc. 14th ACM Symposium on Principles of Progamming Languages, Munich 1987, 114-119.

[80] J. Jaffar, J.L. Lassez, M.J. Maher: *A Logic Programming Language Scheme*, in D. DeGroot, G. Lindstrom (eds.) *Logic Programming: Functions, Relations, Equations*, Prentice Hall, 1986.

[81] J. Jaffar,M.J. Maher: *Constraint Logic Programming: A Survey*, To appear in Journal of Logic Programming.

[82] J. Jaffar, S. Michaylov: *Methodology and Implementation of a Constraint Logic Programming System*, Procs. 4th Int. Conf. on Logic Programming ICLP' 87, MIT Press, 1987, 196-218. Extended version as Tech. Rep., Dep. Comp. Science, Monash Univ., 1986.

[83] J. Jaffar, S. Michaylov, P. Stuckey, R.H.C. Yap: *An Abstract Machine for CLP($\mathcal{R}$)*, Proceedings ACM-SIGPLAN Conference on Programming Language Design and Implementation, 1992, 128-139.

[84] J. Jaffar, S. Michaylov, P. Stuckey, R. Yap: *The CLP($\mathcal{R}$) Language and System*, ACM Transactions on Programming Languages, 14(3), 1992, 339-395.

[85] R. Jagadeesan, K. Pingali, P. Panangaden : *A Fully Abstract Semantics for a First-Order Functional Language with Logic variables*, ACM Transactions on Programming Languages ans Systems, 13(4), 1991, 577-625.

[86] J.A. Jiménez-Martín, J. Mariño-Carballo, J.J. Moreno Navarro: *Efficient Compilation of Lazy Narrowing into Prolog*, LOPSTR 92, Springer LNCS, 1992.

[87] J.P. Jouannaud, C. Kirchner: *Solving Equations in Abstract Algebras: A Rule-Based Survey of Unification*, in : J.L.Lassez,G. Plotkin (eds.) Computational Logic, Essays in Honor of Alan Robinson, The MIT Press, 1991,357-321.

[88] S. Kaplan: *Conditional rewrite rules*, Theoretical Computer Science 33,,1984,175-193.

[89] C. Kirchner, H. Kirchner, M. Rusinowitch: *Deduction with Symbolic Constraints*, Revue Franaise Inteligence Artificielle, Vol 4, 3, 1990, 9-52.

[90] H. Kirchner, C. Ringeissen: *Constraint Solving in Combined Algebraic Domains*, in Procs. 11th. Int. Conf. on Logic Programming ICLP'94, MIT Press, 1994, 617-631.

[91] H. Kuchen, R. Loogen, J. J. Moreno-Navarro, M. Rodríguez-Artalejo: *Graph-based Implementation of a Functional Logic Language*, Procs. ESOP'90, Springer LNCS 432, 1990, 271-290.

[92] H. Kuchen, F.J. López Fraguas: *Result Directed Computing in a Functional Logic Language*, Procs. Second Int. Workshop on Functional/Logic Programming (A. Mück ed.), Munchen, 1993, 14-34. Also as Tech. Report 92-21, RWTH Aachen, 1992.

[93] H. Kuchen, F.J. López Fraguas, J.J. Moreno Navarro, M. Rodríguez Artalejo: *Implementing a Lazy Functional Logic Language with Disequality Constraints*, Procs. Joint Int. Conf and Symp. on Logic Programming, MIT Press, 1992, 207-221.

[94] H. Kuchen, F.J. López Fraguas, J.J. Moreno Navarro, M. Rodríguez Artalejo: *Implementing Disequality in a Lazy Functional Logic Language* , Tech. Report 92-20, RWTH Aachen, 1992.

[95] K. Kunen: *Negation in Logic Programming*, Journal of Logic Programming, 4, 1987, 289-308.

[96] C. Lassez: *Constraint Logic Programming: a Tutorial*, in BYTE Magazine, August 1987.

[97] C. Lassez, M. Maher, K.G. Marriot: *Unification Revisited*, in J. Minker (ed.): *Foundations of Deductive Databases and Logic Programming*, Morgan-Kaufmann, 1988, 587-625.

[98] W. Leler: *Constraint Programming Languages: Their Specification and Generation*, Addison Wesley, 1988.

[99] G. Levi, P.G. Bosco, E. Giovannetti, C. Moiso, C. Palamidessi: *A Complete Semantic Characterization of K-LEAF, a Logic Language with Partial Functions*, Procs. 4th Symp. on Logic Programming, 1987, 1-27.

[100] J. W. Lloyd: *Foundations of Logic Programming*, Springer-Verlag, Second Edition, 1987.

[101] H. Lock: *The Implementation of Logic Functional Programming Languages*, PhD dissertation, Berlin, 1992.

[102] R. Loogen: *From Reduction Machines to Narrowing Machines*, TAPSOFT 91, Springer LNCS 494, 1991, 438-457.

[103] R. Loogen, F. J. López Fraguas, M. Rodríguez Artalejo: *A Demand Driven Computation Strategy for Lazy Narrowing*, Int. Symp. on Programming Language Implementation and Logic Programming (PLILP'93), Springer LNCS, 1993, 184-200.

[104] F.J. López Fraguas: *Diseño de un Compilador de Babel en Prolog*, Trabajo de Investigación de Tercer ciclo, DIA-UCM, 1990.

[105] F.J. López Fraguas: *A General Scheme for Constraint Functional Logic Programming*, Procs. 3rd. Int. Conf. on Algebraic and Logic Programming (ALP'92), Springer LNCS 632, 1992, 213-217.

[106] F.J. López Fraguas, R.M. Pinero Fernández: *Building a Babel Compiler with Logic Programming Techniques*, Workshop on the Integration of Functional and Logic Programming, Granada, Spain, 1990.

[107] F.J. López-Fraguas, M. Rodríguez-Artalejo: *An Approach to Constraint Functional Logic Programming*, Tech. Rep. DIA 91/4, 1991.

[108] M. J. Maher: *Complete Axiomatizations of the Algebras of Finite, Rational and Infinite Trees*, Procs. 3rd IEEE Symp. Logic in Computer Science, 1988, 348-357.

[109] M. J. Maher: *A CLP view of Logic Programming*, Proc. 3rd. Conf. on Algebraic and Logic Programming, LNCS 632, 1992, 364-383.

[110] L. Mandel: *The Semantics of the Untyped Constrained Lambda Calculus*, Technical Report 9319, Institut fur Informatic, Ludwig-Maximilians-Univ., Munchen, October 1993.

[111] A. Middeldorp, E. Hamoen: *Counterexamples to Completeness Results for Basic Narrowing*, Proc. 3rd Int. Conf. on Algebraic and Logic Programming, Springer LNCS 463, 1992.

[112] D. Miller, G. Nadathur: *A Higher-Order Logic Programming Language*, Proc. 3rd Int. Conf. on Logic Programming ICLP'86, MIT Press, 1986, 448-462.

[113] R. Milner: *A Theory of Type Polymorphism in Programming*, Journal of Computer and System Science 17(3), 1978, 348-375.

[114] J.J. Moreno Navarro: *Diseño, semántica e implementación de BABEL: Un lenguaje que integra la programación funcional y lógica*, Tesis doctoral, UPM, Madrid,1989.

[115] J.J. Moreno Navarro: *Default Rules: An Extension of Constructive Negation for Narrowing-based Languages*, Procs. 11th. Int. conf. on Logic Programming ICLP'94, MIT Press,1994,535-549.

[116] J. J. Moreno Navarro, H. Kuchen, R. Loogen, M. Rodríguez Artalejo: *Lazy Narrowing in a Graph Machine*, Procs. 2nd International Conference on Algebraic and Logic Programming ALP'90, Springer LNCS 463, 1990, 298-317. Detailed version appeared as: Aachener Informatik-Bericht Nr. 90-11.

[117] J.J. Moreno Navarro, M. Rodríguez Artalejo: *BABEL: A functional and logic language based and constructor discipline and narrowing*, Procs. 1st International Conference on Algebraic and Logic Programming ALP'88, Springer LNCS 343, 1989, 223-232.

[118] J.J. Moreno Navarro, M. Rodríguez Artalejo: *Logic Programming with Functions and Predicates: The Language BABEL*, J. Logic Programming, 12, 1992, 189-223.

[119] A. Mück: *Compilation of Narrowing*, Procs. Programming Language Implementation and Logic Programming PLILP'90, Springer LNCS 456, 1990, 16-39.

[120] A. Mück,T. Streicher, H. Lock: *A Tiny Constraint Functional Logic Language and Its Continuation Semantics*, Procs. European Symp. on Programming ESOP'94, Springer LNCS.

[121] G. Nadathur, D. Miller: *An overview of λ-Prolog*, Proc. Int. Conf. on Logic Programming ICLP'88, MIT Press, 1988, 810-827.

[122] S.Narain: *A technique for doing lazy evaluation in logic*, The Journal of Logic Programming, 3, 1986, 259-276.

[123] L. Naish: *Negation and Control in PROLOG*, Springer LNCS 238, 1986.

[124] W. Nutt, P. Réty, G. Smolka: *Basic Narrowing Revisited*, J. of Symbolic Computation, 7, 1989, 295-317.

[125] M.J. O'Donnel: *Equational Logic as Programming Language*, MIT Press, 1985.

[126] P. Padawitz: *Computing in Horn Clause Theories*, volume 16 of EATCS Monographs on Theoretical Computer Science, Sptinger, 1988.

[127] S. Peyton-Jones: *The Implementation of Functional programming Languages*, Prentice Hall, 1987.

[128] V. Poirriez: *MLOG: a pragmatic extension of ML with logical variables, unification and suspensions*, Procs. Second Int. Workshop on Functional/Logic Programming (A. Mück ed.), Munich, 1993, 105-110.

[129] M.J. Ramírez Quintana: *Negación Constructiva para Programación Lógica Ecuacional*, Tesis doctoral, UPV, 1993.

[130] M.J. Ramírez Quintana, M. Falaschi: *Conditional Narrowing with Constructive Negation*, Procs. Third Int. Workshop on Extensions of Logic Programming ELP'92, Springer LNCS 660, 1993, 59-79.

[131] U.S. Reddy: *Narrowing as the Operational Semantics of Functional Languages*, Procs. International Symposium on Logic Programming, IEEE Comp. Soc. Press 1985, 138-151.

[132] U.S. Reddy: *Functional Logic Languages*, Part I, Procs. Workshop on Graph Reduction, Springer LNCS 279, 1987, 401-425.

[133] P. Roussel: *Définition et traitement de l"egalité formelle en démonstration automatique*, These de 3eme cycle, Groupe DÍntelligence Artificielle, Univ. Marseille, 1972.

[134] A. Rubio: *Automated Deduction with Constrained Clauses*, Tesi doctoral, UPC, Barcelona, 1994.

[135] K. Sakai, A. Aiba: *CAL: A Theoretical Background of Constraint Logic Programming and its Applications*, J. Symbolic Computation 8, 1989, 589-603.

[136] V. Saraswat: *Concurrent Constraint Programming Languages*, Ph.D. thesis, Carnegie-Mellon University, 1989. Revised version appears as Concurrent Constraint Programming, MIT Press, 1993.

[137] V. Saraswat, M. Rinard, P. Panangaden: *Semantic Foundation of Concurrent Constraint Programming Languages*, Procs. 18th ACM Symp. on Principles of Programming Languages,1991, 333-352.

[138] D.S. Scott: *Domains for Denotational Semantics*, Procs. ICALP'82, Springer LNCS 140, 1982, 577-613.

[139] R.C. Sekar and J.V. Ramakrishnan: *Programming in Equational Logic: Beyond Strong Sequentiality*, Procs. LICS'90, IEEE Comp. Soc. Press, 274-284.

[140] J. Siekmann: *Unification Theory*, Journal of Symbolic Computation, 7, 207-274, 1989.

[141] F.S.K Silbermann,B. Jarayaman: *Set Abstraction in Functional and Logic Programming*, Conference on Functional Programming and Computer Arquitecture, ACM, IFIP, 1989.

[142] J.L. Slagle: *Automated theorem proving for theories with simplifiers, conmutativity and associativity*, Journal of the ACM, 21,4,1974, 622-642.

[143] D.A. Smith: *Constraint Operations for CLP(FT)*, Procs. 8th Int. Conf. on Logic Programming, MIT Press, 1991, 760-774.

[144] M.S. Stickel: *Unification Algorithms for Artificial Intelligence Languages*, PhD Thesis, Carnegie Mellon University, 1976.

[145] P. Stuckey: *Constructive Negation for Constraint Logic Programming*, Proc. Logic in Computer Science Conference LICS'91, 1991, 328-339.

[146] E. Tsang: *Foundations of Constraint Satisfaction*, Academic Press, 1993.

[147] D.H.D. Warren: *Higher-order extensions to Prolog: are they needed?*, in J.E. Hayes, D. Mitchie, Y.H. Yao (eds.), Machine Intelligence 10, Ellis Horwood, 1982, 441-454.

[148] D.H.D. Warren: *An abstract Prolog instruction set*, Technical Report 309, SRI International 1983.

[149] D. Wolz: *Design of a compiler for lazy pattern driven narrowing*, 7th. Workshop on Specification of ADTs, Springer LNCS 534, 1991, 362-379.