Técnicas de diagnóstico y depuración declarativa para lenguajes lógico-funcionales

Rafael Caballero Roldán

Departamento de Sistemas Informáticos y Programación
Universidad Complutense de Madrid

Memoria presentada para optar al título de:
Doctor en C.C. Matemáticas

Director:
Mario Rodríguez Artalejo

Tribunal de Lectura:

Presidente:	Ricardo Peña Marí	U.C. Madrid
Secretario:	Manuel García Clavel	U.C. Madrid
Vocales:	Herbert R. Kuchen	U. Münster
	Manuel Hermegildo Salinas	U.P. Madrid
	María Alpuente Frasnedo	U.P. Valencia

Madrid, 2004

Índice

Resumen

La falta de herramientas auxiliares tales como depuradores o entornos gráficos ha sido señalada en [93] como una posible limitación para el éxito, fuera de ámbitos puramente académicos, de los lenguajes funcionales tipo Haskell [75]. Este mismo razonamiento es aplicable al caso de los lenguajes lógicos [51, 8], lógico-funcionales [38] y en general a todos los llamados *lenguajes declarativos*.

Sin embargo, la incorporación de estas herramientas no es siempre sencilla, ya que a menudo las ideas empleadas para su desarrollo en otros paradigmas no son aplicables a los lenguajes declarativos. Este es el caso de la depuración, donde el enfoque seguido tradicionalmente en programación imperativa, basado en la ejecución paso a paso de los programas, no resulta adecuado para la programación declarativa, especialmente en el caso de los lenguajes llamados *perezosos* en los que el orden de ejecución de las expresiones no viene determinado por la posición en la que éstas aparecen en el programa sino por el momento en que su evaluación resulta necesaria.

Entre las diferentes alternativas presentadas para resolver este problema se encuentra la *depuración declarativa*, presentada primero en el contexto de la programación lógica [82, 51] y adaptada después a la programación funcional [71, 68, 77] y a la programación lógica con restricciones [88]. Esta técnica parte de un *síntoma inicial* (un comportamiento inesperado del programa) y construye un árbol de prueba para el cómputo asociado a dicho síntoma inicial. Este árbol, en el que aparece reflejado el comportamiento del programa durante el cómputo, abstrayendo detalles como el orden de ejecución, es después recorrido por el depurador, que va preguntando al usuario por la validez de los resultados de los subcómputos realizados hasta localizar el origen del error.

La programación lógico-funcional [38] recoge ideas tanto de la programación lógica como

funcional perezosa para dar lugar a lenguajes como $\mathcal{TOY}$ [1, 54] y Curry [39], que tratan de aunar las ventajas de ambos paradigmas. El problema de la construcción de depuradores para estos lenguajes plantea los inconvenientes ya señalados, y ello motiva la investigación de técnicas alternativas a las tradicionales también para estos lenguajes.

En la presente tesis desarrollamos un marco teórico para la depuración declarativa de respuestas incorrectas en lenguajes lógico-funcionales y nos ocupamos así mismo de su implementación. Para ello definiremos un cálculo semántico, basado en la lógica CRWL [35, 36] propuesta como marco semántico para programas lógico-funcionales, que nos permita la definición formal de árboles de cómputo adecuados para la depuración de respuestas incorrectas en nuestro marco. Con objeto de poder llevar estas ideas teóricas a la práctica definiremos además una transformación de programas capaz de producir programas que generan árboles de cómputo como parte de sus resultados, y probaremos la corrección de esta transformación con respecto al cálculo semántico.

Entre nuestros objetivos también está comprobar la viabilidad de la implementación de estas ideas. Para ello hemos incorporado depuradores declarativos de respuestas incorrectas tanto al sistema $\mathcal{TOY}$ como al sistema Curry desarrollado por la Universidad de Münster, y desarrollado técnicas destinadas a mejorar la utilización de estos depuradores.

Como tanto la programación lógica como funcional tienen cabida en nuestro marco teórico, nuestras pruebas de corrección sirven a la vez para probar la corrección de los depuradores basados en estas mismas ideas para la programación lógica y para la programación funcional. Aunque en programación lógica sí es habitual encontrar pruebas de corrección de los depuradores declarativos, no ocurre lo mismo en programación funcional, por lo que nuestro enfoque resulta novedoso en este sentido.

Agradecimientos

El resultado de esta tesis, o al menos lo que de interesante pueda tener, se debe no sólo al autor sino a las personas que han colaborado con él durante su elaboración. A ellos va dedicado este apartado.

Mario Rodríguez Artalejo ha sido un cuidadoso, atento y concienzudo director de tesis. Su paciencia y dedicación han sido fundamentales para el desarrollo del trabajo. Durante este tiempo hemos compartido libros, conversaciones y viajes que me han enriquecido personalmente más allá del aprendizaje científico y los teoremas.

Tengo la suerte de conocer a Paco López Fraguas desde hace 15 años. Lo de la "suerte"es fácil de explicar: dirigió mi proyecto fin de carrera en la Escuela de Informática de la U.P.M., me animó a estudiar Matemáticas y después a presentar la solicitud para mi primera plaza en la Universidad, ha colaborado de forma decisiva en muchos resultados de esta tesis y me ha ayudado y aconsejado en todos los momentos difíciles. Gracias Paco.

Herbert Kuchen y Wolfgang Lux de la Universidad de Münster (Alemania) colaboraron en el desarrollo del depurador declarativo incluido en el compilador de Curry desarrollado en esta Universidad. Agradezco su cálida acogida durante mis estancias en Münster, durante las cuales he escrito gran parte de la tesis. Wolfgang es además coautor de la técnica utilizada en el capítulo 5 para la depuración declarativa de programas con búsqueda encapsulada.

Mercedes Abengózar colaboró en la implementación del depurador declarativo incorporado en el sistema $\mathcal{TOY}$.

Agradezco también a todo el grupo de Programación Declarativa su colaboración y apoyo. En particular a Ana y Puri por sus consejos, a Jaime por el desarrollo de $\mathcal{TOY}$, su paciencia y su buen humor, y a Eva por su simpatía y las correcciones de las transparencias para la presentación. Igualmente doy las gracias al resto de los compañeros del Departamento de Sistemas Informáticos y Programación por crear el ambiente de trabajo que ha permitido el desarrollo de la tesis, y a Luis Hernández su disponibilidad, siempre amable y eficaz.

Parte de los recursos (ayuda para la asistencia a conferencias, máquinas, etc) utilizados en esta tesis han sido aportados por los proyectos de investigación CICYT-TIC98-0445-C03-02/97 ("TREND") y CICYT-TIC2002-01167 ("MELODIAS").

Finalmente, a las personas que han colaborado de forma nada científica y absolutamente imprescindible en el trabajo: mis padres, mi hermana, Gus y Ruta.

Capítulo 1

Introducción

Comenzamos exponiendo cuál ha sido la motivación que nos ha llevado a la elaboración de esta tesis y los objetivos perseguidos. Estos dos puntos ocupan las dos primeras secciones de este capítulo. En la tercera y última sección describiremos brevemente el contenido del resto de los capítulos y de los apéndices finales y enumeraremos las publicaciones que recogen partes del material expuesto en la tesis.

1.1. Motivación

La motivación principal de esta tesis se basa en dos puntos complementarios:

1. Hasta el presente no se ha invertido aún suficiente esfuerzo en el desarrollo de depuradores para lenguajes lógico-funcionales, aunque es un hecho conocido que buena parte de la utilidad real de un sistema pasa por la incorporación de herramientas de este tipo.

2. Las técnicas empleadas habitualmente para la depuración de lenguajes imperativos u orientados a objetos no resultan aplicables en nuestro caso. Una alternativa es la *depuración declarativa*, que ya ha sido empleada en programación lógica y funcional, pero no en programación lógico-funcional.

De aquí surge la idea de desarrollar los fundamentos teóricos para la depuración declarativa de lenguajes lógico-funcionales perezosos y de su posible implementación práctica.

Para profundizar en estos puntos exponemos a continuación algunas ideas claves sobre la programación declarativa. A continuación hablaremos de la importancia de la incorporación de herramientas tales como depuradores para obtener sistemas basados en lenguajes declarativos que puedan tener una utilidad real, y mostraremos las ventajas que puede suponer el empleo de la depuración declarativa en el caso de la programación lógico-funcional.

1.1.1. Programación Declarativa

La idea básica de la programación declarativa es desarrollar lenguajes de programación en los que los cómputos se correspondan con demostraciones en una cierta lógica asociada. Una de las ventajas de este enfoque es que el programador sólo debe describir cuáles son las características del problema, sin tener que preocuparse en detalle de cómo los cómputos son llevados a la práctica. Además, el alto nivel de abstracción de estos lenguajes permite probar formalmente la corrección de los programas y razonar acerca sus propiedades de forma sencilla. Estas ideas han dado lugar a dos corrientes distintas: la programación lógica y la programación funcional.

La programación lógica representa sus programas como fórmulas lógicas, en particular como *cláusulas*, y emplea mecanismos también propios de la lógica matemática (como la resolución SLD) para la resolución de objetivos [51, 8]. Probablemente el lenguaje más famoso de este tipo es Prolog [87], aunque no se trate de un lenguaje lógico "puro". La programación lógica con restricciones [44, 45, 46] amplía el marco tradicional de la programación lógica definiendo las restricciones atómicas sobre un cierto dominio que pueden formar parte de cláusulas y objetivos.

La programación funcional, en cambio, utiliza una lógica ecuacional y sus programas consisten en un conjunto de funciones. Los objetivos se ven en este paradigma como expresiones a evaluar mediante un mecanismo conocido como *reescritura* [29]. Entre los lenguajes de este tipo más conocidos se encuentran Haskell [9] y ML [74].

La programación lógico-funcional [38] se inició en los años 80 recogiendo ideas tanto de la programación lógica como funcional para dar lugar a lenguajes que tratan de aunar las ventajas de ambos paradigmas. Uno de los primeros lenguajes en esta línea es LOGLISP [81] de J.A. Robinson y E.E. Sibert. Otros lenguajes de este mismo tipo desarrollados posteriormente son BABEL [60], o más recientemente $\mathcal{TOY}$ [54, 1] y Curry [39], para los que existen sistemas actualmente disponibles y en los que nos centraremos en esta tesis. Estos lenguajes no son una "mezcla" sin más de principios de programación funcional y lógica, sino que tienen características propias tanto en su fundamentos teóricos como en su implementación, así como en las técnicas y metodologías de programación aplicables.

1.1.2. Herramientas de Desarrollo

La falta de herramientas auxiliares tales como depuradores o entornos gráficos ha sido señalada en [93] como una posible limitación para el éxito fuera de ámbitos puramente académicos de los lenguajes funcionales tipo Haskell [75]. Este mismo razonamiento es aplicable a los casos de los lenguajes lógicos [51, 8], lógico-funcionales [38] y en general a todos los lenguajes declarativos. En efecto, la carencia de estas herramientas en un compilador o intérprete no resulta apreciable cuando se elaboran pequeños programas experimentales de pocas líneas sino en desarrollos complejos reales. De hecho, si el uso de un sistema se va a limitar a programas de pocas líneas, es preferible que éste sea lo más simple posible, ahorrándonos la obligación de pasar por opciones tales como creación de proyectos o selección de una "plantilla" inicial para la aplicación. Es al intentar aplicar el sistema a problemas reales cuando surge la necesidad de tales herramientas a apoyo, cuya ausencia

acaba en la práctica llevando al rechazo del sistema frente a otros mucho más evolucionados en estos aspectos.

Sin embargo, la adopción de estas herramientas en los sistemas para lenguajes declarativos no pasa simplemente por trasladar ideas ya conocidas en otros paradigmas. Por el contrario, a menudo exige el desarrollo de ideas y conceptos nuevos, como sucede en el caso de la depuración.

1.1.3. Depuración Declarativa

Si consideramos detenidamente el desarrollo y construcción de depuradores para el paradigma declarativo, observaremos en seguida que la propia naturaleza de los lenguajes declarativos hace que enfoques tradicionales utilizados, por ejemplo, en los sistemas para lenguajes imperativos, tales como la ejecución del programa paso a paso, no puedan ser aplicados tan fácilmente a este paradigma. Esto se debe al alto nivel de abstracción de estos lenguajes, que produce un gran alejamiento entre el significado del programa y cómo es este ejecutado en la práctica.

En el caso de los lenguajes perezosos, entre los que se encuentran los lenguajes lógico-funcionales considerados en esta tesis, ni siquiera está claro el orden ni el momento en el que serán evaluadas las diferentes partes de una expresión, al depender dicha evaluación del momento en que los resultados sean demandados por otras partes del programa. Esto permite a los sistemas perezosos evitar evaluaciones innecesarias y al programador manejar estructuras de datos potencialmente infinitas, pero dificulta el uso de mecanismos tradicionales tales como los puntos de parada, la evaluación paso a paso o la monitorización del contenido de las variables.

La *depuración declarativa* se encuentra entre las diferentes alternativas encaminadas a la construcción de depuradores para lenguajes declarativos. La idea fue presentada primero en el contexto de la programación lógica [82, 32, 52] y adaptada después a la programación funcional [71, 68, 77] y a la programación lógica con restricciones [88], aunque como indica Lee Naish en [63] la técnica puede ser aplicada a cualquier paradigma.

La idea fundamental de la depuración declarativa es partir de un cómputo erróneo, el llamado *síntoma inicial*, generar un árbol de cómputo adecuado para dicho síntoma inicial, y recorrer dicho árbol con ayuda de un *oráculo externo*, normalmente el usuario, al que se van realizando preguntas acerca de la corrección de los nodos del árbol. La construcción del árbol debe garantizar que cada nodo tenga asociados:

- El resultado de algún subcómputo realizado durante el cómputo principal.
- El fragmento de código del programa depurado responsable de dicho subcómputo.

Los nodos hijos corresponderán entonces a los cómputos auxiliares que han sido necesarios para llegar al resultado almacenado en el nodo padre, teniendo cada uno de ellos a su vez su resultado y fragmento de código asociados. Si se localiza un nodo con un resultado asociado que no es correcto, pero tal que los resultados de sus hijos sí lo son, tendremos que el fragmento de código asociado a dicho nodo ha producido un resultado incorrecto

a partir de resultados auxiliares correctos, y que, por tanto, se trata de un fragmento de código erróneo, que será señalado por el depurador como la causa del error.

El tipo de árbol de cómputo variará para diferentes tipos de errores. En programación lógica se consideran dos tipos de errores susceptibles de ser tratados mediante depuración declarativa:

- Respuestas perdidas: En el conjunto de todas las respuestas obtenidas para un objetivo dado falta alguna respuesta esperada. Un caso particular de esta situación es cuando no se obtiene ninguna respuesta cuando se esperaba al menos una.
- Respuestas incorrectas: Se obtiene una respuesta inesperada para un objetivo determinado.

En programación funcional, como veremos, sólo se considera el segundo tipo de respuestas, las respuestas incorrectas. En programación lógico-funcional, en cambio, sucede como en programación lógica, y podemos hablar tanto de respuestas incorrectas como de respuestas perdidas.

1.2. Objetivos

Nuestro objetivo primordial en esta tesis es desarrollar un marco tanto teórico como práctico para de la depuración declarativa de respuestas incorrectas en lenguajes lógico-funcionales. La extensión de este marco a la depuración de respuestas perdidas queda fuera del objetivo de esta tesis y constituye parte del trabajo futuro planeado.

Podemos precisar con más detalle nuestros objetivos mediante los siguientes 4 puntos:

1. Definir un esquema teórico para la depuración declarativa de respuestas incorrectas (a las que también llamaremos *síntomas positivos*) en el paradigma lógico-funcional, encontrando un árbol de cómputo adecuado para este propósito y probando la corrección de la depuración declarativa basada en este árbol. Para ello definiremos un cálculo semántico basado en la lógica CRWL [35, 36] propuesta como marco semántico para programas lógico-funcionales, y mostraremos cómo obtener los árboles de depuración a partir de las demostraciones en este cálculo.

2. Encontrar una técnica que permita convertir el esquema teórico anterior en una herramienta real, así como probar formalmente la corrección de dicha técnica. Para ello definiremos una transformación de programas en dos fases (transformación de aplanamiento y transformación para la introducción de árboles de cómputo). Probaremos que las respuestas obtenidas utilizando el programa transformado incluyen los árboles de depuración de su cómputo correspondiente en el programa original. Estos árboles serán los utilizados por el depurador.

3. Llevar a la práctica estas ideas, incorporando un depurador declarativo a los sistemas lógico-funcionales $\mathcal{TOY}$ [54, 1] y Curry [39]. Comprobaremos que la transformación de

programas es independiente de los detalles del sistema particular, lo que facilitará su incorporación a sistemas basados en implementaciones completamente diferentes.

4. Estudiar la viabilidad de estos depuradores (centrándonos en el de $\mathcal{TOY}$) discutiendo las heurísticas que permiten el mayor aprovechamiento de la herramienta, tanto desde el punto de vista de recursos consumidos (tiempo y memoria) como en cuanto a las preguntas realizadas al usuario. Con respecto a este último punto introducimos dos ideas: el uso de especificaciones parciales para detectar la validez (o incorrección) de algunos nodos del árbol de forma automática, y una noción decidible de *implicación* que relaciona la validez de algunos nodos con la de otros nodos, evitando preguntas innecesarias al usuario.

1.3. Estructura de la Tesis

El siguiente capítulo repasará el llamado habitualmente *estado de la cuestión* (o *estado del arte*). Allí veremos las soluciones aportadas para la depuración de lenguajes declarativos lógicos (y lógicos con restricciones), funcionales y lógico-funcionales. Aparte del interés en sí mismo que tiene conocer los trabajos relacionados y las diferentes propuestas existentes, este capítulo nos permitirá presentar más a fondo las ideas de la depuración declarativa, y también nuestros objetivos y aportaciones. Para mostrar mediante un ejemplo el esquema general en el que se basa la depuración declarativa hemos definido un mini-depurador declarativo genérico en Haskell, que se puede encontrar en el apéndice C, y que no tiene aún pretensiones de ser un depurador práctico sino que sólo pretende ilustrar las ideas expuestas en este apartado.

Los siguientes cuatro capítulos se corresponden con los cuatro puntos citados en la sección 1.2 como objetivos de la tesis. Los dos primeros, correspondientes con los capítulos 3 y 4, constituyen el cuerpo teórico o primera parte de la tesis. En particular, el capítulo 3 presenta algunos conceptos básicos de programación lógico-funcional y la sintaxis de los lenguajes considerados, a la vez que aprovecha para introducir el cálculo semántico cuyas demostraciones nos servirán como árboles de depuración y prueba formalmente la corrección de un depurador basado en estos árboles. El capítulo 4 define la transformación de programas que permitirá obtener los árboles de depuración en la práctica. Además de la transformación, que se realiza en dos fases, el capítulo también incluye los teoremas que prueban formalmente la corrección de esta transformación desde el punto de vista de la depuración declarativa.

En la segunda parte de la tesis, capítulos 5 y 6, discutimos los aspectos prácticos de la implementación. En particular el capítulo 5 muestra como se han llevado a la práctica las ideas de los capítulos anteriores en los compiladores $\mathcal{TOY}$ [1, 54] y Curry [39] de Münster. Se discuten además aspectos que no habían sido considerados anteriormente como la depuración de programas incluyendo primitivas, y las limitaciones en cuanto al subconjunto de los lenguajes considerados en ambos depuradores. El capítulo 6 se centrará en la navegación de los árboles de depuración y presentará experimentos realizados para determinar la eficiencia de la generación de estos árboles, planteando dos alternativas: generación perezosa e impaciente. También se presentará el entorno gráfico $\mathcal{DDT}$ incorporado en $\mathcal{TOY}$,

comentando las ventajas y desventajas que presenta, así como las diferentes *estrategias de navegación* que incorpora. Finalmente se muestran dos ideas incorporadas en este depurador gráfico encaminadas a reducir el número de preguntas que el usuario debe contestar durante el proceso de depuración.

Para finalizar, el capítulo 7 hará un resumen de los resultados obtenidos y de las consecuencias que se deducen de ellos. También planteará líneas para un trabajo posterior basadas en los resultados de la tesis.

Además del ya reseñado apéndice C, la tesis incluye otros dos apéndices. El apéndice A incluye la demostración de aquellos proposiciones o teoremas que por su excesiva longitud hemos preferido probar por separado. Con ello pretendemos facilitar la lectura de los capítulos en los que se presentan estos resultados, ya que algunos de ellos requieren de gran cantidad de resultados auxiliares y pueden hacer "perder el hilo" al lector. La mayoría de las pruebas no son complicadas y se realizan por inducción, ya sea inducción estructural sobre expresiones o patrones o sobre la profundidad de una prueba en un cálculo semántico. Muchas son, sin embargo, prolijas, sobre todo debido a la necesidad de realizar muchas distinciones de casos. Aunque hemos tratado de ser suficientemente precisos, en ocasiones hemos abreviado la redacción de las demostraciones refiriéndonos a resultados análogos.

El apéndice B incluye una recopilación de diversos programas con errores y sus correspondientes sesiones de depuración. Muchos de los ejemplos están extraídos de artículos sobre depuración declarativa escritos por otros autores, adaptados aquí a la sintaxis del lenguaje $\mathcal{TOY}$. Otros son ejemplos encontrados por causalidad durante el desarrollo de programas en Haskell, $\mathcal{TOY}$ o Curry, y finalmente algunos están escritos con errores "a propósito" para mostrar aspectos determinados del depurador.

Algunas partes de la tesis están basadas en los siguientes trabajos ya publicados:

- [16]: Presenta el pequeño depurador declarativo genérico en Haskell que se puede encontrar en el apéndice C.

- [17]: Introduce los árboles de prueba utilizados en el depurador y que veremos en el capítulo 3.

- [19]: Incluye una primera versión de la transformación de programas que se presenta en el capítulo 4, incluyendo demostraciones acerca de la corrección de la depuración basada en esta transformación. También se presenta la relación de *implicación* que se usará en el capítulo 6 para evitar preguntas innecesarias al usuario, así como un algoritmo para decidir dicha relación.

- [18]: Presenta el depurador declarativo para el sistema Curry de Münster, discutiendo especialmente la depuración declarativa de cómputos que utilizan la *búsqueda encapsulada* [42], una característica particular del lenguaje Curry. En este trabajo se basan diferentes partes del capítulo 5.

- [20]: Presenta el navegador gráfico $\mathcal{DDT}$ incorporado al depurador declarativo de $\mathcal{TOY}$ y que expondremos en el capítulo 6.

Capítulo 2

Panorámica de la Depuración Declarativa

En este capítulo vamos a introducir las ideas fundamentales de la depuración declarativa. Para ello repasaremos su aplicación a la programación lógica (incluyendo su extensión al caso de la programación lógica con restricciones), la programación funcional y la combinación de ambas, la programación lógico-funcional. En cada caso estudiaremos las problemáticas y las soluciones particulares, y discutiremos brevemente algunas de las técnicas de implementación y los sistemas existentes.

Veremos también que a pesar de las diferencias los principios son los mismos en los tres casos y que de hecho permiten definir un esquema de depuración aplicable a cualquier paradigma.

2.1. Programación Lógica y Programación Lógica con Restricciones

Puesto que las primeras ideas acerca de la depuración declarativa provienen del paradigma la programación lógica (ver [51, 8] para una introducción a este paradigma), nos ha parecido adecuado comenzar con una mención de los primeros trabajos en este campo. Sin embargo la ideas precisas serán expuestas en el contexto más amplio de la programación lógica con restricciones (ver por ejemplo [56]). También dedicaremos algunos breves comentarios a la *diagnosis abstracta*, una técnica diferente a la depuración declarativa también utilizada en programación lógica.

2.1.1. Un poco de historia

Fue E.Y. Shapiro quien en 1982 ([82]) introdujo bajo el nombre de *Depuración Algorítmica* un método para localizar errores en programas lógicos (en particular para el lenguaje *Prolog* [24, 87]) basado en la semántica declarativa de estos lenguajes.

Podemos describir a grandes rasgos el proceso de depuración propuesto en este trabajo, ya que contiene gran parte de las ideas claves de la actual depuración declarativa:

La depuración comienza cuando el usuario observa que un programa lógico produce un resultado inesperado como resultado del cómputo de un objetivo. El depurador entonces repite el cómputo paso a paso, pero preguntando a cierto *oráculo* (normalmente el usuario) si los resultados intermedios de obtenidos en cada paso son los esperados. De esta forma, comparando el modelo representado por el programa con el *modelo pretendido* conocido por el oráculo se puede localizar el error. Dicho modelo pretendido es, en caso del trabajo de Shapiro, un subconjunto de la *base de Herbrand*, es decir un conjunto de átomos sin variables.

Posteriores trabajos de G. Ferrand [32] y J.W.Lloyd [51, 52] continúan y amplían la línea propuesta por Shapiro. En [32] se extiende la noción de modelo pretendido a conjuntos de átomos con variables, mientras que en sus trabajos J.W.Lloyd [51, 52] en lugar de tratar con programas formados por cláusulas de Horn considera programas lógicos más generales formados por fórmulas $A \leftarrow W$ con A un átomo y W una fórmula de primer orden general.

En todos las propuestas los depuradores presentados son meta-intérpretes que repiten el cómputo erróneo para analizarlo paso a paso.

2.1.2. Programación lógica con restricciones

La programación lógica con restricciones amplía el marco tradicional de la programación lógica, definiendo las restricciones atómicas sobre un cierto dominio $\mathcal{D}$ que pueden formar parte de cláusulas y objetivos. Para hacer referencia a la programación lógica con restricciones sobre un cierto dominio $\mathcal{D}$ usaremos la notación $\mathcal{PLR}(\mathcal{D})$.

El interés de la depuración declarativa en este caso es doble. En primer lugar está el interés que tiene en sí mismo establecer un marco para depurar programas lógicos incluyendo restricciones. En segundo lugar, la programación lógica sin restricciones se puede ver como el caso particular $\mathcal{PLR}(Herbrand)$, donde las restricciones atómicas son simplemente ecuaciones interpretadas sobre el universo de Herbrand. Por tanto el marco que presentamos a continuación es válido para programación lógica con o sin restricciones.

Numerosos trabajos en este campo se han realizado en el marco del proyecto DiSCiPl (ver `http://discipl.inria.fr/`). La presentación que realizamos a continuación está basada en gran medida en una reelaboración de las ideas propuestas en estos trabajos (en particular en [88]) ya que entendemos que representan el "estado del arte" de la depuración declarativa de programas de $\mathcal{PLR}(\mathcal{D})$ en la actualidad.

Hemos orientado la discusión de forma que nos permita introducir ideas que serán útiles en posteriores capítulos de la tesis. Por simplicidad utilizaremos ejemplos con programas lógicos sin restricciones, y éstas sólo aparecerán en las respuestas computadas, representando la sustitución obtenida mediante restricciones de igualdad.

2.1.3. Preliminares

Los símbolos t, t_i denotarán en el resto de la sección términos. Abreviaremos las secuencias de términos $t_1, \ldots t_n$ mediante la notación $\overline{t_n}$. Análogamente, $\overline{t_n} = \overline{s_n}$ se utilizará para abreviar la conjunción $t_1 = s_1 \wedge \cdots \wedge t_n = s_n$.

Denotaremos los átomos de la forma $p(\overline{t_n})$, con p un símbolo de predicado, por A. C y B representarán respectivamente conjunciones finitas de restricciones atómicas y de átomos. G denotará una conjunción en la que pueden aparecer tanto restricciones atómicas como átomos.

Utilizaremos el símbolo θ para denotar una sustitución de variables por elementos del dominio $\mathcal{D}$, y la notación $\varphi\theta$ para representar la fórmula obtenida al aplicar dicha sustitución a una fórmula φ.

Asumimos implícitamente la conmutatividad y asociatividad de la conjunción ($\wedge$) y la disyunción ($\vee$). *cierto* y *falso* representarán respectivamente la conjunción y la disyunción vacías.

Usaremos la notación $\exists_{-\varphi_1}\varphi_2$ para denotar que la fórmula lógica φ_2 está cuantificada existencialmente sobre todas sus variables salvo sobre las variables libres de φ_1. En el caso de una implicación lógica, escribiremos a menudo $\varphi_1 \rightarrow \exists_{-izq}\varphi_2$ para denotar $\varphi_1 \rightarrow \exists_{-\varphi_1}\varphi_2$. Un convenio análogo nos permitirá escribir $\varphi_1 \leftrightarrow \exists_{-izq}\varphi_2$ en lugar de $\varphi_1 \leftrightarrow \exists_{-\varphi_1}\varphi_2$.

Un programa P estará formado por cláusulas de la forma:

$$p(\overline{l_n}) \leftarrow G$$

Si G es vacío (i.e. *cierto*) escribiremos simplemente

$$p(\overline{l_n})$$

Supondremos sin perdida de generalidad que un mismo símbolo de predicado p no puede aparecer en el mismo programa aplicado a diferente número de términos (por ejemplo se puede suponer que cada aparición de un átomo de la forma $p(\overline{t_n})$ queda renombrada de forma automática a $p/n(\overline{t_n})$). Con esta suposición podemos referirnos sin ambigüedad a la *aridad* de un predicado del programa. La *definición* de un cierto predicado p de aridad n en un programa P está formada por todas las cláusulas de la forma $p(\overline{t_n}) \leftarrow G$.

Supongamos una definición para un predicado p de aridad n dada por m cláusulas de la forma:

$$p(\overline{l_n^1}) \leftarrow G_1$$
$$\ldots$$
$$p(\overline{l_n^m}) \leftarrow G_m$$

Entonces la *compleción* del predicado p es la fórmula:

$$p(\overline{X_n}) \leftrightarrow \exists_{-izq} \bigvee_{i=1}^{m} (\overline{X_n} = \overline{l_n^i} \wedge G_i)$$

donde $\overline{X_n}$ son variables nuevas. La compleción de p se puede expresar también mediante la fórmula equivalente

$$[p(\overline{X_n}) \leftarrow \exists_{-izq} \bigvee_{i=1}^{m} (\overline{X_n} = \overline{l_n^i} \wedge G_i)] \ \wedge \ [p(\overline{X_n}) \rightarrow \exists_{-izq} \bigvee_{i=1}^{m} (\overline{X_n} = \overline{l_n^i} \wedge G_i)]$$

Las variables existenciales del antecedente de una implicación se pueden eliminar, por lo que tenemos

$$[p(\overline{X_n}) \leftarrow \bigvee_{i=1}^{m} (\overline{X_n} = \overline{l_n^i} \wedge G_i)] \ \wedge \ [p(\overline{X_n}) \rightarrow \exists_{-izq} \bigvee_{i=1}^{m} (\overline{X_n} = \overline{l_n^i} \wedge G_i)]$$

Utilizando está notación podemos definir los conjuntos de fórmulas:

- $SI(P)$ formado por todas las fórmulas de la forma

  $$p(\overline{X_n}) \leftarrow \bigvee_{i=1}^{m} (\overline{X_n} = \overline{l_n^i} \wedge G_i)$$

  con p predicado de P.

- $IS(P)$ formado por todas las fórmulas de la forma

  $$p(\overline{X_n}) \rightarrow \exists_{-izq} \bigvee_{i=1}^{m} (\overline{X_n} = \overline{l_n^i} \wedge G_i)$$

  con p predicado de P.

Estaremos sobre todo interesados en el segundo conjunto; es fácil comprobar que el primero ($SI(P)$) es equivalente, en el marco de la lógica de primer orden con igualdad, al propio P.

La notación $(p(\bar{l}_n) \leftarrow G) \in_{var} P$ indicará que $(p(\bar{l}_n) \leftarrow G)$ es una *variante* (i.e. con variables nuevas) de una cláusula del programa P. De forma análoga se define $p(\overline{X_n}) \rightarrow \exists_{-izq} \bigvee_{i=1}^{m} (\overline{X_n} = \overline{l_n^i} \wedge G_i) \in_{var} IS(P)$

Una interpretación pretendida $\mathcal{I}$ de un programa P en $\mathcal{PLR}(\mathcal{D})$ es una extensión de $\mathcal{D}$ que añade una interpretación por cada símbolo de predicado en P.

2.1.4. Síntomas y Errores

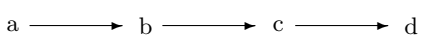
Fue el propio Shapiro el primero en distinguir dos tipos de síntomas iniciales de error, las *respuestas incorrectas* y las *respuestas perdidas*. Vamos a introducir estos dos tipos de síntomas, a los que llamaremos en el contexto de la programación lógica con restricciones *síntomas positivos* y *síntomas negativos* respectivamente, así como sus *errores* asociados por medio de un ejemplo.

Ejemplo 2.1.1. *Caminos en un grafo*

$arco(a, b)$
$arco(b, c)$
% Se nos ha olvidado incluir:
% arco(c,d)

$camino(X, Y) \leftarrow arco(X, Y)$
$camino(X, Y) \leftarrow arco(X, Z) \wedge camino(Y, Z)$
% Hay un error en la última cláusula, que debería ser:
% camino$(X, Y) \leftarrow arco(X, Z) \wedge camino(Z, Y)$

El programa pretende reflejar los caminos en el grafo dirigido:



La *interpretación pretendida* de los predicados es:

- $arco(X, Y)$ sii hay un arco desde X a Y.
- $camino(X, Y)$ sii hay un camino de X a Y de longitud mayor o igual que 1.

Si con el programa del ejemplo lanzamos el objetivo

$$camino(b, Y)$$

se obtendrán dos respuestas computadas mediante resolución SLD:

$$Y = c \wedge \ldots$$
$$Y = b \wedge \ldots$$

Las restricciones que aparecen mediante los puntos suspensivos contendrán los vínculos para otras variables correspondientes a otros objetivos intermedios, pero que no afectan al objetivo inicial (cuya única variable era Y).

La primera respuesta es correcta, pero la segunda es incorrecta: no hay ningún camino de longitud al menos uno de b a si mismo. La definición de *síntoma positivo* expresa formalmente este concepto.

Definición 2.1.1. $C \rightarrow G$ (C restricción, G objetivo) es un síntoma positivo (de un programa P con respecto a una interpretación pretendida $\mathcal{I}$) si C es una respuesta computada para G pero $C \rightarrow G$ no es válida en $\mathcal{I}$.

En nuestro ejemplo, utilizando la segunda respuesta, se tiene que la fórmula

$$Y = b \wedge \cdots \rightarrow camino(b, Y)$$

es un síntoma positivo.

Como veremos posteriormente, un síntoma positivo implica siempre la existencia de un *error positivo* en el programa.

Definición 2.1.2. Un error positivo (en un programa P con respecto a una interpretación pretendida $\mathcal{I}$) es una cláusula $p(\bar{l}_n) \leftarrow G$, a la que llamaremos *cláusula incorrecta*, tal que para alguna restricción C se tiene que $G \wedge C$ es válida en $\mathcal{I}$ pero $p(\bar{l}_n) \wedge C$ no lo es. En este caso diremos que p es un *predicado incorrecto*.

Informalmente una cláusula incorrecta permite inferir hechos incorrectos a partir de hechos correctos. En nuestro ejemplo el error positivo viene dado por la cláusula incorrecta $camino(X, Y) \leftarrow arco(X, Z) \wedge camino(Y, Z)$. En efecto, dada la restricción

$$C \equiv \ X = b \wedge Y = b \wedge Z = c$$

tenemos que en nuestra interpretación pretendida existe un arco de b a c y una camino de b a c, pero no un camino de b a b.

Aún hay un segundo tipo de síntoma de error en este cómputo, pero para detectarlo no basta con mirar las respuestas una a una sino que debe examinarse el cómputo en su conjunto. Así notaremos que en el conjunto de respuestas computadas no se encuentra ninguna respuesta de la forma $Y = d \wedge \ldots$, cuando en el grafo propuesto sí hay un camino desde b hasta d, y que tenemos por tanto una *respuesta perdida*. La definición de *síntoma negativo* recoge esta idea.

Definición 2.1.3. $G \rightarrow \exists_{-Izq}(C_1 \vee \ldots C_k)$ (C_i restricciones, G objetivo) es un síntoma negativo (de un programa P con respecto a una interpretación pretendida $\mathcal{I}$) si para el objetivo G existe un árbol SLD finito cuyas respuestas computadas son $C_1 \ldots C_n$, pero la fórmula $G \rightarrow \exists_{-Izq}(C_1 \vee \ldots C_k)$ no es válida en $\mathcal{I}$.

Es importante resaltar que para la existencia de un síntoma negativo deben poder observarse *todas* las respuestas computadas y esto sólo resulta posible en el caso de objetivos a los que correspondan espacios de búsqueda finitos.

Al igual que sucede con los síntomas positivos, los síntomas negativos también corresponden a un error en el programa al que llamaremos *error negativo*.

Definición 2.1.4. Un error negativo (de un programa P con respecto a una interpretación pretendida $\mathcal{I}$) es una fórmula

$$p(\overline{X_n}) \rightarrow \exists_{-izq} \bigvee_{i=1}^{m} (\overline{X_n} = \overline{l_n^i} \wedge G_i) \in IS(P)$$

tal que para alguna restricción C se tiene que $p(\overline{X_n}) \wedge C$ es válida en $\mathcal{I}$ pero

$$(\exists_{-izq} \bigvee_{i=1}^{m} (\overline{X_n} = \overline{l_n^i} \wedge G_i)) \wedge C$$

no lo es. En este caso se dice que p es un *predicado incompleto.*

En nuestro ejemplo el síntoma negativo es

$$camino(b, Y) \rightarrow \exists_{-Izq}(Y = c \wedge \dots) \vee (Y = b \wedge \dots)$$

mientras que la fórmula de $IS(P)$ y la restricción C son respectivamente

$$arco(X, Y) \rightarrow (X = a \ \wedge \ Y = b) \ \vee \ (X = b \ \wedge \ Y = c)$$

y $C \equiv X = c \wedge Y = d$, por lo que *arco* será el correspondiente predicado incompleto.

El proceso de depuración comenzará cuando se observe un síntoma (ya sea positivo o negativo) por parte del usuario. Por tanto la depuración declarativa no puede utilizarse, por ejemplo, cuando el programa se queda "colgado" sin producir ninguna respuesta para un objetivo dado. El objetivo del depurador será localizar el correspondiente error (positivo o negativo). Para ello en la siguiente sección introduciremos dos cálculos POS y NEG. El primero se utilizará en el caso de síntomas positivos y el segundo cuando se obtenga un síntoma negativo. En cada caso el razonamiento representará un árbol de prueba. Posteriormente veremos que de dichos árboles establecen una relación directa entre síntomas y errores.

2.1.5. Cálculos POS y NEG

Presentamos primero el cálculo POS, que utilizaremos para la depuración en el caso de la aparición de un síntoma positivo.

$POS_\wedge$:

$$\frac{C' \rightarrow C \wedge A \qquad C'' \rightarrow C' \wedge B}{C'' \rightarrow C \wedge A \wedge B}$$

POS_A :

$$\frac{C' \rightarrow C \wedge \overline{l_n} = \overline{t_n} \wedge G}{C' \rightarrow C \wedge p(\overline{t_n})} \quad \text{donde } (p(\overline{l_n}) \leftarrow G) \in_{var} P$$

POS_C :

$$\frac{}{C' \rightarrow C} \qquad \text{donde } C' \text{ es (una forma simplificada de) } C \text{ obtenida por el resolutor}$$

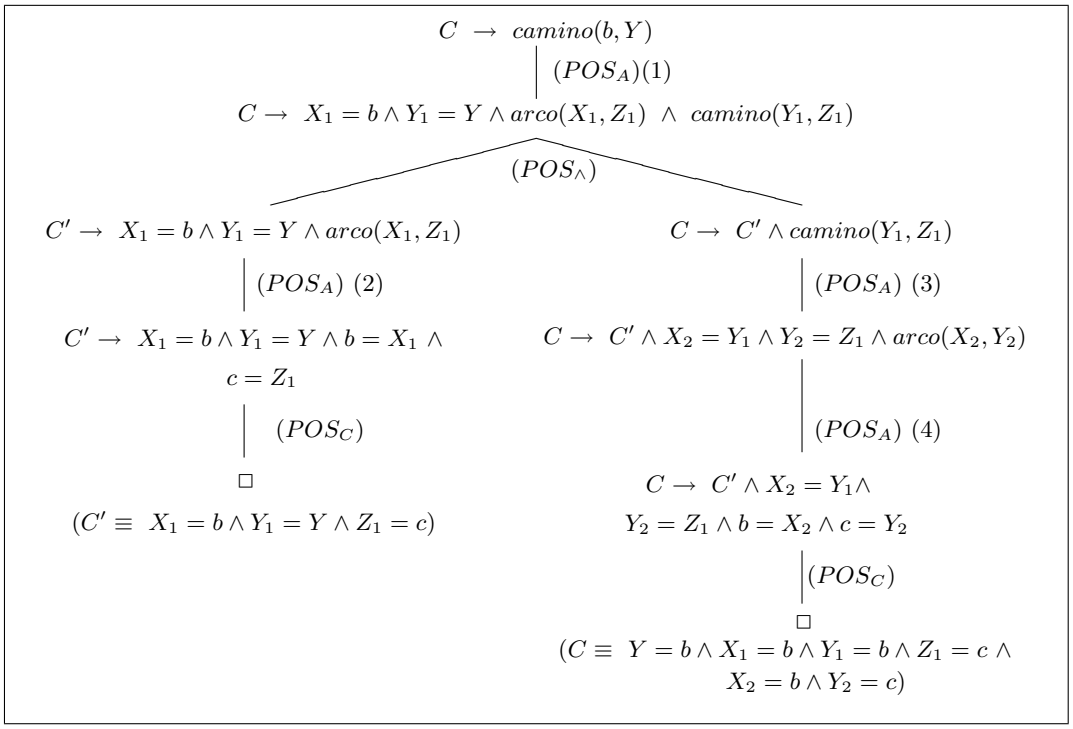


Figura 2.1: Ejemplo de árbol de prueba positivo

El propósito de este cálculo es probar que una restricción C es la respuesta computada para un objetivo G. Como veremos, en el caso de la depuración utilizaremos este cálculo para obtener los árboles de prueba correspondientes a los síntomas positivos.

La primera regla permite tratar los objetivos compuestos de más de una fórmula atómica, indicando que para probar que C'' es una respuesta computada para un objetivo de la forma $C \wedge A \wedge B$ tenemos que encontrar un C' tal que C' sea una respuesta computada para $C \wedge A$ y probar a continuación que C'' es una respuesta computada para $C' \wedge B$. Nótese que debido a la conmutatividad de $\wedge$ un objetivo G siempre se puede escribir como $G \equiv C \wedge B$. La segunda regla (POS_A) nos permite aplicar una (variante de una) regla de programa para el caso en el que tengamos una sola fórmula atómica (como en el caso de la primera premisa de del $POS_\wedge$), mientras que la tercera regla (POS_C) permite simplificar el conjunto de restricciones. Un caso particular de (POS_C) que utilizaremos a menudo es:

$\overline{C \rightarrow C}$

En la figura 2.1 podemos ver un ejemplo de derivación en POS para el ejemplo 2.1.1 escrita en forma de árbol. Llamaremos a los árboles de prueba correspondientes a derivaciones *POS árboles de prueba positivos* o simplemente APP's.

Para eso comenzaremos intentando probar una fórmula genérica de la forma $C \rightarrow camino(b, Y)$. Al finalizar la construcción del árbol obtenemos el valor de C

$$C \equiv Y = b \wedge X_1 = b \wedge Y_1 = b \wedge Z_1 = c \wedge X_2 = b \wedge Y_2 = c$$

Este valor depende de las cláusulas elegidas al aplicar POS_A. En este caso particular hemos obtenido un APP para un síntoma positivo de la forma $Y = b \cdots \rightarrow camino(b, Y)$.

Hemos etiquetado cada nodo con el nombre de la regla de POS utilizada. En el caso de la regla POS_A hemos incluido además un número para etiquetar cada aplicación de dicha regla. Las variantes de regla de programa utilizadas en cada cada caso son:

$(POS_A)(1)$: $camino(X_1, Y_1) \leftarrow arco(X_1, Z_1) \wedge camino(Y_1, Z_1)$

$(POS_A)(2)$: $arco(b, c)$

$(POS_A)(3)$: $camino(X_2, Y_2) \leftarrow arco(X_2, Y_2)$

$(POS_A)(4)$: $arco(b, c)$

Es fácil observar que existe una relación directa entre una prueba en POS y la obtención de una respuesta computada en SLD, tal y como atestigua el siguiente teorema:

Teorema 2.1.1. *C es una respuesta computada por resolución SLD para un objetivo G si y sólo si existe un APP cuyo nodo raíz es $C \rightarrow G$.*

Por tanto las respuestas obtenidas mediante POS son efectivamente respuestas computadas. En la sección 2.1.6 mostraremos el resultado que establece además la corrección de POS.

Al igual que el cálculo positivo nos ha permitido construir árboles de prueba para los síntomas positivos (correspondientes a respuestas incorrectas), podemos utilizar análogamente el siguiente cálculo NEG para construir árboles de prueba para los síntomas negativos (correspondientes a respuestas perdidas).

NEG no se fijará en el computo concreto de una respuesta como hacía POS sino en los cómputos de *todas* las respuestas para un cierto objetivo, probando para ello implicaciones de la forma

$$G \rightarrow \exists_{-G} \bigvee_{i=1}^{m} C_i$$

donde $C_1 \ldots C_m$ serán las respuestas computadas para G. El caso particular $m = 0$ corresponderá por tanto a un *fallo finito* y la implicación anterior se reducirá a $G \rightarrow falso$. Estas son las reglas de NEG:

$NEG_\wedge$:

$$\frac{C \wedge A \rightarrow \exists_{-izq} \bigvee_{i=1}^{m} C_i \qquad \cdots C_i \wedge B \rightarrow \exists_{-izq} \bigvee_{j=1}^{m_i} C_j^i \ (1 \leq i \leq m) \cdots}{C \wedge A \wedge B \rightarrow \exists_{-izq} \bigvee_{i=1}^{m} \bigvee_{j=1}^{m_i} C_j^i}$$

NEG_A :

$$\frac{\cdots C \wedge \overline{l_n^i} = \overline{t_n} \wedge G_i \rightarrow \exists_{-izq} \bigvee_{j=1}^{m_i} C_j^i \ (1 \leq i \leq m) \cdots}{C \wedge p(\overline{t}_n) \rightarrow \exists_{-izq} \bigvee_{i=1}^{m} \bigvee_{j=1}^{m_i} C_j^i}$$

donde $(p(\overline{X}_n) \rightarrow \exists_{-izq} \bigvee_{i=1}^{m} (\overline{X}_n = \overline{l^i}_n \wedge G_i)) \in_{var} IS(P)$

NEG_C :

$$\frac{}{C \to C'}$$ donde C' es (una forma simplificada de) C obtenida por el resolutor

NEG_F :

$$\frac{}{C \wedge B \to falso}$$ donde C es considerado inconsistente por el resolutor

En estas reglas en lugar de usar el programa P se utiliza $IS(P)$ (ver sección 2.1.3). En el caso del programa del ejemplo 2.1.1, $IS(P)$ estará formado por las siguientes reglas:

$$\begin{aligned} arco(X,Y) \to \quad & (X = a \wedge Y = b) \ \vee \ (X = b \wedge Y = c) \\ camino(X,Y) \to \quad & \exists X_1, Y_1, X_2, Y_2, Z_2 \ (X = X_1 \wedge Y = Y_1 \wedge arco(X_1, Y_1)) \vee \\ & (X = X_2 \wedge Y = Y_2 \wedge arco(X_2, Z_2) \wedge camino(Y_2, Z_2)) \end{aligned}$$

Usando NEG se puede escribir un árbol de prueba para una fórmula de la forma:

$$camino(a, Y) \ \to \ \bigvee_{i=1}^{m} C_i$$

Los árboles negativos son de tamaño y complejidad mucho mayores que los positivos. En la figura 2.2 mostramos el árbol negativo de prueba (a los que llamaremos APN's) para una fórmula de la forma $camino(b, Y) \to C$.

El subíndice de la regla NEG aplicada en cada paso se encuentra debajo del nodo padre (que corresponde a la conclusión de la inferencia) para los nodos internos. En los nodos hoja la regla aplicada será NEG_F para fórmulas de la forma $G \to false$, o NEG_C en otro caso.

Estos son los valores correspondientes a las abreviaturas C_i y G_i utilizadas por razones de espacio en la figura:

- $C_1 \equiv Y = c \wedge X_1 = b \wedge Y_1 = c$
- $C_2 \equiv Y = b \wedge X_2 = b \wedge Y_2 = b \wedge Z_2 = c \wedge X_3 = b \wedge Y_3 = c$
- $C_3 \equiv X_2 = b \wedge Y_2 = Y \wedge Z_2 = c$
- $C_4 \equiv Y = a \wedge X_2 = b \wedge Y_2 = a \wedge Z_2 = c \wedge X_4 = a \wedge Y_4 = c \wedge Z_4 = b$
- $C_5 \equiv Y = b \wedge X_2 = b \wedge Y_2 = b \wedge Z_2 = c \wedge X_4 = b \wedge Y_4 = c \wedge Z_4 = c$

- $G_1 \equiv X_1 = b \wedge Y_1 = Y \wedge arco(X_1, Y_1)$
- $G_2 \equiv X_2 = b \wedge Y_2 = Y \wedge arco(X_2, Z_2) \wedge camino(Y_2, Z_2)$

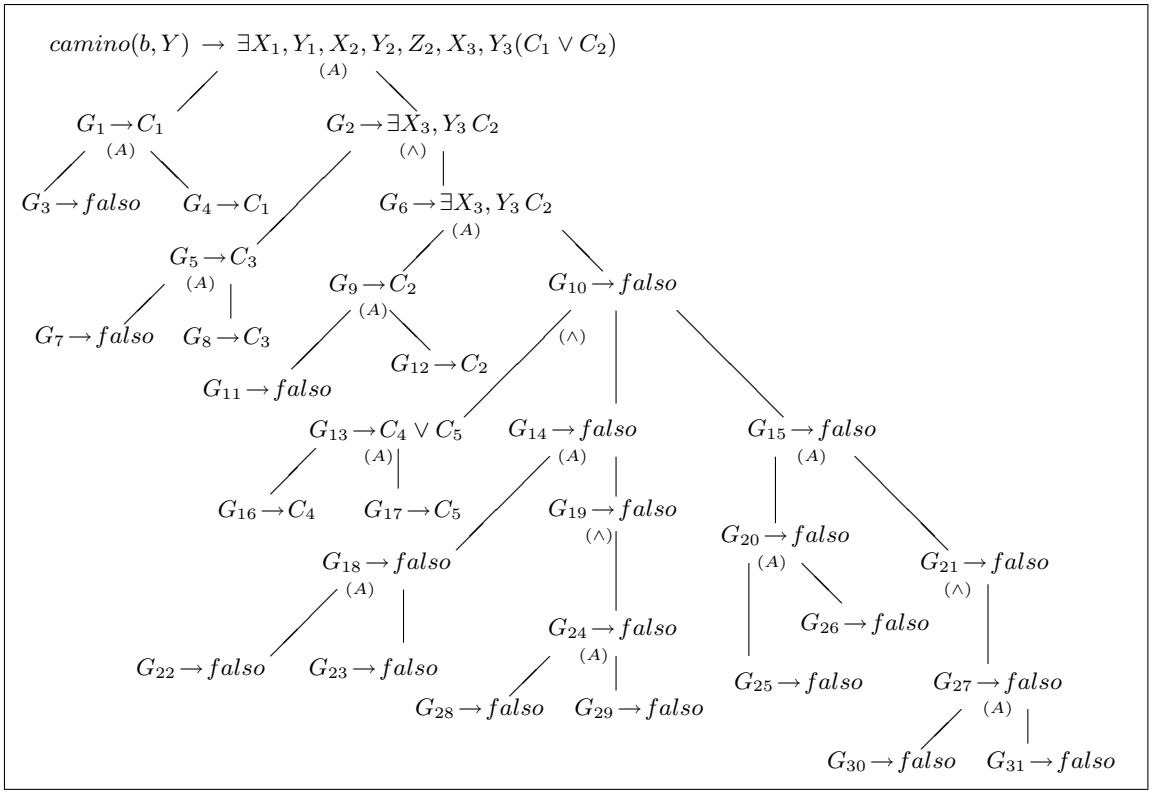


Figura 2.2: Árbol de prueba negativo

- $G_3 \equiv X_1 = b \wedge Y_1 = Y \wedge a = X_1 \wedge b = Y_1$
- $G_4 \equiv X_1 = b \wedge Y_1 = Y \wedge b = X_1 \wedge c = Y_1$
- $G_5 \equiv X_2 = b \wedge Y_2 = Y \wedge arco(X_2, Z_2)$
- $G_6 \equiv C_3 \wedge camino(Y_2, Z_2)$
- $G_7 \equiv X_2 = b \wedge Y_2 = Y \wedge a = X_2 \wedge b = Z_2$
- $G_8 \equiv X_2 = b \wedge Y_2 = Y \wedge b = X_2 \wedge c = Z_2$
- $G_9 \equiv X_2 = b \wedge Y_2 = Y \wedge Z_2 = c \wedge X_3 = Y_2 \wedge Y_3 = Z_2 \wedge arco(X_3, Y_3)$
- $G_{10} \equiv X_2 = b \wedge Y_2 = Y \wedge Z_2 = c \wedge X_4 = Y_2 \wedge Y_4 = Z_2 \wedge arco(X_4, Z_4) \wedge camino(Y_4, Z_4)$
- $G_{11} \equiv X_2 = b \wedge Y_2 = Y \wedge Z_2 = c \wedge a = X_3 \wedge b = Y_3$
- $G_{12} \equiv X_2 = b \wedge Y_2 = Y \wedge Z_2 = c \wedge b = X_3 \wedge c = Y_3$
- $G_{13} \equiv X_2 = b \wedge Y_2 = Y \wedge Z_2 = c \wedge X_4 = Y_2 \wedge Y_4 = Z_2 \wedge arco(X_4, Z_4)$
- $G_{14} \equiv C_4 \wedge camino(Y_4, Z_4)$
- $G_{15} \equiv C_5 \wedge camino(Y_4, Z_4)$
- $G_{16} \equiv X_2 = b \wedge Y_2 = Y \wedge Z_2 = c \wedge X_4 = Y_2 \wedge Y_4 = Z_2 \wedge a = X_4 \wedge b = Z_4$
- $G_{17} \equiv X_2 = b \wedge Y_2 = Y \wedge Z_2 = c \wedge X_4 = Y_2 \wedge Y_4 = Z_2 \wedge b = X_4 \wedge c = Z_4$
- $G_{18} \equiv Y = a \wedge X_2 = b \wedge Y_2 = a \wedge Z_2 = c \wedge X_4 = a \wedge Y_4 = c \wedge Z_4 = b \wedge X_5 = Y_4 \wedge Y_5 = Z_4 \wedge arco(X_5, Y_5)$
- $G_{19} \equiv Y = a \wedge X_2 = b \wedge Y_2 = a \wedge Z_2 = c \wedge X_4 = a \wedge Y_4 = c \wedge Z_4 = b \wedge X_6 = Y_4 \wedge Y_6 = Z_4 \wedge arco(X_6, Z_6) \wedge camino(Y_6, Z_6)$
- $G_{20} \equiv Y = b \wedge X_2 = b \wedge Y_2 = b \wedge Z_2 = c \wedge X_4 = b \wedge Y_4 = c \wedge Z_4 = c \wedge X_7 = Y_4 \wedge Y_7 = Z_4 \wedge arco(X_7, Y_7)$
- $G_{21} \equiv Y = b \wedge X_2 = b \wedge Y_2 = b \wedge Z_2 = c \wedge X_4 = b \wedge Y_4 = c \wedge Z_4 = c \wedge X_8 = Y_4 \wedge Y_8 = Z_4 \wedge arco(X_8, Z_8) \wedge camino(Y_8, Z_8)$
- $G_{22} \equiv Y = a \wedge X_2 = b \wedge Y_2 = a \wedge Z_2 = c \wedge X_4 = a \wedge Y_4 = c \wedge Z_4 = b \wedge X_5 = Y_4 \wedge Y_5 = Z_4 \wedge a = X_5 \wedge b = Y_5$
- $G_{23} \equiv Y = a \wedge X_2 = b \wedge Y_2 = a \wedge Z_2 = c \wedge X_4 = a \wedge Y_4 = c \wedge Z_4 = b \wedge X_5 = Y_4 \wedge Y_5 = Z_4 \wedge b = X_5 \wedge c = Y_5$
- $G_{24} \equiv Y = a \wedge X_2 = b \wedge Y_2 = a \wedge Z_2 = c \wedge X_4 = a \wedge Y_4 = c \wedge Z_4 = b \wedge X_6 = Y_4 \wedge Y_6 = Z_4 \wedge arco(X_6, Z_6)$

- $G_{25} \equiv Y = b \wedge X_2 = b \wedge Y_2 = b \wedge Z_2 = c \wedge X_4 = b \wedge Y_4 = c \wedge Z_4 = c \wedge\ X_7 = Y_4 \wedge Y_7 = Z_4 \wedge a = X_7 \wedge b = Z_5$
- $G_{26} \equiv Y = b \wedge X_2 = b \wedge Y_2 = b \wedge Z_2 = c \wedge X_4 = b \wedge Y_4 = c \wedge Z_4 = c \wedge\ X_7 = Y_4 \wedge Y_7 = Z_4 \wedge b = X_7 \wedge c = Y_7$
- $G_{27} \equiv Y = b \wedge X_2 = b \wedge Y_2 = b \wedge Z_2 = c \wedge X_4 = b \wedge Y_4 = c \wedge Z_4 = c \wedge\ X_8 = Y_4 \wedge Y_8 = Z_4 \wedge arco(X_8, Z_8)$
- $G_{28} \equiv Y = a \wedge X_2 = b \wedge Y_2 = a \wedge Z_2 = c \wedge X_4 = a \wedge Y_4 = c \wedge Z_4 = b \wedge\ X_6 = X_4 \wedge Y_6 = Z_4 \wedge a = X_6 \wedge b = Z_6$
- $G_{29} \equiv Y = a \wedge X_2 = b \wedge Y_2 = a \wedge Z_2 = c \wedge X_4 = a \wedge Y_4 = c \wedge Z_4 = b \wedge\ X_6 = X_4 \wedge Y_6 = Z_4 \wedge b = X_6 \wedge c = Z_6$
- $G_{30} \equiv Y = b \wedge X_2 = b \wedge Y_2 = b \wedge Z_2 = c \wedge X_4 = b \wedge Y_4 = c \wedge Z_4 = c \wedge\ X_8 = Y_4 \wedge Y_8 = Z_4 \wedge a = X_8 \wedge b = Z_8$
- $G_{31} \equiv Y = b \wedge X_2 = b \wedge Y_2 = b \wedge Z_2 = c \wedge X_4 = b \wedge Y_4 = c \wedge Z_4 = c \wedge\ X_8 = Y_4 \wedge Y_8 = Z_4 \wedge b = X_8 \wedge c = Z_8$

En el ejemplo podemos pensar que se ha partido de una implicación de la forma $camino(b, Y) \rightarrow$ y que ha sido el propio cálculo el que ha construido la parte derecha de la implicación.

Al igual que en el caso positivo, el siguiente teorema resalta la equivalencia entre los cómputos SLD y los APN's:

Teorema 2.1.2. *Un objetivo G tiene un espacio finito de búsqueda SLD con m respuestas computadas C_i $(1 \leq i \leq m)$ si y sólo si existe un APN cuya raíz es $G \rightarrow \bigvee_{i=1}^{m} C_i$.*

Al ser sistemas de inferencia, tanto POS como NEG muestran claramente cómo los cómputos en $\mathcal{PLR}(\mathcal{D})$ se corresponden con inferencias lógicas. En el siguiente apartado veremos cómo los árboles de prueba correspondientes a estos dos cálculos pueden ser utilizados para llevar a cabo la diagnosis de los errores en el programa.

2.1.6. Diagnosis mediante árboles de prueba

Cada nodo de los árboles de prueba propuestos en el apartado anterior se puede hacer corresponder a una fórmula lógica, de la que se tiene que poder determinar su validez en la interpretación pretendida del programa. Por ejemplo el nodo raíz del APP de la figura 2.1, $Y = b \wedge \ldots \rightarrow camino(b, Y)$ no es válido en la interpretación pretendida del programa del ejemplo 2.1.1, ya que no existe ningún camino desde b hasta b (el valor de Y dado por la restricción) en el grafo que pretendíamos representar.

El paso de unos nodos hijos $P_1 \ldots P_k$ a su nodo padre Q representa una inferencia en el cálculo correspondiente: los hijos son las premisas de la inferencia y el nodo padre la conclusión. Dichas inferencias pueden representarse como fórmulas lógicas de la forma:

$$P_1 \wedge \cdots \wedge P_k \rightarrow Q$$

Para la depuración estaremos interesados en encontrar *inferencias incorrectas* con respecto al modelo pretendido $\mathcal{I}$ del programa P considerado, es decir inferencias con todas sus premisas P_i válidas en $\mathcal{I}$ pero conclusión Q no válida en $\mathcal{I}$. En términos de los árboles de prueba esto corresponde a encontrar lo que, siguiendo la terminología de [63], llamaremos un *nodo crítico*:

Definición 2.1.5. Llamaremos *nodo crítico* a un nodo incorrecto con todos sus hijos correctos.

En el caso que nos ocupa la corrección de un nodo equivale a su validez en $\mathcal{I}$. Los siguientes resultados de corrección justifican el porqué de nuestro interés por localizar nodos críticos en los árboles de prueba: En el caso de POS la existencia de un nodo de este tipo equivaldrá a encontrar un error positivo en el programa P, mientras que en el caso de NEG cada nodo crítico estará asociado con un error negativo.

Teorema 2.1.3. *Resultado de Corrección en POS.*
Si asumimos un resolutor correcto, las reglas $POS_\wedge$ y POS_C son correctas con respecto a cualquier interpretación $\mathcal{I}$ de predicados sobre el dominio $\mathcal{D}$. Más aún, POS_A es también correcta con respecto a $\mathcal{I}$ si $\mathcal{I}$ es modelo de P.

Corolario 2.1.4. *Cualquier APP finito cuya raíz es un síntoma positivo contiene un nodo crítico que corresponde a la conclusión de una inferencia incorrecta obtenida mediante la regla POS_A. La cláusula utilizada en esta inferencia es una cláusula no válida en $\mathcal{I}$, correspondiente a un error positivo que revela un predicado incorrecto.*

Teorema 2.1.5. *Resultado de Corrección en NEG.*
Si asumimos un resolutor correcto, las reglas $NEG_\wedge$, NEG_C y NEG_F son correctas con respecto a cualquier interpretación $\mathcal{I}$ de predicados del dominio $\mathcal{D}$. Más aún, NEG_A es también correcta con respecto a $\mathcal{I}$ si $\mathcal{I}$ es modelo de $IS(P)$.

Corolario 2.1.6. *Cualquier APN finito cuya raíz es un síntoma negativo contiene un nodo crítico que corresponde a la conclusión de una inferencia incorrecta obtenida mediante la regla NEG_A. El axioma de $IS(P)$ utilizado en esta inferencia no es válido en $\mathcal{I}$, correspondiendo a un error negativo que revela un predicado incompleto.*

En [88] se prueba un refinamiento de estos resultados, en los que de un nodo crítico se obtiene no sólo la correspondiente cláusula incorrecta o predicado incompleto sino además la correspondiente restricción C (ver definiciones 2.1.2 y 2.1.4) que sirve para mostrar la incorrección de la fórmula lógica correspondiente.

Combinando los teoremas 2.1.1 y 2.1.3 se obtiene también un resultado ya conocido (ver por ejemplo teorema 4.1 en [46]):

Corolario 2.1.7. *Resultado de Corrección de SLD con respecto a P.*
Si C es una respuesta computada mediante resolución SLD para un objetivo G en un programa P, entonces la fórmula $C \rightarrow G$ es válida en todas las extensiones de $\mathcal{D}$ que sean modelo de P.

De los teoremas 2.1.2 y 2.1.5 se obtiene un resultado análogo para $IS(P)$:

Corolario 2.1.8. *Resultado de Corrección de SLD con respecto a $IS(P)$.*
Sea P un programa y G un objetivo para el que existe un árbol finito SLD cuyas respuestas computadas son $C_1, \ldots C_n$. Entonces la fórmula

$$G \rightarrow \exists_{-G} \bigvee_{i=1}^{n} C_i$$

es válida en todas las extensiones de $\mathcal{D}$ que sean modelo de $IS(P)$.

Como consecuencia de los lemas 2.1.4 y 2.1.6, la labor del depurador será localizar un nodo crítico correspondiente a una regla POS_A en el caso de un APP o a una regla POS_N para un APN. Para esto habrá que preguntar al usuario acerca de la validez de los nodos correspondientes a dichas reglas así como por la de sus hijos.

En el ejemplo de la figura 2.1, la raíz

$$Y = b \wedge X_1 = b \wedge Y_1 = b \wedge Z_1 = c \ \wedge \ X_2 = b \wedge Y_2 = c \wedge X_3 = b \wedge Y_3 = c \rightarrow camino(b, Y)$$

no es válida en nuestra interpretación pretendida. En efecto, la validez de esta fórmula puede ser ser reducida en este caso a la validez de

$$Y = b \rightarrow camino(b, Y)$$

ya que el resto de las variables no influyen en el lado derecho de la implicación. Y claramente no esperamos que haya un camino (de longitud mayor o igual que 1) entre b y el propio b (correspondería a un bucle en el grafo correspondiente). En cambio el único hijo del nodo raíz:

$$\begin{aligned} Y = b \wedge X_1 = b \wedge Y_1 = b \wedge Z_1 = c \ \wedge \ X_2 = b \wedge Y_2 = c \wedge X_3 = b \wedge Y_3 = c \rightarrow \\ X_1 = b \wedge Y_1 = Y \wedge arco(X_1, Z_1) \wedge camino(Y_1, Z_1) \end{aligned}$$

sí es válido en la interpretación pretendida, ya que equivale a la existencia de un arco entre b y c, arco que es parte del grafo considerado y un camino entre estos dos vértices, que obviamente también existe.

Por tanto la raíz del árbol es un nodo crítico (de hecho el único en el APP) y la cláusula asociada

$$camino(X, Y) \rightarrow arco(X, Z) \wedge camino(Y, Z)$$

puede ser señalada como una cláusula incorrecta como consecuencia de los resultados anteriores.

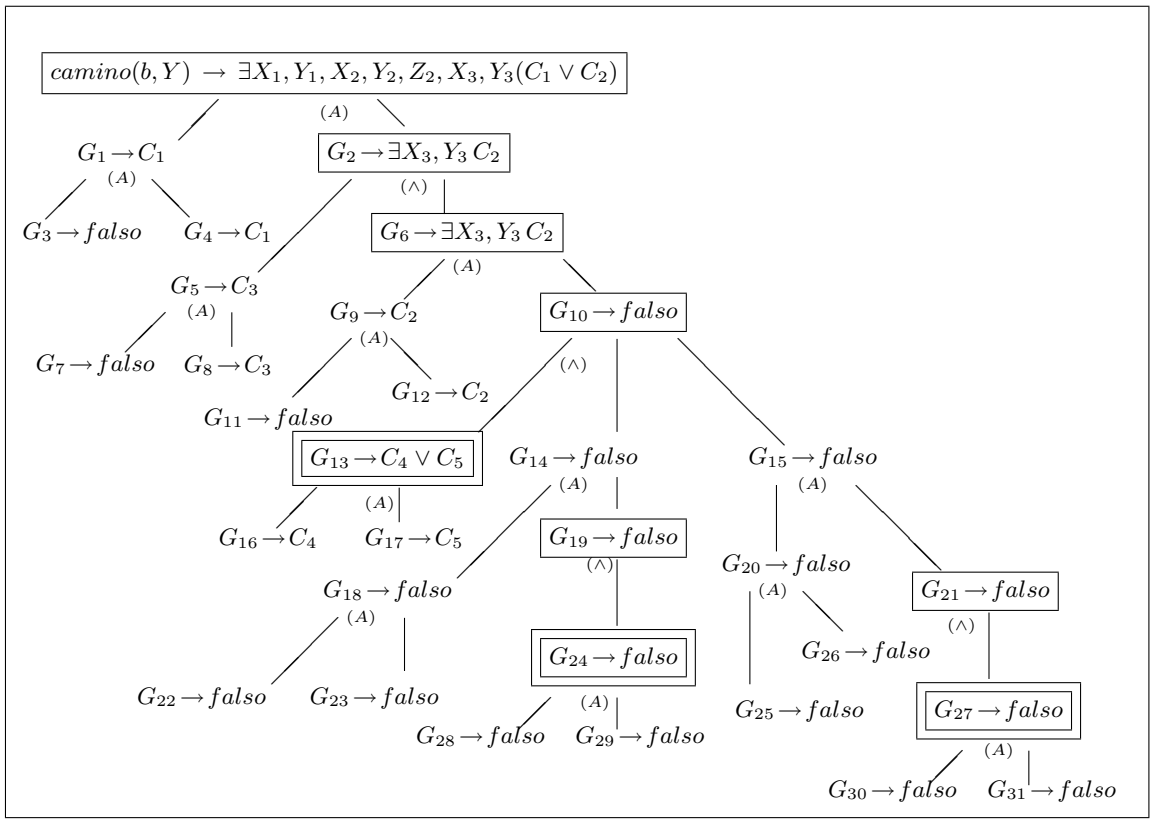


Figura 2.3: Árbol de prueba negativo

En el ejemplo del APN tenemos una mayor cantidad de nodos no válidos. Para encontrar el nodo crítico en este caso en la figura 2.3 hemos enmarcado con un rectángulo los nodos no válidos y con dos los nodos críticos. La raíz, por ejemplo, es un nodo no válido en la interpretación pretendida (de hecho es el síntoma negativo inicial), ya que al existir en nuestro grafo un camino de b a d la fórmula

$$camino(b, Y) \rightarrow \exists \ldots ((Y = c \ldots) \vee (Y = b \ldots)$$

debería incluir en la parte derecha una disyunción con una fórmula de la forma $(Y = d \ldots)$.

Análogamente se puede proceder con el resto de los nodos hasta localizar alguno de los 3 nodos críticos. Tal y como indican los resultados teóricos anteriores, dichos nodos críticos se encuentran asociados a reglas NEG_A y por tanto podemos señalar al predicado utilizado como incompleto. En nuestro caso los tres nodos señalan que el predicado *arco* es incompleto (debido a que en el programa falta la regla $arco(c, d)$). Aunque también el predicado *camino* es incompleto, esto no se detectaría con el árbol de la figura 2.3. La razón es que en este caso es la incompletitud de *arco* y no la de *camino* la que ha motivado el síntoma negativo.

2.1.7. Aspectos relacionados con la implementación

El propósito final de la depuración declarativa es el desarrollo de de herramientas prácticas que faciliten la detección de errores. Por ello los aspectos relativos a la implementación resultan de suma importancia, ya que de ellos depende finalmente que la herramienta llegue a ser útil para el programador. En este campo podemos agrupar los diferentes problemas en tres bloques:

1. Generación del árbol de prueba.
2. Estrategias para la búsqueda de nodos críticos (navegación).
3. Presentación y simplificación de las preguntas al usuario. Reducción del número de preguntas formuladas.

En relación con el árbol de prueba, en [88] se propone utilizar una simplificación del mismo que sólo incluya la raíz y los nodos que pueden ser nodos críticos, es decir las conclusiones de inferencias de POS_A para APP´s y NEG_A para APN's y muestra cómo se puede obtener dicho árbol a partir de árboles de búsqueda, tal y como se hace en el sistema *TkCalypso* desarrollado para la depuración declarativa de programas lógicos con restricciones dentro del proyecto DiSCiPl. Un árbol simplificado T' pueden obtenerse a partir de un árbol de prueba T (positivo o negativo) como sigue:

- La raíz de T' es la raíz de T.
- Dado un nodo x en T', cada hijo suyo y debe verificar:
    - y es descendiente de x en T.

- y corresponde a la conclusión de una regla POS_A o NEG_A (según estemos tratando con árboles positivos o negativos).
- Ningún nodo entre x e y corresponde a la conclusión de una regla POS_A o NEG_A.

El árbol resultado de esta simplificación verifica los siguientes resultados que garantizan la corrección de la simplificación (como probaremos para una simplificación análoga en el paradigma lógico-funcional):

1. Si el árbol original tenía algún nodo crítico, el árbol simplificado también tendrá algún nodo crítico.
2. Todo nodo crítico en el árbol simplificado es crítico en el árbol original.

Al eliminarse los nodos intermedios tipo $POS_\wedge, POS_C$ y $NEG_\wedge, NEG_C, NEG_F$, se reduce el tamaño del árbol considerado. Además las preguntas realizadas al usuario se simplifican, quedando todas ellas de la forma $G \rightarrow C \wedge p(\overline{t_n})$ en el caso positivo y $C \wedge p(\bar{t}_n) \rightarrow \exists_{-izq} \bigvee_{i=1}^{m} \bigvee_{j=1}^{m_i} C_j^i$ en el caso negativo (excepto para el caso especial de la raíz), eliminándose en particular las preguntas en las que interviene más de un átomo.

En relación con la *navegación*, se han propuesto distintas estrategias de búsqueda con el objeto de reducir el número de preguntas realizadas al usuario.

Una de la más comunes consiste en realizar un *recorrido descendente* del árbol: puesto que la raíz se supone incorrecta (es un síntoma) se trata de buscar uno de sus hijos que también lo sea. Si se encuentra tal hijo incorrecto el proceso se repite recursivamente sobre el subárbol cuya raíz es dicho nodo. El proceso acaba cuando se encuentra un nodo sin hijos incorrectos, es decir un nodo crítico.

Otro método propuesto (llamado *divide & query* en [88]) considera para cada nodo dos valores:

- El número de nodos no visitados descendientes del nodo considerado
- El número de nodos no visitados que *no* son descendientes del nodo considerado

Examinando estos valores para todos los nodos del árbol, se selecciona el nodo N para el que ambas cantidades estén más próximas. Si, después de consultar al usuario, N resulta incorrecto el proceso continúa considerando sólo el subárbol cuya raíz es N (puesto que es un árbol con raíz incorrecta debe tener un nodo crítico). En otro caso repite el proceso con el resto del árbol, es decir con el árbol inicial al que se elimina el subárbol correspondiente a N.

La situación se complica porque dada la complejidad de algunas de las preguntas el usuario no es siempre capaz de asegurar si un nodo es válido o no. Por tanto se debe permitir clasificar los nodos no sólo como *correctos* o *incorrectos* sino también como *dudosos*. En presencia de nodos dudosos deja de poder asegurarse la completitud de las estrategias utilizadas.

Un último aspecto, de vital importancia, es la presentación y simplificación de las preguntas. A pesar de que la modificación del árbol vista anteriormente reduce la complejidad de las fórmulas presentadas al usuario, éstas pueden ser aún demasiado complejas, especialmente cuando se trata con síntomas negativos. Por ello se recomienda (por ejemplo en [66, 88]) que en caso de existir tanto síntomas positivos como negativos se utilice en primer lugar el depurador de síntomas positivos. La existencia de ambos tipos de síntomas para un mismo programa no es en absoluto extraña: a menudo la existencia de una cláusula incorrecta conlleva la existencia de un predicado incompleto (como sucede en *camino*).

Por ejemplo, considerando el nodo $G_6 \rightarrow \exists X_3, Y_3 C_2$ del APN de la figura 2.2 (el nodo elegido es la conclusión de una inferencia NEG_A, por lo que estaría igualmente en el árbol simplificado) ésta podría ser una pregunta (sin simplificar) que el depurador podría hacer al usuario a partir del árbol

¿Es la fórmula

$$X_2 = b \wedge Y_2 = Y \wedge Z_2 = c \wedge camino(Y_2, Z_2) \rightarrow \exists X_3, Y_3 (Y = b \wedge X_2 = b \wedge Y_2 = b \wedge Z_2 = c \wedge X_3 = b \wedge Y_3 = c)$$

válida en el modelo pretendido?

Afortunadamente la pregunta se puede simplificar eliminando las variables no relevantes y aplicando sustituciones. Un depurador capaz de hacer tales simplificaciones podría cambiar la pregunta por:

¿Es la fórmula $camino(Y, c) \rightarrow Y = b$ válida en el modelo pretendido?

O incluso

¿Incluye el conjunto $\{Y = b\}$ todas las soluciones esperadas para el objetivo $camino(Y, c)$?

Esta última pregunta puede ser contestada con mayor facilidad; en nuestro grafo existe también un camino desde el nodo a al nodo c que no aparece recogido en el conjunto de respuestas, por lo que contestaríamos que el conjunto no contiene todas las soluciones esperadas (faltaría $Y = a$). Sin embargo tales simplificaciones no son siempre posibles. En particular en programas con restricciones la situación puede llegar a ser ciertamente compleja (ver [11] para ejemplos y propuestas en este sentido).

En relación con la reducción del número de preguntas podemos comentar tres técnicas:

- Tratar, cuando sea posible, de deducir la validez o no validez de otras preguntas anteriores. En el capítulo 5 presentaremos un algoritmo que hemos desarrollado para el caso de programas lógico funcionales con ese propósito (presentado en [19]), además de la demostración de su corrección. Un caso particular y muy simple de este método es evitar la repetición de preguntas.

- Utilizar algún representación parcial de la interpretación pretendida de (parte del) programa, por ejemplo mediante el uso de aserciones [30, 23, 43], que pueda reemplazar ocasionalmente al usuario para determinar la validez (o la no validez) de las fórmulas presentadas.

- Clasificar como *seguros* ciertos predicados, bien por ser del sistema o porque el propio usuario asegura su corrección.

2.1.8. Sistemas

A continuación indicamos algunos de los sistemas de programación lógica que incorporan un depurador declarativo.

- *TkCalypso* ([88]). Anteriormente mencionado, es un depurador declarativo para lenguajes de programación lógica con restricciones desarrollado en el $INRIA$. El interfaz gráfico del sistema permite al usuario seleccionar la estrategia a utilizar y le avisa cuando debido a la existencia de los nodos dudosos no se puede proseguir la depuración, sugiriendo un cambio de estrategia.

- *NUDE* ([66, 63]). Conjunto de herramientas de depuración para *NU-Prolog*, un lenguaje lógico que incluye características funcionales. El depurador permite localizar respuestas incorrectas como perdidas, aunque éste último caso resulta según los autores aún demasiado complejo (el usuario tiene que proporcionar las instancias de átomos no cubiertos).

- *Mercury* (*http://www.cs.mu.oz.au/research/mercury*). Mercury es un lenguaje lógico con características funcionales desarrollado en la Universidad de Melbourne. Incluye un depurador de traza y un depurador declarativo y permite 'saltar' de uno a otro durante el proceso de depuración. El depurador declarativo sirve para detectar respuestas incorrectas y perdidas. Además también se puede utilizar para detectar el origen de una excepción que se ha producido durante un cómputo pero no ha sido tratada tratada.

2.1.9. Diagnosis Abstracta

Como señalamos en la introducción la diagnosis abstracta, también llamada *diagnosis declarativa* no es un tipo de depuración declarativa sino una técnica diferente para la depuración de programas lógicos (ver por ejemplo [25]). Aunque también se trata de comparar la semántica del programa con la interpretación pretendida del mismo, en diagnosis abstracta se utilizan técnicas de interpretación abstracta [26, 27] para tratar de probar ciertas propiedades que se cumplen en el modelo pretendido y que el usuario indica, generalmente mediante el uso de aserciones. Una ventaja con respecto a la depuración declarativa es que no se precisa un síntoma inicial, ni habitualmente interacción alguna con el usuario durante

el proceso de depuración. Un caso sencillo pero ilustrativo de esta técnica son las declaraciones de tipo indicadas por el usuario y el correspondiente análisis de tipos en tiempo de compilación.

El sistema *CIAO* [23] desarrollado en la Universidad Politécnica de Madrid se propone el uso de aserciones con tres propósitos diferentes:

1. Para detectar errores en tiempo de compilación mediante diagnosis abstracta.
2. Para detectar síntomas positivos o negativos durante la ejecución de un programa sin la necesidad de la intervención del usuario.
3. Para evitar algunas de las preguntas al usuario utilizando las aserciones como oráculo cuando ello es posible.

2.2. Un Esquema para la Depuración Declarativa

Como observó Lee Naish en [63] las ideas expuestas en la sección anterior pueden abstraerse para dar lugar a un esquema general de depuración declarativa, aplicable a todo tipo de paradigmas, incluyendo paradigmas no declarativos. En los siguientes apartados exponemos las ideas básicas de este esquema.

2.2.1. Árboles de Cómputo

Citando a [63] (página 4) *"La depuración declarativa puede verse como la búsqueda de un nodo crítico en un árbol"*. Este árbol, al que llamaremos *árbol de cómputo*, debe ser una representación finita de un cómputo con resultado erróneo (el síntoma inicial). Cada nodo del árbol se corresponderá con el resultado de un subcómputo; en particular la raíz del árbol representará el resultado del cómputo principal. También es necesario suponer que el resultado asociado a cada nodo está determinado por los resultados de sus nodos hijos. Por tanto cada nodo representará en este sentido un paso de cómputo y deberá tener asociado el fragmento de código responsable de dicho paso.

El propósito del depurador será detectar fragmentos erróneos de programa a partir del árbol de cómputo. Para ello debe ser capaz de decidir cuando el resultado asociado a un nodo es erróneo (para simplificar llamaremos a estos nodos *nodos erróneos*). Esto se hará normalmente mediante preguntas a un oráculo externo, normalmente el usuario.

Nótese sin embargo que un nodo puede ser erróneo aunque su fragmento de código asociado sea correcto, debido a que su resultado puede haberse obtenido partiendo de resultados ya erróneos. Por eso, el depurador sólo detectará como fragmentos de código erróneos los correspondientes a nodos erróneos sin ningún hijo erróneo (resultado incorrecto obtenido a partir de resultados correctos), a los que llamamos en la sección anterior *nodos críticos* (ver definición 2.1.5).

Por ejemplo en la figura 2.4 vemos un árbol de cómputo genérico con nodos erróneos (los nodos 1 y 2, representados en negrita). Aunque que el resultado del nodo 1 es erróneo,

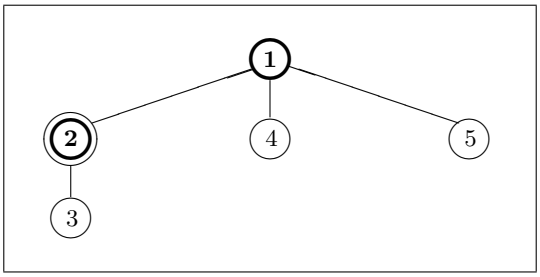


Figura 2.4: Árbol con nodos erróneos y críticos

depende del resultado del nodo 2, también erróneo, por lo que el depurador no podría señalar a 1 y a su fragmento de código asociado como la causa del error. Sí puede hacerlo en cambio con el nodo 2, que es un nodo crítico (señalado con un doble círculo en la figura).

Es importante observar que aunque no podamos asegurar que el nodo 1 corresponda con un paso de cómputo erróneo tampoco podemos asegurar lo contrario. Por tanto el método no garantiza que vayan a encontrar los nodos correspondientes a *todos* los fragmentos de código erróneos que se han utilizado en el cómputo, aunque como establece el teorema siguiente sí puede garantizar que encontrará al menos uno:

Teorema 2.2.1. *Completitud débil del esquema general.*
Todo árbol de cómputo con raíz errónea tiene al menos un nodo crítico.

La demostración de este resultado es sencilla utilizando inducción sobre la profundidad del árbol: en el caso base tendríamos un sólo nodo, la raíz, que sería un nodo crítico. En el caso inductivo examinamos de nuevo la raíz y si no tiene hijos erróneos tenemos que otra vez es el nodo crítico. Si por el contrario tiene algún hijo erróneo consideramos el subárbol del que dicho hijo es raíz, y por hipótesis de inducción contendrá un nodo crítico.

Un corolario sencillo del teorema es:

Corolario 2.2.2. *Todo árbol de cómputo con un nodo erróneo tiene al menos un nodo crítico.*

Aunque el teorema sólo nos asegura que encontraremos un nodo crítico, una vez corregida la causa del error se puede repetir el proceso de forma reiterada. En el árbol de la figura 2.4 una vez que 2 deje de ser erróneo se podrá saber si 1 lo sigue siendo, en cuyo caso pasaría a ser crítico, o no lo es en cuyo caso ya no tendríamos ningún síntoma de error para este computo en particular (lo que por supuesto no garantiza la corrección del programa).

2.2.2. Instancias del Esquema

La propuesta anterior establece una distinción clara entre el concepto de árbol y el algoritmo de depuración utilizado, distinción que no existía en los primeros depuradores

para lenguajes lógicos (que como indicamos en 2.1.1 estaban basados en metaintérpretes). Además no hace referencia a ningún lenguaje ni paradigma particular por lo que puede ser usado para definir depuradores declarativos genéricos. Diferentes tipos de errores requerirán diferentes tipos de árbol y también diferentes definiciones del concepto de nodo erróneo. Es decir constituirán distintas *instancias* del esquema general, pero en todas ellas serán aplicables las mismas ideas (nodo erróneo, paso de cómputo, etc.), así como el teorema de completitud débil 2.2.1. En cambio los resultados de corrección dependen de la instancia en particular y deben probarse en cada caso concreto.

Por ejemplo, aún sin decirlo explícitamente, en la sección 2.1 hemos definido dos instancias de este esquema, en las que los árboles utilizados correspondían, respectivamente, con los árboles de prueba en los cálculos POS y NEG y la noción de nodo erróneo con la no validez en $\mathcal{I}$. En [63] se proponen distintas instancias para respuestas incorrectas y perdidas en programación lógica, programación funcional y programación orientada a objetos.

En [16] propusimos un depurador genérico escrito en Haskell [75] y mostramos su aplicación a instancias para respuestas perdidas e incorrectas en programación lógica (representando previamente los programas lógicos mediante estructuras de datos en Haskell). El objeto de dicho trabajo no era aún escribir un depurador de uso práctico sino presentar de forma clara, mediante el sistema de tipos de Haskell, los componentes del esquema general de depuración declarativa y el concepto de instancia. En el apéndice C mostramos los detalles de la implementación de dicho depurador.

Resulta interesante comentar que depuradores declarativos actuales llevan esta diferencia conceptual al nivel práctico mediante la distinción de dos fases en el proceso de depuración (ver por ejemplo [71]). Estas dos fases son:

1. La fase de generación del árbol de cómputo.
2. El recorrido o *navegación* de dicho árbol.

Esta distinción es necesaria porque cada una de las fases plantea problemáticas y requiere técnicas diferentes, como veremos al hablar de los depuradores funcionales y al discutir nuestros resultados sobre la implementación de depuradores declarativos para programación lógico-funcional.

2.3. Programación Funcional

Vamos a repasar ahora las propuestas de depuración declarativa para el paradigma de programación funcional y en particular para los lenguajes funcionales perezosos (los más próximos a nuestro marco de programación lógico-funcional).

En el apartado siguiente veremos porqué la depuración declarativa es especialmente interesante para este tipo de lenguajes y cuáles han sido los principales trabajos que se han realizado. Posteriormente expondremos las ideas generales, tanto a nivel teórico como desde el punto de vista de la implementación y presentaremos los sistemas existentes de los que tenemos noticia. Finalmente mencionaremos, como hicimos en el paradigma lógico, otras propuestas de depuración distintas de la depuración declarativa.

2.3.1. Depuración Declarativa de Lenguajes Funcionales Perezosos

Al contrario que en el caso de la programación lógica, donde es habitual encontrar depuradores de traza integrados en los sistemas disponibles, tradicionalmente las implementaciones de lenguajes funcionales perezosos tipo Haskell [75] han prescindido de la incorporación de herramientas tales como depuradores. Según expone P. Wadler en [93] esto dificulta la difusión de este tipo de lenguajes fuera del ambiente puramente académico, dónde el desarrollo de aplicaciones reales demanda la existencia de tales herramientas.

La razón principal a la que se puede achacar esta carencia en el caso de la depuración es el alto nivel de abstracción de estos lenguajes, que dificulta enormemente el uso de herramientas tradicionales tales como los depuradores de traza utilizados en programación lógica. En particular características tales como el orden superior, el polimorfismo o la evaluación perezosa hacen la ejecución del cómputo difícilmente "observable" por el usuario. Es por ello que la depuración declarativa resulta particularmente adecuada en estos lenguajes. En cambio en el caso de los lenguajes funcionales impacientes los métodos tradicionales de depuración continúan resultando eficaces (ver por ejemplo [89]).

Es importante resaltar que a diferencia del paradigma lógico, en los trabajos de programación funcional el caso de las respuestas perdidas no es normalmente considerado. La razón es sencilla: en programación funcional no tenemos soluciones múltiples representadas como sustituciones o restricciones para un objetivo dado, sino una única respuesta (o un fallo). Por tanto si al evaluar una expresión tenemos una respuesta perdida existen dos posibilidades:

- Se ha obtenido un respuesta. Esta respuesta debe ser entonces forzosamente una respuesta incorrecta.
- El cómputo ha fallado.

En el primer caso el síntoma negativo tiene asociado de forma inmediata un síntoma positivo (la respuesta incorrecta obtenida en lugar de la respuesta perdida). En el segundo caso lo que se hace es considerar el fallo como una respuesta errónea y por tanto transformar igualmente el síntoma negativo en síntoma positivo (ver por ejemplo [70]).

En cualquier caso también la depuración declarativa de estos lenguajes plantea nuevas dificultades como se señala ya en los primeros trabajos sobre el tema (ver [69, 62], así como [70] para una versión ampliada de [69]).

En [70] H. Nilsson y P. Fritzson proponen un árbol de prueba para programación funcional perezosa basándose una técnica a la que ellos llaman *estrictificación* y que veremos en la siguiente sección. También se presenta en este trabajo un depurador declarativo llamado *LADT* para el lenguaje Freja [67], un subconjunto del lenguaje funcional perezoso Miranda [90]. En relación con esta implementación se discuten dos problemas que siguen hoy en día contándose entre los principales obstáculos para la implementación de un depurador declarativo realmente útil: la cantidad de preguntas formuladas al usuario y el enorme espacio

requerido por el árbol generado. Para el segundo problema los autores proponen una técnica consistente en producir sólo parte del árbol. Si el error no puede ser localizado en este fragmento, entonces el programa es recomputado produciendo otro fragmento de árbol, y así sucesivamente.

En [85] y en [71] H.Nilsson y J.Sparud establecen el árbol de cómputo que se utilizará frecuentemente en trabajos posteriores (por ejemplo en [65, 78, 84, 68]): el EDT (de *Evaluation Dependence Tree*). Además estos autores proponen dos técnicas para la obtención de dicho árbol: la primera mediante una transformación de la máquina abstracta, la segunda mediante una transformación de programas de forma que el programa transformado produzca como resultado el EDT, señalando además las ventajas (eficiencia para la transformación de la máquina abstracta, portabilidad en el caso de la transformación de programas) que se obtienen en cada caso.

Al contrario que en el caso de programación lógica, ninguno de estos trabajos relaciona los árboles de cómputo con un sistema de inferencias lógicas: en [71] se propone su especificación mediante semántica denotacional, mientras que en el resto de trabajos se da una descripción verbal. En nuestros trabajos [17, 19] (que presentaremos en el capítulo 4) obtenemos un árbol equivalente al EDT a partir de un cálculo semántico para programación lógico-funcional y, gracias a esta relación entre la semántica y el árbol de prueba, podemos establecer la corrección del método. También establecemos la corrección de la transformación de programas demostrando la relación entre el programa original y el programa transformado. Dado que nuestro marco abarca tanto la programación lógica como la funcional los resultados obtenidos representan los únicos resultados de corrección que conocemos para el campo de la depuración declarativa de lenguajes funcionales.

En estos trabajos (en particular en [19]) también resolvemos un problema relacionado con la aplicación de la transformación de programas a funciones currificadas (ver [65]) que hacía hasta entonces inviable dicha transformación para programas reales. En [76] se utiliza nuestra propuesta como punto de partida para presentar una alternativa que reduce el tamaño del programa transformado y se define con precisión una transformación de programas basada en combinadores monádicos para Haskell'98.

En la siguiente subsección describiremos informalmente la estructura de un EDT, utilizando la sintaxis de Haskell [75].

2.3.2. El EDT

Como hemos dicho más arriba, el EDT [71] es el árbol de cómputo utilizado comúnmente en programación funcional para la depuración de respuestas incorrectas. Los nodos de este árbol son de la forma $f\ t_1 \ldots t_n \rightarrow t$ donde f corresponde a un símbolo de función del programa y $t_1, \ldots t_n, t$ son términos. Una excepción es la raíz del árbol que es de la forma $expr \rightarrow t$ donde $expr$ es la expresión inicial, aunque a menudo se evita esta excepción asumiendo la existencia de una regla de programa de la forma $main = expr$.

Un nodo $f\ t_1 \ldots t_n \rightarrow t$ indica que t ha sido el resultado producido por una llamada de la forma $f\ t_1 \ldots t_n$ durante el cómputo examinado. Tanto los parámetros t_i como t

deben representarse evaluados hasta donde haya demandado el cómputo completo, y si al acabar el cómputo quedan todavía llamadas sin evaluar, éstas se reemplazarán por un valor especial $\perp$ que representa la falta de información acerca de los resultados de dichas llamadas. De esta forma los valores $t_1 \ldots t_n, t$ nunca incluirán aplicaciones evaluables, lo que redundará en preguntas más sencillas al usuario. Este idea corresponde al proceso de estrictificación propuesto en [70]. A los valores de la forma $f\ t_1 \ldots t_n \rightarrow t$ con t_i, t de la forma indicada los llamaremos *hechos básicos*, siguiendo la nomenclatura que seguimos en [17] para el caso de la programación lógico-funcional.

El modelo pretendido será por tanto un conjunto de hechos básicos y cada pregunta al usuario sobre la validez de un nodo del árbol equivaldrá a preguntar acerca de la pertenencia al modelo pretendido del hecho básico contenido en dicho nodo.

En cuanto a la estructura del árbol, los hijos de cada nodo corresponden las llamadas necesarias para su evaluación. Precisando más: Dado un nodo $f\ t_1 \ldots t_n \rightarrow t$, para determinar sus hijos en él árbol de cómputo se considera la instancia de regla de f utilizada en ese paso, digamos $f\ s_1 \ldots s_n = r$. Entonces los hijos de este nodo se corresponden con las llamadas a funciones que han tenido lugar durante la evaluación r. Aunque aunque el orden de los hijos no es relevante, se suele hacer corresponder la aplicación más interna a la izquierda con el hijo más a la izquierda y así sucesivamente.

Por ejemplo, consideramos el siguiente programa (erróneo):

Ejemplo 2.3.1.

```
from   :: Int -> [Int]
from n = n : from (n+1)


take              :: Int -> [a] -> [a]
take 0 xs           = []
take (n+1) []            = []
take (n+1) (x : xs)  = x : x : take n xs

% La última regla de take es errónea; debería ser:
% take (n+1) (x : xs)  --> x : take n xs

main = take 2 (from 0)
```

En este programa la expresión *main* se evalúa a $[0, 0, 1, 1]$, lo que constituye una respuesta incorrecta (es decir un síntoma positivo). El árbol para el cómputo $main \rightarrow [0, 0, 1, 1]$ se encuentra en la figura 2.5. La raíz del árbol corresponde a la evaluación del a expresión inicial (*main*) con su resultado. Representamos cada nombre de función con un subíndice

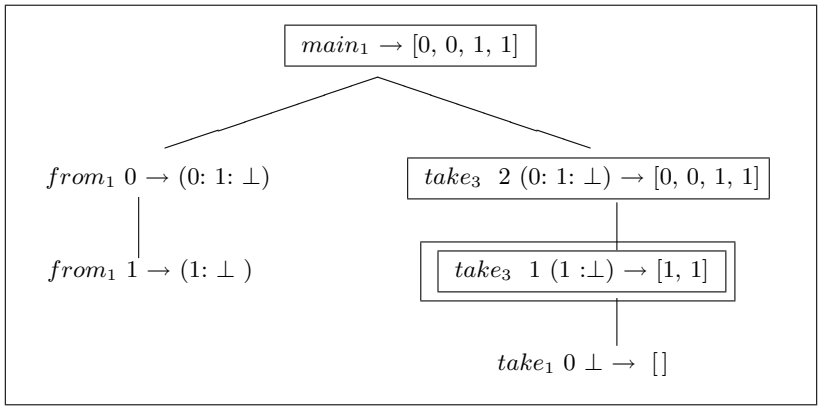


Figura 2.5: Árbol de prueba para el ejemplo 2.3.1

para identificar la regla concreta utilizada en el paso de cómputo (1 para la primera regla de la función que aparece en el programa, y así sucesivamente).

Para determinar los hijos de *main* examinamos su parte derecha, *take* 2 ($from$ 0), en la que vemos dos llamadas de funciones. La llamada más interna es $from$ 0 que corresponderá al primer hijo por la izquierda. Dado que sólo los dos primeros valores de la lista producida por $from$ 0 han sido necesarios durante el cómputo tenemos que el nodo debe ser $from$ 0 $\rightarrow$ (0 : 1 : $\perp$). De manera análoga obtenemos su único hijo, $from$ 1 $\rightarrow$ (1 : $\perp$).

El segundo hijo de la raíz corresponde a la llamada *take*, que tiene dos argumentos, 0 y $from$ 0. Tal y como indicamos más arriba, la llamada $from$ 0 debe ser sustituida por su resultado, es decir (0 : 1 : $\perp$), por lo que tenemos la parte izquierda del hecho básico, *take* 0 (0 : 1 : $\perp$). La parte derecha es el resultado es esta llamada (que coincide con la de *main*), $[0, 0, 1, 1]$, obtenida utilizando la tercera regla de *take*. De forma similar se obtienen los dos nodos restantes.

Hemos señalado en el árbol con un recuadro los nodos incorrectos, y con un doble recuadro el único nodo crítico que señala la tercera regla de *take* como la causa del error. Es importante señalar que durante la fase de depuración el usuario será consultado sobre la validez de hechos básicos que pueden incluir apariciones del símbolo $\perp$, que debe saber interpretar adecuadamente. Por ejemplo la pregunta sobre la validez del hecho básico

$$from\ 0 \rightarrow (0 : 1 : \perp)$$

debe entenderse como ¿Son 0 y 1 los dos primeros valores de la lista producida por $from$ 0? O, con otras palabras, ¿Es el valor producido por $from$ 0 de la forma $(0, 1 : A)$ para algún A? Por tanto la lectura de $\perp$ en la parte derecha de un hecho básico se corresponde con la que se tendría al sustituir el símbolo por una variable cuantificada existencialmente (tal y como se hace explícito en el depurador propuesto en [64]). En cambio la aparición del mismo símbolo $\perp$ en la parte izquierda de un hecho básico debe pensarse como si en su

lugar existiera una variable universal. Por eso la pregunta acerca de la validez de

$$take\ 1\,(1 : \bot) \rightarrow [1, 1]$$

deberá obtener una respuesta negativa; no es cierto que para cualquier lista B se cumpla $take\ 1\,(1 : B) \rightarrow [1, 1]$, aunque sí lo sea para algún caso particular como $B = [1]$.

2.3.3. Sistemas

Vamos a presentar brevemente las dos herramientas basadas en la técnica de depuración declarativa, *Freja* [68] de H. Nilsson y *Buddha* [77, 76, 78] de Bernard Pope, que conocemos como disponibles actualmente.

Freja es un depurador para un subconjunto de Haskell (disponible en la dirección `http://www.ida.liu.se/ henni/`, aunque sólo en versión para SPARC/Solaris) en el que el EDT se obtiene mediante una modificación de la máquina abstracta. En [68] se muestran los detalles de implementación, así como medidas de eficiencia. Para reducir el coste en tiempo y memoria del EDT, el sistema sólo obtiene cada vez una parte del árbol, recomputando el objetivo cuando es necesario para obtener la siguiente porción de EDT. De esta forma se limita el tamaño máximo de memoria requerido por el depurador y se evita uno de los principales inconvenientes de la depuración declarativa, por lo que el depurador es capaz de tratar con programas de tamaño real.

Por ejemplo, si consideramos el siguiente programa erróneo:

Ejemplo 2.3.2.

```
insert:: Int->[Int]->[Int]
insert x [] = [x]
insert x (y:ys) | y > x   =  y: (insert x ys)
        | x < y   = x : (y : ys)
        | otherwise = y : ys

sort::[Int] -> [Int]
sort []  = []
sort (x:xs) = insert x (sort xs)

main = print (sort [1,2,3])
```

Esta es la sesión de depuración en $Freja$:

```
sort [2,1,3]  => [3,1] ?
> no
sort [1,3] => [3,1] ?
> no
sort [3] => [3] ?
> yes
```

```
insert 1 [3] => [3,1] ?
> no
insert 1 [] => [1] ?
> yes
Bug located in function "insert".
Erroneous reduction: insert 1 [3] => [3,1]
```

La navegación consiste en un recorrido descendente del árbol, buscando siempre un hijo erróneo del último nodo erróneo para proseguir.

El sistema Buddha (disponible en `http://www.cs.mu.oz.au/ bjbop/buddha`) es un depurador para Haskell (v 5.04) basado en la transformación de programas. La transformación se hace de fuente a fuente, es decir, dado un programa en Haskell la transformación produce otro programa en Haskell. Este depurador admite la mayor parte de las características de Haskell; en particular el sistema admite la inclusión de operaciones tipo $IO()$ que no influyen en la secuencia de preguntas al usuario. Si el programa contiene alguna operación de lectura, ésta se repetirá durante la sesión de depuración, por lo que el usuario debe dar la misma respuesta para que el cómputo analizado no varíe. El interfaz con el usuario aún está en fase de desarrollo, por lo que éste tiene que ejecutar varios comandos manualmente para transformar el programa y comenzar la sesión de depuración.

Esta es la sesión de depuración en Buddha para el ejemplo anterior 2.3.2:

```
sort
 (: 1 (: 2 (: 3 [])))
 => (: 3 (: 2 (: 1 [])))

(T)rue, (F)alse, (Q)uit : F

insert
 1
 (: 3 (: 2 []))
 => (: 3 (: 2 (: 1 [])))

(T)rue, (F)alse, (Q)uit : F

insert
 1
 (: 2 [])
 => (: 2 (: 1 []))

(T)rue, (F)alse, (Q)uit : F

insert
 1
```

```
 []
 => (: 1 [])

(T)rue, (F)alse, (Q)uit : T

The erroneous definition is: insert
At source location: line = 2, col = 1
In module: Main
In file: Main.hs
```

La principal diferencia con respecto a la sesión de Freja es el orden en el que se visitan los nodos del árbol: en Buddha se comienza por los nodos correspondientes a las llamadas más externas en lugar de por las más internas como se hacía en Freja. En el apéndice B se puede encontrar la sesión análoga para el caso de $\mathcal{TOY}$ (ejemplo B.2.1, pág. 279).

2.3.4. Otras Propuestas

Entro los depuradores disponibles para lenguajes funcionales cabe mencionar dos que no utilizan depuración declarativa: *Hat* [86] y *Hood* [33]. En [22] se hace una comparativa entre dos depuradores y el depurador declarativo Freja.

Para entender el comportamiento de estos dos depuradores podemos considerar el siguiente ejemplo, tomado de [22]:

Ejemplo 2.3.3.

```
main = let xs = [4*2,3+6] :: [Int]
       in (head xs, last xs)

head (x:xs) = x

last (x:xs) = last xs
last [x]    = x
```

El error está en las dos reglas de *last* que deberían aparecer en orden contrario (desde un punto de vista declarativo el error está en la primera regla que permite inferir hechos básicos como $last[1] \rightarrow error$ que no están en la interpretación pretendida del programa). Como consecuencia de este error, al evaluar la expresión *main* se obtiene un fallo cuando llega a aplicarse la función *last* a la lista vacía [ ], para la que no está definida.

Hat es una versión modificada del compilador para Haskell 98 de la universidad de York (disponible en `http://www.cs.york.ac.uk/fp/ART`), que almacena durante un cómputo la secuencia o traza de reducciones (*redexes*), realizadas. En caso de una terminación anormal del programa, o si el resultado no es el esperado, el usuario puede examinar la secuencia de reducciones, comenzando por la última y procediendo hacia atrás. Además el navegador

permite ver cual es la regla asociada a cada paso y sobre qué subexpresión se ha aplicado. En el ejemplo anterior, la secuencia sería de la forma:

$$
\begin{aligned}
&\mathit{last}\ [\,]\\
&\mathit{last}\ (3+6:[])\\
&\mathit{last}\ (\underline{8}:3+6:[])\\
&4*2\\
&\mathit{main}
\end{aligned}
$$

y ya en el primer paso deberíamos darnos cuenta de que $\mathit{last}\ (3+6:[])$ no debería haberse reducido a $\mathit{last}\ [\,]$ sino a $3+6$ y que por tanto debe haber un error en su definición.

El otro depurador disponible, Hood, es una biblioteca de Haskell que permite al usuario señalar los puntos de un programa que quiere observar. Por ejemplo, para observar el comportamiento de la función *last* el programa 2.3.3 podría reescribirse:

```
main = let xs = [4*2,3+6] :: [Int]
       in (head xs, last xs)

head (x:xs) = x

last = observe "last" last'

last' (x:xs) = last xs
last' [x]    = x
```

El primer argumento de *observe* es la etiqueta que sea asociará al punto de observación. Desde un punto de vista declarativo, *observe* comporta como la identidad sobre su segundo argumento, por lo que puede introducirse sin alterar el comportamiento del programa. En este caso obtenemos la respuesta:

```
-- last
{ \ (_ : _ : []) -> throw <Exception>
, \ (_ : []) -> throw <Exception>
, \ [] -> throw <Exception>
}
```

El símbolo _ representa una expresión no evaluada. Nótese que los argumentos no se representan con el valor que alcanzan al final del cómputo, como en los depuradores declarativos, sino con los valores que ha necesitado evaluar la función inspeccionada.

En relación con estas técnicas, en [72] se propone establecer una separación clara entre el programa a depurar y el propio depurador. De esta forma el grafo asociado al cómputo

no necesita ser modificado durante la depuración, y así se asegura la corrección del método. Para lograr este objetivo los autores presentan un evaluador monádico que genera la traza deseada mediante una mónada de estado. Como ejemplos se presentan mónadas para simular las trazas de Freja (es decir el la construcción del EDT), Hood y Hat.

2.4. Programación Lógico-Funcional

Tal y como se indicó en la introducción (apartado 1.1.2), los argumentos de P. Wadler en [93] acerca de la necesidad de integrar herramientas tales como depuradores en los entornos de desarrollo para lenguajes funcionales son perfectamente aplicables también a los lenguajes lógico-funcionales (ver [38] para una introducción a este paradigma).

Esta necesidad, combinada con la ausencia de tales herramientas (tanto de sus fundamentos teóricos como de implementaciones disponibles) es el origen de esta tesis, y hace que este apartado sirva principalmente como introducción informal a los capítulos posteriores. Los ejemplos que utilizaremos, escritos en el lenguaje $\mathcal{TOY}$ [54, 1], también nos servirán para introducir de manera informal las características de este tipo de lenguajes.

2.4.1. Depuración Declarativa de Lenguajes Lógico-Funcionales

Son Naish y Barbour quienes en [64] se plantean por primera vez la depuración de lenguajes que combinan propiedades de los paradigmas lógico y funcional. El lenguaje elegido es una extensión del sistema lógico NU-Prolog descrita en [61] y las ideas propuestas se basan en gran medida en la transformación de las características funcionales del programa al paradigma lógico. Los árboles de prueba considerados están formados por nodos 'lógicos' y nodos 'funcionales' y el depurador trata cada nodo de manera acorde con el paradigma al que pertenece. En esta presentación la programación lógico funcional (a la que llamaremos PLF en ocasiones para abreviar) no se presenta como un nuevo paradigma con su marco teórico propio, sino como una combinación de características de los paradigmas lógico y funcional, y esta idea se extiende también a la depuración declarativa en este entorno.

En diferentes trabajos como [35] y [36] se desarrolla una marco teórico para PLF que constituye la base del lenguaje lógico-funcional $\mathcal{TOY}$ [54, 1] (basado parcialmente en su antecesor $BABEL$ [60]). En estos artículos se presenta un cálculo de reescritura condicional (llamado $CRWL$) que define una semántica adecuada para lenguajes lógico-funcionales. En otros trabajos como [7, 12, 13, 14, 15, 40] se muestra como en el paradigma PLF surgen técnicas de programación y optimizaciones específicas, diferentes a las de la programación lógica y la programación funcional.

Basándonos en el cálculo semántico $CRWL$, en [17] definimos unos árboles de prueba adecuados para la depuración declarativa de respuestas incorrectas dentro del paradigma PLF. Estos árboles corresponden, al igual que sucedía en la programación lógica, con inferencias lógicas en el cálculo semántico $CRWL$, y eso nos permite probar la corrección del

método, tal y como se expondrá en los capítulos 3 y 4. Como hemos indicado en 2.3.1 estos resultado son también relevantes para el paradigma funcional.

En [19] ampliamos los resultados de [17], definiendo con precisión una transformación de programas que permite llevar estos resultados a la práctica para el lenguaje $\mathcal{TOY}$. Además probamos primero que el programa transformado es correcto desde el punto de vista de los tipos y a continuación que el árbol de prueba que obtenemos corresponde efectivamente al árbol de prueba definido en [17]. Estos resultados se expondrán en los capítulos 3 y 4.

También proponemos en este trabajo un método para disminuir el número de preguntas realizadas al oráculo. La idea consiste en deducir, si es posible, la validez (o no validez) de un hecho básico a partir de la validez (o no validez) de otros nodos ya visitados. Para llevar a la práctica este método presentamos un algoritmo, que veremos en el capítulo 6, y probamos su corrección.

En cuanto a sistemas existentes, los únicos que conocemos en este paradigma son el que hemos incorporado al sistema $\mathcal{TOY}$ y que se encuentra disponible en

`http://babel.dacya.ucm.es/toy`

y el que hemos incorporado al compilador de Curry [39] sobre la versión desarrollada en la universidad de Münster por Wolfgang Lux y disponible en

`http://danae.uni-muenster.de/~lux/curry`

Ambos sistemas sirven para la depuración declarativa de respuestas incorrectas y no tratan el caso de las respuestas perdidas. En el capítulo 5 se presentan los detalles acerca de su implementación y su eficiencia.

En [18] discutimos la depuración declarativa de cómputos en los que se utiliza la *búsqueda encapsulada* (descrita en [42]), un método para encapsular cómputos no deterministas utilizado ampliamente en el lenguaje Curry [39] y basado en la primitiva no declarativa *try*.

2.4.2. Respuestas Incorrectas en PLF

El programa del ejemplo 2.4.1 utiliza la técnica conocida como "generación y prueba" para ordenar una lista de números enteros.

Ejemplo 2.4.1.

```
(//) :: A -> A -> A
X // Y = X
X // Y = Y

permSort :: [int] -> [int]
permSort Xs = if (isSorted Y) then Y
                          where Y = permute Xs
```

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

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

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

La sintaxis del programa es similar a la de Haskell con la diferencia de que las variables se escriben en mayúsculas y se utiliza la notación para listas de Prolog (aunque también se puede utilizar la de Haskell, es decir $(Y : Ys)$ en lugar de $[Y|Ys]$).

La idea es ir generando permutaciones de la lista inicial hasta encontrar una que ya tenga los elementos ordenados. Para lograr esto se combina la función *permute*, que genera todas las permutaciones de una lista dada, con la función *isSorted*, que comprueba si una lista de enteros está ordenada. La tercera regla de *isSorted* está definida usando una condición indicando que una lista $[X, Y|Zs]$ con al menos dos elementos está ordenada cuando se verifica que $X <= Y$ y que $[Y|Zs]$ es a su vez una lista ordenada.

Aunque el método de ordenación no es desde luego eficiente, su comportamiento no es tan malo como puede pensarse en principio. En efecto, cada permutación no ordenada es descartada tan pronto como se detectan los dos primeros elementos consecutivos no ordenados, sin que sea necesario generar la permutación entera gracias a la pereza.

La función *permute* tiene dos reglas: la primera indica que la única permutación de una lista vacía es ella misma, y la segunda que para generar todas las permutaciones de una lista no vacía $[X|Xs]$ hay que insertar X de todas las formas posibles en cada permutación de Xs.

La función *insert* se encarga de esta inserción no determinista de un elemento en una lista. Su segunda regla (la primera es de nuevo para el caso trivial de la lista vacía) utiliza el operador infijo // que expresa la elección no determinista entre dos posibilidades. Durante el cómputo el sistema intentará primero la primera regla, pero en caso de que ésta no baste para resolver el objetivo inicial el mecanismo de *backtracking* permitirá utilizar también la segunda regla. Lo mismo sucederá si el sistema logra resolver el objetivo inicial con la primera regla pero el usuario demanda más respuestas. La posibilidad de incluir funciones no deterministas es una de las características de estos lenguajes que los diferencia de los lenguajes funcionales.

En el caso de *insert* la primera posibilidad es que el elemento a insertar se incluya en primera posición y la segunda que se inserte en cualquier posición del resto de la lista. Es en esta segunda posibilidad donde hay un error ya que se nos ha olvidado incluir la cabeza de la lista (representada por la variable Y) como cabeza de la lista resultante. La regla

correcta debería ser:

```
insert X [Y|Ys] =  [X,Y|Ys] //  [Y | insert X Ys]
```

Un ejemplo de objetivo para este ejemplo podría ser

```
TOY>  permSort [3,2,1] == R
```

que tendrá éxito si existe alguna sustitución de R por un término que haga cierta la igualdad. Llamaremos *respuestas* a cada una de estas sustituciones. En este caso obtenemos la respuesta incorrecta

```
R ==  [3]
```

En el capítulo 3 (pág. 46) veremos como es posible definir un árbol (al que llamaremos APA) análogo al EDT de la programación funcional pero para el caso lógico-funcional. A diferencia del caso del EDT el APA se extraerá de un árbol de prueba positivo para el cómputo. En este árbol la relación entre cada nodo y sus hijos se corresponderá con la aplicación de una regla del cálculo semántico que habremos presentado previamente, en un proceso análogo al que hemos visto para la programación lógica en la sección 2.1. Gracias a esta relación entre el APA y el correspondiente cálculo podremos demostrar la corrección del método, probando que para un objetivo con una respuesta incorrecta el correspondiente APA tiene con seguridad un nodo crítico, y que ese nodo tiene asociada una regla de programa incorrecta. Además de dicho nodo crítico se obtendrá la instancia de regla utilizada durante el cómputo, lo que contribuirá a detectar el origen del error. Para nuestro ejemplo el árbol obtenido puede verse en la figura 2.6.

En este árbol hemos señalado con un recuadro los nodos no válidos y con un doble recuadro el único nodo crítico, que señala a la segunda regla de *insert* como la causante del síntoma inicial. Resulta interesante notar la aparición en una de las hojas del árbol del símbolo $\perp$, que corresponde, al igual que sucedía en el caso de la programación funcional, con la evaluación de una llamada a función que no ha sido necesaria durante el cómputo, en este caso la de $insert_1\ 2\ [\,]$, y que por tanto no aparece en el árbol.

En el capítulo 4 indicaremos cómo se puede obtener este árbol utilizando la técnica de transformación de programas y probaremos la corrección de dicha transformación, mientras que en la segunda parte de la tesis discutiremos cómo programar esta transformación en los sistemas $\mathcal{TOY}$ y Curry. El resultado de este trabajo es la incorporación del depurador de respuestas incorrectas para ambos compiladores. La sesión de depuración para el ejemplo 2.4.1 en el sistema $\mathcal{TOY}$ se puede encontrar en el ejemplo B.3.1 del apéndice B, página 293, siendo la sesión en Curry idéntica (salvo cambios triviales de sintaxis). En este apéndice se puede encontrar una colección de ejemplos extraídos de diversos trabajos sobre depuración declarativa adaptados a la sintaxis de $\mathcal{TOY}$.

2.4.3. Respuestas Perdidas en PLF

Vimos en 2.3.1 que dentro del paradigma funcional las respuestas perdidas podían reducirse al caso de las respuestas incorrectas. Sin embargo en los lenguajes lógico-funcionales, y

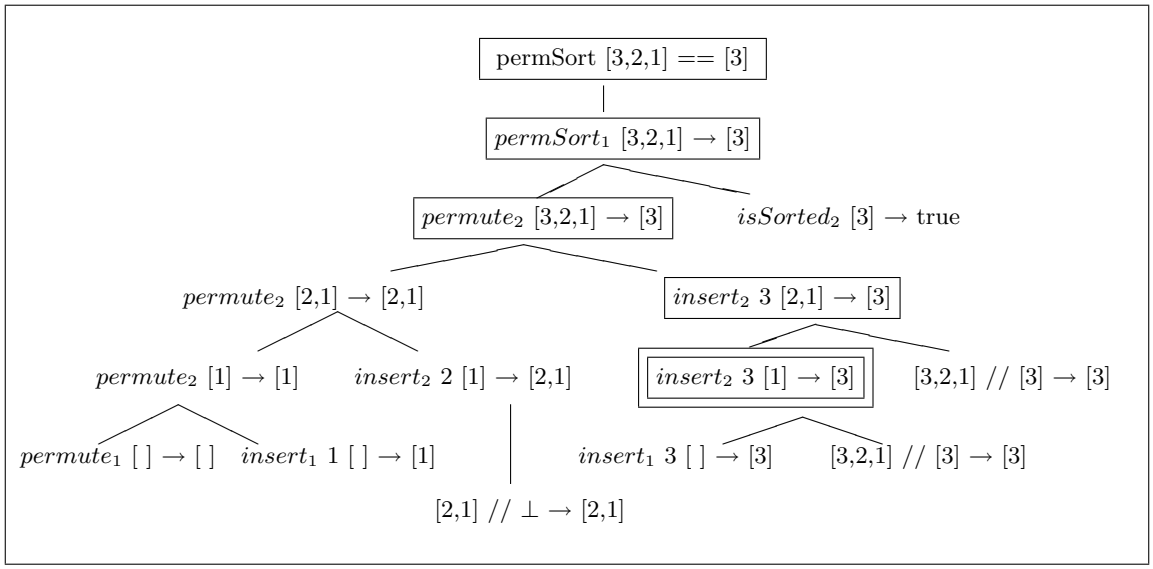


Figura 2.6: Árbol de prueba para el ejemplo 2.4.1

debido a la existencia de cómputos indeterministas, vuelven a encontrarse los dos síntomas de error que aparecían en el caso de la programación lógica: síntomas positivos (respuestas incorrectas) y síntomas negativos (respuestas perdidas).

Por ejemplo consideramos el siguiente programa en $\mathcal{TOY}$:

Ejemplo 2.4.2.

```
merge :: [A] -> [A] -> [A]
merge Xs     []     = Xs
merge [X|Xs] [Y|Ys] = [X|merge Xs [Y|Ys]]
merge [X|Xs] [Y|Ys] = [Y|merge [X|Xs] Ys]
```

La función no determinista *merge* pretende mezclar dos listas de todas las formas posibles pero respetando el orden interno de cada una de ellas. Por ejemplo *merge* $[1,2]$ $[3,4] \rightarrow [3,1,4,2]$ y *merge* $[1,2]$ $[3,4] \rightarrow [1,2,3,4]$ son válidos en la interpretación pretendida mientras que *merge* $[1,2]$ $[3,4] \rightarrow [3,2,4,1]$ no lo es ya que 2 aparece delante de 1 en el resultado y debería respetar el orden seguido en el argumento $[1,2]$.

Sin embargo, el programa del ejemplo 2.4.2 contiene un error, ya que se nos ha olvidado incluir una regla para el caso de que la primera lista sea vacía, es decir una regla de la forma:

```
merge [] Ys = Ys
```

Un ejemplo de objetivo para este programa podría ser

```
merge [1,2]  [3] == R
```

El objetivo tendrá solución para aquellas sustituciones de la variable R por términos que hagan cierta la igualdad. En este ejemplo, y debido a que el programa es erróneo, el conjunto de respuestas es incompleto, tal y como muestra la siguiente sesión en $\mathcal{TOY}$:

```
TOY> merge [1,2] [3] == R
      yes
      R ==  [ 1, 3, 2 ]

more solutions (y/n/d) [y]? y
      yes
      R ==  [ 3, 1, 2 ]

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

En primer lugar el sistema resuelve el objetivo encontrando la respuesta dada por la sustitución $R \mapsto [1,3,2]$ (que se muestra en $\mathcal{TOY}$ como $R = [1,3,2]$). A continuación, y a requerimiento del usuario se encuentra otra respuesta más, $R \mapsto [3,1,2]$, pero cuando el usuario pide una tercera respuesta el sistema ya no es capaz de encontrarla. Las dos respuestas obtenidas son correctas, pero el conjunto formado por ambas en un conjunto de respuestas incompleto para el objetivo inicial, ya que se esperaba que también se obtuviera una tercera respuesta $R \mapsto [1,2,3]$ que es por tanto una respuesta perdida.

Veamos otro ejemplo en esta misma línea, pero utilizando restricciones aritméticas, otra de las características del sistema $\mathcal{TOY}$.

Ejemplo 2.4.3.

```
interval     :: int -> int -> [int]
interval X Y = []                    <== X > Y
interval X Y = X : interval (X+1) Y <== X < Y

% La segunda regla es errónea. Debería ser:
% interval X Y = X : interval (X+1) Y <== X <= Y

member :: A -> [A] -> bool
member X (Y : Ys) = true           <== X == Y
member X (Y : Ys) = member X Ys
```

En este caso la función *interval* tiene como argumentos dos números enteros X e Y y debe generar la lista de todos los números enteros comprendidos entre ambos, es decir la lista $[X, X+1, \ldots Y]$. En el caso de que sea $X > Y$ la lista devuelta debe ser la lista vacía. Para definir esta función se han utilizado reglas con condiciones: la primera regla sólo se utilizará si su condición $X > Y$ se verifica, y la segunda sólo si se cumple $X < Y$. Sin

embargo la definición de esta función no es correcta ya que el caso $X = Y$ no queda cubierto por ninguna de las dos reglas. La otra función parte del programa es *member* y sirve para saber cuando un elemento es parte de una lista.

Combinando ambas funciones podemos probar un objetivo como

```
TOY> Y>0, member X (interval Y 3)
```

En este caso tenemos un objetivo compuesto por dos objetivos atómicos: la restricción aritmética $Y > 0$ y *member* X (*interval* Y 3) que $\mathcal{TOY}$ interpreta implícitamente como *member* X (*interval* Y 3) == *true*. El objetivo tendrá éxito para los valores X e Y tales que Y verifica la restricción $Y > 0$ y X pertenece al intervalo de enteros $[Y, 3]$. Utilizando el resolutor para operaciones aritméticas sobre números reales el sistema encuentra dos respuestas:

```
TOY> Y>0, member X (interval Y 3)
     yes
      Y == X
            { Y>0.0 }
            { Y<3.0 }

more solutions (y/n/d) [y]? y
      yes
      Y== -1.0+X
            { X>1.0 }
            { X<3.0 }
```

que escritas utilizando una notación análoga a la del apartado 2.1.2 serían

$$Y = X \wedge Y > 0 \wedge Y < 3 \quad \vee \quad Y = X - 1 \wedge X > 1 \wedge X < 3$$

En este caso también se tendría una respuesta perdida puesto que por ejemplo $X = 3 \wedge Y = 3$ es una respuesta esperada para el objetivo pero no está cubierta por las dos respuestas obtenidas.

Tal como indicamos al comienzo de esta sección, hasta el presente no existe ningún marco adecuado para la depuración declarativa de respuestas perdidas para PLF, ni para PLF con restricciones. En esta tesis nos ocupamos, en cambio, de la depuración declarativa de respuestas incorrectas en PLF, tanto desde el punto de vista de los fundamentos teóricos como de la implementación práctica.

2.4.4. Otras Propuestas

El entorno de desarrollo gráfico *CIDER* presentado en [41] incluye un depurador gráfico de trazas para el lenguaje Curry. El depurador muestra el proceso de evaluación de una expresión paso a paso. La expresión se muestra como una árbol dónde la subexpresión que va a ser reducida en el siguiente paso aparece marcada en rojo. El usuario tiene la

posibilidad de fijar puntos de parada en el código para 'saltar' aquellas partes del programa que no está interesado en inspeccionar.

Al igual que en el caso de la programación lógica, también en el paradigma lógico-funcional se ha empleado el método de diagnosis abstracta. En [3, 4] se sigue esta idea, pero incorporando además un mecanismo basado en técnicas de desplegado para tratar de corregir automáticamente el programa. Como es habitual en este tipo de depuradores no se requiere un síntoma de incorrección inicial, aunque sí una especificación (parcial) ejecutable del modelo pretendido. Los autores presentan además un prototipo de nombre *BUGGY* disponible en

`http://www.dsic.upv.es/users/elp/soft.html`

Capítulo 3

Árboles de Prueba Positivos para Programación Lógico-Funcional

En el capítulo anterior hemos repasado los conceptos básicos de la depuración declarativa, así como su aplicación a diferentes paradigmas. En este vamos a comenzar a aplicar estas ideas a la depuración de respuestas incorrectas en lenguajes lógico-funcionales perezosos. Tanto el presente capítulo como el siguiente constituyen el núcleo teórico de la tesis, encargándose cada uno de ellos de uno de los dos primeros objetivos descritos en la sección 1.2 (pág. 4) Los contenidos básicos de estos capítulos se encuentran en los trabajos [17, 19].

Ya en la sección 2.4.2 del capítulo anterior esbozamos algunas ideas acerca de los árboles de prueba que podríamos utilizar para la depuración de respuestas incorrectas en programación lógico-funcional. Sin embargo en ese punto no éramos aún capaces de definir esas ideas con precisión ni de probar formalmente su corrección. Para lograr este objetivo necesitamos en primer lugar introducir los conceptos teóricos básicos del paradigma lógico-funcional, lo que haremos en la siguiente sección. Avanzando en esta dirección, la sección 3.2 definirá la semántica de los programas PLF mediante un cálculo semántico SC. Las deducciones en este cálculo se propondrán como árboles de prueba en la sección 3.3, donde también se presenta una simplificación de los árboles de prueba, los *árboles de prueba abreviados*, que serán los que finalmente utilizaremos para la depuración declarativa de respuestas incorrectas en PLF. Esta sección y el capítulo terminan probando la corrección de la depuración basada en los árboles de prueba abreviados, cumpliendo por tanto el primero de los objetivos apuntados al comienzo.

Para facilitar la lectura del capítulo se han separado las demostraciones de las proposiciones y teoremas que requieren demasiado espacio. Todas estas demostraciones pueden consultarse en el apéndice A.

3.1. Preliminares

En esta sección presentamos los conceptos básicos acerca de la sintaxis, la disciplina de tipos y la semántica declarativa de los lenguajes lógico-funcionales, siguiendo la formalización presentada en [36] y utilizando la sintaxis particular del lenguaje $\mathcal{TOY}$ [1, 54].

3.1.1. Tipos y Signaturas

A partir de ahora supondremos la existencia de un conjunto infinito numerable $TVar$ de *variables de tipo* $\alpha,\ \beta,\ \ldots$, así como de otro conjunto, igualmente infinito numerable, $TC = \bigcup_{n\in\mathbb{N}} TC^n$ de *constructoras de tipo* C. Los tipos $\tau \in Type$ tienen como sintaxis:

$$\tau ::= \alpha \ \ (\alpha \in TVar) \mid (C\ \tau_1 \ldots \tau_n) \ \ (C \in TC^n) \mid (\tau \to \tau') \mid (\tau_1, \ldots, \tau_n)$$

donde "$\to$" asocia a la derecha y $(\tau_1, \ldots, \tau_n)$ representa un tipo producto. Por convenio supondremos que $C\ \overline{\tau}_n$ abrevia $(C\ \tau_1 \ldots \tau_n)$ y que $\overline{\tau}_n \to \tau$ abrevia $\tau_1 \to \cdots \to \tau_n \to \tau$. El conjunto de variables de tipo que aparecen en un tipo τ se denotará por $tvar(\tau)$.

Un tipo τ se dice *monomórfico* sii $tvar(\tau) = \emptyset$, y *polimórfico* en caso contrario. A los tipos que no incluyen ninguna aparición de "$\to$" los llamaremos *tipos de datos*. Una *signatura polimórfica* sobre TC es una tripleta $\Sigma = \langle TC,\ DC,\ FS\rangle$, donde $DC = \bigcup_{n\in\mathbb{N}} DC^n$ y $FS = \bigcup_{n\in\mathbb{N}} FS^n$ son conjuntos numerables de declaraciones de *constructoras de datos* y *símbolos de funciones definidas* respectivamente. Cada $c \in DC^n$ de aridad n tiene una declaración de tipo principal

$$c\ ::\ \overline{\tau}_n \to C\ \overline{\alpha}_k$$

con $n, k \geq 0$ y donde:

- Las variables de tipo $\alpha_1, \ldots, \alpha_k$ son diferentes dos a dos.
- $\tau_1 \ldots \tau_n$ son tipos de datos.
- Se verifica que $tvar(\tau_i) \subseteq \{\alpha_1, \ldots, \alpha_k\}$ para todo $1 \leq i \leq n$ (i.e. se cumple la llamada *propiedad de transparencia*).

También supondremos que todo $f \in FS^n$ de aridad n tiene una declaración de tipo principal

$$f\ ::\ \overline{\tau}_n \to \tau$$

donde cada τ_i con $i = 1 \ldots n$ y τ son tipos cualesquiera. Supondremos que las declaraciones de tipo, tanto para las constructoras como para las funciones, forman parte de su declaración y por tanto de la signatura del programa. Para cada signatura Σ, llamaremos $\Sigma_\perp$ al resultado de extender Σ con una nueva constructora de datos $\perp\ ::\ \alpha$ que representa un valor indefinido perteneciente a todos los tipos. Como convenio de notación escribimos $c, d \in DC$, $f, g \in FS$ y $h \in DC \cup FS$, y definimos la *aridad* de $h \in DC^n \cup FS^n$ como $ar(h) = n$.

3.1.2. Expresiones y Patrones

A partir de ahora siempre supondremos la existencia de una signatura Σ, aunque en ocasiones no la mencionemos explícitamente. Sea Var un conjunto numerable de *variables de datos* $X, Y, \ldots$ disjunto de $TVar$ y Σ, a las que llamaremos simplemente *variables* cuando no haya posible confusión con las variables de tipo. Podemos definir el conjunto de las *expresiones parciales* $e \in Exp_\perp$ mediante la siguiente sintaxis:

$$e ::= \perp \mid X \mid h \mid (e\ e')$$

donde $X \in Var,\ h \in DC \cup FS$. Las expresiones de la forma $(e\, e')$ representan la aplicación de la expresión e (actuando como una función) a la expresión e' (que representa el papel de argumento). Para la representación de tuplas asumiremos la existencia de una cantidad infinita numerable de constructoras $tup_n \in DC^n,\ n \geq 0$, con declaración de tipo

$$tup_n :: \alpha_1 \to \cdots \to \alpha_n \to (\alpha_1, \ldots, \alpha_n)$$

Además, usaremos por convenio la notación $(e_1, \ldots, e_n)$ para representar tuplas de n componentes como "azúcar sintáctico" para la expresión $tup_n\ e_1 \ldots e_n$. Supondremos, como es habitual, que la aplicación asocia a la izquierda y que la expresión $(e_0\, e_1 \ldots\ e_n)$ abrevia $((\ldots(e_0\, e_1)\ldots)\, e_n)$. El conjunto de variables en e se representará por $var(e)$. Una expresión e se dice *cerrada* sii $var(e) = \emptyset$, y *abierta* en otro caso. Una expresión e es llamada *lineal* sii toda $X \in var(e)$ aparece una única vez en e. Llamaremos *flexibles* a las expresiones de la forma $X\ e_1 \ldots e_m$ con $X \in Var, m > 0$ y *rígidas* a las expresiones $h\ e_1 \ldots e_m$ con $m \geq 0, h \in DC \cup FS$. Las expresiones rígidas pueden dividirse a su vez en *activas*, cuando $h \in FS^n$ con $m \geq n$, o *pasivas* en caso contrario.

Los *patrones parciales* $t \in Pat_\perp \subset Exp_\perp$ se definen siguiendo la sintaxis:

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

donde $X \in Var$, $c \in DC^n$, $0 \leq m \leq n$, $f \in FS^n$, $0 \leq m < n$ y donde los t_i representan a su vez patrones parciales. Los patrones parciales representan *aproximaciones* de los valores de las expresiones. Siguiendo las ideas de la semántica denotacional [37], veremos $Pat_\perp$ como el conjunto de elementos finitos de un dominio semántico. La siguiente definición establece una relación de orden entre los patrones parciales:

Definición 3.1.1. Definimos el *orden de aproximación* $\sqsubseteq$ como el menor orden parcial sobre $Pat_\perp$ que satisface las siguientes propiedades:

1. $\perp \sqsubseteq t$, para todo $t \in Pat_\perp$.
2. $h\ \bar{t}_m \sqsubseteq h\ \bar{s}_m$ siempre que estas dos expresiones sean patrones y $t_i \sqsubseteq s_i$ para todo $1 \leq i \leq m$.

El conjunto $Pat_\perp$, y en general cualquier conjunto ordenado parcialmente, puede convertirse en un dominio semántico mediante la técnica conocida como *compleción por ideales* (ver,

por ejemplo, [59]). Los elementos de $Pat_\perp$ se pueden representar mediante árboles finitos, mientras que los de su compleción precisarán de árboles posiblemente infinitos.

Los patrones parciales de la forma $f\ t_1 \ldots t_m$ con $f \in FS^n$ y $m < n$ son una representación conveniente de las funciones como valores (ver [36]). Una expresión o patrón en el que no aparece $\perp$ será llamada *total*. Denotaremos por Exp y Pat los conjuntos de expresiones y patrones totales, respectivamente. Realmente el símbolo $\perp$ nunca aparece en el texto de un programa pero puede aparecer durante una sesión de depuración, tal y como ya discutimos en la secciones 2.3.1 y 2.4 del capítulo anterior.

3.1.3. Sustituciones

Una *sustitución total* es una aplicación $\theta : Var \to Pat$ con una extensión única $\hat{\theta} : Exp \to Exp$ que será representada igualmente por θ. $Subst$ será el conjunto de todas las sustituciones totales, mientras que $Subst_\perp$ representará el conjunto de todas las sustituciones parciales $\theta : Var \to Pat_\perp$ definidas de manera análoga. Llamamos *dominio* de una sustitución θ, y lo denotamos por $dom(\theta)$, al conjunto de variables X tales que $\theta(X) \neq X$, y *rango* de θ, denotado por $ran(\theta)$, a

$$\bigcup_{X \in dom(\theta)} var(\theta(X))$$

Como es habitual

$$\theta = \{X_1 \mapsto t_1, \ldots, X_n \mapsto t_n\}$$

representará la sustitución con dominio $\{X_1, \ldots, X_n\}$ que verifica $\theta(X_i) = t_i$ para todo $1 \leq i \leq n$. Por convenio escribiremos $e\theta$ en lugar de $\theta(e)$, y $\theta\sigma$ para representar la composición de θ y σ tal que $e(\theta\sigma) = (e\theta)\sigma$ para cualquier e. Dado un subconjunto $\mathcal{X} \subseteq dom(\theta)$ definimos la *restricción* $\theta\restriction_{\mathcal{X}}$ como la sustitución θ' tal que $dom(\theta') = \mathcal{X}$ y $\theta'(X) = \theta(X)$ para toda $X \in \mathcal{X}$. También definimos la *unión disjunta* $\theta_1 \uplus \theta_2$ de dos sustituciones θ_1 y θ_2 con dominios disjuntos, como la sustitución θ que cumple $dom(\theta) = dom(\theta_1) \cup dom(\theta_2)$, $\theta(X) = \theta_1(X)$ para toda $X \in dom(\theta_1)$, y $\theta(Y) = \theta_2(Y)$ para toda $Y \in dom(\theta_2)$.

La aplicación id de Var en si misma es llamada *sustitución identidad*, mientras que cualquier sustitución ρ que sea una biyección de Var en si misma será llamada *renombramiento*. Se dice que dos expresiones e y e' son *variantes* sii existe algún renombramiento ρ tal que $e\rho = e'$. El *orden de subsunción* sobre Exp se define mediante la condición $e \leq e'$ sii $e' = e\theta$ para algún $\theta \in Subst$. Cuando $e \leq e'$ se dice que e es más general que e', o también que e' es una instancia de e. Sobre $Exp_\perp$ puede definirse un orden similar, que puede a su vez ser extendido a $Subst_\perp$, definiendo $\theta \leq \theta'$ sii $\theta' = \theta\sigma$ para algún $\sigma \in Subst_\perp$. Dado un conjunto de variables $\mathcal{X}$, utilizaremos las notaciones $\theta \leq \theta'[\mathcal{X}]$ (respectivamente $\theta \leq \theta'[\backslash\mathcal{X}]$) para indicar que $X\theta' = X\theta\sigma$ se cumple para algún $\sigma \in Subst_\perp$ y para toda $X \in \mathcal{X}$ (respectivamente, para toda $X \notin \mathcal{X}$). Otro concepto que nos será útil más adelante es el *orden de aproximación* sobre $Subst_\perp$, definido por la condición $\theta \sqsubseteq \theta'$ sii $\theta(X) \sqsubseteq \theta'(X)$, para toda $X \in Var$.

Hasta ahora hemos considerado *sustituciones de datos*. Las *sustituciones de tipos* pueden definirse de forma similar como aplicaciones $\theta_t : TVar \rightarrow Type$ con una extensión única $\hat{\theta_t} : Type \rightarrow Type$, que representaremos simplemente como θ_t. $TSubst$ representará el conjunto de todas las sustituciones de tipos. La mayor parte de las definiciones y conceptos introducidos hasta ahora para las sustituciones de datos (como dominio, rango, composición, renombramiento, etc.) son igualmente válidos para las sustituciones de tipos, y se utilizarán cuando resulte necesario.

3.1.4. Expresiones Bien Tipadas

Basándonos en el sistema de tipos de Milner [58, 28] introducimos a continuación el concepto de expresión bien tipada.

Definimos un *contexto* como cualquier conjunto T de supuestos de tipo para variables de datos de la forma $X :: \tau$, tal que T no incluye dos supuestos diferentes para la misma variable. El *dominio* $dom(T)$ y el *rango* $ran(T)$ de un contexto son, respectivamente, los conjuntos de todas las variables de datos y variables de tipo que aparecen en T. Para toda variable $X \in dom(T)$, el único tipo τ tal que $(X :: \tau) \in T$ se representará como $T(X)$. La notación $(h :: \tau) \in_{var} \Sigma$ se utilizará para indicar que Σ incluye una declaración de tipo $h :: \tau$, quizá con las variables de tipo renombradas.

Los *juicios de tipo* $(\Sigma, T) \vdash_{WT} e :: \tau$ se obtendrán mediante la utilización de las siguientes *reglas de inferencia de tipos* :

VR $(\Sigma, T) \vdash_{WT} X :: \tau$, si $T(X) = \tau$

ID $(\Sigma, T) \vdash_{WT} h :: \tau\sigma_t$,
si $(h :: \tau) \in_{var} \Sigma_\perp,\ \sigma_t \in TSubst$

AP $(\Sigma, T) \vdash_{WT} (e\ e_1) :: \tau$,
si $(\Sigma, T) \vdash_{WT} e :: (\tau_1 \rightarrow \tau)$, $(\Sigma, T) \vdash_{WT} e_1 :: \tau_1$, para algún $\tau_1 \in Type$

Nótese que estas reglas de inferencia de tipos pueden aplicarse a tipos polimórficos porque las declaraciones de tipo incluidas en la signatura Σ se interpretan como esquemas de tipo, tal como indica la regla de inferencia **ID**.

Una secuencia de la forma $(\Sigma, T) \vdash_{WT} e_1 :: \tau_1,\ \ldots, (\Sigma, T) \vdash_{WT} e_n :: \tau_n$ se abreviará por $(\Sigma, T) \vdash_{WT} \overline{e}_n :: \overline{\tau}_n$, mientras que $(\Sigma, T) \vdash_{WT} a :: \tau,\ (\Sigma, T) \vdash_{WT} b :: \tau$ se abreviará escribiendo $(\Sigma, T) \vdash_{WT} a :: \tau :: b$.

Una expresión $e \in Exp_\perp$ se dice que está *bien tipada* con respecto a una signatura Σ sii se puede encontrar un contexto T y un tipo τ, tales que es posible derivar un juicio de tipo

de la forma $(\Sigma, T) \vdash_{WT} e :: \tau$. Se dice que una expresión es *polimórfica* cuando admite más de un tipo. Toda expresión bien tipada tiene siempre un *tipo principal* (TP), que será el más general de todos los tipos admitidos. Un patrón cuyo TP determina el TP de todos sus subpatrones se llamará *transparente*. En [36] se pueden encontrar más detalles sobre estos conceptos.

3.1.5. Programas Bien Tipados

Un *programa bien tipado* P es un conjunto de *reglas de tipo bien tipadas* para los símbolos de función de la signatura. La reglas de programa de una función $f \in FS^n$ con tipo principal $f :: \overline{\tau}_n \to \tau$ son de la forma

$$(R) \quad \underbrace{f\ t_1 \ldots t_n}_{\text{lado izquierdo}} \ \to \ \underbrace{r}_{\text{lado derecho}} \ \Leftarrow \ \underbrace{C}_{\text{condición}}$$

y deberán cumplir las siguientes condiciones:

(i) $t_1 \ldots t_n$ debe ser una secuencia lineal de patrones transparentes, y r debe ser una expresión.

(ii) La *condición* C será una secuencia de *condiciones atómicas* $C_1, \ldots, C_k$, donde cada C_i puede ser o bien una *igualdad estricta* de la forma $e == e'$, con $e, e' \in Exp$, o bien una *aproximación* de la forma $d \to s$, con $d \in Exp$ y $s \in Pat$.

(iii) Además la condición C debe ser *admisible* respecto al conjunto de variables $\mathcal{X} =_{def} var(f\ \overline{t}_n)$. Esto significa, por definición, que el conjunto de aproximaciones en C debe admitir una reordenación $d_1 \to s_1, \cdots, d_m \to s_m$ $(m \geq 0)$ que verifique las tres condiciones siguientes:

   a) Para todo $1 \leq i \leq m$: $var(s_i) \cap \mathcal{X} = \emptyset$

   b) Para todo $1 \leq i \leq m$, s_i es lineal y para todo $1 \leq j \leq m$ con $i \neq j$ $var(s_i) \cap var(s_j) = \emptyset$.

   c) Para todo $1 \leq i \leq m, 1 \leq j \leq i$: $var(s_i) \cap var(d_j) = \emptyset$.

(iv) Existe un contexto T con dominio $var(R)$, que permite tipar correctamente la regla de programa cumpliendo las siguientes condiciones:

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

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

   c) Para cada $(e == e') \in C$ existe algún $\mu \in Type$ tal que $(\Sigma, T) \vdash_{WT} e :: \mu :: e'$.

   d) Para cada $(d \to s) \in C$ existe algún $\mu \in Type$ que verifica $(\Sigma, T) \vdash_{WT} d :: \mu :: s$.

En el lenguaje de programación $\mathcal{TOY}$ [54] las reglas de programa se escriben de una forma ligeramente diferente:

$$(R) \quad \underbrace{f\ t_1 \ldots t_n}_{\text{lado izquierdo}} \rightarrow \underbrace{r}_{\text{lado derecho}} \Leftarrow \underbrace{IE}_{\text{igualdades estrictas}} \quad \text{where} \quad \underbrace{DL}_{\text{definiciones locales}}$$

Es decir, la condición C de la regla de programa aparece dividida en dos partes: Una parte IE con las igualdades estrictas $e == e'$, y otra parte DL con las aproximaciones $d \rightarrow s$, entendidas como *definiciones locales* sobre las variables del patrón s. Esto motiva la condición (iii) vista anteriormente. De hecho:

- Los items (iii) (a), (iii) (b) exigen que las variables definidas localmente sean diferentes entre sí, y nuevas con respecto a las variables que aparecen en la parte izquierda de la regla de programa, que representan los parámetros formales.
- El ítem (iii) (c) asegura que las variables definidas en la definición local que ocupa la posición i puede ser utilizada en la definición local que ocupa la posición j sólo si $j > i$. En particular, esto implica que no se admiten definiciones locales recursivas.

Informalmente, el significado de una regla de programa (R) como la descrita más arriba, es que una llamada a la función f puede convertirse en r cuando los parámetros de llamada pueden encajar en los patrones t_i, y tanto las igualdades estrictas como las definiciones locales pueden ser satisfechas. Una igualdad estricta $e == e'$ se satisface evaluando e y e' a un mismo patrón total. Una definición local $d \rightarrow s$ se satisface evaluando d a algún patrón (puede que parcial) que encaje con s. En la sección 3.2 expondremos formalmente la semántica de las reglas de programa.

3.1.6. Un Programa Sencillo

A continuación presentamos un programa sencillo, escrito siguiendo la sintaxis del lenguaje de programación $\mathcal{TOY}$. En $\mathcal{TOY}$ cada definición local $d \rightarrow s$ se escribe como $s = d$, y debe aparecer en la posición requerida por las condiciones de admisibilidad explicada en la sección 3.1.5. En los programas se permite la utilización de operadores infijos como : para construir expresiones de la forma (X:Xs), que se entienden como ((:) X Xs). La signatura del programa puede inferirse fácilmente a partir de las declaraciones de tipo que aparecen en el texto. En particular, las declaraciones data incluyen la información completa acerca de las constructoras de tipo y los tipos principales de las constructoras de datos. El tipo de las listas y el de valores lógicos (tipo *bool*) se incluyen como comentarios al tratarse de tipos predefinidos en $\mathcal{TOY}$. Obsérvese que las constructoras de listas se representan en este programa mediante los símbolos [] y : (un operador infijo), al igual que se hace en Haskell [75], aunque en $\mathcal{TOY}$ también se permite utilizar la notación [X|Xs], habitual en Prolog [87].

Ejemplo 3.1.1.

```
% data [A] = [] | A : [A]
head :: [A] -> A
head (X:Xs) = X

tail :: [A] -> [A]
tail (X:Xs) = Xs

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

twice :: (A -> A) -> A -> A
twice F X = F (F X)

drop4 :: [A] -> [A]
drop4 = twice twice tail

from  :: nat -> [nat]
from N = N : from N

data nat = z | suc nat

plus :: nat -> nat -> nat
plus z       Y = Y
plus (suc X) Y = suc (plus X Y)

times :: nat -> nat -> nat
times z       Y = z
times (suc X) Y = plus (times X Y) X

take :: nat -> [A] -> [A]
take z       Xs     = []
take (suc N) []     = []
take (suc N) (X:Xs) = X : take N Xs

(//)   :: A -> A -> A
X // Y = X
X // Y = Y

data person = john | mary  | peter | paul  | sally | molly |
              rose | tom   |  bob  | lisa  | alan  | dolly |
              jim  | alice
```

```
parents :: person -> (person,person)
parents peter = (john,mary)
parents alan  = (paul,rose)
parents paul  = (john,mary)
parents dolly = (paul,rose)
parents sally = (john,mary)
parents jim   = (tom,sally)
parents bob   = (peter,molly)
parents alice = (tom,sally)
parents lisa  = (peter,molly)

ancestor :: person -> person
ancestor X = Y // Z // ancestor Y // ancestor Z
             where
                (Y,Z) = parents X

% data bool = true | false

related :: person -> person -> bool
related X Y = true <== ancestor X == ancestor Y
```

El significado pretendido de las funciones del programa se deduce con facilidad de sus nombres y definiciones, mientras que su aridad viene dada por el número de parámetros formales de sus reglas. En particular, drop4 (función que elimina los 4 primeros elementos de una lista) tiene aridad 0, a pesar de su declaración de tipo. Las dos últimas funciones ejemplifican la utilización de igualdades estrictas y aproximaciones. Además, las funciones ancestor y (//) son *no deterministas* (la función (//) ya fue utilizada en el ejemplo 2.4.1, pág, 39) ya que cualquier llamada a una de ellas con parámetros fijos puede producir más de un resultado. Por ejemplo, ancestor alan puede producir tanto paul como rose, john o mary.

Algunas de las reglas de programa del ejemplo son *incorrectas* con respecto a la interpretación pretendida de sus respectivas funciones. En concreto, la segunda regla de la función times y la única regla de from son erróneas; las versiones correctas deberían ser:

$$\mathsf{times\ (suc\ X)\ Y} \rightarrow \mathsf{plus\ (times\ X\ Y)\ Y} \qquad \mathsf{from\ N} \rightarrow \mathsf{N : from\ (suc\ N)}$$

En las siguiente sección daremos una definición formal de “interpretación pretendida”, definición que será necesaria a la hora de probar formalmente los resultados de corrección para nuestra técnica de depuración.

3.2. La Semántica de los Programas

Tras haber introducido los conceptos básicos ya estamos preparados para dotar de semántica a nuestros programas. Esta presentación está basada en trabajos previos de nuestro grupo de investigación [35, 36], pero adaptada a las necesidades de la depuración declarativa.

3.2.1. El Cálculo Semántico SC

En [36] se presentó un *cálculo de reescritura* llamado $GORC$ que permite deducir de un programa P aquellas igualdades estrictas y aproximaciones que pueden considerarse válidas en la semántica de P. Informalmente, podemos decir que una aproximación $e \to t$ se cumple cuando $t \in Pat_\bot$ representa un valor parcialmente definido que aproxima el valor de $e \in Exp_\bot$; mientras que una igualdad estricta $e == e'$ se cumple cuando se cumplen $e \to t$ y $e' \to t$ para algún patrón total $t \in Pat$.

En nuestro trabajo utilizaremos el *Cálculo Semántico SC*, una variante de $GORC$ que presentamos en [17] con objeto de definir un marco lógico adecuado para la depuración declarativa de respuestas incorrectas en lenguajes lógico-funcionales perezosos. Formalmente, SC consta de las siguientes reglas de inferencia:

$$\textbf{BT} \quad e \to \bot$$

$$\textbf{RR} \quad X \to X \quad \text{with} \quad X \in Var$$

$$\textbf{DC} \quad \frac{e_1 \to t_1 \ \ldots \ e_m \to t_m}{h\ \overline{e}_m \to h\ \overline{t}_m} \quad h\ \overline{t}_m \in Pat_\bot$$

$$\textbf{JN} \quad \frac{e \to t \quad e' \to t}{e \ == \ e'} \quad t \in Pat \text{ (patrón total)}$$

$$\textbf{AR+FA} \quad \frac{e_1 \to t_1 \ \ldots \ e_n \to t_n \quad \boxed{\dfrac{C \quad r \to s}{f\ \overline{t}_n \to s}} \quad s\ \overline{a}_k \to t}{f\ \overline{e}_n\ \overline{a}_k \to t} \quad \begin{array}{l}(f\ \overline{t}_n \to r \Leftarrow C) \in [P]_\bot, \\ t \neq \bot\end{array}$$

En todas las reglas SC, $e, e_i \in Exp_\bot$ son expresiones parciales, $t_i, t, s \in Pat_\bot$ son patrones parciales y $h \in DC \cup FS$. La notación $[P]_\bot$ de la regla $AR + FA$ representa el conjunto $\{(l \to r \Leftarrow C)\theta \mid (l \to r \Leftarrow C) \in P,\ \theta \in Subst_\bot\}$ de las instancias parciales de reglas de P.

Nótese que $AR+FA$ es la única regla SC que depende del programa dado. Debe entenderse como la aplicación consecutiva de dos pasos, cuya especificación mostramos a continuación por separado:

$$\textbf{AR} \quad \frac{e_1 \to t_1 \ \ldots \ e_n \to t_n \quad f\ \overline{t}_n \to s \quad s\ \overline{a}_k \to t}{f\ \overline{e}_n\ \overline{a}_k \to t} \quad \begin{array}{l} f \in FS^n \\ t \neq \bot \end{array}$$

$$\textbf{FA} \quad \frac{C \quad r \ \to s}{f\ \overline{t}_n \to s} \quad (f\ \overline{t}_n \to r \Leftarrow C) \in [P]_\bot$$

La regla $AR + FA$ formaliza los pasos que deben llevarse a cabo para para computar un *patrón parcial* t como aproximación al valor de la aplicación de función $f\ \overline{e}_n\ \overline{a}_k$. Estos pasos son:

1. Computar unos *patrones* parciales t_i adecuados como aproximaciones de los argumentos e_i.

2. Aplicar una instancia de regla de programa $(f\ \bar{t}_n \to r \Leftarrow C) \in [P]_\perp$, verificar la condición C, y computar un patrón parcial adecuado s como aproximación del lado derecho r.

3. Computar t como una aproximación para $s\ \overline{a}_k$.

Si se tiene que $k > 0$, entonces f debe ser una función de orden superior que devuelve un valor funcional representado por el patrón s. Si por el contrario $k = 0$, la regla $AR + FA$ puede simplificarse tomando $r \to t$ como segunda premisa del paso FA, cambiando $f\ \bar{t}_n \to s$ por $f\ \bar{t}_n \to t$ en AR y en FA, y omitiendo la premisa $s\ \overline{a}_k \to t$ de AR. En el resto de la tesis utilizaremos esta simplificación cuando convenga sin indicarlo explícitamente.

Hay que señalar que en SC no se pueden aplicar las reglas AR y FA separadamente; siempre deben aparecer combinadas en un paso $AR + FA$. Sin embargo nos resultará útil fijarnos en los pasos FA de las deducciones en SC, ya que son éstos los únicos pasos que dependen de las reglas del programa. Además, la conclusión de un paso FA es una aproximación especialmente simple, de la forma $f\ \bar{t}_n \to s$ (con t_i, $s \in Pat_\perp$). Las aproximaciones de esta forma serán llamadas *hechos básicos* en el resto del trabajo. Tanto los hechos básicos como las definiciones locales son aproximaciones, pero se utilizan con diferentes propósitos. Un hecho básico $f\ \bar{t}_n \to s$ afirma que el patrón parcial s (que puede ser no lineal) aproxima el resultado de $f\ \bar{t}_n$, una llamada con con el número exacto de parámetros requerido por la aridad de f, y con argumentos $t_i \in Pat_\perp$ que representan las aproximaciones parciales de los parámetros de f necesarias para computar s como resultado de la llamada.

Las otras reglas de inferencia en SC son más sencillas de entender. A partir de ahora utilizaremos la notación $P \vdash_{SC} \varphi$ para indicar que la aproximación φ puede deducirse del programa P utilizando las reglas de inferencia de SC. Por ejemplo, tomando como P el programa ejemplo de la sección 3.1.6 se pueden hacer las siguientes demostraciones en SC:

1. $P \vdash_{SC}$ from X $\to$ X:$\perp$.
2. $P \vdash_{SC}$ from X $\to$ X:X:$\perp$.
3. $P \vdash_{SC}$ head (from X) $\to$ X.
4. $P \vdash_{SC}$ parents alice $\to$ (tom,sally).
5. $P \vdash_{SC}$ ancestor alan $\to$ john.
6. $P \vdash_{SC}$ ancestor alan $\to$ mary.
7. $P \vdash_{SC}$ ancestor alice $\to$ john.
8. $P \vdash_{SC}$ ancestor alice $\to$ mary.
9. $P \vdash_{SC}$ ancestor alan == ancestor alice.

Por ejemplo, la demostración de $P \vdash_{SC}$ head (from X) $\to$ X:$\perp$ sería de la siguiente forma:

$$\dfrac{\dfrac{\dfrac{\dfrac{}{\mathsf{X} \to \mathsf{X}}\,(\mathsf{RR}) \quad \dfrac{}{\mathsf{from\ X} \to \perp}\,(\mathsf{BT})}{\mathsf{X{:}from\ X} \to \mathsf{X} : \perp}\,(\mathsf{DC})}{\boxed{\mathsf{from\ X} \to \mathsf{X{:}}\perp}}\,(\mathsf{AR{+}FA}) \quad \dfrac{}{\mathsf{X} \to \mathsf{X}}\,(\mathsf{RR})}{\mathsf{from\ X} \to \mathsf{X} : \perp}\,(\mathsf{AR{+}FA}) \quad \dfrac{}{\boxed{\mathsf{head\ (X{:}}\perp) \to \mathsf{X}}}\,(\mathsf{RR})$$

$$\mathsf{head\ (from\ X)} \to \mathsf{X}$$

En la demostración hemos incluido entre paréntesis el nombre de la regla aplicada en cada paso. En el primer paso AR+FA la instancia de regla utilizada es $\mathsf{from\ X} \to \mathsf{X{:}}\perp$, mientras que en la segunda aplicación de esta regla la instancia elegida es $\mathsf{head\ X{:}}\perp \to \mathsf{X}$. En este ejemplo se observa como el hecho de trabajar con patrones parciales nos permite especificar una semántica no estricta con la simplicidad de una semántica estricta.

Los ejemplos anteriores muestran que la semántica de las aproximaciones es consistente con su utilización como definiciones locales en las reglas de programa, pero diferente del significado de la igualdad. Por ejemplo, $\mathsf{from\ X} \to \mathsf{X{:}}\perp$ sólo significa que el valor parcial $\mathsf{X{:}}\perp$ *aproxima* el valor de $(\mathsf{from\ X})$, no que el valor de $(\mathsf{from\ X})$ sea *igual* a $\mathsf{X{:}}\perp$. Existe una relación formal entre las aproximaciones y el orden de aproximación definido sobre $Pat_\perp$ en la sección 3.1.2. Ésta y otras propiedades básicas de SC se establecen en el siguiente resultado:

Proposición 3.2.1. *Para cualquier programa P se cumple:*

1. *Para todo t, $s \in Pat_\perp$: $P \vdash_{SC} t \to s$ sii $t \sqsupseteq s$. Además en este caso cualquier prueba de $P \vdash_{SC} t \to s$ sólo utiliza las reglas DC, BT y RR.*
2. *Para todo $t \in Pat_\perp$: $P \vdash_{SC} t \to t$.*
3. *Para todo $t \in Pat_\perp$, $s \in Pat$: $P \vdash_{SC} t \to s$ sii $t \equiv s$.*
4. *Para todo t, $s \in Pat_\perp$: si $P \vdash_{SC} t \to s$ entonces se tiene $P' \vdash_{SC} t \to s$ para todo programa P'.*
5. *Para todo $e \in Exp_\perp$, $t \in Pat_\perp$ y θ, $\theta' \in Subst_\perp$ tales que $P \vdash_{SC} e\theta \to t$ y $\theta \sqsubseteq \theta'$, se tiene que $P \vdash_{SC} e\theta' \to t$ con una demostración SC con la misma estructura y tamaño.*
6. *Para todo $e \in Exp_\perp$, $s \in Pat_\perp$ tal que $P \vdash_{SC} e \to s$, se tiene también que $P \vdash_{SC} e\theta \to s\theta$ para cualquier* sustitución total $\theta \in Subst$.

Demostración Ver pág. 171, apéndice A.

3.2.2. Objetivos y Sistemas Correctos de Resolución de Objetivos

Un *objetivo bien tipado* G tiene la misma forma que una condición bien tipada. En particular, debe cumplir los criterios de admisibilidad definidos en la sección 3.1.5, pero con respecto al conjunto vacío de variables. Por tanto un objetivo será una secuencia de

igualdades estrictas y de aproximaciones, a las que también llamamos en este contexto *definiciones locales*. A menudo nos interesará distinguir las variables definidas mediante definiciones locales del resto de las variables del objetivo. Con este propósito definimos, para cada objetivo G, el conjunto:

$$def(G) = \bigcup_{(e \to s) \in G} var(s)$$

Vamos a decir que una sustitución $\theta \in Subst_\perp$ es una *solución* para un objetivo G cuando se cumpla:

1. $P \vdash_{SC} G\theta$.
2. θ es de la forma $\theta \equiv (\theta_p \biguplus \theta_l)$, donde:
   - $\theta_p \in Subst$, $dom(\theta_p) \subseteq var(G) - def(G)$, $ran(\theta_p) \cap def(G) = \emptyset$.
   - $\theta_l \in Subst_\perp$, $dom(\theta_l) = def(G)$,

Por tanto los valores tomados por las variables de $var(G) - def(G)$ no deben depender de las variables definidas localmente $def(G)$. Por ejemplo, utilizando el programa del ejemplo 3.1.1 de la sección 3.1.5 con el objetivo $G \equiv$ (plus X Y == R, R → Y) tenemos que la sustitución $\theta = \{\text{X} \mapsto \text{z}, \text{R} \mapsto \text{Y}\ \}$ no es una solución para G, a pesar de que se cumple $P \vdash_{SC} G\theta$, ya que la variable no local R queda definida a partir de la variable local Y. Por el contrario sí sería una solución la sustitución $\theta' = \{\text{X} \mapsto \text{z}, \text{Y} \mapsto \text{R}\ \}$.

Un sistema de programación lógico-funcional debe *resolver* objetivos, produciendo sustituciones como respuestas computadas. Formalmente: Un *sistema de resolución de objetivos* GS es un sistema que produce una secuencia ordenada de respuestas computadas para cada programa P y objetivo G. Cada respuesta computada debe ser una sustitución $\theta \in Subst_\perp$ de la forma $\theta \equiv (\theta_p \biguplus \theta_l)$ donde θ_p, θ_l cumplen la relación expresada en el punto 2 de la definición de solución. Utilizaremos la notación $G \Vdash_{GS,P} \theta$ para indicar que θ es una de las repuestas computadas por el sistema GS para el objetivo G usando el programa P.

La primera parte de la respuesta computada, que hemos representado por θ_p y a la que llamaremos *respuesta producida*, da valores totales a las variables no definidas localmente, y no afecta ni en su dominio ni en su rango a las variables definidas localmente. La segunda parte, θ_l o *respuesta local*, se encarga de dar valores, quizá parciales, a todas las variables definidas localmente.

En los sistemas reales sólo la sustitución θ_p se muestra al usuario, aunque obviamente también θ_l es computada internamente. Es por ello que el punto 2 de la definición de solución establece la independencia de θ_p con respecto a θ_l, ya que de otra manera la información mostrada al usuario dependería de variables cuyo valor se desconoce. Por tanto, decir que θ_p es una respuesta producida para un objetivo G por un sistema de resolución de objetivos GS equivale a decir que existe alguna sustitución θ_l que da valores a las variables locales y tal que se verifica $G \Vdash_{GS,P} (\theta_p \biguplus \theta_l)$.

En el caso de objetivos sin variables definidas localmente los conceptos de respuesta computada y respuesta producida son equivalentes, es decir se tiene $\theta \equiv \theta_p$, y en estos casos hablaremos simplemente de respuestas.

Algunos ejemplos de objetivos y respuestas computadas y producidas por el sistema $\mathcal{TOY}$ para el programa del ejemplo 3.1.1 de la sección 3.1.5:

1. El objetivo G ≡ from X → H, head H == R, se escribiría en sintaxis $\mathcal{TOY}$ como head H == R where H = from X. Una respuesta computada para este objetivo sería { R ↦ X} ⊎ { H ↦ X:⊥}, lo que se representa en el interfaz de salida de $\mathcal{TOY}$ como R==X. En este caso $def(G) = \{H\}$ y $var(G) - def(G) = \{R, X\}$. El hecho de que la respuesta producida $\theta_p =$ { R ↦ X} no incluya X en su dominio significa que no ha sido necesario vincular esta variable para resolver el objetivo, y que por consiguiente $\theta_p(X) = X$.

2. El objetivo related alan X == true tiene como respuesta (tanto producida como computada) {X ↦ alice}, entre otras.

3. El objetivo take (suc (suc z)) (from X) == Xs tiene una única respuesta, {Xs ↦ X:X:[]}, que es una respuesta *incorrecta* con respecto a la interpretación pretendida del programa.

4. El objetivo head (tail (map (times N) (from X))) == Y pregunta por el segundo elemento de la lista infinita que contiene el producto de N por las lista de los números naturales consecutivos comenzando en X. Las dos primeras respuestas computadas (y producidas) por $\mathcal{TOY}$ son {N ↦ z, Y ↦ z}, que es una respuesta *correcta*, y {N ↦ suc z, Y ↦ z}, una respuesta *incorrecta*. Esto es porque la definición incorrecta de la función times hace que la expresión (times (suc z)) sólo puede evaluarse a z. La respuesta válida {N ↦ suc z, Y ↦ suc X} esperada por el usuario pero no producida por el sistema es una *respuesta perdida*. Como ya comentamos en el capítulo anterior, en este trabajo no tratamos la depuración declarativa de respuestas perdidas.

El concepto de respuesta incorrecta, utilizado de manera intuitiva en estos ejemplos, será definido formalmente más adelante (definición 3.3.3, pág. 65).

El cálculo SC no es un cálculo operacional: puede utilizarse, por ejemplo, para comprobar si dados un programa P, un objetivo G y una sustitución θ se verifica $P \vdash_{SC} G\theta$, pero no para averiguar qué sustituciones θ verifican esta relación, es decir cuáles son las respuestas computadas por un sistema de resolución de objetivos en el sentido expuesto en el apartado 3.2.2. Esto se debe a la regla $AR + FA$, que precisa la utilización de una *instancia adecuada* de regla de programa para su aplicación, pero no nos dice cómo determinar tal instancia.

Los sistemas de programación lógico-funcional $\mathcal{TOY}$ [1, 54] y Curry [39] basan su implementación en estrategias de *estrechamiento necesario* [5, 6, 53, 91]. En [1, 53] pueden encontrarse las ideas fundamentales empleadas en la implementación del sistema $\mathcal{TOY}$. En este trabajo no vamos a discutir los detalles particulares de la implementación de ningún

sistema concreto. En lugar de esto, para demostrar los resultados teóricos del capítulo 4, asumiremos que el sistema de resolución de objetivos del sistema elegido satisface la siguiente propiedad:

Definición 3.2.1. Decimos que un sistema de resolución de objetivos GS es *correcto* con respecto al cálculo SC cuando toda respuesta computada θ obtenida para un objetivo G mediante un programa programa P es una solución para G mediante P en SC.

Por tanto la corrección asegura que las respuestas computadas pueden probarse en el cálculo semántico, es decir que siempre que se verifique $G \Vdash_{GS,P} \theta$ se cumplirá también $P \vdash_{SC} G\theta$. Pensamos que los sistemas de resolución de $\mathcal{TOY}$ [1, 54] y Curry [39] son correctos con respecto al cálculo SC en el sentido técnico de esta definición, pero en esta tesis no probamos formalmente esta creencia, lo que sería una tarea muy complicada, como cualquier prueba formal sobre la corrección de un sistema de software complejo. La investigación teórica sobre estrategias de estrechamiento necesario [5, 6, 91] incluye resultados de completitud. Sin embargo la completitud se puede perder en las implementaciones prácticas debido a la estrategia de exploración del espacio de búsqueda que adopte el sistema. De hecho esto sucede para los sistemas $\mathcal{TOY}$ y Curry. En el capítulo 4, veremos un tipo de completitud más débil (definición 4.4.3, pág. 96) asociada a los cómputos de depuración que pensamos que sí se cumple en los sistemas $\mathcal{TOY}$ y Curry y que es suficiente para nuestros propósitos.

3.3. Árboles de Prueba para la Depuración Declarativa de Respuestas Incorrectas

Tras los preliminares de las secciones anteriores ya estamos preparados para abordar el objetivo marcado en la introducción: encontrar un árbol análogo al EDT de programación funcional descrito en la sección 2.4.2 pero para programación lógico-funcional. Además queremos que este árbol (del que anticipamos un ejemplo en la figura 2.5, pág. 33) corresponda a una prueba en un cálculo lógico adecuado, algo que como vimos se había logrado en el caso de la depuración declarativa de lenguajes lógicos pero no en el del paradigma funcional. El cálculo SC de la sección anterior nos permitirá cumplir estos requerimientos.

3.3.1. Demostraciones SC como Árboles de Prueba

En la sección 3.2.2 introdujimos el concepto de respuesta computada, asumiendo la existencia de un sistema resolutor de objetivos. En el resto del trabajo supondremos además que este sistema es correcto con respecto al cálculo semántico SC. La notación $P \vdash_{SC} G\theta$ significará por convenio que $P \vdash_{SC} \varphi\theta$ se cumple para cada componente atómica φ de G.

Dado un objetivo atómico G, cada deducción en SC que prueba $P \vdash_{SC} G\theta$ puede representarse mediante un *árbol de prueba* (AP) que contiene objetivos atómicos en sus nodos, con $G\theta$ en la raíz, y tal que el contenido de cada nodo puede inferirse a partir de sus nodos hijos mediante alguna regla de inferencia SC. Si el objetivo inicial G no es atómico, entonces

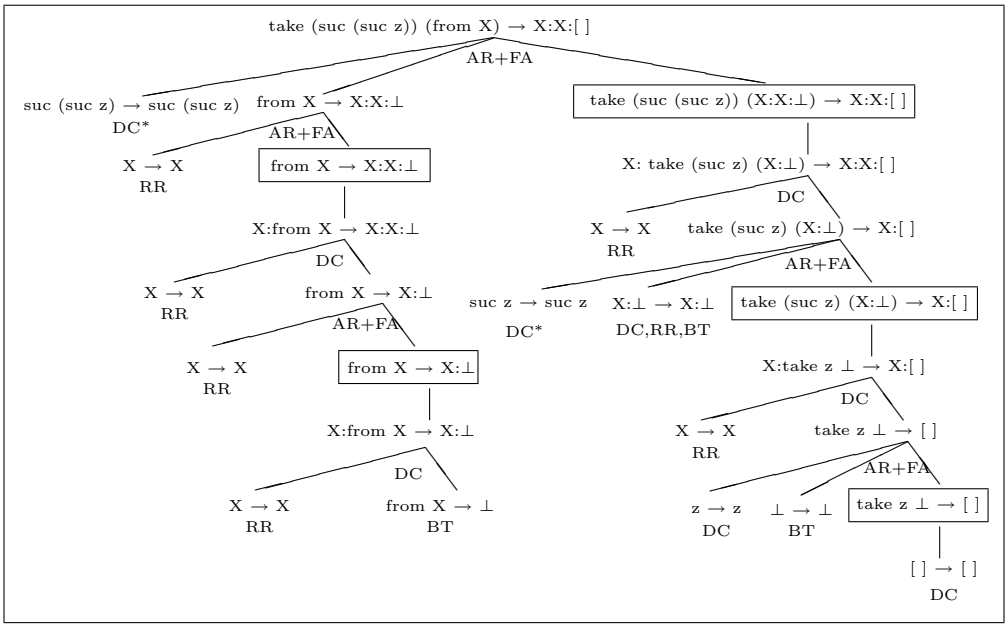


Figura 3.1: Árbol de Prueba en el cálculo semántico SC

cada deducción en SC que prueba $P \vdash_{SC} G\theta$ puede representarse mediante una familia de árboles de prueba, uno para cada deducción $P \vdash_{SC} \varphi\theta$ correspondiente a un componente atómico de G. Por abuso del lenguaje hablaremos de "árbol de prueba"también en este caso.

Por ejemplo, tal y como indicamos anteriormente $\theta = \{\mathsf{Xs} \mapsto \mathsf{X{:}X{:}[]}\}$ es una respuesta computada para el objetivo $G = \mathsf{take\ (suc\ (suc\ z))\ (from\ X)} == \mathsf{Xs}$, tomando como programa P el presentado en la sección 3.1.6. Cualquier prueba de la forma $P \vdash_{SC} G\theta$ debe incluir una deducción de $P \vdash_{SC} \mathsf{take\ (suc\ (suc\ z))\ (from\ X)} \rightarrow \mathsf{X{:}X{:}[]}$, representada por el árbol de prueba de la Fig. 3.1. Nótese que los hechos básicos que aparecen como conclusiones de pasos FA aparecen inscritos en un recuadro. Por razones de espacio se han abreviado las demostraciones de algunos de los nodos internos. Así, las hojas de la figura que aparecen etiquetadas con DC^* corresponden en el árbol completo a nodos internos que pueden probarse en SC mediante la aplicación repetida de la regla DC. Igualmente los nodos que aparecen etiquetados con más de una regla corresponden a nodos internos que pueden ser demostrados aplicando la secuencia de reglas SC indicada.

3.3.2. Árboles de Prueba Abreviados

Tal y como veremos en la siguiente sección, nuestro propósito es utilizar los árboles de prueba para la depuración declarativa. Los únicos nodos relevantes para este propósito son los correspondientes a conclusiones de pasos FA. Esto es debido a que el resto de reglas de

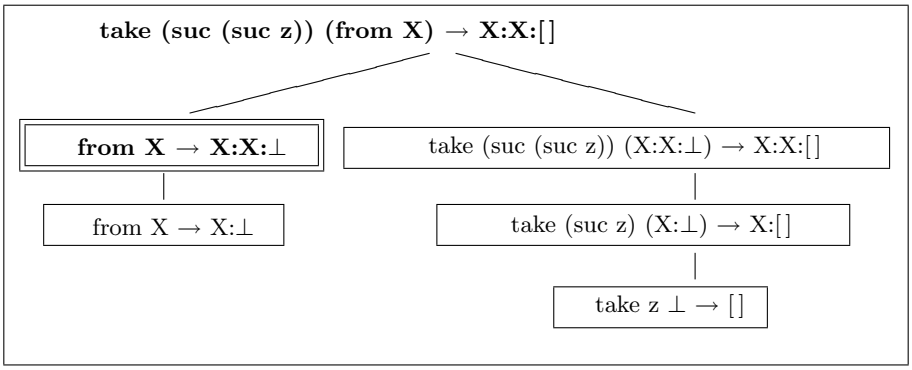


Figura 3.2: APA correspondiente al AP de la Fig. 3.1

inferencia SC son independientes del programa, tal y como comentamos anteriormente, y por tanto no pueden dar lugar a pasos incorrectos. Por esta razón vamos a asociar a cada árbol de prueba un *árbol de prueba abreviado* (al que llamaremos a menudo APA), obtenido mediante la eliminación de todos los nodos del AP, salvo la raíz, que no se corresponden con la conclusión un paso FA. El APA correspondiente a un AP se construye como se indica a continuación:

- La raíz del APA es la raíz del AP.
- Los hijos de un nodo que ya forma parte del APA, son los descendientes más cercanos del correspondiente nodo en el AP tales que corresponden con la conclusión de un paso FA.

En todo APA, cada nodo, excepto la raíz, está implícitamente asociado a la instancia de la regla de programa utilizada en el paso FA correspondiente, cuya conclusión es precisamente el hecho básico $f\ \bar{t}_n \rightarrow s$ que constituye el propio nodo. Es interesante recordar que $t_1, \ldots, t_n$, s son patrones parciales que no pueden contener ninguna llamada a función reducible.

Como ejemplo concreto, la Fig. 3.2 muestra el APA que se obtiene a partir del AP de la Fig. 3.1.

Aunque la definición de APA es bastante intuitiva nos interesa establecer también una definición más formal que será utilizada en algunas demostraciones. Para ello introducimos primero algunas definiciones auxiliares, que nos serán útiles también en capítulos posteriores. Para cada árbol de prueba T representaremos como $apa(T)$ su APA asociado. Dado un árbol cualquiera T usaremos la notación *raíz(T)* para referirnos a su raíz. La expresión *subárboles(T,N)* se referirá la secuencia de árboles hijos del nodo $N \in T$ enumerados de izquierda a derecha y que representaremos en forma de lista. Escribiremos simplemente *subárboles(T)* para referirnos a *subárboles(T,raíz(T))*, y *subárbol(T,N)* para el subárbol de T cuya raíz es N. Mediante *hijos(T,N)* nos referiremos a los nodos hijos del nodo N en

el árbol T, enumerados de izquierda a derecha y representados igualmente por una lista. También abreviaremos *hijos(T,raíz(T))* escribiendo *hijos(T)*. Si se tiene que *hijos(T,N)* = *[]* y $N \in T$, entonces N es un nodo-hoja de T. La notación *árbol(r,l)* se referirá al árbol de raíz r y de árboles hijos enumerados de izquierda a derecha en la secuencia l. Por tanto para todo árbol T se verifica T = *árbol(raíz(T),subárboles(T))*. Por último, utilizaremos el símbolo '++' para representar la concatenación de subárboles.

Dado un árbol de prueba T tal que *subárboles(T)* $= [a_1, \ldots, a_k]$, se cumple:

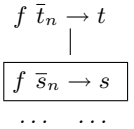
$$apa(T) = \textit{árbol(}\ \textit{raíz(T)},\ apa'(a_1)\ +\!+\ \ldots\ +\!+\ apa'(a_k)\ \textit{)}$$

$$apa'(T) = \begin{cases} [apa(T)] & \text{si } \textit{raíz(T)} \text{ es la concl. de un paso FA no trivial} \\ \textit{subárboles(apa(T))} & \text{en otro caso} \end{cases}$$

Observación 3.3.1. *Aunque la definición de APA especifica con precisión el orden de los subárboles de cada nodo, nos interesará admitir también como posibles APAs cualquier reordenación de estos subárboles. Esto no afectará a la depuración, ya que es fácil comprobar que todo nodo crítico en un APA sigue siéndolo en un subárbol en el que se intercambian los subárboles correspondientes a dos nodos hermanos.*

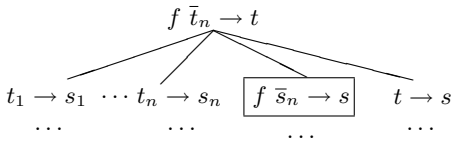
La siguiente proposición, también muy sencilla, establece la forma de los APAs correspondientes a hechos básicos.

Proposición 3.3.1. *Sea P un programa y $f\ \bar{t}_n \to t$ un hecho básico con $t \neq \bot$ tal que $P \vdash_{SC} f\ \bar{t}_n \to t$, y sea T un AP de esta derivación. Entonces $apa(T)$ es de la forma:*



Para ciertos $s_i \in Pat_\bot$ tales que $t_i \to s_i$ para cada $i = 1 \ldots n$ y para $s \in Pat_\bot$ tal que $t \to s$.

Demostración. El paso SC aplicado en la raíz de T debe corresponder a la regla $AR{+}FA$, y T será de la forma:



El único descendiente inmediato de la raíz conclusión de un paso FA no trivial es $\boxed{f\ \bar{s}_n \to s}$, ya que las pruebas de los nodos $t_i \to s_i$ para $i = 1 \ldots n$ y de $t \to s$ pueden hacerse únicamente mediante pasos DC, BT y RR, al tratarse de aproximaciones entre patrones (apartado 1 de la proposición 3.2.1, pág. 57). ■

Para finalizar este apartado veamos algunas propiedades básicas que relacionan los APAs con las pruebas en SC:

Proposición 3.3.2. *Sea P un programa cualquiera, y sean $e, e' \in Exp_\bot$ y $t \in Pat_\bot$. Entonces se cumplen:*

1. *Si* $P \vdash_{SC} e \to t$ *con un árbol de prueba* T *y* $t \sqsupseteq s$ *para algún valor* $s \in Pat_\perp$*, entonces se puede probar* $P \vdash_{SC} e \to s$ *mediante un árbol de prueba* T' *de profundidad menor o igual a la de* T *y tal que* $apa(T)$ *y* $apa(T')$ *sólo difieren en la raíz.*

2. $P \vdash_{SC} e\ e' \to t$ *sii existe un* $s \in Pat_\perp$ *tal que* $P \vdash_{SC} (e\ s \to t, e' \to s)$*. Además ambas pruebas admiten APAs que sólo difieren en la raíz.*

3. $P \vdash_{SC} e\ e' \to t$ *sii existe un* $s \in Pat_\perp$ *tal que* $P \vdash_{SC} (e \to s,\ s\ e' \to t)$*. Además ambas pruebas admiten APAs que sólo difieren en la raíz.*

Demostración Ver pág. 179, apéndice A.

3.3.3. Modelos Pretendidos

Los modelos pretendidos de los programas lógicos, tal como aparecen en [32, 51], pueden representarse como conjuntos de fórmulas atómicas pertenecientes a la base de Herbrand del programa. Sin embargo, la utilización del *universo abierto de Herbrand* (i.e. el conjunto de términos con variables) da lugar a una semántica con más información, tal como se indica en [31]. En nuestro entorno de programación lógico-funcional, el análogo natural al universo abierto de Herbrand es el conjunto $Pat_\perp$ de todos los patrones parciales junto con el orden de aproximación $\sqsubseteq$. De manera similar, el análogo natural de la base de Herbrand abierta sería la colección de todos los hechos básicos $f\ \bar{t}_n \to s$. La definición siguiente se basa en estas ideas:

Definición 3.3.1. Una *interpretación de Herbrand* es un conjunto $\mathcal{I}$ de hechos básicos que para toda $f \in FS^n$ y patrones parciales arbitrarios $t, \bar{t}_n$ cumple los tres requisitos siguientes:

1. $(f\,\bar{t}_n \to \perp) \in \mathcal{I}$.
2. Si $(f\ \bar{t}_n \to s) \in \mathcal{I}$, $t_i \sqsubseteq t'_i, s \sqsupseteq s'$ entonces también $(f\,\bar{t}'_n \to s') \in \mathcal{I}$.
3. Si $(f\ \bar{t}_n \to s) \in \mathcal{I}$, y θ es una sustitución *total*, entonces $(f\ \bar{t}_n \to s)\theta \in \mathcal{I}$.

Esta definición de interpretación de Herbrand es más simple que la dada en [36], donde se presenta una noción más general de interpretación bajo el nombre de *álgebra*. La desventaja de esta definición más sencilla es que no nos permite considerar interpretaciones que no sean de Herbrand. Sin embargo, también en [36] se demuestra que el modelo mínimo que cualquier programa es una interpretación de Herbrand, por lo que no parece demasiado restrictivo suponer que también lo es el modelo pretendido. Por tanto en nuestro modelo de depuración asumiremos que el modelo pretendido de un programa es una interpretación de Herbrand $\mathcal{I}$. Las interpretaciones de Herbrand pueden ordenarse entre si mediante la inclusión de conjuntos.

Un programa correcto P debe ser conforme a su interpretación pretendida $\mathcal{I}$. Para formalizar esta idea, necesitamos algunas definiciones que damos a continuación.

Definición 3.3.2. Diremos que un objetivo G es *válido* en la interpretación de $\mathcal{I}$ si toda igualdad estricta o aproximación $\varphi \in G$ puede probarse en el cálculo $SC_{\mathcal{I}}$. Este cálculo consta de las reglas BT, RR, DC y JN de SC junto con la regla de inferencia $FA_{\mathcal{I}}$ siguiente:

$$\textsf{FA}_{\mathcal{I}} \qquad \frac{e_1 \to t_1 \ \ldots \ e_n \to t_n \quad s\ \overline{a}_k \to t}{f\ \overline{e}_n\, \overline{a}_k \to t} \qquad \begin{array}{l} t \text{ patrón}, t \neq \bot, s \text{ patrón} \\ (f\ \overline{t}_n \ \to \ s) \in \mathcal{I} \end{array}$$

Por ejemplo, suponiendo la interpretación pretendida natural $\mathcal{I}$ para el programa ejemplo de la sección 3.1.6, se tiene que las siguientes aproximaciones e igualdades estrictas son válidas en $\mathcal{I}$:

1. from X → X:suc X:⊥
2. take (suc (suc z)) (from X) → X:suc X:[]
3. ancestor alan == ancestor alice

La primera aproximación pertenece incluso a $\mathcal{I}$. En general, puede probarse que todo hecho básico $f\ \overline{t}_n \to s$ es válido en $\mathcal{I}$ sii $(f\ \overline{t}_n \to s) \in \mathcal{I}$.

A partir de la noción de interpretación pretendida podemos precisar ahora el concepto de respuesta producida incorrecta:

Definición 3.3.3. Sean P un programa, $\mathcal{I}$ su interpretación pretendida, G un objetivo y θ_p una respuesta producida para G por algún sistema de resolución de objetivos usando P. Entonces decimos que θ_p es una respuesta (producida) incorrecta si $G(\theta_p \uplus \theta_l)$ no es válido en $\mathcal{I}$ para ninguna sustitución θ_l con $dom(\theta_l) = def(G)$.

Es decir que la respuesta producida es θ_p incorrecta si $G\theta_p$ no es válida en la interpretación pretendida para ningún valor de sus variables locales.

A continuación definimos la *denotación* de una expresión y la noción de *modelo* de un programa dado:

Definición 3.3.4.

- La denotación de la expresión e es el conjunto

$$[\![e]\!]^{\mathcal{I}} = \{s \in Pat_{\bot} \mid e \to s \text{ es válida en } \mathcal{I}\}$$

- $\mathcal{I}$ es modelo de P ($\mathcal{I} \models P$) sii y toda regla de programa de P es válida en $\mathcal{I}$.
- Una regla de programa $l \to r \ \Leftarrow C$ es válida en $\mathcal{I}$ ($\mathcal{I} \models l \to r \ \Leftarrow C$) sii para toda sustitución $\theta \in Subst_{\bot}$, se tiene que $\mathcal{I}$ satisface la instancia de regla $l\theta \to r\theta \Leftarrow C\theta$.
- $\mathcal{I}$ satisface una instancia de regla $l' \to r' \ \Leftarrow C'$ sii o bien $\mathcal{I}$ no satisface C' o bien $[\![l']\!]^{\mathcal{I}} \supseteq [\![r']\!]^{\mathcal{I}}$.

- $\mathcal{I}$ satisface una condición instanciada $C' = \varphi_1, \ldots \varphi_k$ sii para $i = 1 \ldots k$, $\mathcal{I}$ satisface φ_i.
- $\mathcal{I}$ satisface $d' \rightarrow s' \in C'$, sii $[\![d']\!]^{\mathcal{I}} \supseteq [\![s']\!]^{\mathcal{I}}$. Puede comprobarse que $[\![d']\!]^{\mathcal{I}} \supseteq [\![s']\!]^{\mathcal{I}}$ sii $s' \in [\![d']\!]^{\mathcal{I}}$.
- $\mathcal{I}$ satisface $l' == r' \in C'$, sii $[\![l']\!]^{\mathcal{I}} \cap [\![r']\!]^{\mathcal{I}} \cap Pat \neq \emptyset$.

La relación fundamental entre programas y modelos se establece en el siguiente resultado que aparece probado en [36] para una noción de modelo más general que la de modelo de Herbrand dada aquí. La demostración de este resultado puede encontrarse en el apéndice A.

Teorema 3.3.3. *Sea P un programa y G un objetivo. Entonces:*
(a) Si $P \vdash_{SC} G$ entonces G es válido en cualquier modelo de Herbrand de P.
(b) $\mathcal{M}_P = \{f\ \bar{t}_n \rightarrow s \mid P \vdash_{SC} f\ \bar{t}_n \rightarrow s\}$ es el menor modelo de Herbrand *de P con respecto al orden de inclusión.*
(c) Si G es válido en $\mathcal{M}_P$ entonces $P \vdash_{SC} G$.

Demostración Ver pág. 185, apéndice A.

Combinando este teorema con la definición 3.3.4 y con la corrección del sistema de resolución de objetivos razonable, se obtiene de manera inmediata el siguiente resultado:

Proposición 3.3.4. *Sea P un programa, G un objetivo y GS un sistema de resolución de objetivos correcto. Sea θ_p una respuesta incorrecta producida por GS para G usando P. Entonces debe existir alguna regla de programa en P que no es válida en $\mathcal{I}$.*

Demostración. Por ser θ_p una respuesta producida, se tiene que debe existir una sustitución θ_l tal que $(\theta_p \biguplus \theta_l)$ es una respuesta computada, es decir $G \Vdash_{GS,P} (\theta_p \biguplus \theta_l)$. Como GS es correcto (def. 3.2.1) ha de cumplirse $P \vdash_{SC} G(\theta_p \biguplus \theta_l)$. Como además θ_p es errónea por la definición 3.3.3 se tiene que $G(\theta_p \biguplus \theta_l)$ no es válida en $\mathcal{I}$, lo que por el teorema 3.3.3 apartado (a) significa que $\mathcal{I}$ no es modelo de Herbrand de P, y por la definición 3.3.4 esto nos lleva a que P contiene alguna regla de programa incorrecta con respecto a $\mathcal{I}$. ■

Esta proposición predice la existencia de al menos una regla de programa incorrecta siempre que se observe una respuesta computada incorrecta. En el caso de nuestro programa ejemplo, $\theta = \{\mathsf{Xs} \mapsto \mathsf{X{:}X{:}[]}\}$ es una respuesta computada incorrecta para el objetivo $G = \mathsf{take\ (suc\ (suc\ z))\ (from\ X)} == \mathsf{Xs}$, porque $G\theta$ no es válido en el modelo pretendido del programa. Por la proposición 3.3.4, debe haber alguna regla incorrecta en P que es la responsable de la respuesta incorrecta. En este caso la regla incorrecta causa del error es la regla que define from.

Siempre que una regla de programa $l \rightarrow r \Leftarrow C$ no es válida en el modelo pretendido $\mathcal{I}$, debe haber alguna sustitución $\theta \in Subst_{\perp}$ tal que la instancia de regla $l\theta \rightarrow r\theta \Leftarrow C\theta$ no es satisfecha $\mathcal{I}$, lo que significa que

1. $\varphi\theta$ es válida en $\mathcal{I}$ para todo $\varphi \in C$.

2. $r\theta \to s$ es válida en $\mathcal{I}$ para algún $s \in Pat_\perp$ tal que $(l\theta \to s) \notin \mathcal{I}$.

En nuestro ejemplo, la instancia incorrecta de regla que define from es la propia regla. Ciertamente N:from N → N:N:⊥ es válida en $\mathcal{I}$, pero (from N → N:N:⊥) $\notin \mathcal{I}$. Esto se corresponde con el ítem 2, con N:N:⊥ haciendo el papel de s.

A efectos de la depuración práctica, la proposición 3.3.4 debería ser refinada hasta proporcionar un método *efectivo* que pueda utilizarse para detectar instancias incorrectas de reglas de programa a partir de la observación de una respuesta computada incorrecta. En la siguiente sección veremos que esto puede conseguirse mediante la depuración declarativa basada en los APAs como árboles de cómputo.

3.3.4. Corrección de la Depuración Declarativa con APAs

Nuestro esquema para la depuración declarativa de respuestas incorrectas en programación lógico-funcional se base en el esquema general presentado en la sección 2.2. Partimos para ellos de programas y objetivos de PLF que suponemos bien tipados en el sentido indicado en la sección 3.1.4. También supondremos la existencia de un modelo pretendido para cada programa, representado como un conjunto de hechos básicos, tal como se explicó en la sección 3.3.3. Así mismo, debemos suponer que los cómputos los lleva a cabo un sistema de resolución de objetivos razonable con respecto al cálculo semántico SC presentado en la sección 3.2.1. Finalmente, siempre que un cómputo obtenga una sustitución θ como respuesta computada para un objetivo G mediante el programa P, supondremos la existencia de un APA *testigo* de la demostración de $P \vdash_{SC} G\theta$, que será utilizado como árbol de prueba por el depurador. La generación efectiva de estos árboles será discutida en el capítulo 4.

Para probar la corrección de la depuración declarativa con APAs comenzamos por definir formalmente el concepto de nodo incorrecto:

Definición 3.3.5. Sea P un programa, $\mathcal{I}$ su modelo pretendido y *apa* un APA obtenido a partir de algún árbol de prueba testigo de una demostración SC hecha con respecto al programa P. Entonces un nodo de *apa* se considera *incorrecto* cuando su contenido no es válido en $\mathcal{I}$. A menudo utilizaremos como sinónimos de incorrecto los términos *erróneo* o *no válido*.

La propiedad expresada en la siguiente proposición nos será útil más adelante:

Proposición 3.3.5. *La raíz de un APA nunca es un nodo crítico.*

Demostración
Ver pág. 190, apéndice A.

El siguiente teorema garantiza la corrección lógica de la depuración declarativa con APAs:

Teorema 3.3.6. *Sea P un programa, G un objetivo, GS un sistema de resolución de objetivos correcto y $\theta \equiv (\theta_p \biguplus \theta_l)$ una sustitución tal que $G \Vdash_{GS,P} \theta$. Supongamos que disponemos de un APA testigo de $P \vdash_{SC} G\theta$, el cual debe existir debido a la corrección de*

GS, y que θ_p es una respuesta producida incorrecta. En estas circunstancias la depuración declarativa basada en el esquema general de la sección 2.2 que toma el APA como árbol de cómputo cumple las dos propiedades siguientes:

(a) Completitud: el APA tiene un nodo crítico.

(b) Corrección: cada nodo crítico del APA tiene asociada una instancia de regla de programa que es incorrecta con respecto al modelo pretendido del programa P.

Demostración
Llamemos $\mathcal{I}$ al modelo pretendido, *apa* al APA, y *ap* al *AP* a partir del cual se ha obtenido *apa*. Al ser θ_p una respuesta incorrecta se tiene que $G(\theta_p \uplus \nu)$, es decir la raíz de *apa* y de *ap*, no es válida en $\mathcal{I}$ (definición 3.3.3), por lo que el ítem (a) se sigue inmediatamente del teorema 2.2.1 (pág. 28). Para probar el ítem (b), consideremos un nodo crítico cualquiera del *apa*, que será distinto de la raíz como consecuencia de la proposición 3.3.5 (pág. 67). El nodo correspondiente en *ap* debe contener un hecho básico $f\ \bar{t}_n \rightarrow s$ que no es válido en $\mathcal{I}$ y que aparece como conclusión de un paso FA. En este paso FA se ha utilizado una instancia de regla de programa, digamos $(f\ \bar{t}_n \rightarrow r \Leftarrow C) \in [P]_\perp$. Por tanto los hijos de $f\ \bar{t}_n \rightarrow s$ en *ap* contendrán a la aproximación $r \rightarrow s$ además de a todas las componentes atómicas de C. En *apa* los hijos de $f\ \bar{t}_n \rightarrow s$ no son necesariamente éstos mismos; pero como *apa* se ha construido a partir de *ap*, resulta que tanto $r \rightarrow s$ como C pueden inferirse a partir de los hijos de $f\ \bar{t}_n \rightarrow s$ en *apa* mediante pasos de inferencia SC distintos de FA y por tanto correctos en toda interpretación de Herbrand. Además todos los hijos de $f\ \bar{t}_n \rightarrow s$ en *apa* tienen que ser válidos en $\mathcal{I}$, ya que son hijos de un nodo crítico (def. 2.1.5, pág. 20). De esto puede concluirse que tanto C como $r \rightarrow s$ son válidos en $\mathcal{I}$ mientras que $f\ \bar{t}_n \rightarrow s$ no lo es; y esto significa que la instancia de regla $(f\ \bar{t}_n \rightarrow r \Leftarrow C) \in [P]_\perp$ es incorrecta en $\mathcal{I}$. ■

Este teorema constituye una versión práctica de la proposición 3.3.4, así como una interpretación lógica de los árboles de cómputo. Hasta donde sabemos ningún resultado similar ha sido probado hasta ahora en otras propuestas de depuración declarativa para programación funcional o lógico-funcional: [64, 65, 69, 68, 71, 76, 77, 78, 85, 84].

Como ejemplo concreto, consideremos de nuevo el AP de la Fig. 3.1 (pág. 61) y su APA correspondiente de la Fig. 3.2 (pág. 62). Como dijimos anteriormente, este AP corresponde al cómputo de la respuesta errónea $\theta_p = \{\mathsf{Xs} \mapsto \mathsf{X{:}X{:}[]}\}$ para el objetivo $G \equiv$ take (suc (suc z)) (from X) == Xs, utilizando el programa ejemplo de la sección 3.1.6[1]. En la Fig. 3.2, los hechos básicos correspondientes a nodos erróneos aparecen en negrita, mientras que el único nodo crítico está inscrito en un rectángulo doble. El razonamiento del teorema 3.3.6 apunta aquí a la instancia de regla de programa utilizada en el paso FA correspondiente al nodo crítico, que es from N → N:from N, y por tanto a la regla de programa de *from* como causa del error.

En [17] presentamos un método para extraer el APA testigo de un cómputo particular a partir de la representación formal del cómputo en un cálculo de pruebas para lenguajes

[1]Hablando con precisión el AP de este cómputo debería tener en la raíz la igualdad estricta take (suc (suc z)) (from X) == X:X:[]; pero el AP de la Fig. 3.1 representa la parte interesante de la deducción.

lógico-funcionales perezosos sin definiciones locales ni igualdad estricta. Este resultado teórico dependía de una particular formalización de la técnica de estrechamiento, y no mostraba un método para una implementación de una herramienta de depuración declarativa para sistemas de programación PLF. En el capítulo 4 veremos como obtener los APA mediante transformación de programas, lo que nos permitirá construir las herramientas de depuración discutidas en la segunda parte de la tesis.

Capítulo 4

Generación de APAs mediante transformación de programas

En el capítulo anterior hemos definido los árboles de prueba abreviados (APAs), y hemos probado que resultan adecuados para la depuración de respuestas incorrectas en programación lógico-funcional. El siguiente problema con el que nos enfrentamos es el de la construcción efectiva de tales árboles.

En el caso de la programación lógica los cálculos semánticos que definen los árboles de prueba resuelven el problema desde un punto de vista conceptual, al poder utilizarse como cálculos operacionales. En otras palabras, en el paradigma lógico es posible escribir un programa que, partiendo de un objetivo, sea capaz de construir un árbol de prueba adecuado aplicando paso a paso las reglas de un cálculo similar a los presentados en la sección 2.1.5 (pág. 13). Por supuesto, esto no significa que la construcción de árboles de prueba para programas lógicos sea una tarea inmediata, ya que subsisten numerosos problemas, como el consumo de recursos del árbol generado (tanto en tiempo como en memoria requerida), que pueden complicar la puesta en práctica de la solución conceptual.

En el paradigma funcional la definición del árbol de prueba EDT (presentado en la sección 2.3.1, pág. 30) no conduce al desarrollo de un mecanismo formal para su generación. En la siguiente sección repasaremos las propuestas que se han hecho en este paradigma para la generación de los EDT.

La situación para el caso lógico-funcional es similar a la del paradigma funcional: los APAs definidos en el capítulo anterior están definidos formalmente como árboles de prueba en el cálculo semántico SC. Sin embargo, este cálculo no puede utilizarse como un cálculo operacional, como hemos ya discutido en el apartado 3.2.2 (pág. 57) del capítulo anterior. Por tanto, para la generación efectiva de los APAs tendremos que recurrir a las mismas técnicas que se utilizan en programación funcional para la generación de los EDTs. A pesar de todo el cálculo SC también nos será útil en esta ocasión, ya que nos permitirá probar formalmente la corrección de la técnica empleada, algo que hasta ahora no había podido hacerse para el paradigma funcional ni lógico-funcional.

La siguiente sección repasa brevemente las dos alternativas para la construcción de

los EDTs que se han propuesto en programación funcional y justifica nuestra elección, la transformación de programas. En la sección 4.2 comentaremos un problema existente en las primeras versiones de la transformación de programas y cuya solución constituye una de nuestras aportaciones. A continuación presentaremos formalmente la transformación de programas, que hemos dividido en dos fases. La primera fase, descrita en la sección 4.3, consistirá en un *aplanamiento* inicial del programa. Aunque el concepto de aplanamiento no es nuevo, sí lo es su aplicación a la depuración declarativa, y nos permitirá presentar la transformación de programas de un modo más claro y fácil de comprender que en nuestra propuesta [19]. La sección 4.4 describe la segunda fase, en la que se define la transformación que, partiendo de un programa plano, producirá un programa capaz de devolver árboles como parte de sus cómputos. En esta sección comprobaremos además que, dado un objetivo G y una respuesta incorrecta producida θ, el programa transformado calcula efectivamente un árbol que puede ser utilizado para la depuración, y que un depurador basado en este árbol encontrará, suponiendo un oráculo capaz de detectar nodos erróneos, una regla de programa incorrecta. Esto probará la corrección de la técnica.

Al igual que sucedía con el capítulo anterior hemos separado las demostraciones más prolijas, que pueden encontrarse el apéndice A. Los contenidos que presentamos a continuación se basan en una reelaboración de nuestros trabajos previos [17, 19].

4.1. La Elección de la Transformación de Programas

En los mismos artículos [85, 71] en los que H.Nilsson y J.Sparud proponen el EDT como árbol de prueba adecuado para la depuración declarativa de programas funcionales, se discuten dos métodos alternativos para la generación de estos árboles: la modificación de la máquina abstracta que lleva a cabo los cómputos y la transformación de programas.

La modificación de la máquina abstracta ha sido desarrollada en trabajos posteriores como [84, 68]. La principal ventaja de esta propuesta con respecto a la transformación de programas es la eficiencia. En [68] se muestra experimentalmente que el depurador declarativo para Freja (comentado en la sección 2.3.3, pág. 34) basado en estas técnicas es capaz de tratar programas funcionales de tamaño medio, empleando para ello algunos de los programas del banco de pruebas *nofib* [73]. La principal desventaja, en cambio, es que las modificaciones son dependientes de la máquina y no son, por tanto, fácilmente portables de un sistema a otro.

Por el contrario la técnica basada en la transformación de programas es mucho menos eficiente que la basada en la modificación de la máquina abstracta, pero a cambio sólo depende de las características particulares del sistema para la implementación de una única función impura (llamada *dirt* en [64, 65, 78]) y $dVal$ en nuestra presentación), lo que la hace mucho más portable.

Son dos las razones que nos han llevado a preferir la transformación de programas en esta tesis:

- La portabilidad que acabamos de mencionar, que nos ha permitido, reutilizando las

mismas técnicas, incluir un depurador declarativo en el sistema $\mathcal{TOY}$ [19], basado en la traducción a Prolog del código lógico-funcional, y en el sistema Curry de Münster [18], que genera código para una máquina abstracta. Una transformación de programas similar ha sido así mismo empleada por Bernard Pope para escribir el depurador declarativo Buddha [76, 77] para Haskell (descrito brevemente en la sección 2.3.3, pág. 34).

- Además resulta más sencillo probar formalmente la corrección de la transformación de programas que la de la modificación del intérprete o máquina abstracta modificada. Esto nos permitirá en las secciones siguientes establecer resultados teóricos de corrección dentro del marco teórico presentado en el capítulo anterior.

Aunque en [85, 71] se esboza una transformación de programas, ésta se limita a programas de primer orden y plantea problemas cuando se aplica a programas con parámetros de orden superior, por lo que no es aplicable más que al subconjunto de programas funcionales de primer orden. En [19] solucionamos este problema, que comentamos en la próxima sección.

4.2. Funciones Currificadas

Informalmente, todos los enfoques basados en transformación de programas se basan en lograr que las funciones definidas en el programa transformado devuelvan pares (res, ap), donde res debe ser el resultado que devuelve la función en el programa original y ap el correspondiente árbol de prueba. Desde el punto de vista de los tipos, la transformación de una función de aridad n $f \in FS^n$ tendría el siguiente aspecto:

$$f \;::\; \tau_1 \;\to\; \cdots \;\to \tau_n \;\to\; \tau \quad \Rightarrow \quad f^{\mathcal{T}} \;::\; \tau_1^{\mathcal{T}} \;\to\; \cdots \;\to \tau_n^{\mathcal{T}} \;\to\; (\tau^{\mathcal{T}}, \mathsf{cTree})$$

donde cTree es un tipo de datos que representa los árboles de prueba y $\tau_i^{\mathcal{T}}$, $\tau^{\mathcal{T}}$ son las transformaciones de los tipos τ_i, τ respectivamente. Esta transformación de tipos equivale a la identidad en el caso de tipos de datos (es decir, tipos en los que no aparece la constructora de tipos "$\to$"), pero resulta importante en el caso de *tipos de orden superior*, cuya transformación debe incluir el tipo cTree.

Consideremos el siguiente ejemplo, cuyas reglas de función y definiciones de datos ya conocemos, pues eran parte del ejemplo 3.1.1 (pág. 52):

Ejemplo 4.2.1.

```
data nat = z | suc nat

plus :: nat -> nat -> nat
plus z       Y = Y
plus (suc X) Y = suc (plus X Y)
```

```
twice :: (A -> A) -> A -> A
twice F X = F (F X)

drop4 :: [A] -> [A]
drop4 =  twice twice tail

from :: nat -> [nat]
from N = N: from N % regla errónea

head :: [A] -> A
head (X:Xs) = X

tail :: [A] -> [A]
tail (X:Xs) = Xs

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

Los tipos de las funciones se transforman entonces de la siguiente forma:

plus:: nat $\rightarrow$ nat $\rightarrow$ nat	$\Rightarrow$	plus$^{\mathcal{T}}$:: nat $\rightarrow$ nat $\rightarrow$ (nat, cTree)
twice :: (A $\rightarrow$ A) $\rightarrow$ A $\rightarrow$ A	$\Rightarrow$	twice$^{\mathcal{T}}$ :: (A $\rightarrow$ (A,cTree)) $\rightarrow$ A $\rightarrow$ (A, cTree)
drop4 :: [A] $\rightarrow$ [A]	$\Rightarrow$	drop4$^{\mathcal{T}}$:: ([A] $\rightarrow$ ([A], cTree), cTree)
from:: nat $\rightarrow$ [nat]	$\Rightarrow$	from$^{\mathcal{T}}$:: nat $\rightarrow$ ([nat], cTree)
head :: [A] $\rightarrow$ A	$\Rightarrow$	head$^{\mathcal{T}}$ :: [A] $\rightarrow$ (A, cTree)
tail :: [A] $\rightarrow$ [A]	$\Rightarrow$	tail$^{\mathcal{T}}$ :: [A] $\rightarrow$ ([A], cTree)
map :: (A $\rightarrow$ B) $\rightarrow$ [A] $\rightarrow$ [B]	$\Rightarrow$	map$^{\mathcal{T}}$ :: (A $\rightarrow$ (B,cTree)) $\rightarrow$ [A] $\rightarrow$ ([B], cTree)

El tipo de drop4 tiene la forma $(\tau^{\mathcal{T}}, cTree)$ porque drop4 es una función de aridad 1, al estar definida por reglas de programa sin parámetros.

Como se señala en [65, 78], la aplicación ingenua de este enfoque conduce a errores de tipo cuando se utilizan funciones currificadas para construir expresiones que se pasan como parámetros a otras funciones de orden superior. Por ejemplo, la expresión (map drop4) estaría bien tipada en el programa original, pero si la transformamos simplemente como (map$^{\mathcal{T}}$ drop4$^{\mathcal{T}}$) obtendríamos una expresión mal tipada, porque el tipo de drop4$^{\mathcal{T}}$ (un tipo producto) no encaja con el tipo esperado por map$^{\mathcal{T}}$ para su primer parámetro (un tipo funcional). En general, el tipo del resultado devuelto por $f^{\mathcal{T}}$ cuando se aplica a m argumentos

depende de la relación entre m y la aridad de f. Por ejemplo tanto (map (plus z)) como (map plus) son expresiones bien tipadas en el programa original, pero si las transformamos directamente a ($\mathsf{map}^{\mathcal{T}}$ ($\mathsf{plus}^{\mathcal{T}}$ z)) y ($\mathsf{map}^{\mathcal{T}}$ $\mathsf{plus}^{\mathcal{T}}$) respectivamente, tendremos que la primera continua estando bien tipada mientras que la segunda no.

Como posible solución a este problema, los autores de [65] sugieren modificar la traducción de forma que una función currificada de aridad $n > 0$ siempre devuelva un resultado de tipo ($\tau^{\mathcal{T}}$, cTree) al aplicarse a su primer parámetro. El tipo transformado de la función plus, siguiendo esta idea, sería $\mathsf{plus}^{\mathcal{T}}$:: nat → (nat → (nat, cTree), cTree).

Sin embargo, tal y como se señala en el mismo [65], tal transformación daría lugar a programas transformados que producen cómputos muy ineficientes, además de producir árboles de prueba con gran cantidad de nodos inútiles. Por eso los autores de [65] afirman: *"Sería deseable disponer de una transformación intermedia, que sólo se ocupe de las funciones currificadas cuando resulte necesario. Se está investigando si es posible hacer esto sin efectuar análisis detallados del programa"*.

Nuestra transformación de programas resuelve este problema, transformando cada función currificada f de aridad n, en n funciones currificadas $f_0^{\mathcal{T}}, \ldots, f_{n-2}^{\mathcal{T}}, f^{\mathcal{T}}$ con aridades 1, 2, $\ldots n-1$ y n, respectivamente, y con tipos adecuados. La función $f_m^{\mathcal{T}}$ $(0 \leq m \leq n-2)$ se usará para transformar las apariciones de f aplicada a m parámetros, mientras que $\mathrm{f}^{\mathcal{T}}$ traducirá las apariciones de f aplicada a $n-1$ parámetros. Por ejemplo, (map plus) quedará transformado como ($\mathsf{map}^{\mathcal{T}}$ $\mathsf{plus_0}^{\mathcal{T}}$), utilizando la función auxiliar $\mathsf{plus_0}^{\mathcal{T}}$:: nat → (nat → (nat,cTree), cTree). La aplicación de una función f de aridad n a n o más parámetros deberá ser transformada con la ayuda de definiciones locales, una técnica ya empleada en [65, 71, 78]. En nuestra propuesta esto se hará durante la fase previa de aplanamiento descrita en la siguiente sección.

La solución será similar para el caso de la aplicación parcial de constructoras de datos currificadas, las cuales también pueden provocar errores de tipo si se transforman directamente, sin la ayuda de funciones auxiliares (piénsese por ejemplo en ($\mathsf{twice}^{\mathcal{T}}$ suc), con suc una constructora de aridad 1). Hasta donde sabemos este problema no había sido señalado hasta nuestros trabajos [17, 19].

Nuestra propuesta incrementa el número de funciones en el programa transformado, pero las funciones auxiliares son utilizadas sólo cuando son necesarias para evitar errores de tipo como los señalados anteriormente. En [76] Bernard Pope y Lee Naish comentan nuestra solución y proponen una mejora que reduce la cantidad de funciones auxiliares a utilizar, y por tanto el tamaño del programa generado. La idea consiste en utilizar una única función auxiliar aux_m para transformar todas las aplicaciones de funciones f de aridad n a m parámetros con $m < n$. Nuestra experiencia es que en la eficiencia del depurador no influye el tamaño del programa generado sino el coste (tanto en memoria como en tiempo) requerido por la generación del árbol, y éste no varía con la propuesta de [76].

4.3. Primera Fase: Aplanamiento

Tal y como comentamos en la introducción de este capítulo la transformación de programas que proponemos puede dividirse en dos fases:

1. Un aplanamiento inicial del programa. El programa aplanado carecerá de llamadas anidadas pero conservará una semántica equivalente al programa original, en un sentido que precisaremos en los teoremas 4.3.4 y 4.3.5.

2. Introducción de funciones auxiliares y transformación para devolver los árboles de cómputo. Esta segunda fase partirá de un programa que, gracias al punto anterior, supondremos previamente aplanado.

En esta sección explicamos la primera etapa, dejando para la sección 4.4 la descripción y los resultados teóricos relacionados con la segunda parte de la transformación.

4.3.1. Programas Planos

Intuitivamente el aplanamiento de un programa consiste en la eliminación de las llamadas a función anidadas, tanto del lado derecho como de las condiciones de cada regla de programa. Nuestra transformación de aplanamiento además lleva todas las llamadas de función a definiciones locales, asegurando que el lado derecho de la regla en un programa plano es siempre un patrón. Como veremos enseguida, esta transformación puede llevarse a cabo sin alterar la semántica mediante la introducción de nuevas definiciones locales. La idea del aplanamiento no es nueva; puede encontrarse ya en trabajos de 15 años atrás, como [10]. Esta técnica jugaba un papel importante en la semántica operacional del lenguaje K-LEAF, uno de los primeros lenguajes lógico-funcionales [34]. La aportación de nuestra propuesta consiste en probar formalmente la equivalencia semántica del programa aplanado con respecto al programa original, además de su utilización como etapa inicial para la transformación destinada a la obtención de árboles de cómputo.

La definición formal de programa plano es la siguiente:

Definición 4.3.1.

- Decimos que un programa P es un *programa plano* cuando toda regla de programa $(f\ \bar{t}_n \to r \Leftarrow C) \in P$ verifica:
  - $r \in Pat_\perp$.
  - C es una condición plana.
- Una condición C es una *condición plana* si cada una de sus componentes atómicas es de una de las dos formas siguientes:
  - $t == t' \in C$ con $t, t' \in Pat_\perp$.
  - $e \to s$, con $e \in Exp_\perp$, $s \in Pat_\perp$, y donde e es una expresión plana.

- Una expresión e es una *expresión plana* si es de una de las siguientes formas:
  - $e \equiv t$, con $t \in Pat_\perp$.
  - $e \equiv f\ \bar{t}_n$ con $f \in FS^n$ y $t_i \in Pat_\perp$ para $i = 1 \ldots n$.
  - $e \equiv X\ t$, con $X \in Var, t \in Pat_\perp$.

  Utilizaremos el término *llamada* para designar a las expresiones planas que no son patrones.

Por ejemplo, el programa 3.1.1 (pág. 52) no es un programa plano, ya que contiene reglas de programa que no cumplen los criterios de la definición 4.3.1, como:

- map F (X:Xs)= F X : map F Xs
- twice F X = F (F X)
- times (suc X) Y = plus (times X Y) X
- drop4 = twice twice tail

4.3.2. Cálculo de Aplanamiento

La transformación de programas para la obtención de APA's descrita en la siguiente sección partirá de programas planos. Por ello debemos definir una transformación de *aplanamiento* que pueda emplearse para convertir un programa que no es plano en un programa plano equivalente. Este el objetivo de la trasformación $\Rightarrow_{\mathcal{A}}$ definida en la figura 4.1, en la que se basa la siguiente definición:

Definición 4.3.2.

- Sean R, $R_{\mathcal{A}}$ dos reglas de programa tales que $R \Rightarrow_{\mathcal{A}} R_{\mathcal{A}}$. Diremos que $R_{\mathcal{A}}$ es el aplanamiento de R, o también que $R_{\mathcal{A}}$ se ha obtenido de R por aplanamiento.
- Sea un programa P con signatura Σ. Se llama aplanamiento de P al programa $P_{\mathcal{A}}$ con signatura Σ que se obtiene al aplicar la transformación $\Rightarrow_{\mathcal{A}}$ a todas las reglas de P. También diremos en este caso que $P_{\mathcal{A}}$ se ha obtenido de P por aplanamiento, y utilizaremos la notación $P \Rightarrow_{\mathcal{A}} P_{\mathcal{A}}$ para expresar la relación entre ambos programas.

Según indica la figura 4.1, para aplanar una regla de programa debemos aplanar tanto su parte derecha (una expresión) como sus condiciones. Sin embargo, el aplanamiento de una expresión e no es, en general, otra expresión en el sentido definido en la sección 3.1.2, sino una expresión u acompañada por un conjunto de definiciones locales D, lo que queda reflejado en la notación $e \Rightarrow_{\mathcal{A}} (u\ ;\ D)$. Cada una de las definiciones locales de D será una aproximación de la forma $a \rightarrow X$ con $X \in Var$ y $a \in Exp_\perp$, y pasará a ser parte de la condición de la regla transformada. El aplanamiento de expresiones está dividido en casos (reglas (PL_2)-(PL_6)), de forma que a toda expresión le corresponde una única regla de

Aplanamiento de una regla de función:

$$(\mathbf{PL_1}) \quad \frac{r \Rightarrow_{\mathcal{A}} (u \ ; \ C_r) \qquad C \Rightarrow_{\mathcal{A}} C_{\mathcal{A}}}{f \ \overline{t}_n \to r \Leftarrow C \quad \Rightarrow_{\mathcal{A}} \quad f \ \overline{t}_n \to u \Leftarrow C_{\mathcal{A}}, C_r}$$

Aplanamiento de una expresión:

$$(\mathbf{PL_2}) \quad \bot \Rightarrow_{\mathcal{A}} \ (\bot \ ; \ )$$

$$(\mathbf{PL_3}) \quad X \ \Rightarrow_{\mathcal{A}} \ (X \ ; \ ) \text{ para } X \in Var$$

$$(\mathbf{PL_4}) \quad \frac{a_1 \Rightarrow_{\mathcal{A}} (u_1 \ ; C_1) \qquad R \ a_2 \ldots a_k \Rightarrow_{\mathcal{A}} (u \ ; C)}{X \ \overline{a}_k \ \Rightarrow_{\mathcal{A}} \ (u \ ; C_1, \ X \ u_1 \to R, C)} \qquad \begin{array}{l} X \ \overline{a}_k \text{ flexible }, k > 0, \\ R \text{ nueva variable} \end{array}$$

$$(\mathbf{PL_5}) \quad \frac{e_1 \Rightarrow_{\mathcal{A}} (u_1 \ ; C_1) \ \ldots \ e_m \Rightarrow_{\mathcal{A}} (u_m \ ; C_m)}{h \ \overline{e}_m \ \Rightarrow_{\mathcal{A}} \ (h \ \overline{u}_m \ ; C_1, \ldots, C_m)} \qquad h \ \overline{e}_m \text{ rígido y pasivo}$$

$$(\mathbf{PL_6}) \quad \frac{e_1 \Rightarrow_{\mathcal{A}} \ (u_1 \ ; C_1) \quad \ldots \quad e_n \Rightarrow_{\mathcal{A}} \ (u_n \ ; C_n) \quad S \ \overline{a}_k \Rightarrow_{\mathcal{A}} \ (u \ ; D)}{f \ \overline{e}_n \ \overline{a}_k \quad \Rightarrow_{\mathcal{A}} \quad (u \ ; C_1, \ldots, C_n, f \ \overline{u}_n \to S, D)}$$

para $f \ \overline{e}_n \ \overline{a}_k$ expresión rígida y activa, $n \geq 0, k \geq 0$,
$f \in FS^n$, S variable nueva

Aplanamiento de condiciones:

$$(\mathbf{PL_7}) \quad \frac{C_1 \ \Rightarrow_{\mathcal{A}} \ D_1 \qquad C_2 \ \Rightarrow_{\mathcal{A}} \ D_2}{C_1, C_2 \ \Rightarrow_{\mathcal{A}} \ D_1, D_2}$$

$$(\mathbf{PL_8}) \quad \frac{l \ \Rightarrow_{\mathcal{A}} \ (u_1 \ ; C_1) \qquad r \ \Rightarrow_{\mathcal{A}} \ (u_2 \ ; C_2)}{l == r \ \Rightarrow_{\mathcal{A}} \ C_1, C_2, u_1 == u_2}$$

$$(\mathbf{PL_9}) \quad \frac{e \ \Rightarrow_{\mathcal{A}} \ (u \ ; C)}{e \to s \ \Rightarrow_{\mathcal{A}} \ C, u \to s} \qquad \text{si } e \text{ no es una expresión plana}$$

$$(\mathbf{PL_{10}}) \quad \frac{}{e \to s \ \Rightarrow_{\mathcal{A}} \ e \to s} \qquad \text{si } e \text{ ya es una expresión plana}$$

Figura 4.1: Reglas del Cálculo de Aplanamiento

aplanamiento. Finalmente, las condiciones se tratan mediante cuatro reglas. La regla (PL_7) para condiciones no atómicas, que indica que deben aplanarse las subcondiciones atómicas una a una, dos reglas para los dos tipos de condiciones atómicas: igualdades estrictas (PL_8), y aproximaciones (PL_9), además de la regla (PL_{10}), necesaria para que se cumpla la (deseable) propiedad de que el resultado de aplanar un programa plano es el mismo programa, tal y como establecerá la proposición 4.3.2. Cuando una regla dependa de varias premisas (como ocurre por ejemplo con PL_5) se sobrentenderemos que el aplanamiento efectuado por cada premisa utiliza variables nuevas diferentes. Esto se podría formalizar arrastrando en todas las reglas del cálculo un "conjunto de variables nuevas no usadas aún", pero quedaría una presentación engorrosa que hemos preferido evitar.

Es interesante resaltar que, al contrario de lo que sucede con las expresiones, el resultado de aplanar una condición sí es una nueva condición.

Las reglas (PL_1)-(PL_{10}) constituyen un método efectivo de aplanamiento. Por ejemplo, aplicando la transformación de aplanamiento a la regla

$$\mathsf{times\ (suc\ X)\ Y} \rightarrow \mathsf{plus\ (times\ X\ Y)\ X}$$

del ejemplo 3.1.1 obtenemos:

$$\dfrac{\dfrac{\dfrac{\mathrm{X} \Rightarrow_{\mathcal{A}} (\ X;\ )\quad \mathrm{Y} \Rightarrow_{\mathcal{A}} (\ Y;\ )\quad S_2 \Rightarrow_{\mathcal{A}} (\ S_2;\ )}{(\mathrm{PL}_6)\quad \text{times X Y} \Rightarrow_{\mathcal{A}} (\ S_2;\ \text{times X Y} \rightarrow S_2)} \quad \mathrm{X} \Rightarrow_{\mathcal{A}} (\ X;\ )\quad S_1 \Rightarrow_{\mathcal{A}} (\ S_1;\ )}{(\mathrm{PL}_6)\quad \text{plus (times X Y) X} \Rightarrow_{\mathcal{A}} (S_1\ ;\ \text{times X Y} \rightarrow S_2,\ \text{plus } S_2\ \mathrm{X} \rightarrow S_1)}}{(\mathrm{PL}_1)\quad \text{times (suc X) Y} \rightarrow \text{plus (times X Y) X} \Rightarrow_{\mathcal{A}} \text{times (suc X) Y} \rightarrow S_1 \Leftarrow \text{times X Y} \rightarrow S_2,\ \text{plus } S_2\ \mathrm{X} \rightarrow S_1}$$

Las reglas que no aparecen etiquetadas corresponden a aplicaciones de la regla (PL_3). La regla resultante queda, utilizando la notación de $\mathcal{TOY}$:

```
times (suc X) Y = S1
                  where S2 = times X Y
                        S1 = plus S2 X
```

En este ejemplo se aprecia que las reglas de aplanamiento trasladan todas las llamadas a definiciones locales de la forma $e \rightarrow X$, cuya parte izquierda es expresión plana y cuya parte derecha es una variable variable, o más en general un patrón. La siguiente proposición establece con precisión la estructura de las reglas del programa plano con respecto al programa original.

Proposición 4.3.1. *Sea P un programa y $P_{\mathcal{A}}$ su aplanamiento. Sean también dos reglas de programa (R) $(f\ \bar{t}_n \rightarrow r \Leftarrow C) \in P$, $(R_{\mathcal{A}})$ $(f\ \bar{t}_n \rightarrow r_{\mathcal{A}} \Leftarrow C_{\mathcal{A}}) \in P_{\mathcal{A}}$ tales que $R\ \Rightarrow_{\mathcal{A}}\ R_{\mathcal{A}}$. Entonces se cumple:*

(a) $r_{\mathcal{A}} \in Pat_{\perp}$.

(b) Cada condición atómica en $C_{\mathcal{A}}$ ha de ser de una de las siguientes formas:

- *Una igualdad estricta de la forma $t == t'$, con $t, t' \in Pat_{\perp}$.*
- *Una aproximación $e \to s$, con e una expresión plana, donde a su vez existen tres posibilidades:*
  - *$s \equiv X$, $X \in Var$, con X una variable nueva introducida por las regla (PL_4) o (PL_6), $X \notin var(R)$, y $e \in Exp_{\perp}$ una expresión plana. Estas son las aproximaciones nuevas introducidas por el aplanamiento.*
  - *$s, e \in Pat_{\perp}$ tal que para algún e' se tiene $e' \to s \in C$. Estas aproximaciones provienen por tanto de la transformación de una aproximación con lado izquierdo no plano del programa original.*
  - *$e \to s \in C$, es decir una aproximación que ya era parte del programa original.*

Demostración
Por inducción sobre la estructura de la demostración de $R \Rightarrow_{\mathcal{A}} R_{\mathcal{A}}$. Véase pág. 191, apéndice A.

La siguiente proposición indica que la transformación de aplanamiento deja inalterados los programas que ya eran planos y viceversa.

Proposición 4.3.2. *Sea P un programa. Entonces $P \Rightarrow_{\mathcal{A}} P$ sii P es un programa plano.*

Demostración
Véase pág. 194, apéndice A.

Por ejemplo la transformación de aplanamiento aplicada al ejemplo 4.2.1 (pág. 72) produciría el el siguiente programa plano:

Ejemplo 4.3.1. *Aplanamiento del ejemplo 4.2.1.*

```
data nat = z | suc nat

plus :: nat -> nat -> nat
plus z       Y = Y
plus (suc X) Y = suc U
                 where U = plus X Y
twice :: (A -> A) -> A -> A
twice F X = V
            where U = F X
                  V = F U

drop4 :: [A] -> [A]
drop4 = U
        where U = twice twice tail
```

```
from :: nat -> [nat]
from N = N: U
    where U = from  N

head :: [A] -> A
head (X:Xs) = X

tail :: [A] -> [A]
tail (X:Xs) = Xs

map :: (A -> B) -> [A] -> [B]
map F []    = []
map F (X:Xs)= U : V
              where U = F X
                    V = map F Xs
```

4.3.3. Corrección del Aplanamiento

Acabamos de ver en el apartado anterior que las reglas de aplanamiento propuestas sirven en efecto para aplanar programas lógico-funcionales. Sin embargo no hemos verificado aún que en el programa aplanado se puedan probar los mismos objetivos que en el programa original, ni tampoco que el programa aplanado sea correcto desde el punto de de los tipos. Estos resultados se establecen a continuación.

Comenzamos comprobando que el aplanamiento de programas correctamente tipados conduce a programas correctamente tipados:

Teorema 4.3.3. *Sea P un programa bien tipado y Σ su signatura. Sea $P_{\mathcal{A}}$ un programa tal que $P \Rightarrow_{\mathcal{A}} P_{\mathcal{A}}$. Entonces $P_{\mathcal{A}}$ es un programa bien tipado con signatura Σ.*

Demostración
Ver pág. 197, apéndice A.

Seguidamente debemos comprobar la equivalencia semántica entre el programa original y el programa aplanado. La idea intuitiva tras el siguiente resultado es que un programa P es capaz de computar una respuesta θ para un objetivo dado si y sólo si su aplanamiento $_{\mathcal{A}}$ es capaz de computar una respuesta θ' para el objetivo aplanado, donde θ' es una extensión de θ a las variables nuevas introducidas durante el aplanamiento.

Teorema 4.3.4. *(Corrección semántica del aplanamiento)*
Sean P y $P_{\mathcal{A}}$ dos programas tales que $P \Rightarrow_{\mathcal{A}} P_{\mathcal{A}}$. Sea $\theta \in Subst_{\perp}$. Entonces:

a) Las expresiones aplanadas son correctas en el siguiente sentido:

Sean $e, e_{\mathcal{A}}$ expresiones y $C_{\mathcal{A}}$ una condición tales que $e \Rightarrow_{\mathcal{A}} (e_{\mathcal{A}}\,;\, C_{\mathcal{A}})$. Sea $V_{\mathcal{A}}$ el conjunto de variables nuevas introducidas durante el aplanamiento de e que podemos

suponer distintas de las variables en θ, es decir

$$V_{\mathcal{A}} \cap (dom(\theta) \cup ran(\theta)) = \emptyset$$

Entonces para todo patrón parcial t se cumple

$$P \vdash_{SC} e\theta \rightarrow t \quad sii \quad P_{\mathcal{A}} \vdash_{SC} (e_{\mathcal{A}}(\theta \biguplus \nu) \rightarrow t,\ C_{\mathcal{A}}(\theta \biguplus \nu))$$

para algún $\nu \in Subst_{\perp}$ con $dom(\nu) \subseteq V_{\mathcal{A}}$. Además ambas pruebas admiten APAs que sólo difieren en la raíz.

b) Las condiciones aplanadas son también correctas en el siguiente sentido:

Sean $C, C_{\mathcal{A}}$ dos condiciones tales que $C \Rightarrow_{\mathcal{A}} C_{\mathcal{A}}$. Sea $V_{\mathcal{A}}$ el conjunto de variables nuevas introducidas durante el aplanamiento de C que podemos suponer disjunto con las variables en θ. Entonces:

$$P \vdash_{SC} C\theta \quad sii \quad P_{\mathcal{A}} \vdash_{SC} C_{\mathcal{A}}(\theta \biguplus \nu)$$

para algún $\nu \in Subst_{\perp}$ con $dom(\nu) \subseteq V_{\mathcal{A}}$. Además ambas pruebas admiten APAs que sólo difieren en la raíz.

Demostración
Ver pág. 205, apéndice A.

El siguiente resultado establece que el programa aplanado también es capaz de probar las mismas condiciones (atómicas o no) que el programa original, aunque éstas no hayan sido aplanadas previamente:

Teorema 4.3.5. *Sean P y $P_{\mathcal{A}}$ dos programas tales que $P \Rightarrow_{\mathcal{A}} P_{\mathcal{A}}$ y sea C una condición cualquiera. Entonces:*

$$P \vdash_{SC} C \quad sii \quad P_{\mathcal{A}} \vdash_{SC} C$$

Demostración
Este resultado es una consecuencia directa del anterior. Su demostración se puede consultar en la pág. 233, apéndice A.

Por tanto en el ejemplo anterior podemos lanzar directamente el objetivo sin aplanar (head (drop4 (map plus (from z)))) (suc z) == X con respecto al programa aplanado 4.3.1, y obtendremos la misma respuesta { X ↦ suc z }.

4.4. Segunda Fase: Generación de los Árboles de Cómputo

Acabamos de ver como todo programa P se puede transformar en un programa aplanado equivalente $P_{\mathcal{A}}$. En esta sección mostramos como transformar el programa aplanado $P_{\mathcal{A}}$ en un nuevo programa, que denotaremos por $P^{\mathcal{T}}$, en el que cada función devolverá junto a su resultado original un árbol de cómputo, y probaremos la corrección de la transformación.

4.4.1. Representación de los Árboles de Cómputo

Todo programa transformado incluirá la definición del tipo de datos cTree, utilizado para la representación de los árboles de cómputo (los APAs) y definido de la siguiente forma:

```
data cTree        =  ctVoid | ctNode funId [arg] res [cTree]
type arg, res     =  pVal
type funId, pVal  =  string
```

Un árbol de cómputo de la forma (ctNode f ts s rl cts) se corresponde con una llamada a una función f con argumentos ts y resultado s, donde rl indica la regla de función utilizada para evaluar la llamada, mientras que la lista cts está compuesta por los árboles hijos asociados a todas las llamadas (de las definiciones locales, del lado derecho y de las condiciones de la regla utilizada) que han resultado necesarias para evaluar s. puede que el cómputo principal, debido a la evaluación perezosa, sólo necesite aproximaciones parciales de los resultados de los cómputos intermedios. Por tanto, tanto ts como s representan valores posiblemente parciales, representados mediante patrones parciales; y (f ts → s) representa el *hecho básico* sobre cuya validez se preguntará al oráculo durante la depuración.

La constructora ctVoid se utiliza para representar un árbol de cómputo vacío, y será devuelta por funciones de las que se puede asegurar a priori que no contienen ningún error (en particular, las primitivas y las funciones auxiliares introducidas durante la transformación). Finalmente, la definición de arg, res, funId, pVal y rule son sinónimos del tipo cadena de caracteres, tipo que hemos elegido por simplicidad. De hecho, nuestros depuradores declarativos, incluidos en los sistemas $\mathcal{TOY}$ y Curry, utilizan una representación más estructurada en el lugar de cadenas de caracteres. En concreto, los valores del tipo rule representan instancias de reglas del programa que permiten mostrar al usuario la instancia de regla asociada a un nodo crítico. Esta representación más estructurada será discutida en el capítulo 5.

4.4.2. Transformación de la Signatura de los Programas

Como hemos visto en la sección 4.3 la signatura del programa aplanado $P_{\mathcal{A}}$ coincide con la del programa inicial. En cambio esto no sucederá en la segunda fase de la transformación. Si el programa plano $P_{\mathcal{A}}$ (y por tanto P) tiene una signatura $\Sigma = \{TC,\ DC,\ FS\}$, el programa transformado $P^{\mathcal{T}}$ tendrá una signatura $\Sigma^{\mathcal{T}}$ definida como:

$$
\begin{aligned}
\Sigma^{\mathcal{T}} = \{\ & TC \uplus \{cTree\}, \\
& DC \uplus \{ctVoid :: cTree, \\
& \qquad ctNode :: string \to [pVal] \to pVal \to string \to [cTree] \to cTree\}, \\
& FS' \uplus FS_{aux} \cup \{\, ctClean :: [(pVal, cTree)] \to [cTree]\,\}\ \}
\end{aligned}
$$

El símbolo de *unión disjunta* $\uplus$ se utiliza aquí para indicar que no hay coincidencia de nombres entre los símbolos definidos en los dos conjuntos cuya unión se considera. Supondremos que el depurador genera/modifica los nombres de los nuevos símbolos de forma

que se verifica esta propiedad en cada caso. Por simplicidad conservaremos el nombre de la función original para la función transformada y haremos lo mismo con los símbolos de constructora.

La constructora de tipos cTree y las constructoras de datos ctNode y ctVoid ya han sido introducidos en la sección anterior y se utilizan para representar los árboles de cómputo. El papel de la función ctClean se explicará en la subsección 4.4.3.

Los conjuntos FS' y FS_{aux} se obtienen a partir de los símbolos de función FS y constructoras de datos DC de la signatura de $P_{\mathcal{A}}$ según se indica a continuación:

- Por cada función n-aria $f \ :: \ \tau_1 \to \cdots \to \tau_n \to \tau$ de FS, se incluye en FS' una función $f^{\mathcal{T}}$ de aridad n y tipo principal:

  $$f^{\mathcal{T}} \ :: \ \tau_1^{\mathcal{T}} \to \cdots \to \tau_n^{\mathcal{T}} \to (\tau^{\mathcal{T}}, cTree)$$

  y en FS_{aux} $n-1$ funciones auxiliares $f_m^{\mathcal{T}}$ con $0 \leq m < n-1$, donde cada $f_m^{\mathcal{T}}$ es de aridad $(m+1)$ y tiene tipo principal:

  $$f_m^{\mathcal{T}} \ :: \ \tau_1^{\mathcal{T}} \to \cdots \to \tau_{m+1}^{\mathcal{T}} \to ((\tau_{m+2} \to \cdots \to \tau_n \to \tau)^{\mathcal{T}}, cTree)$$

  donde la transformación de tipos $^{\mathcal{T}}$, de la que ya hablamos en la sección 4.2 de este capítulo, se define como:

  $$\begin{array}{lll} \alpha^{\mathcal{T}} & = \alpha & (\alpha \in TVar) \\ (C\ \overline{\tau}_n)^{\mathcal{T}} & = C\ \overline{\tau}_n^{\mathcal{T}} & (C \in TC^n) \\ (\mu \to \nu)^{\mathcal{T}} & = \mu^{\mathcal{T}} \to (\nu^{\mathcal{T}}, cTree) & \end{array}$$

- Análogamente, por cada constructora de datos $c \ :: \ \tau_1 \to \cdots \to \tau_n \to \tau$ de aridad n en DC, se incluirán en FS_{aux} n funciones auxiliares $c_m^{\mathcal{T}}$ con $0 \leq m < n$, donde cada función c_m tiene aridad $(m+1)$ y tipo principal:

  $$c_m^{\mathcal{T}} \ :: \ \tau_1^{\mathcal{T}} \to \cdots \to \tau_{m+1}^{\mathcal{T}} \to ((\tau_{m+2} \to \cdots \to \tau_n \to \tau)^{\mathcal{T}}, cTree)$$

  Obsérvese que $\tau^{\mathcal{T}}$ es igual a τ siempre que τ sea un tipo que no incluya la constructora de tipos de orden superior "$\to$". Como esto sucede en el tipo principal de los argumentos y del resultado de las constructoras en $\mathcal{TOY}$, el tipo principal de las funciones auxiliares $c_m^{\mathcal{T}}$ será:

  $$c_m^{\mathcal{T}} \ :: \ \tau_1 \to \cdots \to \tau_{m+1} \to ((\tau_{m+2} \to \cdots \to \tau_n \to \tau)^{\mathcal{T}}, cTree)$$

4.4.3. Transformación de las Reglas de Programa

Según hemos visto en el apartado anterior podemos dividir los símbolos de función del programa transformado en tres apartados: las funciones auxiliares del conjunto FS_{aux}, las funciones transformadas en FS' y la función auxiliar ctClean. La definición de las reglas de programa de cada símbolo de función se hace de la siguiente forma:

Transformación de una regla de función:

$$(\mathbf{TR_1}) \quad \frac{t_1 \Rightarrow^{\mathcal{T}} s_1 \ \ldots \ t_n \Rightarrow^{\mathcal{T}} s_n \quad r \Rightarrow^{\mathcal{T}} s \quad C \Rightarrow^{\mathcal{T}} (C^{\mathcal{T}}\,;\,(r_1,T_1),\ldots,(r_m,T_m))}{\begin{array}{l} f\ \bar{t}_n \to r \Leftarrow C \ \Rightarrow^{\mathcal{T}} \\ \quad f^{\mathcal{T}}\ \bar{s}_n \ \to (s,T) \Leftarrow C^{\mathcal{T}},\ ctNode\ "f.k"\ \ [dVal\, s_1,\ldots,\ dVal\, s_n]\ \ (dVal\, s) \\ \qquad\qquad (ctClean\ [(dVal\, r_1,T_1),\ldots,\ (dVal\, r_m,T_m)]) \to T \end{array}}$$

donde $f\ \bar{t}_n \to r \Leftarrow C$ es la k-ésima regla de f en $P_{\mathcal{A}}$, T variable nueva

Transformación de un patrón:

$$(\mathbf{TR_2}) \quad \frac{}{\bot \Rightarrow^{\mathcal{T}} \bot}$$

$$(\mathbf{TR_3}) \quad \frac{}{X \ \Rightarrow^{\mathcal{T}} \ X}$$

Si X ∈ Var

$$(\mathbf{TR_4}) \quad \frac{t_1 \ \Rightarrow^{\mathcal{T}} \ s_1 \ \ldots \ t_m \Rightarrow^{\mathcal{T}} \ s_m}{h\ \bar{t}_m \ \Rightarrow^{\mathcal{T}} \ h\ \bar{s}_m}$$

Para $h \in DC^m$

$$(\mathbf{TR_5}) \quad \frac{t_1 \ \Rightarrow^{\mathcal{T}} \ s_1 \ \ldots \ t_m \Rightarrow^{\mathcal{T}} \ s_m}{h\ \bar{t}_m \ \Rightarrow^{\mathcal{T}} \ h_m^{\mathcal{T}}\ \bar{s}_m}$$

Para $h \in DC^n$ con $n > m$, o $h \in FS^n$, con $n > m+1$

$$(\mathbf{TR_6}) \quad \frac{t_1 \ \Rightarrow^{\mathcal{T}} \ s_1 \ \ldots \ t_m \Rightarrow^{\mathcal{T}} \ s_m}{f\ \bar{t}_m \ \Rightarrow^{\mathcal{T}} \ f^{\mathcal{T}}\ \bar{s}_m}$$

Para $f \in FS^{m+1}$

Transformación de condiciones:

$$(\mathbf{TR_7}) \quad \frac{C_1 \Rightarrow^{\mathcal{T}} (D_1\,;\,(r_1,T_1),\ldots,(r_k,T_k)) \quad C_2 \Rightarrow^{\mathcal{T}} (D_2\,;\,(r_{k+1},T_{k+1}),\ldots,(r_m,T_m))}{C_1, C_2 \ \Rightarrow^{\mathcal{T}} \ (D_1, D_2\,;\,(r_1,T_1),\ \ldots,\ (r_m,T_m))}$$

$$(\mathbf{TR_8}) \quad \frac{l \ \Rightarrow^{\mathcal{T}} \ u \quad r \ \Rightarrow^{\mathcal{T}} \ v}{l == r \Rightarrow^{\mathcal{T}} (u == v\,;\,)}$$

$$(\mathbf{TR_9}) \quad \frac{e \Rightarrow^{\mathcal{T}} u \quad t \Rightarrow^{\mathcal{T}} v}{e \to t \Rightarrow^{\mathcal{T}} (u \to v\,;\,)}$$

Si $e \in Pat_{\bot}$

$$(\mathbf{TR_{10}}) \quad \frac{s \Rightarrow^{\mathcal{T}} u \qquad t \Rightarrow^{\mathcal{T}} v}{X\ s \to t \ \Rightarrow^{\mathcal{T}} \ (X\ u \to (v,T)\,;\,(v,T))}$$

Si $X \in Var,\ T$ variable nueva

$$(\mathbf{TR_{11}}) \quad \frac{t_1 \Rightarrow^{\mathcal{T}} s_1 \ \ldots \ t_n \Rightarrow^{\mathcal{T}} s_n \qquad t \Rightarrow^{\mathcal{T}} v}{f\ \bar{t}_n \to t \ \Rightarrow^{\mathcal{T}} \ (f^{\mathcal{T}}\ \bar{s}_n \to (v,T)\,;\,(v,T))}$$

Si $f \in FS^n,\ T$ variable nueva

Figura 4.2: Reglas del Cálculo de Introducción de Árboles de Cómputo

- Cada función auxiliar $f_m^{\mathcal{T}}$ precisa $m+1$ argumentos y devuelve una aplicación parcial de la función $f_{m+1}^{\mathcal{T}}$ formando par con el árbol de cómputo ctVoid. Excepcionalmente $f_{n-2}^{\mathcal{T}}$ devolverá una aplicación parcial de f. Las funciones auxiliares $c_m^{\mathcal{T}}$ se definen de forma similar, salvo porque $c_{n-1}^{\mathcal{T}}$ devuelve una aplicación total de la constructora de datos c:

$$\begin{array}{lcllcl}
f_0^{\mathcal{T}}\ X_1 & \to & (f_1^{\mathcal{T}}\ X_1, \mathrm{ctVoid}) & c_0^{\mathcal{T}}\ X_1 & \to & (c_1^{\mathcal{T}}\ X_1, \mathrm{ctVoid}) \\
f_1^{\mathcal{T}}\ X_1\ X_2 & \to & (f_2^{\mathcal{T}}\ X_1\ X_2, \mathrm{ctVoid}) & c_1^{\mathcal{T}}\ X_1\ X_2 & \to & (c_2^{\mathcal{T}}\ X_1\ X_2, \mathrm{ctVoid}) \\
\ldots & & & \ldots & & \\
f_{n-2}^{\mathcal{T}}\ \overline{X}_{n-1} & \to & (f\ \overline{X}_{n-1}, \mathrm{ctVoid}) & c_{n-1}^{\mathcal{T}}\ \overline{X}_n & \to & (c\ \overline{X}_n, \mathrm{ctVoid})
\end{array}$$

  El propósito de estas funciones auxiliares será, como veremos en el siguiente punto, sustituir a las aplicaciones parciales de funciones y constructoras, que en el programa transformado también deben devolver un par formado por un resultado y un árbol de cómputo tal y como comentamos en la sección 4.2.

- A cada regla de programa R de $P_{\mathcal{A}}$ le corresponderá una regla de programa $R^{\mathcal{T}}$ tal que $R \Rightarrow^{\mathcal{T}} R^{\mathcal{T}}$, donde la transformación $\Rightarrow^{\mathcal{T}}$ está definida en la figura 4.2.

  Obsérvese que la regla (TR_1) de esta figura sólo se pueden aplicar a reglas de programas planos, por lo que debido a las propiedades de los programas planos, y en particular por la forma de las condiciones de las reglas de programas planos establecida en la proposición 4.3.1, se tiene que en estas reglas todos los símbolos t_i, s_i, u, v, r, l, s representan patrones.

  Como se puede ver en la regla (TR_1), la regla transformada devolverá pares de la forma (s,T), donde s proviene de la transformación del lado derecho de la regla original en $P_{\mathcal{A}}$ y T es un valor de tipo cTree. La definición de T puede verse en la misma regla, en forma de definición local (más abajo discutiremos detalladamente las reglas (TR_2)-(TR_{11})).

  El árbol devuelto por una regla de función transformada siempre se construye mediante la constructora ctNode. El primer argumento de la constructora es el identificador de la regla, para el que hemos usado el nombre de la regla seguido de un punto y un número denotando el orden de aparición de la regla con respecto a las otras reglas de la misma función en $P_{\mathcal{A}}$ (y por tanto en P).

  El segundo y tercer argumentos de ctNode son, respectivamente, las representaciones de los argumentos y del resultado producido por la función. Para construir estas representaciones se utiliza la función dVal :: A → pVal. Esta función, definida en forma de función primitiva en $\mathcal{TOY}$, es una función "impura"[1], es decir sin significado

[1] En cualquier caso, el comportamiento de dVal necesario para garantizar la corrección de la técnica puede ser especificado formalmente, como se verá en la definición del cálculo $dValSC$, (def. 4.4.2, pág. 91).

declarativo, muy similar a la función dirt de [78, 76]. Una expresión de la forma (dVal a) devuelve una representación de la aproximación al valor a que se ha obtenido durante el cómputo principal. Para ello dVal reemplazará todas las llamadas que formen parte de a y que hayan sido evaluadas durante el cómputo por una representación del valor obtenido, mientras que las llamadas que no hayan sido evaluadas se reemplazarán por la cadena "_", representando el valor $\perp$. Así mismo dVal renombrará los identificadores de las funciones auxiliares $f_m^{\mathcal{T}}$ y $c_m^{\mathcal{T}}$ a f y c respectivamente. De esta forma los patrones que representen resultados computados se trasladarán a la signatura original.

Como último argumento de ctNode tenemos la lista de árboles hijos, que se corresponden, como vimos en el capítulo 3, con los APA's de cada una de las llamadas existentes en la regla, tanto en el lado derecho como en las condiciones. En este caso el lado derecho r no puede contener ninguna llamada (la proposición 4.3.1 nos asegura que se trata de un patrón), por lo que sólo necesitamos recopilar los árboles obtenidos en las llamadas incluidas en las condiciones. Estos árboles aparecen en la regla (TR_1) como $T_1, \ldots, T_m$. Sin embargo, en lugar de incluir la lista formada por estos árboles como cuarto argumento de ctNode, la regla (TR_1) aplica la función ctClean a una lista de pares de la forma $(dVal\ r_i, T_i)$, donde cada r_i se corresponde con el valor asociado a la raíz del correspondiente árbol T_i. Esto se hace así para eliminar tanto los árboles correspondientes a llamadas que no han sido evaluadas durante el cómputo (para las que $dVal\ r_i$ devolverá "_"), como los árboles triviales ctVoid producidos por las funciones auxiliares. La función ctClean, que describimos a continuación, se encarga de convertir esta lista de pares en una lista de árboles tal y como requiere el tipo de ctNode.

- La definición de la función ctClean es la siguiente:

```
ctClean:: [(pVal,cTree)]-> [cTree]
ctClean [ ] = [ ]
ctClean ((PVal,CTree):Ts) = if (PVal=="_")
                            then CleanTs
                            else if (CTree==ctVoid)
                                 then CleanTs
                                 else (CTree:CleanTs)
                            where CleanTs = ctClean Ts
```

La primera regla se encarga de las listas vacías y no requiere explicación. La segunda regla comprueba en primer lugar si el subcómputo asociado al árbol CTree ha sido requerido. Si no ha sido así (PVal=="_") el árbol se elimina de la lista y se procede a aplicar ctClean recursivamente al resto de la lista. Si el subcómputo ha sido requerido para el cómputo principal pero CTree está asociado a una función auxiliar (CTree==ctVoid), entonces también se elimina del árbol porque los nodos asociados a funciones auxiliares nunca pueden ser nodos críticos. En otro caso (subcómputo

requerido y no asociado a una función auxiliar) CTree pasa a formar parte de la lista resultado.

Vamos a examinar un poco más detalladamente las reglas de transformación de la figura 4.2. En primer lugar es interesante observar que, al contrario de lo que sucedía con las reglas de aplanamiento de la figura 4.1, no existen reglas para la transformación de expresiones generales sino sólo de patrones. Esto es consecuencia de la proposición 4.3.1 (pág. 78), que nos asegura que las únicas expresiones distintas de patrones en un programa aplanado aparecen en aproximaciones de la forma $X\ s \to t$ con $X \in Var$ y $t, s \in Pat_\perp$, o de la forma $f\ \bar{t}_n \to t$ con $f \in FS^n$ y $t_i \in Pat_\perp$ para todo $1 \leq i \leq n$, y ambas posibilidades son tratadas directamente por las reglas (TR_{10}) y (TR_{11}) respectivamente. El propósito de las reglas de transformación de patrones (TR_2)-(TR_6) es tan sólo la sustitución de las aplicaciones parciales por llamadas a las funciones auxiliares, sustitución que se aplica tanto a los argumentos (premisas $t_i \Rightarrow^{\mathcal{T}} s_i$ de (TR_1)), como al lado derecho (premisa $r \Rightarrow^{\mathcal{T}} s$ de (TR_1)) y a las condiciones (premisas de (TR_7)-(TR_{11})).

Las condiciones se transforman mediante las reglas (TR_7)-(TR_{11}). La transformación de una condición C produce un par de la forma (C', L), donde C' es la condición transformada y L es una secuencia de pares de la forma (r_i, T_i) donde T_i es un árbol de cómputo asociado a una llamada en C y r_i es el resultado del subcómputo cuyo APA viene representado por T_i. En particular (TR_7) se ocupa de las condiciones no atómicas, recopilando los resultados de transformar cada componente. (TR_8) se encarga de las igualdades estrictas, que como sabemos por la proposición 4.3.1 que están compuestas por patrones en el programa aplanado, por lo que la transformación sólo necesita introducir las funciones auxiliares allá donde sean precisas, devolviendo una secuencia vacía de pares *(valor,árbol)*. Algo similar sucede con la regla (TR_9), destinada al tratamiento de aproximaciones entre patrones. Finalmente, las reglas (TR_{10}) y (TR_{11}) se encargan, como hemos dicho más arriba, de las posibles aproximaciones que incluyan llamadas. Como en el programa transformado todas las llamadas devolverán pares *(valor,árbol)*, estas dos reglas introducen una nueva variable T en el lado derecho de la aproximación para recoger el valor de tipo cTree, además de devolver el par correspondiente como único elemento de la secuencia L.

Dado un patrón t a menudo utilizaremos la notación $t^{\mathcal{T}}$ para referirnos al patrón que se obtiene al transformar t según las reglas (TR_1)-(TR_{11}), es decir tal que se verifica $t \Rightarrow^{\mathcal{T}} t^{\mathcal{T}}$. Análogamente $C^{\mathcal{T}}$ representará la transformada de una condición plana C. Para indicar que el programa $P^{\mathcal{T}}$ se ha obtenido transformando el programa aplanado $P_{\mathcal{A}}$ utilizaremos la notación $P_{\mathcal{A}} \Rightarrow^{\mathcal{T}} P^{\mathcal{T}}$. Hay notar que en este caso, y a diferencia de lo que ocurría con la transformación de aplanamiento, la notación $P_{\mathcal{A}} \Rightarrow^{\mathcal{T}} P^{\mathcal{T}}$ no implica sólo la transformación de las reglas de programa de $P_{\mathcal{A}}$ mediante la aplicación de las reglas (TR_1)-(TR_{11}) de la figura 4.2, sino la transformación completa descrita en esta sección.

Por ejemplo, sea $P_{\mathcal{A}}$ el programa aplanado del ejemplo 4.3.1 (pág. 79). Entonces el programa $P^{\mathcal{T}}$ tal que $P_{\mathcal{A}} \Rightarrow^{\mathcal{T}} P^{\mathcal{T}}$ es el siguiente:

Ejemplo 4.4.1. *Transformación del programa del ejemplo 4.3.1.*

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% definiciones auxiliares
type funId, pVal = string
type arg, res = pVal

data cTree = ctVoid | ctNode funId [arg] res  [cTree]

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

data nat = z | suc nat

plus :: nat -> nat -> (nat, cTree)
plus z Y = (Y,T)
    where T = ctNode "plus.1" [dVal z, dVal Y]
              (dVal Y) (ctClean [])
plus (suc X) Y = (suc U, T)
        where (U,T1) = plus X Y
          T = ctNode "plus.2" [dVal (suc X),dVal Y]
             (dVal (suc U)) (ctClean [(dVal U,T1)])

twice :: (A -> (A,cTree)) -> A -> (A, cTree)
twice F X = (V,T)
    where (U, T1) = F X
              (V, T2) = F U
          T = ctNode "twice.1" [dVal F,dVal X] (dVal V)
                    (ctClean [(dVal U,T1), (dVal V, T2)])

drop4 :: ([A] -> ([A],cTree),cTree)
drop4 = (U,T)
        where (U,T1) = twice twice0 tail
          T = ctNode "drop4.1" [] (dVal U)
                    (ctClean [(dVal U,T1)])

from :: nat -> ([nat],cTree)
from N = (N: U, T)
    where (U,T1) = from  N
          T = ctNode "from.1" [dVal N] (dVal (N:U))
                    (ctClean [(dVal U,T1)])

head :: [A] -> (A,cTree)
head (X:Xs) = (X,T)
    where T = ctNode "head.1" [dVal (X:Xs)] (dVal X)
                          (ctClean [])
```

```
tail :: [A] -> ([A],cTree)
tail (X:Xs) = (Xs,T)
    where T = ctNode "tail.1" [dVal (X:Xs)] (dVal Xs)
                          (ctClean [])

map :: (A -> (B,cTree)) -> [A] -> ([B],cTree)
map F []    = ([],T)
    where T = ctNode "map.1" [dVal F, dVal []] (dVal [])
                          (ctClean [])

map F (X:Xs)= (U : V, T)
        where (U,T1) = F X
              (V,T2) = map F Xs
          T = ctNode "map.2" [dVal F, dVal (X:Xs)]
                     (dVal (U:V))
                     (ctClean [(dVal U,T1),(dVal V,T2)])



%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% función ctClean
ctClean:: [(pVal,cTree)]-> [cTree]
ctClean [ ] = [ ]
ctClean ((PVal,CTree):Ts) = if (PVal=="_")
                                 then CleanTs
                                 else if (CTree==ctVoid)
                                      then CleanTs
                                      else (CTree:CleanTs)
                            where CleanTs = ctClean Ts


%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% funciones auxiliares

s0::nat -> (nat, cTree)
s0 X = (suc X, ctVoid)

plus0 :: nat -> (nat->(nat,cTree),cTree)
plus0 X = (plus X, ctVoid)

twice0 :: (A -> (A,cTree)) -> (A -> (A,cTree),cTree)
twice0 X = (twice X, ctVoid)
```

```
map0 :: (A -> (B,cTree)) -> ([A] -> ([B],cTree),cTree)
map0 X = (map X, ctVoid)
...
```

Por simplicidad en el ejemplo hemos usado para cada función transformada $f^{\mathcal{T}}$ el mismo nombre que tenía la función f original. Los puntos suspensivos indican que en el programa real aparecerían más funciones auxiliares, como por ejemplo la correspondiente a la constructora de listas : cuando está aplicada a 0 argumentos, o las de las funciones primitivas del lenguaje (éstas serán discutidas en el capítulo 5).

4.4.4. El programa P_{solve}

Para probar la corrección de la técnica nos va a interesar incluir el objetivo que ha dado lugar al síntoma inicial como parte del programa que va a ser depurado mediante la incorporación de una nueva función, a la que llamaremos *solve*. De esta forma las transformaciones de aplanamiento y de introducción de árboles de cómputo se aplicarán automáticamente también a los objetivos, simplificando el razonamiento. El objetivo inicial pasará entonces a ser de la forma $solve == true$. La siguiente definición introduce el programa P_{solve}, construido a partir de estas ideas.

Definición 4.4.1. Sea P un programa con signatura $\Sigma = \langle TC,\ DC,\ FS\rangle$. Sea G un objetivo y $\theta \equiv \theta_p \biguplus \theta_l$ una respuesta computada para G mediante P en SC. Definimos entonces el programa P_{solve} sobre la signatura extendida $\Sigma_{solve} = \langle TC,\ DC,\ FS \cup \{solve\}\rangle$, con *solve* una nueva[2] función de aridad 0, como el programa que contiene todas las reglas de P además de la regla adicional $solve\ = true \Leftarrow G\theta_p$.

Si P y $G\theta$ lo están bien tipado resulta inmediato comprobar que P_{solve} también lo estará tras la inclusión de la nueva regla de *solve*, y que el tipo principal de esta función será $solve :: bool$.

La relación entre el APA para $G\theta$ en P y el de $solve \rightarrow true$ en P_{solve} se muestra en la siguiente proposición:

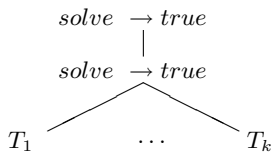
Proposición 4.4.1. *Sea P un programa, GS un sistema correcto de resolución de objetivos, G un objetivo y $\theta \equiv (\theta_p \biguplus \theta_l)$ tal que $G\ \Vdash_{GS,P}\ \theta$. Sea P_{solve} el programa construido a partir de P, G y θ como indica la definición 4.4.1. Entonces:*

(i) Se cumple $P_{solve} \vdash_{SC} solve \rightarrow true$ con un árbol de prueba tal que en la raíz se ha aplicado un paso $AR + FA$ utilizando θ_l como instancia de la regla de solve.

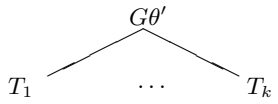
(ii) Para cada árbol de prueba T de $P_{solve} \vdash_{SC} solve \rightarrow true$ en el que se haya utilizado en la raíz un paso $AR+FA$ con una sustitución $\nu \in Subst_{\perp}$ tal que $dom(\nu) = def(G)$

[2]Suponemos sin pérdida de generalidad que $solve \notin FS$. En otro caso bastaría con utilizar otro nombre n, $n \notin FS$ para la nueva función.

para obtener la instancia de solve aplicada, se cumple que que $\theta' \equiv (\theta_p \biguplus \nu)$ *es una solución para* G *mediante* P *en* SC*, y que* T *tiene un* APA *asociado de la forma*



para ciertos subárboles $T_1, \ldots, T_k$ *tales que existe un* APA *asociado a una prueba para* $P \vdash_{SC} G\theta'$ *de la forma:*



Demostración
Ver pág. 236, apéndice A.

4.4.5. El cálculo *dValSC*

Uno de nuestros objetivos va a ser probar que la semántica de cada función transformada en el programa transformado $P^{\mathcal{T}}$ coincide con su semántica en el programa original P, salvo porque sus llamadas también devuelven valores de tipo cTree a partir de los cuales se pueden obtener fácilmente los APAs asociado al cómputo.

Sin embargo para esto debemos resolver un problema: el cálculo semántico SC presentado en el apartado 3.2.1 (pág. 55), en el que nos venimos basando hasta el momento, no permite probar ninguna aproximación de la forma dVal e → t para ningún $e \in Exp_\bot$, $t \in Pat_\bot$, ya que la única regla de SC aplicable en este caso es $AR{+}FA$ utilizando alguna instancia de regla de programa para dVal. Sin embargo dVal es una primitiva incorporada directamente del sistema y no definida por reglas de programa. Esta limitación sobre hechos básicos con dVal se extiende a hechos básicos $f\ \bar{t}_n \to t$ cualesquiera en el programa transformado, ya que la regla (TR_1) de la figura 4.2 (pág. 84) incluye llamadas a la primitiva dVal en todas las funciones transformadas.

Para solventar este inconveniente vamos a considerar un cálculo SC extendido para tratar con aproximaciones de dVal. La definición de este cálculo depende de algunos conceptos auxiliares, que agrupamos en la siguiente definición:

Definición 4.4.2. Sean $P_{\mathcal{A}}, P^{\mathcal{T}}$ tales que $P_{\mathcal{A}} \Rightarrow^{\mathcal{T}} P^{\mathcal{T}}$. Sea $\Sigma = \langle TC,\ DC,\ FS\rangle$ la signatura de $P_{\mathcal{A}}$ y $\Sigma^{\mathcal{T}} = \langle TC^{\mathcal{T}},\ DC^{\mathcal{T}},\ FS^{\mathcal{T}}\rangle$ la signatura de $P^{\mathcal{T}}$. Sea $Pat_\bot$ el conjunto de todos los patrones parciales en Σ, y $Pat^{\mathcal{T}}_\bot$, $Subst^{\mathcal{T}}_\bot$ respectivamente los conjuntos formados por los patrones parciales y las sustituciones parciales en $\Sigma^{\mathcal{T}}$. Entonces:

1. Llamamos *patrones invertibles* sobre $\Sigma^{\mathcal{T}}$ al conjunto $Pat_{inv} \subset Pat^{\mathcal{T}}_{\perp}$ dado por la siguiente definición:

$$Pat_{inv} = \perp \mid X \mid h^{\mathcal{T}}_m\ \bar{t}_m \mid c\ \bar{t}_m \mid f^{\mathcal{T}}\ \bar{t}_m$$

donde $X \in Var$, $h^{\mathcal{T}}_m$ es una de las funciones auxiliares introducidas durante la transformación de $P_{\mathcal{A}} \Rightarrow^{\mathcal{T}} P^{\mathcal{T}}$, $t_i \in Pat_{inv}$, para $i = 1 \ldots m$, $c \in DC^m$ y $f \in FS^{m+1}$.

2. Llamamos *sustituciones invertibles* al conjunto $Subst_{inv} \subset Subst^{\mathcal{T}}_{\perp}$ dado por la siguiente definición:

$$Subst_{inv} = \{\theta \in Subst^{\mathcal{T}}_{\perp} \mid \theta(X) \in Pat_{inv} \text{ para toda } X \in dom(\theta)\}$$

3. Dada una regla $(R) \in P^{\mathcal{T}}$, vamos a llamar $NV_{(R)}$ a las variables nuevas introducidas durante la transformación que ha dado lugar a la regla, y $OV_{(R)}$ al resto (es decir a las variables que provienen de la regla original).

4. Llamamos conjunto de *instancias admitidas* de $P^{\mathcal{T}}$ al conjunto

$$\{\ (R)(\theta_1 \biguplus \theta_2) \mid (R) \in P^{\mathcal{T}},\ \begin{array}{l} dom(\theta_1) \subseteq OV_{(R)},\ \theta_1 \in Subst_{inv}\ , \\ dom(\theta_2) \subseteq NV_{(R)},\ \theta_2 \in Subst^{\mathcal{T}}_{\perp}\ \} \end{array}$$

5. Dado un patrón $t \in Pat_{\perp}$ vamos a denotar por $\ulcorner t \urcorner$ a la representación de t como un valor de tipo $pVal$.

6. Dado $s \in Pat_{inv}$ definimos $s^{\mathcal{T}^{-1}}$ de la siguiente manera:

(INV$_1$)	$s^{\mathcal{T}^{-1}}$	$= s$	si $s \in Pat_{\perp}$.
(INV$_2$)	$(h^{\mathcal{T}}_m\ \bar{t}_m)^{\mathcal{T}^{-1}}$	$= h\ \bar{t}^{\mathcal{T}^{-1}}_m$	si $h \in DC \cup FS$.
(INV$_3$)	$(f^{\mathcal{T}}\ \bar{t}_m)^{\mathcal{T}^{-1}}$	$= f\ \bar{t}^{\mathcal{T}^{-1}}_m$	si $f \in FS$.
(INV$_4$)	$(c\ \bar{t}_m)^{\mathcal{T}^{-1}}$	$= c\ \bar{t}^{\mathcal{T}^{-1}}_m$	si $c \in DC$, $c\ \bar{t}_m \notin Pat_{\perp}$.

7. Definimos el cálculo semántico $dValSC$ como un cálculo que incluye las reglas del cálculo SC (descritas en el apartado 3.2.1, pág. 55), además de la siguiente regla adicional:

$$\textbf{DV} \quad dVal\ s\ \rightarrow\ u \qquad \text{para cualquier } u \sqsubseteq \ulcorner s^{\mathcal{T}^{-1}} \urcorner \text{ con } s \in Pat_{inv}$$

Además, la regla $AR + FA$ del cálculo $dValSC$ utilizará instancias admitidas de en lugar de instancias generales.

Por tanto el cálculo $dValSC$ es una extensión del cálculo semántico SC con un regla adicional para tratar las llamadas a dVal, y con la limitación de $AR+FA$ a instancias admitidas. Es evidente que este cálculo define la misma semántica que SC para programas que no utilicen la primitiva dVal, es decir que para todo programa P y aproximación $e \rightarrow t$ tales

que $P \vdash_{SC} e \to t$ se tiene igualmente $P \vdash_{dValSC} e \to t$. Esto nos permitirá utilizar las propiedades ya vistas para SC cuando nos encontremos ante demostraciones en $dValSC$ que no utilicen dVal.

La introducción de la regla DV supone asumir que el comportamiento de dVal es correcto, es decir que esta función realmente calcula la representación como valor de tipo pVal de su argumento, lo que parece razonable al no ser dVal una función definida por el usuario sino una función del sistema. El conjunto Pat_{inv} de los patrones invertibles (punto 1 de la definición) corresponde a aquellos patrones de $P^{\mathcal{T}}$ que corresponden a la transformación de un patrón en $P_{\mathcal{A}}$.

Nótese que $Pat_{inv} \neq Pat_{\perp}^{\mathcal{T}}$, ya que la signatura del programa transformado incluye los nuevos símbolos dVal, ctClean, ctNode y ctVoid con los que en principio se pueden construir patrones, como por ejemplo (dVal ctVoid) que no corresponden a la transformación de un patrón en $P_{\mathcal{A}}$. Lo mismo sucede con algunas aplicaciones de los símbolos de $FS^{\mathcal{T}} \cup DC^{\mathcal{T}}$: por ejemplo un patrón de la forma $(h_3\ t_1\ t_2)$ para alguna función auxiliar h_3 no corresponde a la transformación de ningún patrón de $P_{\mathcal{A}}$ ya que en caso contrario h_3 debería aparecer aplicado a 3 y no a 2 argumentos (ver regla de transformación (TR_5) de la figura 4.2, la única que introduce funciones auxiliares).

La definición 4.4.2 también introduce una transformación ${}^{\mathcal{T}^{-1}}$ que convierte los patrones invertibles Pat_{inv} en los valores de $Pat_{\perp}^{\mathcal{T}}$ de los que provienen. La siguiente proposición asegura que ${}^{\mathcal{T}^{-1}}$ está bien definida y que su resultado es el esperado:

Proposición 4.4.2. *Sean $P_{\mathcal{A}}, P^{\mathcal{T}}$ tales que $P_{\mathcal{A}} \Rightarrow^{\mathcal{T}} P^{\mathcal{T}}$. Sea $\Sigma = \langle TC,\ DC,\ FS \rangle$ la signatura de $P_{\mathcal{A}}$, $Pat_{\perp}$ el conjunto de los patrones parciales sobre Σ y Pat_{inv} definido como indica el punto 1 de la definición 4.4.2. Entonces:*

1. *Para todo $t \in Pat_{\perp}$, $t^{\mathcal{T}} \in Pat_{\mathrm{inv}}$ y $(t^{\mathcal{T}})^{\mathcal{T}^{-1}} = t$.*
2. *Para todo $t \in Pat_{\mathrm{inv}}$, $t^{\mathcal{T}^{-1}} \in Pat_{\perp}$ y $(t^{\mathcal{T}^{-1}})^{\mathcal{T}} = t$.*

Demostración

1. El primer punto se puede comprobar por inducción estructural sobre $t \in Pat_{\perp}$ examinando las reglas de transformación (TR_2)-(TR_6) de la figura 4.2, aplicables a t para obtener $t^{\mathcal{T}}$.

   - $t \equiv \perp$, $t \equiv X$ con $X \in Var$, o $t \equiv h\ \bar{t}_m$ con $h \in DC^m$.
     Entonces $t^{\mathcal{T}} =$ (TR_2, TR_3 o TR_4) $= t \in Pat_{inv}$. Además $(t^{\mathcal{T}})^{\mathcal{T}^{-1}} = t^{\mathcal{T}^{-1}} =$ (INV_1) $= t$.
   - $t \equiv h\ \bar{t}_m$, $t_i \in Pat_{\perp}$ para $i = 1 \ldots m$ y o bien $h \in DC^n$, $n > m$ o $h \in FS^n$ $n > m + 1$.
     En este caso $t^{\mathcal{T}} = (\mathrm{TR}_5) = (h_m^{\mathcal{T}}\ \bar{t}_m^{\mathcal{T}}) \in Pat_{inv}$ ya que $t_i^{\mathcal{T}} \in Pat_{inv}$ para $i = 1 \ldots m$ por hipótesis de inducción. Por otra parte

$$(t^{\mathcal{T}})^{\mathcal{T}^{-1}} = (h_m^{\mathcal{T}}\ \bar{t}_m^{\mathcal{T}})^{\mathcal{T}^{-1}} = (\mathrm{INV}_2) = h\ ((\bar{t}_m^{\mathcal{T}})^{\mathcal{T}^{-1}}) = (\text{h. i.}) = h\ \bar{t}_m$$

- $t \equiv f\ \bar{t}_m,\ f \in FS^{m+1}$ con $t_i \in Pat_\perp$ para $i = 1 \ldots m$.
  Entonces $t^{\mathcal{T}} = (\mathrm{TR}_6) = (f^{\mathcal{T}}\ \bar{t}_m^{\mathcal{T}}) \in Pat_{inv}$, ya que por hipótesis de inducción $t_i^{\mathcal{T}} \in Pat_{inv}$ para $i = 1 \ldots m$. Además:

$$(t^{\mathcal{T}})^{\mathcal{T}^{-1}} = (f^{\mathcal{T}}\ \bar{t}_m^{\mathcal{T}})^{\mathcal{T}^{-1}} = \ (\mathrm{INV}_3) = f\ ((\bar{t}_m^{\mathcal{T}})^{\mathcal{T}^{-1}}) = \ (\text{h. i.})\ = f\ \bar{t}_m$$

2. Por inducción estructural sobre $t \in Pat_{inv}$, examinando las distintas formas que puede tener t y comprobando que en cada caso una única regla (INV_1)-(INV_4) es aplicable:

   - $t \in Pat_\perp$. Esta posibilidad cubre los casos $t \equiv \perp$, $t \equiv X$ con $X \in Var$ y $t \equiv c\ \bar{t}_m$ con $c \in DC^m$ y $t_i \in Pat_\perp$.
     Se tiene por (INV_1) que $t^{\mathcal{T}^{-1}} \equiv t \in Pat_\perp$ y $(t^{\mathcal{T}^{-1}})^{\mathcal{T}} = t^{\mathcal{T}} = (\mathrm{TR}_2,\ \mathrm{TR}_3$ o $\mathrm{TR}_4) = t$.
   - Si $t \equiv c\ \bar{t}_m$ con $c \in DC^m$ pero $t \notin Pat_\perp$, entonces la regla (INV_4) es aplicable, teniéndose $t^{\mathcal{T}^{-1}} = c\ \bar{t}_m^{\mathcal{T}^{-1}}$ y teniéndose $c\ \bar{t}_m^{\mathcal{T}^{-1}} \in Pat\ \perp$ por hipótesis de inducción. Además

$$(t^{\mathcal{T}^{-1}})^{\mathcal{T}} = c\ \bar{t}_m^{\mathcal{T}^{-1}} = (\ \mathrm{TR}_4\ ) = c\ (\bar{t}_m^{\mathcal{T}^{-1}})^{\mathcal{T}} = \ (\text{h.i.})\ = c\ \bar{t}_m$$

   - Los dos casos restantes, $t \equiv f^{\mathcal{T}}\ \bar{t}_m$ y $t \equiv h_m\ \bar{t}_m$ son similares, obteniéndose el resultado por hipótesis de inducción tras aplicar respectivamente (INV_3) o (INV_2) para ver que $t^{\mathcal{T}^{-1}} \in Pat_\perp$ y (TR_5) o (TR_6) para comprobar que $(t^{\mathcal{T}^{-1}})^{\mathcal{T}} = t$.

■

Es interesante observar que las reglas (INV_1)-(INV_4), y con ellas transformación $^{\mathcal{T}^{-1}}$, son sencillas de implementar en la práctica, ya que se basan en un examen sintáctico del valor a que se aplican.

Sin embargo aún queda una cuestión pendiente: la regla DV sólo se encarga de los casos en los que dVal aparece aplicada a valores $s \in Pat_{inv}$ y no a expresiones generales. Para asegurar que esto es suficiente en la práctica observamos en primer lugar que en el programa transformado sólo aparece dVal aplicada a argumentos de la forma indicada:

Proposición 4.4.3. *Sean $P_{\mathcal{A}}, P^{\mathcal{T}}$ programas tales que $P_{\mathcal{A}} \Rightarrow^{\mathcal{T}} P^{\mathcal{T}}$. Supongamos que el símbolo dVal no forma parte de la signatura de $P_{\mathcal{A}}$. Entonces todas las expresiones de la forma $dVal\ e$ que aparecen en $P^{\mathcal{T}}$ verifican $e \equiv t^{\mathcal{T}}$ para algún patrón t, y por tanto $e \in Pat_{\mathrm{inv}}$.*

Demostración

La única de las reglas de transformación (TR_1)-(TR_{11}) de la figura 4.2 (pág. 84) que introduce la primitiva $dVal$ en el programa transformado es (TR_1), que incluye diversas llamadas a la primitiva en la última aproximación de la condición de la regla transformada. Vamos a examinar cada una de las apariciones de $dVal$:

- $dVal\ s_i$ con $i = 1 \ldots n$. De las premisas de la regla tenemos $t_i \Rightarrow^{\mathcal{T}} s_i$ con $t_i \in Pat$ para cada $i = 1 \ldots s_i$, al estar cada t_i en el lado izquierdo de una regla del programa original. Por tanto $s_i \equiv t_i^{\mathcal{T}}$ para $i = 1 \ldots s_i$.

- $dVal\ s$. En la premisa tenemos que $r \Rightarrow^{\mathcal{T}} s$ con r el lado derecho del programa original y por el apartado a) la proposición 4.3.1 (pág. 78) r es un patrón. Por tanto $s \equiv r^{\mathcal{T}}$.

- $dVal\ r_i$ con $i = 1 \ldots m$. Todos los r_i provienen de la transformación de la condición C en la regla del programa original. Examinando las reglas que se encargan de la transformación de condiciones, (TR_7)-(TR_{11}), se tiene que todos los r_i son introducidos bien por la regla (TR_{10}) o por la regla (TR_{11}), donde aparecen representados por el valor v que se obtienen de la transformación de un patrón t, por lo que del apartado a) la proposición 4.3.1 se tiene que estos valores son siempre patrones.

Recordemos además que estamos suponiendo que dVal no aparece en el programa aplanado, es decir no es utilizado por el usuario en el programa original, lo que resulta razonable al ser esta primitiva parte del sistema únicamente para su uso por el depurador. □

El requerimiento de que dVal, y más en general de que ninguno de los símbolos dVal, ctNode, ctVoid, ctClean estén en la signatura de $P_{\mathcal{A}}$ no supone ninguna restricción en la práctica, al tratarse de símbolos introducidos por la transformación de programas que no deben ser utilizados por el usuario en sus programas. En el resto de los resultados asumiremos que esto sucede sin indicarlo explícitamente.

La proposición 4.4.3 nos asegura que en el programa transformado sólo aparece dVal aplicada a patrones transformados del programa original. Esto resulta suficiente para asegurar que siempre se podrá utilizar la regla DV a las aplicaciones de dVal durante las demostraciones en $dValSC$, debido al requerimiento adicional de que la regla $AR+FA$ sólo utilice instancias admitidas, tal y como asegura la siguiente proposición:

Proposición 4.4.4. *Sean $P_{\mathcal{A}}$, $P^{\mathcal{T}}$ tales que $P_{\mathcal{A}} \Rightarrow^{\mathcal{T}} P^{\mathcal{T}}$. Sean $c, c^{\mathcal{T}}$ condiciones atómicas tales que c es una condición plana y $c \Rightarrow^{\mathcal{T}} (c^{\mathcal{T}}\ ;\ sec)$ para cierto valor sec. Sea $\theta \in Subst_{\perp}^{\mathcal{T}}$ de la forma $\theta \equiv \theta_1 \uplus \theta_2$, con $dom(\theta_1) \subseteq Var(c)$ y $\theta_1(X) \in Pat_{\mathrm{inv}}$ para todo $X \in dom(\theta_1)$ tal que $P^{\mathcal{T}} \vdash_{dValSC} c^{\mathcal{T}}\theta$. Entonces todas las expresiones de la forma (dVal t) que pueden aparecer en un paso de demostración de $P^{\mathcal{T}} \vdash_{dValSC} c^{\mathcal{T}}\theta$ verifican $\mathsf{t} \in Pat_{\mathsf{inv}}$.*

Demostración
En $c^{\mathcal{T}}$ no puede aparecer dVal al estar suponiendo que este símbolo no puede estar en c. En $c^{\mathcal{T}}\theta$ el símbolo dVal sólo puede aparecer, por la estructura de θ, en lugar de alguna variable nueva introducida durante la demostración. Examinando las reglas de transformación de la figura 4.2 (pág. 84) tenemos que ninguna de estas variables aparece nunca aplicada a otro valor por lo que en $c^{\mathcal{T}}\theta$ no puede haber ninguna expresión (dVal t).

De la proposición 4.4.3 tenemos que en las reglas de $P^{\mathcal{T}}$ las llamadas a dVal son siempre de la forma dVal s para $s \in Pat_{inv}$ (en particular $s \equiv t^{\mathcal{T}}$ para $t \in Pat_{\perp}$), por lo que en la demostración aparecerán como dVal $s(\theta_1 \uplus \theta_2)$ debido a la utilización de instancias admisibles por parte del cálculo $dValSC$. Como s no puede incluir variables nuevas (al

corresponder al transformado de un patrón) se tiene $s(\theta_1 \uplus \theta_2) = s\theta_1$. Veamos que $s\theta_1 \in Pat_{inv}$ por inducción estructural sobre $s \in Pat_{inv}$:

- $s \equiv \bot$. Entonces $s\theta_1 = \bot$, y $\bot \in Pat_{inv}$.
- $s \equiv X$, $X \in Var$, y X no es una variable nueva por lo que $\theta_1(X) \in Pat_{inv}$ por la definición de θ_1.
- $s \equiv h\ \bar{t}_m$. Entonces $s\theta_1 = h\ \bar{t}_m\theta_1$. Como $s \in Pat_{inv}$ se tiene que $t_i \in Pat_{inv}$ y el resultado se tiene por hipótesis de inducción. ■

Para finalizar este apartado introducimos la definición de una propiedad que deben cumplir los sistemas resolutores de objetivos de los sistemas sobre los que se implemente el depurador.

Definición 4.4.3. Dado un sistema de resolución de objetivos GS, decimos que

- GS es *completo con respecto a la depuración* sii para cada programa P, objetivo G y sustitución θ verificando $G \Vdash_{GS,P} \theta$ tales que $P^{\mathcal{T}}_{solve} \vdash_{dValSC} solve^{\mathcal{T}} == (true, ct)$, con P_{solve} definido a partir de P, G y θ como indica la definición 4.4.1, y con ct un valor total de tipo $cTree$, se verifica

$$(solve^{\mathcal{T}} == (true, Tree)) \Vdash_{GS,P^{\mathcal{T}}_{solve}} \{Tree \mapsto ct\}$$

  siendo $\{Tree \mapsto ct\}$ la primera respuesta obtenida por GS para el objetivo $(solve^{\mathcal{T}} == (true, Tree))$.

- GS es *razonable con respecto a la depuración* si es completo con respecto a la depuración y correcto con respecto a $dValSC$ en el sentido de la definición 3.2.1[3].

La noción de completitud con respecto a la depuración establece que que la prueba de la igualdad estricta $(solve^{\mathcal{T}} == (true, ct))$ hecha en $dValSC$ asegura que el sistema será capaz de obtener la respuesta el objetivo $(solve^{\mathcal{T}} == (true, Tree))$. Intuitivamente este requerimiento expresa que los objetivos del programa transformado que se planteen para una sesión de depuración a partir de una respuesta que el sistema GS ha calculado previamente también pueden ser resueltos con el sistema GS. Se trata de un requerimiento natural y confirmado por la experiencia práctica con sistemas tales como $\mathcal{TOY}$ y Curry. La verificación formal de que un sistema concreto cumple este requisito es una tarea muy complicada que no se aborda en esta tesis.

4.4.6. Corrección de la Técnica

El primer resultado que vamos a establecer, análogamente a lo que hicimos en el caso de la transformación de aplanamiento, prueba que el programa transformado es correcto desde el punto de vista de los tipos:

[3]Esta definición se hizo con respecto al cálculo SC, pero su extensión al caso $dValSC$ es inmediata.

Teorema 4.4.5. *Sea P un programa bien tipado. Sean $P_{\mathcal{A}}$ y $P^{\mathcal{T}}$ dos programas tales que $P \Rightarrow_{\mathcal{A}} P_{\mathcal{A}}$ y $P_{\mathcal{A}} \Rightarrow^{\mathcal{T}} P^{\mathcal{T}}$. Entonces $P^{\mathcal{T}}$ es un programa bien tipado.*

Demostración
Ya hemos visto (teorema 4.3.3) que al estar P bien tipado $P_{\mathcal{A}}$ también lo está. Por tanto para probar el teorema basta con comprobar que si $P_{\mathcal{A}}$ está bien tipado $P^{\mathcal{T}}$ también lo estará. Esta demostración se puede encontrar en la pág. 238, apéndice A.

Ahora podemos probar el siguiente teorema:

Teorema 4.4.6. *Sea P un programa y f una función de aridad n definida en P. Sean $P_{\mathcal{A}}$ y $P^{\mathcal{T}}$ programas tales que $P \Rightarrow_{\mathcal{A}} P_{\mathcal{A}}$ y $P_{\mathcal{A}} \Rightarrow^{\mathcal{T}} P^{\mathcal{T}}$. Sean además $\bar{t}_n$, patrones parciales cualesquiera sobre la signatura de P y t un patrón sobre la misma signatura, $t \neq \bot$. Entonces*

(i) Si $P \vdash_{SC} f\ \bar{t}_n \rightarrow t$ y apa es un APA para esta demostración, que por la proposición 3.3.1 (pág. 63) debe tener la forma $\overset{f\ \bar{t}_n \rightarrow t}{\underset{T}{|}}$ para cierto árbol T.
Entonces se cumple $P^{\mathcal{T}} \vdash_{dValSC} f^{\mathcal{T}}\ \overline{t_n^{\mathcal{T}}} \rightarrow (t^{\mathcal{T}}, ct)$, donde $ct :: cTree$ es un patrón total representando T.

(ii) Si $P^{\mathcal{T}} \vdash_{dValSC} f^{\mathcal{T}}\ \overline{t_n^{\mathcal{T}}} \rightarrow (t^{\mathcal{T}}, ct)$ entonces $P \vdash_{SC} f\ \bar{t}_n \rightarrow t$. Además si ct es un patrón total, entonces representa un árbol T tal que $\overset{f\ \bar{t}_n \rightarrow t}{\underset{T}{|}}$ es un APA para $P \vdash_{SC} f\ \bar{t}_n \rightarrow t$.

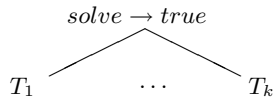
Demostración
Ver pág. 251, apéndice A.

Como se observa en el enunciado del teorema el programa transformado no obtiene el APA completo, pero éste puede generar a posteriori simplemente construyendo el árbol de raíz $f\ \bar{t}_n \rightarrow t$ y único subárbol hijo el árbol T calculado por el programa transformado. Por tanto el resultado prueba la corrección de la transformación desde el punto de vista del cálculo semántico para los objetivos de la forma $f\ \bar{t}_n \rightarrow t$.

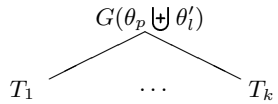
La generalización a un objetivo cualquiera se puede conseguir introduciendo éste en el programa como condición de una nueva regla de programa *solve*, tal y como indica la definición 4.4.1 (pág. 90). El siguiente resultado utiliza esta idea para establecer el comportamiento del sistema, suponiendo un sistema de resolución de objetivos razonable con respecto a la depuración.

Teorema 4.4.7. *Sea P un programa con signatura $\Sigma = \langle TC,\ DC,\ FS \rangle$, G un objetivo y GS un sistema de resolución de objetivos razonable para la depuración. Sea θ_p una respuesta producida por GS para G usando P. Sea P_{solve} construido como se indica en la definición 4.4.1. Sean $P_{solve}^{\mathcal{A}}$, $P_{solve}^{\mathcal{T}}$ programas tales que $P_{solve} \Rightarrow_{\mathcal{A}} P_{solve}^{\mathcal{A}}$ y $P_{solve}^{\mathcal{A}} \Rightarrow^{\mathcal{T}} P_{solve}^{\mathcal{T}}$.*

Entonces el objetivo $solve^{\mathcal{T}} == (true, Tree)$ tiene éxito con respecto al programa $P_{solve}^{\mathcal{T}}$ utilizando el sistema GS. Más aún, la primera respuesta producida vincula Tree *a un valor de tipo* cTree *representando un árbol de la forma:*



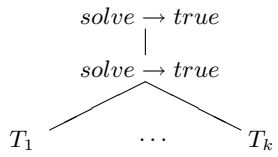
para ciertos árboles $T_1, \ldots, T_k$*. Además, existe una sustitución* θ'_l *tal que* $P \vdash_{SC} G(\theta_p \biguplus \theta'_l)$ *se puede probar y tiene un APA asociado a algún árbol de prueba testigo de esta derivación de la forma:*



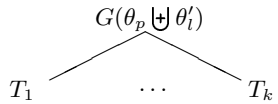
Demostración
Dividimos la demostración en pasos:

1. Al ser θ_p una respuesta producida debe existir una sustitución θ_l tal que $G \Vdash_{GS,P} (\theta_p \biguplus \theta_l)$, es decir que $(\theta_p \biguplus \theta_l)$ es una respuesta computada para G mediante GS usando el programa P.

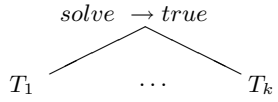
2. Entonces por el apartado (i) de la proposición 4.4.1 (pág. 90) se cumple $P_{solve} \vdash_{SC} solve \rightarrow true$ con un paso de derivación en la raíz la sustitución θ_l para obtener la instancia de $solve$ aplicada. Al ser $dom(\theta_l) = def(G)$, podemos aplicar el apartado (ii) de la misma proposición para tener que el APA de esta prueba es de la forma



para ciertos subárboles $T_1, \ldots, T_k$ tales que existe una sustitución θ'_l cumpliendo $P \vdash_{SC} G(\theta_p \biguplus \theta'_l)$ con un APA asociado a algún árbol de prueba testigo de esta derivación de la forma:



3. Por el punto anterior y del apartado (i) del teorema 4.4.6 (pág. 97) se tiene que se cumple $P^{\mathcal{T}}_{solve} \vdash_{dValSC} solve^{\mathcal{T}} \rightarrow (true, ct)$ con ct un patrón total representando



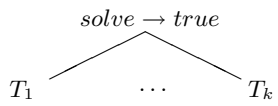
4. Al ser $(true, ct)$ totales se puede probar $P^{\mathcal{T}}_{solve} \vdash_{dValSC} solve^{\mathcal{T}} == (true, ct)$, comenzando con un paso JN y utilizando el valor $(true, ct)$ como el valor total al que se refiere dicha regla. Las premisas serán entonces $P^{\mathcal{T}}_{solve} \vdash_{dValSC} solve^{\mathcal{T}} \rightarrow (true, ct)$, que se tiene por el punto anterior, y $P^{\mathcal{T}}_{solve} \vdash_{dValSC} (true, ct) \rightarrow (true, ct)$ que se puede probar aplicando reiteradamente las reglas DC, RR y BT.

5. Por ser GS completo con respecto a la depuración se tiene del punto anterior que $(solve^{\mathcal{T}} == (true, Tree)) \Vdash_{GS, P^{\mathcal{T}}_{solve}} \{Tree \mapsto ct\}$, con ct el indicado por el teorema y siendo esta la primera respuesta obtenida por el sistema para el objetivo $(solve^{\mathcal{T}} == (true, Tree))$.

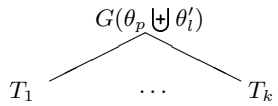
6. Del punto anterior y por ser GS correcto con respecto a $dValSC$ se tiene que se puede probar $P^{\mathcal{T}}_{solve} \vdash_{dValSC} solve^{\mathcal{T}} == (true, ct)$. En el árbol de esta prueba se encuentra entonces la prueba de $P^{\mathcal{T}}_{solve} \vdash_{dValSC} solve^{\mathcal{T}} \rightarrow (true, ct)$.

7. Del punto anterior tenemos que es posible aplicar el apartado (ii) del teorema 4.4.6, de donde tenemos que se puede probar $P_{solve} \vdash_{SC} solve \rightarrow true$ y que además ct representa un APA para esta prueba salvo por la ausencia de la raíz.

8. Como además se tiene $P \vdash_{SC} G(\theta_p \biguplus \theta_l)$ por el primer punto, podemos aplicar la proposición 4.4.1, apartado (ii) al punto anterior para deducir que ct representa un árbol



y se cumple $P \vdash_{SC} G(\theta_p \biguplus \theta'_l)$ con un APA asociado a esta prueba de la forma



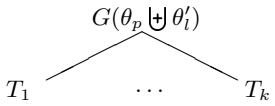
tal y como queríamos demostrar.

■

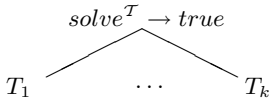
El árbol obtenido siguiendo el procedimiento difiere del APA buscado en la raíz. Sin embargo la proposición 3.3.5 (pág. 67) nos asegura que la raíz "perdida" no era un nodo crítico, y como el APA debe contener un nodo crítico éste se conservará en el árbol obtenido, lo que garantiza que este árbol resulta adecuado para la depuración. El siguiente teorema utiliza estas ideas para establecer la corrección de un depurador basado en la transformación de programas para la depuración de respuestas incorrectas de programas lógico-funcionales:

Teorema 4.4.8. *Sea P un programa, G un objetivo, GS un sistema de resolución de objetivos razonable con respecto a SC y a $dValSC$ y θ_p una respuesta incorrecta producida por GS para G utilizando P. Sea T el árbol representado por el valor $Tree$ de tipo $cTree$ indicado en el teorema 4.4.7. Supongamos además que existe un oráculo capaz de determinar cuando un hecho básico pertenece a la interpretación pretendida $\mathcal{I}$ de P. Entonces un depurador declarativo basado en el esquema general de la sección 2.2 que utilice T como árbol de prueba localizará una regla incorrecta en el programa P.*

Demostración
Sea θ'_l la sustitución indicada en el teorema 4.4.7 tal que $P \vdash_{SC} G(\theta_p \biguplus \theta'_l)$ con un APA *apa* asociado a un árbol de prueba testigo de la derivación de la forma:



Por otra parte y también por como consecuencia del teorema 4.4.7 el árbol T obtenido por el sistema mediante el programa $P^{\mathcal{T}}_{solve}$ es:



que difiere de *apa* sólo en la raíz. El depurador utilizará el oráculo para navegar T, asumiendo que $solve^{\mathcal{T}} \rightarrow true$ no está en la interpretación pretendida del programa P (suposición razonable teniendo en cuenta la regla que define *solve* en P_{solve} y el hecho de que el usuario haya considerado θ_p una respuesta incorrecta para G).

Como la raíz de *apa* es errónea pero no puede ser un nodo crítico por la proposición 3.3.5, debe tener algún hijo erróneo n, que será también hijo de $solve^{\mathcal{T}} \rightarrow true$, por lo que la raíz de T tampoco puede ser un nodo crítico. Ahora bien, al haber al menos un nodo erróneo en T (el nodo n) este árbol tiene un nodo crítico, y todo nodo crítico de T es nodo crítico de *apa* (y viceversa) al diferir ambos árboles sólo en la raíz, que en ambos casos es errónea pero no crítica. Y, según el teorema 3.3.6 (pág. 67), este nodo crítico, por estar en *apa*, tiene asociada una instancia de regla de programa que es incorrecta con respecto al modelo pretendido del programa P_{solve}. Como la regla correspondiente regla no puede corresponder a la única regla para *solve*, ya que esta sólo aparece en la raíz que no es crítica, esta regla corresponderá a una regla de P, por lo que puede ser señalada con seguridad por el depurador como una regla incorrecta. ■

En la práctica la validez de la raíz del árbol T no se preguntará al oráculo, asumiéndose siempre como errónea porque se corresponde con la raíz errónea de un APA de $G(\theta_p \biguplus \theta_l)$. Es interesante observar que los teoremas 4.4.7 y 4.4.8 no sólo prueban la corrección del método, sino que indican los pasos a seguir por el depurador para la detección de una regla incorrecta a partir de una respuesta producida incorrecta.

Por ejemplo, consideremos el objetivo G definido como

(head (drop4 (map plus (from z)))) (suc z) == X

El lado izquierdo de la igualdad estricta aplica el quinto elemento de la lista (map plus (from z)) al valor (suc z). Debido al error en la regla de from, este quinto elemento es plus z (en lugar de suc (suc (suc (suc z)))) y la respuesta obtenida para el objetivo con respecto al programa del ejemplo 4.2.1 es { X ↦ suc z }, que es el síntoma inicial. Dado el programa, el objetivo y la respuesta, para obtener el depurador puede seguir los siguientes pasos:

1. Aplicar la respuesta computada al objetivo, y añadir al programa una nueva función:

```
solve ::  bool
solve = true <==(head (drop4 (map plus (from z)))) (suc z)==suc z
```

2. Se aplana el program P_{solve} según las reglas (PL_1)-(PL_{10}). El resultado es el programa del ejemplo 4.3.1 (pág. 79), pero incluyendo la función solve aplanada:

```
solve ::  bool
solve = true <== U==suc z
        where
         V1= from z
         V2= map plus V1
         V3= drop4
         V4= V3 V2
         V5= head V4
         U = V5 (suc z)
```

3. Se transforma el programa aplanado según las reglas (TR_1)-(TR_{11}). El resultado es el programa del ejemplo 4.4.1 (pág. 87), pero aumentado con la función solve transformada:

```
solve ::  (bool,cTree)
solve = (true,T) <== U==suc z
    where (V1,T1)= from z
          (V2,T2)= map plus0 V1
          (V3,T3)= drop4
          (V4,T4)= V3 V2
          (V5,T5)= head V4
          (U ,T6) = V5 (suc z)
          T = ctNode "solve.1"  (dVal true)
               (ctClean [(dVal V1, T1), (dVal V2, T2),
                         (dVal V3, T3), (dVal V4, T4),
                         (dVal V5, T5), (dVal U, T6)])
```

4. Se lanza el objetivo solve == (true, T). La respuesta computada, contendrá un vínculo de T a un valor de tipo cTree con la forma indicada en el teorema 4.4.7.

5. Se procede a examinar el árbol obtenido en el paso anterior con ayuda del oráculo.

Todo este proceso se lleva a cabo automáticamente por el depurador. El usuario sólo señala que ha habido una respuesta incorrecta y tras unos instantes empieza a recibir preguntas por parte del depurador. En el capítulo siguiente explicaremos detalladamente cómo se realiza este proceso en las implementaciones existentes en los sistemas $\mathcal{TOY}$ [1, 54] y Curry [39].

Capítulo 5

Depurador de Respuestas Incorrectas para los Sistemas $\mathcal{TOY}$ y Curry

Los dos capítulos anteriores describen los fundamentos teóricos en los que basamos nuestra propuesta para la depuración declarativa de respuestas incorrectas en lenguajes lógico-funcionales. En este capítulo y en el siguiente nos concentraremos en cambio en los aspectos prácticos, discutiendo cómo se pueden incorporar estas ideas a un sistema real. Para facilitar la exposición hemos divido esta discusión en dos partes:

- Modificaciones que hay que realizar en el sistema para que éste sea capaz de producir los árboles de cómputo asociados a las respuestas incorrectas. Este es el propósito del presente capítulo.
- Diferentes propuestas para el recorrido o *navegación* del árbol de cómputo. A esto dedicaremos el capítulo próximo. Allí hablaremos así mismo de la eficiencia del depurador desde el punto de vista del consumo en tiempo y en memoria requerido para la generación del árbol de cómputo, así como de su utilidad real para un usuario: cantidad y dificultad de las preguntas planteadas.

En relación con el primer punto presentamos en este capítulo los dos depuradores declarativos que, basándonos en las ideas expuestas en esta tesis, hemos incorporado a los sistemas lógico-funcionales $\mathcal{TOY}$ [1, 54] (desarrollado por el Grupo de Programación Declarativa del Departamento de Sistemas Informáticos y Programación, Universidad Complutense de Madrid) y Curry [39] (desarrollado por Wolfgang Lux y Herbert Kuchen, Institut für Wirtschaftsinformatik, Westfälische Wilhems-Universität Münster), que pueden encontrarse en

```
http://babel.dacya.ucm.es/toy
```

y en

`http://danae.uni-muenster.de/~lux/curry`

respectivamente.

Aunque los lenguajes soportados por ambos sistemas son muy similares, los dos compiladores siguen técnicas distintas para la compilación de los programas fuente y están escritos en lenguajes diferentes:

- El compilador de $\mathcal{TOY}$ traduce el código fuente a código Prolog [87]. Este código es después interpretado por el sistema SICStus Prolog [83]. El propio compilador está también escrito en Prolog.
- El compilador de Curry de la universidad de Münster está escrito en Haskell [75] y traduce el código Curry [39] a un código intermedio propio que es ejecutado posteriormente por una máquina abstracta [55] escrita en lenguaje C.

A pesar de estas diferencias, y gracias a la independencia del compilador concreto que proporciona la técnica de la transformación de programas, la estructura del depurador es muy similar en ambos casos. Por ello expondremos más detalladamente la estructura del depurador incorporado en $\mathcal{TOY}$, limitándonos en el caso del depurador para Curry a comentar las peculiaridades que lo diferencian del anterior.

La siguiente sección tratará de las funciones primitivas y de en qué condiciones podemos admitir su incorporación a los programas que queremos depurar. En las secciones 5.2 y 5.3 comentaremos el proceso seguido para para la incorporación del depurador a cada uno de los sistemas y sus particularidades, comenzando por $\mathcal{TOY}$. Una de las características principales que distinguen a Curry de $\mathcal{TOY}$ es la *búsqueda encapsulada* [42]. La depuración de programas incluyendo esta característica se discute en la sección 5.4, basada en el trabajo [18]. Para finalizar, en la sección 5.5 muestra las dos limitaciones actuales de los depuradores: no se permiten programas incluyendo operaciones de entrada/salida, ni tampoco programas que hagan uso de restricciones que no sean de igualdad estricta. En particular, las restricciones de desigualdad y las restricciones aritméticas disponibles en TOY no están aún soportadas por el depurador.

5.1. Traducción de las Funciones Primitivas

Las funciones primitivas son funciones implementadas directamente en el compilador y disponibles en todos los programas. Podemos por tanto considerarlas parte de cualquier signatura, a pesar de no encontrarse definidas mediante reglas de función. Ejemplos típicos de funciones primitivas son los operadores aritméticos $+, -, *, /$, o las funciones trigonométricas tales como *sin* o *cos*.

Estas funciones plantean un problema a la hora de la depuración basada en la transformación de programas, ya que no tenemos reglas de función que transformar. En efecto, el código de una función primitiva normalmente no está escrito ni siquiera en el mismo lenguaje que estamos depurando, por lo que no tiene sentido pensar en aplicarle el esquema

de transformación de programas visto en los capítulos anteriores. Debemos considerar este código, y por tanto el tipo de la función primitiva, como "intocables" para el depurador.

Sin embargo, las funciones primitivas se encuentran frecuentemente como parte de los programas a depurar, y limitar la utilización del depurador a programas sin funciones primitivas resultaría excesivamente restrictivo. En esta sección vamos a ver como extender el marco de la transformación de programas a los programas con primitivas.

Para facilitar la exposición hemos dividido el conjunto de las primitivas en dos grupos:

- Primitivas *seguras*. Primitivas tales que el tipo de sus argumentos y de su valor de salida son tipos de datos, es decir no incluyen el símbolo "→".
- Primitivas *no seguras*. Las primitivas tales que alguno de sus argumentos o su valor de salida no es un tipo de datos.

El conjunto de las primitivas seguras esta formado por todas las primitivas disponibles en $\mathcal{TOY}$ y Curry con las siguientes excepciones:

- En $\mathcal{TOY}$: Los combinadores de order superior utilizados en la entrada/salida.
- En Curry: Como en el caso anterior los combinadores de order superior utilizados en la entrada/salida, y además la primitiva try utilizada en la búsqueda encapsulada de Curry [42].

La incorporación de las primitivas seguras es sencilla, tal y como se muestra en el apartado 5.1.1. En cambio la incorporación de primitivas no seguras no es tan inmediata, por las razones que comentaremos en el apartado 5.1.2. En el caso de la entrada/salida se unen además otras complicaciones, que comentaremos en el apartado 5.5.1, que nos han llevado a no permitir las operaciones de entrada/salida en programas sobre los que se va a utilizar el depurador declarativo (si se utiliza alguna de estas operaciones el depurador da un mensaje de aviso durante la transformación de programas e interrumpe el proceso de depuración).

En cambio sí se permite la depuración de programas Curry incluyendo la primitiva try, gracias a la solución particular para la inclusión de esta primitiva que veremos en la sección 5.4.

5.1.1. Incorporación al Depurador de Primitivas Seguras

Como acabamos de indicar, vamos a llamar seguras a las primitivas p con tipo principal $p :: \tau_1 \to \ldots \tau_n \to \tau$, donde ni en τ_i para $i = 1 \ldots n$, ni en τ aparece el símbolo $\to$.

Resulta razonable suponer que, al ser las primitivas parte del sistema, cualquier hecho básico correspondiente a una primitiva p es válido en el modelo pretendido de cualquier programa. En el caso de las primitivas seguras podemos asegurar además que dentro del código de la primitiva, que como hemos dicho consideramos inaccesible para el depurador, no se van a efectuar internamente llamadas a funciones definidas por el usuario, al no tener

argumentos de tipo funcional. Es decir, que el tipo de las primitivas seguras nos garantiza que no pueden "ocultar" en su código la utilización de alguna regla de función incorrecta definida por el usuario. Como veremos esto no ocurre en el caso de las primitivas no seguras.

Esto no significa que una primitiva segura polimórfica no pueda recibir valores de orden superior como parámetro; sólo que podemos asegurar que estos valores no serán utilizados como funciones, i.e. aplicados a argumentos, debido a que en su tipo principal no se encuentra el símbolo $\rightarrow$.

Por tanto en el "imaginario" subárbol de prueba correspondiente a una llamada a una primitiva segura estaría formado únicamente por nodos válidos. Es fácil razonar (lo veremos en este mismo capítulo, en la proposición 5.3.1 de la página 114) que los nodos válidos de un APA pueden eliminarse sin alterar sus nodos críticos, y esto es lo que hemos hecho en el depurador. En otras palabras: en los árboles de prueba utilizados durante la depuración no habrá ningún nodo asociado a una función primitiva segura, al considerarse estos nodos válidos y ser consecuentemente eliminados.

Para llevar esta idea a la práctica en los depuradores de $\mathcal{TOY}$ y Curry hemos considerado a estas primitivas exactamente como a constructoras de datos durante la fase de introducción de árboles de cómputo. Esta analogía no es arbitraria porque ni las constructoras ni las primitivas están definidas por reglas de función, y en ambos casos sus aplicaciones totales no devolverán árboles de cómputo en el programa transformado. Las aproximaciones correspondientes a aplicaciones de constructoras no aparecen en el APA, e igualmente sucederá con las correspondiente a funciones primitivas seguras.

Así pues, siguiendo las ideas expuestas en el apartado 4.4.2 (pág. 82) para las constructoras, a cada función primitiva segura $p :: \tau_1 \rightarrow \ldots \tau_n \rightarrow \tau$ le corresponderán en el programa transformado n funciones

$$
\begin{array}{l}
p_0^{\mathcal{T}} :: \tau_1 \rightarrow \cdots \rightarrow \tau_n \rightarrow (\tau, cTree) \\
p_0^{\mathcal{T}}\ X_1 \quad\ \rightarrow\ (p_1^{\mathcal{T}}\ X_1, \text{void}) \\
\\
\ldots \\
\\
p_{n-1}^{\mathcal{T}} :: \tau_1 \rightarrow \cdots \rightarrow \tau_n \rightarrow (\tau, cTree) \\
p_{n-1}^{\mathcal{T}}\ \overline{X}_n \ \rightarrow\ (p\ \overline{X}_n, \text{void})
\end{array}
$$

Cada aplicación de p a m argumentos con $m < n$ se sustituirá por la aplicación de la función $p_m^{\mathcal{T}}$ a esos mismos argumentos. Las aplicaciones totales no se sustituirán, como se hacía en el caso de las constructoras.

5.1.2. Primitivas No Seguras

La razón por la que la solución anterior no es aplicable a las primitivas no seguras es doble. Para comprenderlo mejor supongamos por un momento que la función map de tipo principal (A → B) → [A] → [B], cuya definición puede consultarse, por ejemplo, en el programa 4.2.1 de la página 72, fuera una primitiva. Debido a la aparición de → en el tipo de su primer argumento se trataría de una primitiva no segura. Entonces:

1. Al no tener acceso al código de las funciones primitivas no podemos modificar su tipo. Por ejemplo en el caso de map su tipo en el programa transformado debería ser según la transformación descrita en el apartado 4.4.2 (pág. 82): (A → (B,cTree)) → [A] → ([B],cTree), que no es compatible con el tipo original (A → B) → [A] → [B]. Pero al tratarse (en nuestra suposición) de una función primitiva, y por tanto parte del sistema, map conservará en el programa transformado su tipo original, y el programa obtenido será incorrecto desde el punto de vista de los tipos.

   En el caso de las primitivas seguras esto no sucede ya que resulta sencillo probar que la transformación de un tipo τ sin apariciones de → es el propio τ, evitándose este problema.

2. Aunque consiguiéramos evitar el error de tipos la primitiva puede "ocultar" la llamada a alguna función errónea del programa recibida como parámetro. Por ejemplo continuemos pensando en map como función primitiva y consideremos un programa como:

   ```
   % incrementa todos los elementos de L en una unidad
   incList :: [int] -> [int]
   incList L = map inc L

   % incrementa en uno un valor entero
   inc:: int -> int
   inc X = X+2
   ```

   Evidentemente la función inc es errónea, y de hecho un objetivo como incList [1,2] == R computaría la respuesta incorrecta { R ↦ [3,4] }, pero en el programa no existe ninguna llamada a inc y por tanto en el APA obtenido no puede haber ningún nodo crítico asociado a esta función, ya que su llamada se encuentra oculta en el código de map al que estamos suponiendo que no tenemos acceso.

No pensamos que sea imposible depurar programas que incluyan primitivas no seguras, pero hasta el momento no conocemos una solución general. Afortunadamente, y como ya hemos indicado, en el sistema $\mathcal{TOY}$ no existen primitivas no seguras con excepción de los combinadores de entrada/salida, que no están admitidos en la depuración por éstas y por otras razones adicionales que veremos en el apartado 5.5.1. Estás mismas funciones tampoco están admitidas en Curry. En cambio, y como también hemos indicado en la introducción,

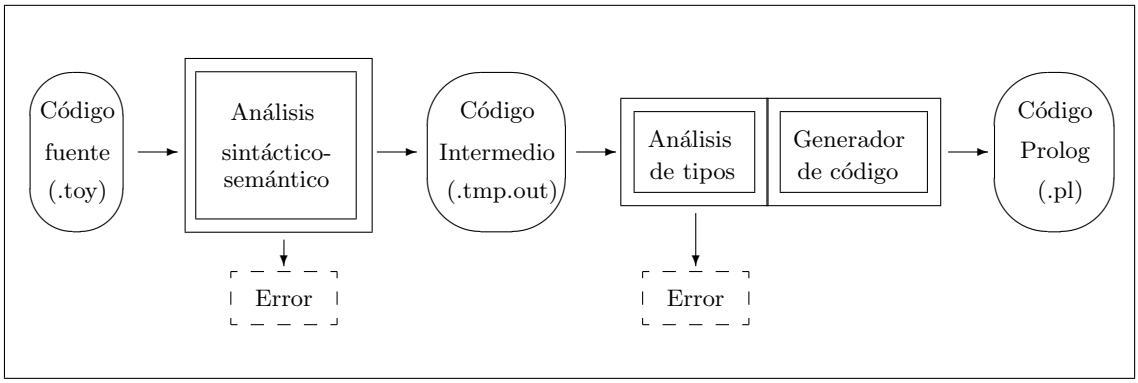


Figura 5.1: Compilación de un programa en $\mathcal{TOY}$

la otra primitiva no segura de Curry, try, sí se admite en los programas a depurar gracias a la solución particular que veremos en el apartado 5.4.

5.2. El depurador declarativo de $\mathcal{TOY}$

En esta sección vamos a describir brevemente la estructura del depurador declarativo incorporado a $\mathcal{TOY}$. Para ello nos fijaremos en primer lugar en la estructura del compilador, todavía sin tener en cuenta el depurador, así como el contexto en el que el usuario decide comenzar la depuración. Esta información nos ayudará a entender mejor las modificaciones realizadas para la incorporación del depurador declarativo, expuestas en el apartado 5.2.3. Finalmente describiremos la primitiva dVal y el tipo de datos de su valor de salida, pVal.

5.2.1. Estructura del Compilador

La figura 5.1 muestra, de forma simplificada pero suficiente para nuestros propósitos, el proceso de compilación de un programa en el sistema $\mathcal{TOY}$, que puede dividirse en dos fases.

La primera fase del compilador toma el fichero con el programa fuente (.toy) como entrada y lo lee, analizándolo sintácticamente para comprobar que se ajusta a la sintaxis del lenguaje. Esta fase suele separarse en dos: análisis lexicográfico y análisis sintáctico, pero por simplicidad hemos identificado ambas en el diagrama. Durante esta primera fase el compilador de $\mathcal{TOY}$ también se realiza comprobaciones semánticas tales como asegurar que todo símbolo utilizado está definido en el programa, o que todas las reglas de una misma función tienen el mismo número de parámetros. Si durante alguna de estas comprobaciones se detecta un error, se muestra un mensaje indicativo al usuario y el proceso se detiene. En otro caso se genera un fichero con el mismo nombre que el del programa fuente pero extensión .tmp.out conteniendo la información analizada. Este fichero está formado por hechos Prolog, lo que facilitará su lectura en la segunda fase al estar el compilador escrito en este lenguaje.

La segunda fase recibe como entrada el fichero intermedio generado por la fase anterior y consta a su ver de dos partes: el inferidor de tipos y el generador de código. El inferidor de tipos comprueba que el programa no contiene errores de tipo. Si se encuentra algún error se indica al usuario y el proceso se detiene. En otro caso se pasa a la generación de código en la que finalmente se general el fichero Prolog (extensión .pl) resultado de la compilación. Una descripción detallada de la generación de código Prolog a partir de código $\mathcal{TOY}$ puede encontrarse en el informe de $\mathcal{TOY}$ [1], basado en un desarrollo de las ideas expuestas en [53].

5.2.2. Inicio de una Sesión de Depuración

Como ya vimos en el capítulo 2, $\mathcal{TOY}$ es un sistema interactivo: tras compilar el programa deseado el usuario puede plantear los objetivos que desee desde la línea de comandos, y el sistema buscará las posibles respuestas mostrándolas una a una. Tras cada respuesta el usuario tiene la posibilidad de indicar que dicha respuesta ha sido incorrecta y que desea utilizar el depurador declarativo para encontrar la fuente del error.

Por ejemplo, supongamos que el usuario ha escrito un programa para ordenar listas de números, y que el programa incluye en particular la definición de función sort con este propósito. Entonces podemos imaginar que el usuario plantea al sistema un objetivo como el siguiente, en el que desea ordenar la lista [3,2,1]:

```
TOY> sort [3,2,1] == R
```

La respuesta de $\mathcal{TOY}$ es:

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

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

En este ejemplo el sistema contesta diciendo que ha encontrado una respuesta para el objetivo representada por la sustitución { R $\mapsto$ [3,2,1] }. A continuación el sistema pregunta si debe buscar más soluciones (respuesta y), abandonar este objetivo n, o depurar d. El usuario escogerá d si, como en este caso, considera que la respuesta anterior es una respuesta incorrecta, y desea comenzar la depuración.

5.2.3. Incorporación del Depurador Declarativo

A partir de la figura 5.1 se deduce que hay dos posibles ficheros de partida para la producción del programa transformado:

- El fichero fuente (.toy) del programa que queremos transformar. En este caso se haría una transformación "fuente a fuente", cuyo resultado sería otro fichero .toy conteniendo el programa transformado. Este programa transformado puede ser posteriormente compilado por el sistema como un programa cualquiera, siguiendo las fases descritas en la figura 5.1.

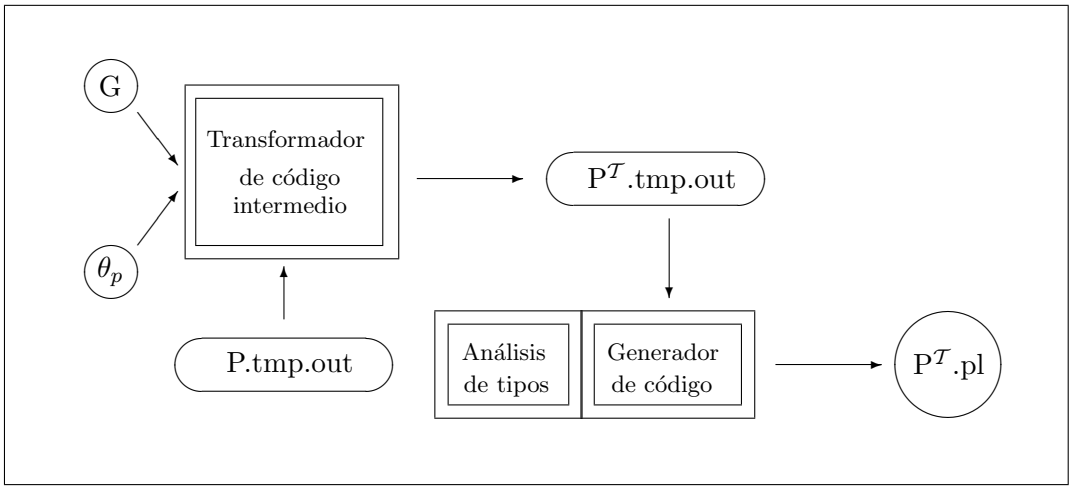


Figura 5.2: Generación de un programa transformado en $\mathcal{TOY}$

- La segunda posibilidad es utilizar el fichero intermedio generado por el compilador al final de la primera fase (de extensión .tmp.out en la figura 5.1) y que está compuesto como ya hemos explicado por hechos Prolog. A partir de este fichero se produciría entonces otro fichero .tmp.out, correspondiente al programa transformado. Podemos asegurar que este fichero intermedio existe en el momento de comenzar la depuración, puesto que como hemos visto en el apartado anterior esto ocurre cuando el usuario detecta una respuesta incorrecta producida utilizando un programa ya compilado. Por tanto en ese momento no sólo el programa fuente (*.toy*) está disponible sino también sus correspondientes ficheros intermedio (*.tmp.out*) y final (*.pl*).

La segunda posibilidad resulta más conveniente por razones de eficiencia. En efecto:

- Leer y tratar el fichero fuente, en formato de texto, es más costoso que leer el fichero intermedio formado por cláusulas Prolog. Eligiendo el fichero intermedio evitamos por tanto la repetición de la fase de análisis sintáctico del programa fuente.

- Evitamos al mismo tiempo el análisis sintáctico y semántico del programa transformado, lo que podemos hacer si suponemos que nuestra herramienta va a generar código sin errores sintácticos. Tampoco es preciso realizar la comprobación de tipos gracias al teorema 4.4.5 que asegura la corrección en cuanto a tipos del programa transformado si el programa original es correcto. Y el programa original es correcto porque ha sido compilado previamente a la depuración, como hemos indicado en el apartado 5.2.2.

Aprovechando estas ideas se ha introducido la transformación de programas en el sistema siguiendo el esquema de la figura 5.2. Como se ve en la figura, se parte del fichero intermedio *P.tmp.out* del programa P que queremos depurar, así como del objetivo G y de la respuesta incorrecta producida θ_p. Como hemos dicho la existencia del fichero *P.tmp.out* está asegurada, pero en cualquier caso el sistema comprueba su existencia y si hubiera sido

eliminado (por ejemplo el usuario lo ha borrado tras la compilación) lo generaría de nuevo a partir del programa fuente.

A este fichero intermedio se le aplica la transformación de programas descrita en el capítulo 4, lo que lleva a un fichero intermedio transformado $P^{\mathcal{T}}.tmp.out$. Finalmente se utiliza el generador de código del compilador de $\mathcal{TOY}$ para producir el programa transformado compilado, $P^{\mathcal{T}}.pl$.

Hay que notar que el fichero intermedio contiene exactamente la misma información que el fichero fuente .toy original, sólo que estructurada mediante hechos Prolog, por lo que la transformación de programas resulta aplicable directamente.

Vamos a detallar el proceso llevado a cabo por el transformador de programas intermedios. Suponemos que en el programa inicial no se incluyen operaciones de entrada/salida. Si durante la transformación de programas se detecta que en el programa se utiliza entrada/salida se avisa al usuario y se interrumpe el proceso.

1. En primer lugar el transformador genera internamente la representación de la nueva regla de programa $\mathsf{solve} = \mathsf{true} <== \mathsf{G}\theta\sigma$ donde σ se define como $\sigma = \{\mathsf{X}_1 \mapsto \mathsf{c}_1, \ldots, \mathsf{X}_n \mapsto \mathsf{c}_n\}$, con $\{\mathsf{X}_1, \ldots, \mathsf{X}_n\}$ las variables en $\mathsf{G}\theta_p$ y $\{\mathsf{c}_1, \ldots, \mathsf{c}_n\}$ constantes nuevas que no aparecen en la signatura del programa contenido en *P.tmp.out*. Así se asegura que estas variables no se instanciarán durante el cómputo, ya que en otro caso estaríamos depurando una respuesta producida diferente, y se garantiza la completitud con respecto a la depuración (definición 4.4.3, pág. 96).

2. Se considera el conjunto S formado por las reglas de función de *P.tmp.out* junto con la regla para solve del punto anterior. Para cada regla $R \in S$:

   - Se aplica las transformación de aplanamiento definida por las reglas (PL_1)-(PL_{10}) de la figura 4.1 (pág 77) obteniendo una regla $S_{\mathcal{A}}$. En esta etapa las funciones primitivas contenidas en la regla se tratan igual que las funciones definidas.
   - Se aplica a $S_{\mathcal{A}}$ la transformación (TR_1)-(TR_{11}) de la figura 4.2 (pág 84). Durante esta etapa las primitivas contenidas en la regla se tratan como si fueran constructoras. La regla transformada obtenida pasa a ser parte del fichero de salida. $P^{\mathcal{T}}.tmp.out$.

3. Se añaden al programa transformado las funciones auxiliares para constructoras, funciones y primitivas indicadas en los apartados 4.4.3 (para las constructoras y funciones) y 5.1 (para las primitivas). Obsérvese que podemos asegurar que el programa sólo contiene primitivas seguras por lo que el esquema descrito para estas funciones en la sección 5.1 resulta aplicable. Esto es así porque estamos suponiendo que el programa no contiene operaciones de entrada/salida, con lo que aseguramos en particular que no se utilizan los combinadores de entrada/salida de orden superior, que son las únicas primitivas no seguras en $\mathcal{TOY}$.

4. Por último se añaden al programa de salida la definición del tipo de datos cTree y las reglas de la función ctClean.

La generación del programa transformado y su compilación, siguiendo el esquema de la figura 5.2, requieren un consumo de tiempo apenas apreciable, menor que el de la compilación del programa original. Esto no significa que el usuario no tenga que esperar apenas desde que decide utilizar el depurador hasta que la navegación comienza. Tras la generación del programa aún hay que obtener el árbol de cómputo y este proceso, descrito en el capítulo siguiente, sí exigirá un consumo elevado de recursos, tanto en tiempo como en memoria y constituye el mayor inconveniente de la depuración declarativa basada en la transformación de programas.

5.2.4. La primitiva dVal

Esta primitiva, con tipo principal dVal :: A → pVal ya fue comentada en el apartado 4.4.3 del capítulo anterior (pág. 83). Su objetivo es devolver una representación de su parámetro tal que pueda usarse durante la navegación para reconstruir el hecho básico asociado a cada nodo. Aunque en el citado apartado suponíamos por simplicidad que el valor pVal era simplemente una cadena de caracteres (y así es en el caso del depurador de Curry), en el depurador de $\mathcal{TOY}$ nos ha resultado más interesante hacer que el tipo pVal fuera un tipo más estructurado, que nos permitiera manejar de forma sencilla la información asociada a cada hecho básico. El tipo pVal está predefinido en $\mathcal{TOY}$ como:

```
data pVal = pValBottom | pValVar [char] | pValChar char |
            pValNum [char] | pValApp [char] [pVal]
```

La constructora de datos pValBottom se utilizará como representación de las llamadas que hayan quedado sin evaluar durante el cómputo, mientras que pValVar, pValChar y pValNum representarán una variable, un carácter y un valor numérico respectivamente. La constructora pValApp representa la aplicación de su primer argumento a los parámetros de tipo pVal incluidos en la lista dada como segundo argumento. Las cadenas de caracteres se representarán como aplicaciones de la constructora de listas (:).

Por ejemplo si asumimos la existencia de una regla de programa id X = X, los siguientes objetivos se cumplen con la respuesta indicada:

```
TOY> dVal id == R
   yes
   R == (pValApp "id" [])

TOY> dVal (id 3) == R
   yes
   R == pValBottom

TOY> F == A, dVal F == R where F = id 3
   yes
   A = 3
   R == (pValNum "3")
```

El primer caso representa la aplicación de *id* a 0 argumentos, y así lo indica la representación pValApp "id"[]. En el segundo el valor (id 3) corresponde a una llamada aún no evaluada y por tanto el valor devuelto es pValBottom. En el tercer objetivo en cambio la llamada sí que ha sido evaluada previamente y por eso dVal devuelve su valor computado pValNum 3.

5.3. El depurador Declarativo de Curry

Tal y como comentamos al comienzo del capítulo la estructura del depurador de Curry, desarrollado en colaboración con Wolfgang Lux de la universidad de Münster, es muy similar a la del depurador de $\mathcal{TOY}$. En Curry en lugar de generarse un fichero intermedio de cláusulas Prolog se genera un fichero en un formato XML conocido como *Flat Curry* que puede consultarse en

`http://www.informatik.uni-kiel.de/ mh/curry/flat`

que es sobre el que se aplica la transformación de programas. Aparte de esta diferencia, el esquema del depurador sigue siendo el reflejado en la figura 5.2. En el fichero Flat Curry se encuentra la misma información que en el fichero original salvo por la eliminación de las definiciones de funciones locales y expresiones lambda, que han sido transformadas en nuevas funciones del programa por el procedimiento conocido como *lambda-lifting* [48].

Las únicas diferencias entre ambos depuradores se deben a las diferencias en los lenguajes, y se discuten en los siguientes apartados, dejándose para la siguiente sección el tratamiento de la búsqueda encapsulada.

5.3.1. Módulos y Funciones Fiables

Una diferencia de Curry con respecto a $\mathcal{TOY}$ es la existencia de módulos análogos a los de Haskell [75]. Esto no supone ningún problema para la transformación de programas, ya que en el fichero *Flat Curry* que el depurador toma como entrada los nombres de las funciones han sido modificados, anteponiéndoles el nombre del módulo en el que están definidas cuando resulta necesario para evitar conflictos entre diversos módulos. Si tenemos un único módulo éste recibe automáticamente el nombre de main aunque el usuario no lo indique explícitamente. Por tanto funciones con los mismos nombres en módulos distintos pasan a tener en el fichero Flat Curry nombres diferentes, y la transformación de programas puede aplicarse sin ninguna modificación. Por ejemplo, en la sesión de depuración del programa ejemplo B.4.1 (pág. 300) del apéndice B se pueden encontrar hechos básicos como

$$\mathsf{main.primes}() \rightarrow [2,3,4,5,6\,|\,\_]$$

refiriendo así al usuario a la función primes del módulo main.

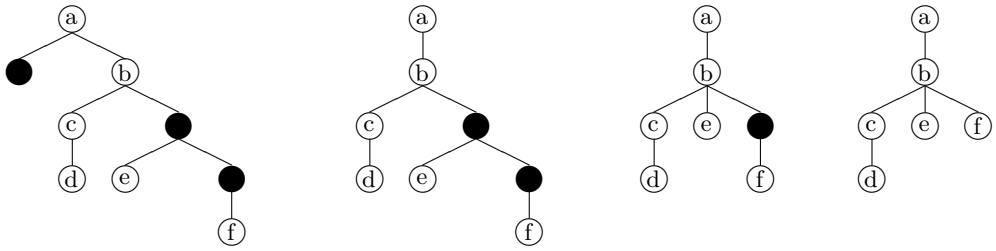


Figura 5.3: Eliminación de nodos de un árbol

Hemos aprovechado esta característica de Curry para incorporar la posibilidad para al usuario de indicar que ciertos módulos completos son *fiables*. Esto significa que el navegador nunca hará preguntas sobre nodos asociados a funciones definidas en estos módulos, considerándolos válidos. Nótese que, a diferencia de las primitivas seguras, las funciones fiables sí son transformadas normalmente, ya que en este caso sí se dispone de reglas de programa que transformar. La diferencia entre la transformación de una función fiable y la de una función cualquiera es que el código transformado de la función fiable incluye en sus reglas transformadas una "marca" para avisar al navegador de que sus nodos se deben tomar como válidos, aunque no así los de sus hijos, ya que las funciones fiables pueden incluir funciones de orden superior cuyos parámetros o resultados sean a su vez funciones.

En la práctica los nodos asociados a funciones fiables son eliminados del APA final utilizando la siguiente operación:

Definición 5.3.1. Sea T un árbol y $N \in T$ un nodo tal que $N \neq$ *raíz(T)*. Sea M el padre de N en T (es decir $N \in hijos(T, M)$). Entonces la notación $T - N$ representará el nuevo árbol obtenido al eliminar N de T, dejando los subárboles hijos de N como subárboles hijos de M.

La operación de eliminación tiene algunas propiedades interesantes que enuncia la siguiente proposición. En el enunciado, y de aquí en adelante, utilizaremos la notación $|T|$ para representar el número de nodos en un árbol T.

Proposición 5.3.1. *Sea P un programa y T un árbol de cómputo obtenido con respecto al programa P. Sea $\mathcal{I}$ la interpretación pretendida de P con respecto a la cual se determina la validez o no validez de los nodos de T. Sea por último $N \in T$ tal que* $\mathrm{N} \neq \mathrm{raíz(T)}$. *Entonces se verifica:*

1. $|T - N| = |T| - 1$.
2. *Todos los nodos de $T - N$ están en T.*
3. *Sean $M, M' \in (T - N)$ tales que $M' \in hijos(T - N, M)$. Entonces $M' \in hijos(T, M)$.*

4. Si $N \in T$ es válido, entonces:

(i) T tiene algún nodo crítico si y sólo T-N tiene algún nodo crítico.

(ii) Todo nodo crítico en T-N es también crítico en T.

Demostración

Los 3 primeros puntos son obvios a partir de la definición 5.3.1: en $T - N$ hay un nodo menos que en T, la operación de eliminación no añade nodos nuevos, y si un nodo M' es hijo de un nodo M en $T - N$ sucede lo mismo en T. Probamos entonces los apartados del punto 4, comenzando por el segundo:

(ii) Sea $N \in T$ válido tal que $T - N$ tiene un nodo crítico B. Al ser B crítico todos sus hijos en $T - N$ son válidos. Además $B \in T$ (apartado 2 de la proposición), y todo hijo de B en T es también hijo de B en $T - N$ (apartado 3), por lo que todos los hijos de B en T son válidos y B es crítico también en T.

(i) Examinemos las dos implicaciones contenidas en este apartado por separado.

Sea $N \in T$ válido, y supongamos que T contiene algún nodo crítico. Debemos probar que entonces $T - N$ también contiene un nodo crítico. Si N tiene algún hijo M no válido se tiene que por el corolario 2.2.2 (pág. 28) que *subárbol(T,M)* contiene un nodo crítico, y al verificarse *subárbol(T,M) = subárbol(T-N,M)* dicho nodo crítico lo será también en T. Si por el contrario todos los hijos de N en T son válidos, nos fijamos en un nodo crítico B de T y probamos que B también es crítico en $T - N$. Observamos en primer lugar que $B \neq N$ (ya que B no es válido y N sí), y por tanto $B \in (T - N)$. Hay dos situaciones posibles:

1. Si $N \notin hijos(T, B)$ entonces $hijos(T, B) = hijos(T - N, B)$ y por tanto B crítico en $T - N$.
2. Si $N \in$ *hijos(T,B)*, por la definición de $T - N$ todos los nodos de *hijos(T,N)* pasan a formar parte de *hijos(T-N,B)*. Pero como estamos asumiendo que todos ellos son válidos se tiene que B es todavía un nodo crítico.

Para la segunda implicación, sea $N \in T$ válido tal que $T - N$ tiene un nodo crítico B. Entonces por el apartado (ii) este nodo es también crítico en T.

■

El punto 4 de la proposición asegura que la eliminación de nodos válidos es segura, en el sentido de que si encontramos un nodo crítico en el árbol simplificado dicho nodo también es crítico en el árbol original (apartado (ii)), mientras que la no existencia de nodos críticos en el árbol simplificado asegura que tampoco los había en el árbol original (apartado (i)).

La figura 5.3 muestra un ejemplo en el que los nodos fiables han sido marcados en negro. El árbol de la izquierda es el APA inicial, mientras que en cada paso hacia la derecha se ha eliminado el nodo fiable más cercano a la raíz según la operación definida anteriormente. Es

interesante comprobar que esta eliminación modifica la estructura del árbol, aumentando el número de hijos de los nodos no fiables.

Para incluir esta posibilidad en el depurador de Curry hemos tenido que realizar los siguientes cambios con respecto a las ideas utilizadas en el depurador de $\mathcal{TOY}$:

1. El tipo CTree que representa los árboles de cómputo está en Curry definido de la siguiente forma:

   ```
   data CTree =  CTreeNode FuncName [Arg] Result RuleNumber [CTree] |
                 EmptyCTreeNode [CTree] |
                 CTreeVoid

   type FuncName, RuleNumber = String
   type Arg, Result = PVal
   type PVal = String
   ```

   El papel de las constructoras CTreeNode y CTreeVoid es análogo al de ctNode y ctVoid respectivamente en el caso de $\mathcal{TOY}$ (definidas en el apartado 4.4.1, pág. 82). La constructora EmptyCTreeNode se emplea para las funciones fiables. Como se puede ver no almacena información sobre el nodo actual, que será eliminado, sino sólo sobre sus hijos que le sustituirán.

2. Durante la transformación de programas, a las funciones fiables no se les aplica la regla (TR_1) de la figura 4.2 (pág. 84), sino la regla alternativa:

   $$\frac{t_1 \Rightarrow^{\mathcal{T}} s_1 \ \ldots \ t_n \Rightarrow^{\mathcal{T}} s_n \quad r \Rightarrow^{\mathcal{T}} s \quad C \Rightarrow_{\mathcal{A}} (C^{\mathcal{T}}; (r_1, T_1), \ldots, (r_m, T_m))}{\begin{array}{l} f\ \overline{t}_n \to r \Leftarrow C \ \ \Rightarrow^{\mathcal{T}} \ f\ \overline{s}_n \ \to (s, T) \Leftarrow C^{\mathcal{T}}, \\ \quad EmptyCTreeNode\ (ctClean\ [(dVal\, r_1, T_1), \ldots,\ (dVal\, r_m, T_m)]) \to T \end{array}}$$

   La estructura de la regla es idéntica a la de (TR_1), excepto por la construcción del árbol T, en la que se ha suprimido toda la información relativa a la regla por corresponder a una regla fiable.

3. También se ha modificado la función ctClean para eliminar los nodos de tipo EmptyCTreeNode:

   ```
   ctClean :: [(String,CTree)] -> [CTree]
   ctClean []         = []
   ctClean ((p,x):xs) =
      if p=="_" then rest
                else case x of
                       CTreeVoid            -> rest
                       EmptyCTreeNode trees -> trees ++ rest
                       CTreeNode _ _ _ _ _  -> x : rest
               where
                rest = ctClean xs
   ```

Para comprender este código hay que observar en primer lugar que en Curry las variables de tipo y de programa se escriben comenzando con minúscula, al contrario de lo que sucede en $\mathcal{TOY}$, mientras que las constructoras de tipo y de datos comienzan con mayúscula cuando en $\mathcal{TOY}$ lo hacen con minúscula. Aparte de las diferencias de sintaxis, el propósito de esta función es análogo a su correspondiente versión en $\mathcal{TOY}$ (definida en la página 86). La única diferencia está en el tratamiento de los nodos con constructora EmptyCTreeNode que, como se ve en la regla, son simplemente sustituidos por sus hijos, llevando así a la práctica la operación de eliminación de la definición 5.3.1 tal y como se describe en el ejemplo de la figura 5.3.

Un módulo marcado por defecto como fiable es el *preludio* de Curry, módulo incluido automáticamente en todo programa Curry y que incluye funciones usuales tales como map o take. En el programa B.4.1 (pág. 300) del apéndice B se puede ver una sesión de depuración para un programa que utiliza estas funciones.

Entre el trabajo futuro para el depurador de $\mathcal{TOY}$ está la posibilidad de incluir funciones fiables, ya sea marcando módulos completos como fiables como acabamos de ver en el caso de Curry o funciones individuales antes de comenzar la navegación. Si esto se hiciera el tipo de datos para representar los árboles de cómputo en $\mathcal{TOY}$ sería el mismo que hemos visto para Curry, al igual que sucedería con la función ctClean. Actualmente, y como solución alternativa, el navegador gráfico $\mathcal{DDT}$ incluido en el depurador de $\mathcal{TOY}$, que veremos en la sección 6.2 (pág. 136) da al usuario la posibilidad de marcar como fiables las funciones que desee así como de eliminar los nodos asociados a estas funciones durante la propia navegación.

5.3.2. Definiciones Locales Generales

En Curry se permiten definiciones locales del mismo tipo que en Haskell [75], y por tanto más generales que en $\mathcal{TOY}$, además de *expresiones lambda*. Como hemos señalado en el fichero en formato *Flat Curry* que el depurador toma como entrada todas estas definiciones locales han sido ya convertidas en nuevas funciones del programa y por tanto la transformación de programas es aplicable también en este caso. Un ejemplo puede verse en el programa B.4.2 (pág. 302) del apéndice B.

El problema surge a la hora de la navegación, donde se debe dar la suficiente información al usuario para permitirle identificar a qué expresión lambda o definición local se refieren los hechos básicos presentados. La idea es incluir como parte del nombre de la función asociada a la definición local el nombre de la función en la que estaba definida, así como la posición en la que se encontraba. Por ejemplo un hecho básico como

$$\mathsf{main.new.newQueue(Empty,\ 1)} \rightarrow \mathsf{Queue\ 1\ Empty}$$

tomado de la sesión de depuración del citado programa B.4.2, se refiere a una función newQueue definida localmente como parte de la función new del módulo principal main. Análogamente, y de la misma sesión de depuración, el hecho básico

main.listToQueue._#lambda1(Empty, 1) → Queue 1 Empty

se refiere a una llamada a la primera función lambda (según el orden textual) dentro de la función listToQueue del módulo main.

5.4. Búsqueda Encapsulada

La búsqueda encapsulada incluida en Curry permite el control, desde el propio lenguaje, de pasos de cómputo no deterministas. Se utiliza habitualmente para definir estrategias de búsqueda diferentes de la usual búsqueda en profundidad, como por ejemplo la búsqueda en anchura. Aunque es muy habitual en los programas Curry no puede depurarse con las técnicas que hemos presentado hasta el momento, al estar basada en una primitiva de las que calificamos como no segura en el apartado 5.1. En [18] estudiamos una solución para la depuración de programas utilizando la búsqueda encapsulada. Esta solución está implementada en el depurador declarativo de Curry y es la que comentamos a continuación.

5.4.1. Un Ejemplo de Programa erróneo en Curry

Vamos a utilizar un ejemplo sencillo de programa erróneo para explicar los conceptos básicos de la búsqueda encapsulada y de su depuración.

Ejemplo 5.4.1. *Último elemento de una lista.*

```
append:: [a]->[a] ->  [a]
append []  ys = ys
append (x:xs)  ys = x : append xs ys

last:: [a]->a
last xs | append [y] ys =:= xs = y where y,ys free
```

Salvo las ya comentadas diferencias de sintaxis, la definición de la función append es análoga a la que se haría en $\mathcal{TOY}$ para la concatenación de listas. El propósito de la función last es obtener el último elemento de la lista que recibe como parámetro. La única regla de esta función dice que y es el último elemento de una lista xs si existe algún valor ys tal que append [y] ys es xs. El símbolo =:= representa la igualdad estricta en Curry. Escrita en sintaxis $\mathcal{TOY}$ la regla sería:

```
last Xs = Y  <== append [Y] Ys == Xs
```

Las variables que aparecen en la condición pero no en el lado derecho de la regla deben indicarse explícitamente en Curry seguidas de la palabra reservada free, como se ve en la definición de last. Evidentemente la regla es errónea, ya que define el primer y no el último elemento de una lista.

En Curry los objetivos son expresiones y la respuesta computada el resultado de evaluar la expresión a algún patrón total, incluyendo posibles sustituciones para algunas variables de la expresión. En este sentido el cómputo asociado a la evaluación de una expresión e en Curry se corresponde con el de un objetivo de la forma $e == X$ en $\mathcal{TOY}$, con X una variable que no aparece en e. En nuestro ejemplo, la evaluación de la expresión last [1,2,3] produce en Curry la respuesta 1, incorrecta con respecto a la interpretación pretendida del programa.

5.4.2. La Primitiva try

Volviendo a la búsqueda encapsulada, ésta se basa en la primitiva del lenguaje try de tipo

$$\mathsf{try::(a \to Success) \to [a \to Success]}$$

Success es una constructora de un tipo al que pertenece un único valor, devuelto por la primitiva de aridad 0 success.

Aunque no tiene una semántica declarativa, try puede definirse en términos de su comportamiento operacional. El cómputo de una llamada de la forma try g busca patrones totales t tales que x=:=t sea una respuesta computada para el objetivo g x en Curry. Hay 3 posibles tipos de resultados:

1. La lista vacía []. Esto significa que no existe ningún patrón total t tal que una llamada g t pueda tener éxito. De esta forma try se puede utilizar para controlar el fallo en Curry.

2. Una lista con un sólo elemento [g']. En este caso existe un patrón total t tal que g t tiene éxito empleando sólo pasos deterministas. Por ejemplo, utilizando el ejemplo 5.4.1, la llamada try (λx $\to$ x =:= last [1,2,3]) devolvería la lista [λx $\to$ x =:= 1]. Al elemento en la lista de salida se le llama *forma resuelta* del objetivo g inicial. Evidentemente a menudo estamos interesados en saber no sólo que existe un valor t como el indicado, sino en conocer su valor. Para esto podemos utilizar la función

   $$\mathsf{unpack\ g \mid g\ x = x\ where\ x\ free}$$

   y evaluar entonces un objetivo como

   $$\mathsf{unpack\ head\ (try\ (\lambda x \to x =:= last\ [1,2,3])}$$

   para obtener el valor 1. Hay que indicar que aquí estamos forzando la introducción de la búsqueda encapsulada para nuestra explicación, aunque ésta no es realmente necesaria en relación con el ejemplo 5.4.1. En el programa B.4.3 (pág. 304) del apéndice B presenta un ejemplo de aplicación de la búsqueda encapsulada en un contexto en el que su uso resulta realmente adecuado.

3. Una lista con más de un elemento. Esto quiere decir que durante el cómputo de g x la búsqueda encapsulada a encontrado un paso no determinista, es decir un punto en el que más de una regla de una función es aplicable. La lista devuelta corresponde a las posibles continuaciones del cómputo. Por ejemplo el resultado de evaluar la expresión try (λx $\rightarrow$ append x [] =:= [1,2,3]) es la lista

   ```
   [(λx → x =:= [] & append [] [] =:= [1,2,3]),
    (λx → let y,ys free in
          x =:= (y:ys) & append (y:ys) [] =:= [1,2,3])]
   ```

   Cada uno de los dos valores correspondería a la continuación mediante una de las dos reglas de append. La primera obliga a que x sea la lista vacía y la segunda a que sea una lista de la forma y:ys. El operador & representa en Curry la conjunción de dos predicados.

Utilizando este predicado resulta sencillo definir funciones que representen, por ejemplo, la búsqueda en anchura de soluciones para un cierto objetivo g del tipo requerido por el tipo de try:

```
bfs g =   step [g]
  where   step     []          = []
          step     (g:gs)      = collect (try g) gs
          collect  []          gs    = step gs
          collect  [g]         gs    = g : step gs
          collect  (g1:g2:gs)  gs'   = step (gs' ++ g1:g2:gs)
```

La función bfs devuelve la lista (posiblemente infinita) de formas resueltas de g. Para ello utiliza una función auxiliar step que generaliza la bfs a una lista de objetivos. La función step llama a otra función definida localmente collect pasándole como primer parámetro el resultado de evaluar try sobre el primer elemento de la lista y como segundo parámetro el resto de la lista. Si el resultado devuelto por try es la lista vacía collect recolecta los resultados obtenidos a partir del resto de la lista. Si por el contrario try ha devuelto un único valor, éste corresponde a un cómputo finalizado en forma resuelta y collect lo recopila junto con los resultados obtenidos del resto de la lista. Finalmente si hay más de una posible continuación, éstas se almacenan al final de la lista para que en la siguiente llamada a step se evalúe try sobre el siguiente elemento de la lista, simulándose de esta forma la búsqueda en anchura.

5.4.3. Depurando Programas con Búsqueda Encapsulada

La función primitiva try incluye parámetros de orden superior y por tanto es una primitiva no segura. Como hemos visto en la sección 5.1 esto significa que el tipo de la función

deja de ser adecuado en el programa transformado, además de que los cómputos, quizás erróneos, ocurridos dentro de la primitiva quedan ocultos al depurador.

Consideremos en primer lugar el primer problema, el error de tipos. El tipo de $\mathsf{try}^{\mathcal{T}}$ se debería obtener aplicando la transformación de tipos descrita en el apartado 4.4.2 (pág. 82) al tipo principal de try, (a → Success) → [a → Success]. El resultado sería:

$$\mathsf{try}^{\mathcal{T}} :: (\mathsf{a} \to (\mathsf{Success},\mathsf{CTree})) \to ([\mathsf{a} \to (\mathsf{Success},\mathsf{CTree})],\mathsf{CTree})$$

Pero al ser try una primitiva no podemos realizar esta transformación, al no disponer de sus reglas de programa, teniendo que conformarnos con la función original try también en el programa transformado. El problema viene porque el resto de las funciones del programa transformado precisan de la función transformada $\mathsf{try}^{\mathcal{T}}$ con el tipo indicado y la sustitución de esta función por la función try original con el tipo sin transformar produciría un error de tipo, tal y como indicamos en la sección 5.1. Lo que vamos a hacer es escribir una función $\mathsf{try}^{\mathcal{T}}$ en el programa transformado que sea capaz de convertir el tipo de sus argumentos y valor de salida al tipo de estos valores en la primitiva try original, de forma que ésta pueda seguir utilizándose en el programa transformado.

Podemos apreciar tres diferencias entre el tipo principal y el que debería ser el tipo transformado: el tipo de su parámetro, el árbol de cómputo asociado al valor devuelto por la función y el tipo de los valores en la lista que constituye el resultado.

Teniendo estas diferencias en cuenta podemos incluir en el programa transformado las siguiente reglas de función:

```
try^T:: (a → (Success,CTree)) → ([a → (Success,CTree)],CTree)
try^T g = (map wrap (try (unwrap g)),CTreeVoid)

unwrap:: (a → (Success, CTree)) → a → Success
unwrap   g   x = r where (r,t) = g x

wrap:: ((a → Success) → a → (Success, CTree)
wrap      g   x = (g x,CTreeVoid)
```

Es fácil comprobar que la regla de función $\mathsf{try}^{\mathcal{T}}$ tiene ahora el tipo de su declaración de tipos. La función unwrap se encarga de extraer el valor, sin árbol de cómputo, asociado al parámetro g, mientras que wrap convierte la lista producida por try, de tipo [a → Success], en una lista de tipo [a → (Success,CTree)] tal y como requiere el tipo del resultado de $\mathsf{try}^{\mathcal{T}}$. Los árboles de cómputo asociados tanto al valor de salida de la función transformada como a los cómputos en la lista de resultados son árboles vacíos.

Con esta solución se puede ya incluir la búsqueda encapsulada en los programas a depurar sin obtener errores de tipos. Sin embargo el segundo problema, la posible pérdida de partes del APA, y quizás del (o los) nodo(s) crítico(s), aún subsiste. Por ejemplo el objetivo

```
unpack (head (try (λx → x =:= last [1,2,3])))
```

devuelve de nuevo el resultado incorrecto 1.

Suponiendo que las funciones head y unpack incluidas en el preludio de Curry y por tanto marcadas como fiables, la sesión de depuración para el objetivo anterior es:

```
Entering debugger...

No error has been found
```

El depurador no llega a hacer ninguna pregunta al usuario y termina la sesión sin encontrar el error. En un principio esto es sorprendente porque el objetivo, recordando la técnica para la depuración explicada en la sección 4.4.6 (pág. 96), se incluye en el programa como una regla de función:

$$\mathsf{solve} \mid 1 =:= \mathsf{unpack}\ (\mathsf{head}\ (\mathsf{try}\ (\lambda \mathsf{x} \rightarrow \mathsf{x} =:= \mathsf{last}\ [1,2,3]))) = \mathsf{true}$$

y esta función incluye una llamada a last en su condición que debería dar lugar a un nodo hijo de la raíz en el APA final. La razón por la que esto no ocurre es que la transformación se aplica, como hemos dicho, al programa Flat Curry, en el que las declaraciones locales y expresiones lambda han sido transformadas en nuevas funciones del programa. El programa a transformar incluye entonces reglas del estilo:

```
solve | 1 =:= unpack (head (try solve.#lambda1 )) = true
solve.#lambda1 x = x =:= last [1,2,3]
```

Ahora tenemos que solve no incluye ninguna llamada a last. La llamada a last ha pasado a formar parte de la función auxiliar solve.#lambda1, para la que no hay ninguna llamada en solve. La única aparición de solve.#lambda1 en solve es como argumento de orden superior para try, que es dónde se "pierde" el nodo crítico asociado a last.

Un examen más cuidadoso de lo que ocurre nos revela que es en realidad la función auxiliar unwrap la que obtiene el árbol para last (llamado t dentro de la función). Posteriormente wrap añade el árbol vacío a la forma resuelta obtenida, cuando en realidad sería deseable que el árbol devuelto fuera el árbol t extraído por unwrap.

La solución consiste en devolver el árbol de cómputo no a través del resultado sino a través del argumento de unwrap:

```
unwrap :: (a→(Success,CTree)) → (a,CTree) → Success
unwrap   g   (x,t)   | r =       t =:= t' where (r,t') = g x

wrap :: ((a,CTree) → Success) → a → (Success, CTree)
wrap      g   x       | g (x,t)   = (success,t) where t free
```

Con estas definiciones try' todavía devuelve valores de la forma (valor, CTreeVoid) como el resto de las primitivas, pero cada elemento en la lista valor (de tipo $\alpha \rightarrow$ (Success,CTree)) producirá el árbol de cómputo asociado a su correspondiente subcómputo, si éste tiene éxito. La condición t =:= t' en unwrap hace que la función que se pasa a try devuelva en su

argumento el árbol de cómputo del objetivo original g, mientras que la condición g (x,t) en wrap sirve para instanciar el valor de la variable t al árbol de cómputo asociado a la forma resuelta.

Volviendo al objetivo de nuestro ejemplo, ahora el depurador es capaz de localizar la regla errónea:

```
Entering debugger...

Considering the following basic facts:
1. solve#lambda1(1)  -> Success
Are all of them valid? (y/n) n

Considering the following basic facts:
1. last([1,2,3])  -> 1
Are all of them valid? (y/n) n

Considering the following basic facts:
1. append([1], [2,3])  -> [1,2,3]
Are all of them valid? (y/n) y

 ** Function last is incorrect (last([1,2,3])  -> 1) **
```

La primera pregunta se refiere a la expresión lambda del objetivo inicial

$$\mathsf{unpack\ (head\ (try\ (\lambda x \rightarrow x =:= last\ [1,2,3])))}$$

y pregunta si es correcto que ésta tenga éxito con el parámetro 1, o lo que es lo mismo que si 1 =:= last [1,2,3] debe tener éxito, lo que es incorrecto.

Otro ejemplo de sesión de depuración para búsqueda encapsulada se puede encontrar en el ya citado ejemplo B.4.3 (pág. 304) del apéndice B.

5.5. Limitaciones del depurador

El depurador de respuestas incorrectas $\mathcal{TOY}$ tiene actualmente dos limitaciones:

1. Los programas considerados no deben contener operaciones de entrada/salida.
2. Tampoco deben hacer uso de restricciones que no sean de igualdad estricta. En particular, las restricciones de desigualdad y las restricciones aritméticas disponibles en TOY no están aún soportadas por el depurador.

En el caso de Curry sólo la primera limitación es relevante, al no permitir este sistema el uso de restricciones aritméticas ni de desigualdad.

Vamos a explicar las razones de cada una de estas limitaciones en el ámbito de $\mathcal{TOY}$. La discusión del primer apartado es igualmente válida en Curry simplemente adaptando la sintaxis.

5.5.1. Entrada/Salida

Comenzamos por la limitación del depurador declarativo a programas sin entrada /salida. La descripción detallada de la entrada/salida en $\mathcal{TOY}$ (que es similar a la de Haskell [75] o Curry [39]) se puede consultar en [1], por lo que aquí nos limitaremos a referirnos a las primitivas que definen la entrada/salida básica sin profundizar en su significado.

Son varias las razones que impiden la depuración de programas en estas condiciones. Para explicarlas vamos a utilizar un ejemplo:

Ejemplo 5.5.1. *Programa con entrada salida*

```
pregunta:: io unit
pregunta = putStr "Introduzca un dígito: " >> getLine >>= answer

answer:: [char] -> (io unit)
answer [C|Rs] = isNumber (ord C)

isNumber:: int -> (io unit)
isNumber N = if (N>=0 && N<10) then putStrLn "Gracias!"
                               else putStrLn "No es un dígito válido!"

false && Y = false
true && Y = Y
```

El programa pide un dígito utilizando la primitiva de entrada/salida getLine y si el valor introducido corresponde efectivamente a un dígito escribe por pantalla el mensaje *"Gracias!"*, mientras que si no lo es escribe *"No es un dígito válido!"* utilizando en cualquier caso la primitiva putStrLn. Para combinar diferentes operaciones de entrada salida se utilizan los combinadores (definidos como funciones primitivas)

$$\begin{array}{ll} (>>)::\ & (\mathsf{io\ A}) \rightarrow (\mathsf{io\ B}) \rightarrow (\mathsf{io\ B}) \\ (>>=)::\ & (\mathsf{io\ A}) \rightarrow (\mathsf{A} \rightarrow (\mathsf{io\ B})) \rightarrow (\mathsf{io\ B}) \end{array}$$

cuyo uso se puede ver en la única regla de la función de pregunta. En $\mathcal{TOY}$ también se puede utilizar la notación do [50], pero ésta se reduce internamente a (>>=) y a (>>).

El programa tiene un error en la función answer, que trata de convertir el primer carácter de la lista de caracteres recibida como entrada en su valor numérico. Para ello utiliza la función ord que devuelve el *código ascii* asociado a un carácter. Pero con esta definición se tiene, por ejemplo, answer "2" $\rightarrow$ 50 (50 es el código ascii de '2'), cuando lo que se desea es answer "2" $\rightarrow$ 2 . La definición correcta sería: answer [C|Rs] = isNumber ((ord C) - (ord '0')) , ya que '0' es el dígito con menor código ascii y todos los dígitos tienen códigos consecutivos.

Podemos observar que el programa tiene un comportamiento anómalo en la respuesta al siguiente objetivo:

```
TOY> pregunta == R
Introduzca un dígito: 5
No es un dígito válido!

      yes
      R == << process >>
```

La respuesta R == <<process>> corresponde a un valor de tipo io unit que no puede ser mostrado al usuario al tratarse de un tipo abstracto de datos.

Veamos ahora las dificultades que plantea la depuración de este objetivo:

1. No se puede siquiera hablar de una respuesta incorrecta para el objetivo. En efecto, el mensaje *"No es un dígito válido!"* no es parte de la respuesta, que es { R $\mapsto$ <<process>> }, que es lo que el usuario espera de un programa con entrada/salida y de la que no se puede asegurar su validez/no validez a falta de más información sobre el valor real de <<process>>, valor que como hemos dicho no está disponible para el usuario. Si el depurador produjera un árbol de cómputo a partir de este cómputo, el nodo inicial, único hijo de la raíz, sería pregunta $\rightarrow$ <<process>>, que es un nodo del que el usuario no puede determinar su validez.

2. El programa utiliza, como mencionamos más arriba, la primitiva, (>>=):: (io A)$\rightarrow$(A$\rightarrow$ (io B))$\rightarrow$(io B), que es no segura debido a que sus argumentos tienen tipos funcionales (por ejemplo (A$\rightarrow$ (io B))). Tal y como explicamos en la sección 5.1, una función de esta clase puede hacer internamente llamadas a funciones incorrectas del programa. Este es el caso del ejemplo, porque la única llamada a la función incorrecta answer se hace desde el código del combinador (>>=) y por tanto no formaría parte de un hipotético APA del cómputo. Obsérvese que en el programa no hay ninguna llamada a answer, por lo que ningún nodo del APA estaría asociado a esta función.

3. Finalmente, también como consecuencia de ser (>>=) no segura, al transformar el programa obtendríamos un programa mal tipado, ya que el argumento answer de (>==) pasaría a tener un tipo [char] $\rightarrow$ (io unit, cTree), incompatible con el tipo esperado (A $\rightarrow$ (io B)).

4. Un problema añadido, pero que también se debe tener en cuenta, se tendría durante la repetición del cómputo mediante el programa transformado, ya que si incluye las operaciones de entrada/salida del programa original, éstas volverán a ejecutarse, y el usuario verá aparecer por la pantalla mensajes inesperados (correspondientes a las operaciones de salida), o como el sistema se queda esperando un valor (correspondiente a las operaciones de entrada). No parece tener sentido que el usuario tenga que repetir los mismos datos de entrada que ya proporcionó durante el cómputo para poder obtener el árbol de cómputo utilizado durante la depuración.

Debido al primer punto la idea descrita para la inclusión de programas con búsqueda encapsulada en el apartado 5.4.3 no es válida aquí. Una posible solución para una incorporación

de estas operaciones al depurador en el futuro sería definir un conjunto de primitivas para entrada/salida específico para la depuración declarativa. Estas nuevas primitivas deberían tener como tipo principal la transformación del tipo principal de las primitivas originales, y deberían ser capaces de devolver los árboles asociados a los cómputos producidos por sus argumentos funcionales, además de conseguir que los "efectos laterales" (salida a pantalla o a fichero) formaran parte de los hechos básicos y de la respuesta computada.

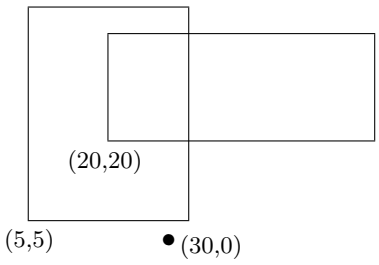
5.5.2. Programas con Restricciones

El sistema $\mathcal{TOY}$ incorpora la posibilidad de realizar cómputos que impliquen restricciones aritméticas sobre números reales, al igual que restricciones de desigualdad. Ya mostramos un ejemplo de cómputos de este tipo al hablar de respuestas perdidas (ejemplo 2.4.3, pág. 43). Aquí vamos a utilizar otro programa, más adecuado para nuestra discusión:

Ejemplo 5.5.2. *Representación de figuras en el plano*

```
type punto = (real,real)
type figura = punto -> bool

rectan:: punto -> real -> real -> figura
rectan  (X,Y) LX LY (X2,Y2) = true
                                <== X2 >= X, X2 <X+LX, Y2<=Y, Y2<Y+LY

intersecc:: figura -> figura -> figura
intersecc F1 F2 P = true <== F1 P, F2 P

fig1 :: figura
fig1 = rectan (20,20) 50 20

fig2::figura
fig2 = rectan (5,5) 30 40
```

El programa define figuras en el plano mediante su función característica. Por simplicidad hemos limitado el ejemplo a la definición de rectángulos sobre el plano de lados paralelos a los ejes de coordenadas. Tal y como indica el tipo de la función rectan un rectángulo de este tipo viene determinado por un punto, que determina su esquina inferior izquierda, y dos números reales que determinan sus dimiensiones. Por tanto la interpretación pretendida de esta función es $(rectan\ (x,y)\ l_x\ l_y(x_2,y_2) \rightarrow true) \in \mathcal{I}$ si y sólo sí el punto (x_2,y_2) está en el rectángulo cuya esquina inferior izquierda tiene coordenadas (x,y) y lados de longitud l_x y l_y. Por ejemplo las funciones fig1 y fig2 representan los rectángulos:



La función intersecc representa la intersección de dos figuras. Su interpretación pretendida es $(intersecc\ f_1\ f_2\ p \rightarrow true) \in \mathcal{I}$ si y sólo si p es un punto de $f_1 \cap f_2$. Por ejemplo, el punto $(30, 0)$ de la figura no corresponde, evidentemente, a la intersección de los dos rectángulos. Sin embargo el objetivo

```
TOY> intersecc fig1 fig2 (30,0)
```

tiene éxito, y constituye una respuesta incorrecta que puede ser depurada con el depurador de $\mathcal{TOY}$ (o de Curry). El motivo del error está en la condición Y2 <= Y de rectan, que debería ser Y2 >= Y. La sesión de depuración puede encontrarse en el ejemplo B.2.7 (pág. 290) del apéndice B.

En el cómputo anterior no se hace uso de las restricciones aritméticas: en ningún momento se utiliza ninguna llamada a una función aritmética con variables no instanciadas. Sin embargo el siguiente objetivo (que se puede utilizar en $\mathcal{TOY}$ activando previamente el resolutor de restricciones aritméticas activando previamente el resolutor de restricciones aritméticas con el comando */cflpr*) sí incluye cómputos de este tipo:

```
TOY> intersecc fig1 fig2 (X,Y)

      yes
      R == true
            { Y=<5.0 }
            { X>=20.0 }
            { X<35.0 }
```

En este caso el usuario pregunta por posibles valores para (X,Y) en la intersección de las dos figuras. $\mathcal{TOY}$ muestra las restricciones que deben cumplir las dos variables. Debido al error en el programa la región representada por estas restricciones no corresponde con la intersección de ambas figuras e incluye por ejemplo al punto (30,0) utilizado en el cómputo anterior. El conjunto de restricciones esperado era:

```
            { X>=20.0 }
```

```
{ X<35.0 }
{ Y>=20.0 }
{ Y<40.0 }
```

Sin embargo, y al igual que nos ocurría en el programa con entrada/salida, no podemos hablar estrictamente de una respuesta computada incorrecta en el sentido de la definición 3.3.3, (pág. 65, ya que la respuesta en este sentido está formada únicamente por la sustitución $\{ \mathsf{R} \mapsto \mathsf{true} \}$.

Si aún así quisiéramos depurar el cómputo asociado con esta pregunta nos encontraríamos con nodos como $(\mathsf{rectan\ (20{,}20)\ 50\ 20\ (X2{,}Y2)} \rightarrow \mathsf{true})$ que, al no incluir las restricciones asociadas a las variables X2, Y2, no representan realmente lo ocurrido durante el cómputo.

Para poder depurar adecuadamente estos cómputos tendríamos que disponer de un marco teórico que considerara las restricciones, de forma que éstas pasaran a formar parte tanto de las respuestas como de los hechos básicos obtenidos en el APA. Tendríamos entonces hechos básicos de la forma $f\ \bar{t}_n \rightarrow t \Leftarrow C$, con C una restricción. En nuestro ejemplo tendríamos por ejemplo un nodo como

$$\mathsf{rectan\ (20{,}20)\ 50\ 20\ (X2{,}Y2)} \rightarrow \mathsf{true} \Leftarrow \mathsf{X2} >= 20 \wedge \mathsf{X2} < 70 \wedge \mathsf{Y2}<=20 \wedge \mathsf{Y2}<40$$

En este caso el usuario si sería capaz de decidir que este nodo no es válido, ya que las restricciones representan puntos (X2,Y2) en una región distinta a la pretendida para el rectángulo de esquina inferior (20,20) y lados 50 y 20.

Este cambio implica la generalización de todos los resultados teóricos de esta tesis: desde el cálculo semántico y la noción de interpretación pretendida hasta la transformación de programas utilizada. Tal y como se sugiere en el ejemplo 2.4.3 (pág. 43) mediante una noción suficientemente general de restricción, incluyendo disyunciones e igualdades, se podrían incorporar a este marco las *respuestas perdidas*, no tratadas en esta tesis. Este marco general para la depuración de restricciones y respuestas perdidas es programación lógico-funcional no existe hasta el momento y constituye unos de los posibles trabajos futuros sugeridos en la sección 7.2 (pág. 166).

Capítulo 6

Navegación de los Árboles de Cómputo

Una vez obtenido el árbol de cómputo, el depurador declarativo debe recorrerlo, investigando la validez de sus nodos hasta localizar un nodo crítico. Esto se hace con ayuda del usuario, al que el depurador va preguntado acerca de la validez de los hechos básicos contenidos en los nodos de árbol. Llamamos *navegación* a este recorrido del árbol, y *estrategia de navegación* al algoritmo seguido por el depurador durante la navegación. El uso de diferentes estrategias de navegación puede tener consecuencias sobre:

- El número de preguntas realizadas al usuario y su complejidad. En efecto, uno de los mayores inconvenientes habitualmente asociados a la depuración declarativa es el gran número de preguntas que el depurador necesita realizar, así como su complejidad para el usuario. En este capítulo compararemos dos estrategias guiadas por el depurador: la estrategia *descendente* y la estrategia *pregunta y divide*, mostrando sus características (ambas fueron introducidas informalmente en el apartado 2.1.7, pág. 23). También veremos las ventajas en este sentido que puede tener el permitir la inspección libre del árbol por parte del usuario utilizando las opciones a este efecto disponibles en el navegador.

- La eficiencia del depurador. El otro inconveniente mayor de la depuración declarativa es el coste requerido, tanto en memoria como en tiempo, para la evaluación del árbol de cómputo. Algunas estrategias (en particular la estrategia *descendente*) permiten la navegación perezosa del árbol, evitando así su evaluación completa y mejorando la eficiencia en cuanto a consumo de recursos del depurador. Por desgracia la evaluación perezosa es incompatible con las técnicas que hasta ahora conocemos para implementar la otra estrategia (la *pregunta y divide*) que resulta preferible en muchos casos para lograr una menor cantidad de preguntas al usuario (punto anterior) así como con algunas de las características más interesantes del depurador. Valoraremos mediante datos empíricos el coste de ambas posibilidades.

El depurador declarativo del sistema Curry (descrito en la sección 5.3, pág. 113) incluye un navegador descendente en modo texto similar al que forma parte de otros depuradores declarativos para lenguajes funcionales como [68, 78]. El depurador de $\mathcal{TOY}$ (descrito en la sección 5.2, pág. 113) también incluye un navegador de este tipo, pero ofrece además la posibilidad de utilizar un navegador gráfico llamado $\mathcal{DDT}$ (iniciales de *D*eclarative *D*ebugging *T*ool) [20]. La principal ventaja de $\mathcal{DDT}$ es la incorporación de diferentes estrategias al navegador, así como el permitir la inspección libre del árbol del cómputo por parte del usuario lo que, como acabamos de comentar, en ocasiones facilita la localización del nodo crítico. El mayor inconveniente de $\mathcal{DDT}$ es que la implementación actual exige la evaluación completa del árbol antes de la navegación, con el coste computacional que veremos.

Las dos secciones siguientes, 6.1 y 6.2, presentan los dos navegadores incluidos en $\mathcal{TOY}$ (el de Curry es análogo al descendente de $\mathcal{TOY}$, por lo que no lo comentaremos), a la vez que sirven de introducción a las dos estrategias consideradas. La comparación empírica de estas dos estrategias, analizando el número de preguntas realizadas al oráculo en cada caso para un conjunto de ejemplos, se puede ver en el último apartado de la sección 6.2. La sección 6.3 compara la eficiencia, en términos de memoria y tiempo, requeridos para la evaluación perezosa del árbol, permitida por el evaluador descendente, y para su evaluación completa (impaciente) necesaria para el uso de $\mathcal{DDT}$. Finalmente, la sección 6.4 propone dos técnicas para la detección automática de la validez o no validez de algunos nodos en el árbol, lo que puede contribuir a disminuir las preguntas realizadas al usuario por el depurador.

La mayor parte de este capítulo está basada en el trabajo [20], excepto la descripción del algoritmo de consecuencia de la sección 6.4 así como la demostración de sus propiedades, que están tomadas de [19].

6.1. Estrategia de Navegación Descendente

En este apartado vamos a presentar brevemente el navegador en modo texto incorporado a $\mathcal{TOY}$ y a Curry. Este navegador utiliza la llamada *estrategia descendente* por lo que su presentación nos servirá a la vez como presentación de esta estrategia. El siguiente apartado introduce el algoritmo seguido por la estrategia, mientras que el siguiente comenta la implementación particular realizada en $\mathcal{TOY}$ y en Curry mediante un ejemplo concreto.

6.1.1. Algoritmo de Navegación Descendente

El navegador textual incluido en $\mathcal{TOY}$ y en Curry es similar a los navegadores incorporados al depurador declarativo *Buddha* [77, 76, 78] para Haskell [75], y al depurador de Freja [68]. Todos ellos funcionan en modo texto y navegan el árbol siguiendo el siguiente esquema:

Navegador utilizando la estrategia descendente

Sea T un árbol y R = raíz(T), con R no válido.

- Si todos los hijos de R en T son válidos
* - Parar, señalando a R como nodo crítico.*
- En otro caso
* - Seleccionar un hijo no válido N ∈ hijos(T,R)*
* - Repetir el proceso recursivamente sobre subárbol(T,N).*

La siguiente proposición muestra que la navegación siguiendo esta estrategia conduce siempre a la localización de un nodo crítico.

Proposición 6.1.1. *Sea T un árbol con raíz incorrecta. Entonces la estrategia de navegación descendente finaliza en un número finito de pasos, señalando a un nodo crítico de T.*

Demostración

En cada llamada recursiva la estrategia utiliza un nuevo árbol *T' = subárbol(T,N)*, con $N \in$ *hijos(T,R)*, por lo que se verifica $0 < |T'| < |T|$, ya que $R \notin T'$ y al menos $N \in T'$. Esto garantiza que el algoritmo termina en un número finito de pasos porque aunque en ningún paso intermedio se encontrara con un nodo con todos los hijos válidos, la disminución del tamaño garantiza que se llegará finalmente a un árbol-hoja, que sí cumple esta condición de parada.

Para comprobar que el nodo señalado es crítico en T podemos proceder por inducción sobre $|T|$. Si $|T| = 1$ el árbol es una hoja y al ser su raíz no válida ésta es un nodo crítico, tal y como señalará la estrategia. Si $|T| > 1$ y su raíz es crítica la estrategia también la señalará correctamente sin efectuar ningún paso recursivo. En otro caso se procederá recursivamente sobre T' = *subárbol(T,N)* con N no válido. Al ser $|T'| < |T|$ y *raíz(T')* un nodo no válido la hipótesis de inducción la estrategia localizara un nodo crítico en T' que, al ser T' subárbol de T, también será nodo crítico en T. ■

Hay que observar que implícitamente la validez de esta proposición depende de la corrección del test de validez, es decir de las respuestas del usuario. Si éste no responde adecuadamente a las preguntas del navegador la estrategia puede proporcionar resultados incorrectos (aunque la condición de terminación sí se mantiene).

6.1.2. Un Ejemplo

El programa $\mathcal{TOY}$ de la figura 6.1 tiene como propósito aproximar el número

$$\frac{1+\sqrt{5}}{2} = 1'618033...$$

```
fib                 = [1, 1 | fibAux 1 1]
fibAux N M          = [N+M | fibAux N (N+M)]

goldenApprox        = (tail fib) ./. fib

infixr 20 ./.
[X | Xs]  ./.  [Y | Ys]  = [ X/Y | Xs ./. Ys]

tail [X|Xs]         = Xs

take 0 L            = []
take N [X|Xs]       = [X| take (N-1) Xs] <== N>0
```

Figura 6.1: Aproximaciones de la proporción aúrea

conocido como la *proporción aúrea* o la *divina proporción*. Es bien conocido que este número se puede aproximar utilizando la sucesión de Fibonacci 1, 1, 2, 3, 5, . . . , en la que cada término de la secuencia después del segundo, es la suma de los dos que le preceden. Si llamamos $fib(i)$ al iésimo término de la sucesión se cumple la siguiente propiedad, en la que se basa el programa:

$$\lim_{n\to\infty} \frac{fib(n+1)}{fib(n)} = \frac{1+\sqrt{5}}{2}$$

La función fib del programa representa una lista infinita conteniendo los términos de la sucesión de Fibonacci. Esta función utiliza una función auxiliar $fibAux$. Dados dos valores enteros X_0 y X_1, la llamada $fibAux\ X_0\ X_1$ debe computar la lista infinita $[X_2, X_3, \ldots]$, en la que $X_k = X_{k-2} + X_{k-1}$, para cualquier $k \geq 2$. Obsérvese que esta lista estará formada por todos los términos a partir del tercero en adelante de la sucesión de Fibonacci para la llamada fibAux 1 1 incluida en el lado derecho de fib. La función goldenApprox computará la lista infinita $[fib(2)/fib(1),\ fib(3)/fib(2), \ldots]$ utilizando para ello el operador infijo ./., que devuelve el resultado obtenido al dividir dos listas infinitas término a término. El significado del resto de las funciones es fácil de deducir de su definición.

Por ejemplo, algunos hechos básicos del modelo pretendido $\mathcal{I}$ del programa son:

$$\begin{aligned}\mathcal{I} = \{\ & \ldots, \mathsf{fib} \to \bot, \ldots, \mathsf{fib} \to [1 \mid \bot],\ \ldots,\ \mathsf{fib} \to [1,1,2,3,5,8 \mid \bot],\ \ldots, \\ & \ldots,\ \mathsf{fibAux\ 1\ 1} \to [2,3,5, \mid \bot],\ \ldots,\ \mathsf{fibAux\ 10\ 20} \to [30,50,80,120 \mid \bot],\ \ldots, \\ & \ldots,\ \mathsf{goldenApprox} \to [1, 2, 1.5, 1.66, 1.6 \mid \bot],\ \ldots\}\end{aligned}$$

entre otros. En particular, el hecho básico goldenApprox → [1, 2, 1.5, 1.66, 1.6 | _] está en la interpretación pretendida ya que

$$[\frac{fib(2)}{fib(1)}, \frac{fib(3)}{fib(2)}, \frac{fib(4)}{fib(3)}, \frac{fib(5)}{fib(4)}, \frac{fib(6)}{fib(5)}] = [1/1, 2/1, 3/2, 5/3, 8/5] = [1, 2, 1'5, 1'66, 1'6]$$

(redondeando a dos decimales por simplicidad).

Sin embargo, si tratamos de resolver el objetivo take 5 goldenApprox == R el sistema produce la respuesta incorrecta

$$\sigma = \{\mathsf{R} \mapsto [1, 2, 1.5, 1.33, 1.25\ ]\}$$

mostrando así que debe haber alguna regla incorrecta en el programa.

Es interesante observar que en este caso la detección del síntoma inicial no es tan sencilla como en otros ejemplos. Es posible que el usuario sólo llegue a notar el error al considerar un mayor número de términos de la secuencia, notando que ésta no tiende hacia el número deseado 1′618033... sino hacia 1′0.

La figura 6.2 muestra un APA asociado a este cómputo. Los nodos cuyos hechos básicos no son válidos en el modelo pretendido aparecen en negrita. Hay dos nodos críticos, recuadrados en la figura, y ambos corresponden a la función fibAux, que es, por tanto, incorrecta. En efecto, en la definición

$$\mathsf{fibAux\ N\ M} = [\mathsf{N{+}M} \mid \mathsf{fibAux}\ \underline{\mathsf{N}}\ (\mathsf{N{+}M})]$$

la variable N subrayada debería ser una M, es decir:

$$\mathsf{fibAux\ N\ M} = [\mathsf{N{+}M} \mid \mathsf{fibAux\ M\ (N{+}M)}]$$

Tras iniciar el depurador, el sistema $\mathcal{TOY}$ permite al usuario elegir entre el depurador textual y el depurador gráfico. Si el usuario elige el depurador en modo texto, se obtendrá una sesión de depuración similar a la siguiente:

```
Consider the following facts:
1: goldenAprox → [1.0, 2.0, 1.5, 1.33, 1.25 | _ ]
2: take 5.0 [1.0, 2.0, 1.5, 1.33, 1.25 | _ ] → [1.0, 2.0, 1.5, 1.33, 1.25]
Are all of them valid? ([y]es / [n]ot) / [a]bort) n
Enter the number of a non-valid fact followed by a fullstop: 1.

Consider the following facts:
1: fib → [1.0, 1.0, 2.0, 3.0, 4.0, 5.0 | _ ]
2: tail [1.0, 1.0, 2.0, 3.0, 4.0, 5.0 | _ ] → [1.0, 2.0, 3.0, 4.0, 5.0 | _ ]
3: fib → [1.0, 1.0, 2.0, 3.0, 4.0 | _ ]
4: [1.0, 2.0, 3.0, 4.0, 5.0 | _ ] ./.
                              [1.0, 1.0, 2.0, 3.0, 4.0 | _ ] → [1.0, 2.0, 1.5, 1.33, 1.25 | _ ]
Are all of them valid? ([y]es / [n]ot) / [a]bort) n
Enter the number of a non-valid fact followed by a fullstop: 1.
```

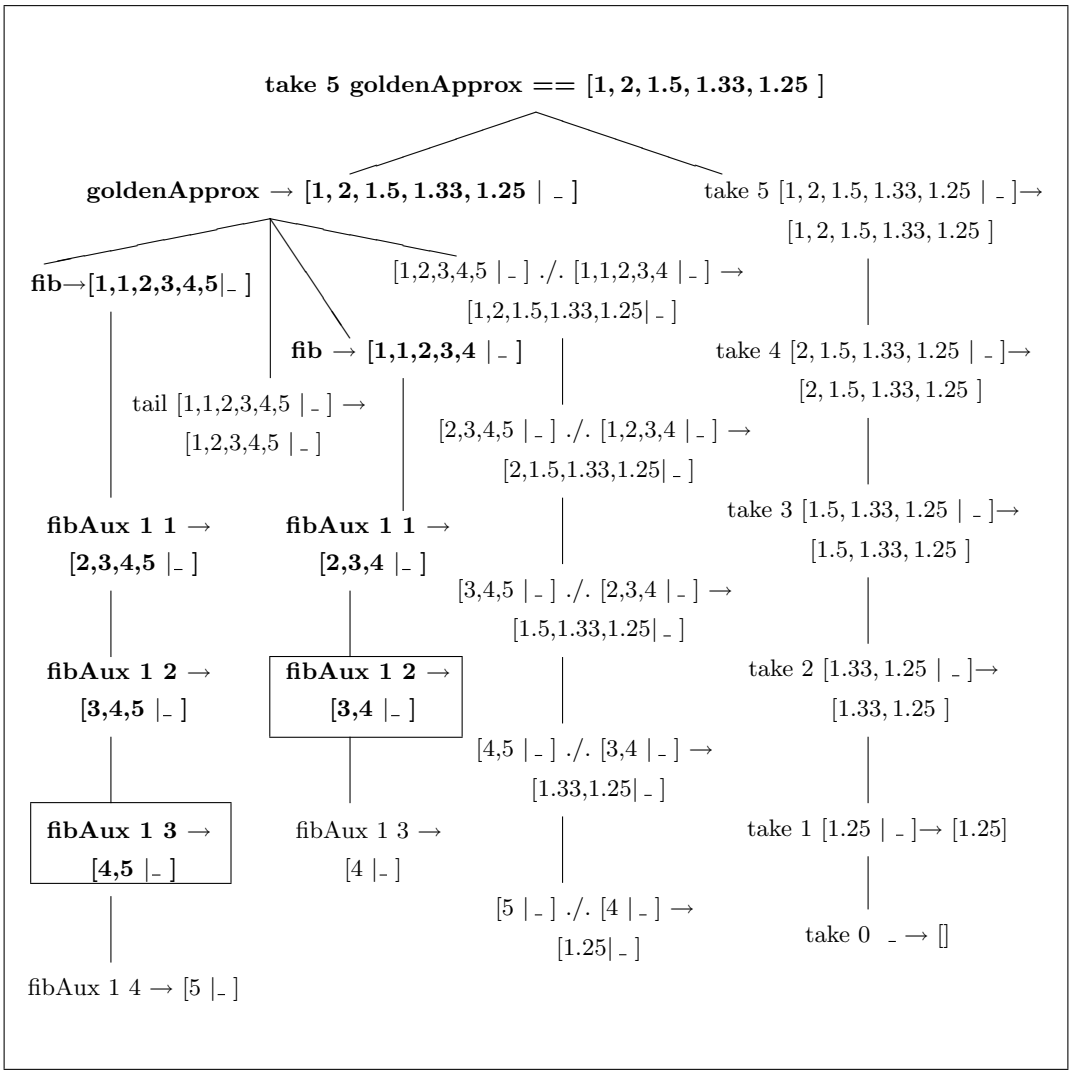


Figura 6.2: APA para el objetivo take 5 goldenApprox == [1, 2, 1.5, 1.33, 1.25]

```
Consider the following facts:
1: fibAux 1.0 1.0 → [2.0, 3.0, 4.0, 5.0 | _ ]
Are all of them valid? ([y]es / [n]ot) / [a]bort) n

Consider the following facts:
1: fibAux 1.0 2.0 → [3.0, 4.0, 5.0 | _ ]
Are all of them valid? ([y]es / [n]ot) / [a]bort) n

Consider the following facts:
1: fibAux 1.0 3.0 → [4.0, 5.0 | _ ]
Are all of them valid? ([y]es / [n]ot) / [a]bort) n

Consider the following facts:
1: fibAux 1.0 4.0 → [5.0 | _ ]
Are all of them valid? ([y]es / [n]ot) / [a]bort) y

Rule number 1 of the function fibAux is wrong.
Buggy node: fibAux 1.0 3.0 → [4.0, 5.0 | _ ]
```

Como se puede ver en el teorema 4.4.8 (pág. 100) el árbol utilizado para la depuración difiere del APA sólo en la raíz, que corresponde a la función solve introducida internamente para incorporar el objetivo al programa transformado. En cualquier caso esta raíz se asume como errónea y no se muestra al usuario. En su lugar, y tal como indica el método de navegación descendente, el navegador comienza mostrando los dos hijos de la raíz. De los dos nodos el primero es erróneo mientras y el segundo correcto. El usuario señala consecuentemente al primer nodo como incorrecto para proseguir la depuración.

En otros depuradores declarativos como Buddha [77, 76, 78] el navegador va preguntando al usuario acerca de la validez de cada hijo, uno tras otro, hasta encontrar uno incorrecto o comprobar que todos son válidos. Sin embargo en los depuradores de $\mathcal{TOY}$ y Curry hemos considerado que es preferible mostrar todos los nodos a la vez, permitiendo al usuario elegir el nodo incorrecto, si lo hubiera, sin que sea necesario introducir información alguna sobre la validez de los nodos restantes. Esta idea, sugerida por Wolfgang Lux, tiene como principal ventaja que si el usuario detecta rápidamente un nodo no válido en la lista no tiene que detenerse a considerar la validez del resto.

La siguiente pregunta realizada por el depurador consta de cuatro hechos básicos (los cuatro nodos hijos del nodo goldenAprox → [1.0, 2.0, 1.5, 1.33, 1.25 | _] del APA de la figura 6.2), de los cuales tanto el primero como el tercero son no válidos. El usuario sólo puede elegir uno de ellos, ya que como sabemos el depurador sólo busca un nodo crítico. En este ejemplo de sesión de depuración el usuario ha señalado el primer nodo como no válido, por lo que la navegación prosigue por este camino hasta localizar el nodo crítico fibAux 1.0 3.0 → [4.0, 5.0 | _] y señalar a la única regla para la función fibAux como incorrecta.

En el apéndice B se pueden encontrar otros ejemplos de sesiones de depuración utilizando el navegador textual.

6.2. $\mathcal{DDT}$: un Navegador Visual

Como ya hemos indicado, los dos depuradores existentes para lenguajes funcionales presentados en el capítulo 2 (sección 2.3.3, pág. 34) incluyen un navegador descendente similar al descrito anteriormente, al igual que lo hacen $\mathcal{TOY}$ y Curry. Sin embargo ya en [88] se sugiere la utilización de un entorno gráfico que permita examinar el árbol en su conjunto y que permita la utilización de diferentes estrategias al usuario. En esta sección presentamos un interfaz gráfico de este tipo que hemos incorporado al sistema $\mathcal{TOY}$. El siguiente apartado introduce el navegador gráfico, mientras que el siguiente muestra una de sus principales ventajas sobre el navegador textual: la posibilidad que tiene el usuario de moverse libremente por el árbol de depuración. El navegador gráfico también permite una navegación guiada, contando para ello con dos posibles estrategias: la estrategia descendente que ya hemos visto al hablar del depurador textual y la estrategia *pregunta y divide* que veremos en el apartado 6.2.3. La sección finaliza con el apartado 6.2.4, en el que se compara experimentalmente el número de preguntas realizadas al usuario por las dos estrategias. El contenido de esta sección es una reelaboración del trabajo [20].

6.2.1. Introducción a $\mathcal{DDT}$

$\mathcal{DDT}$ es parte del depurador declarativo incluido en $\mathcal{TOY}$ y por tanto de la distribución de este sistema. A diferencia del resto del código de $\mathcal{TOY}$, no está escrito en SICStus Prolog [83] sino en Java [47]. Hemos decidido utilizar este lenguaje por dos razones:

1. Java incluye numerosas facilidades para el diseño de interfaces gráficos. En particular dispone de clases que permiten representar estructuras en forma de árbol (como la clase javax.swing.JTree) lo que ha facilitado el desarrollo del navegador.

2. El sistema SICStus Prolog [83], en el que está escrito el resto del sistema $\mathcal{TOY}$, dispone de un protocolo de comunicación con Java, lo que facilita la interacción de ambas partes.

Cuando el usuario detecta un síntoma inicial e indica al sistema que desea comenzar la depuración, $\mathcal{TOY}$ pregunta si prefiere utilizar el navegador textual o $\mathcal{DDT}$. En caso de seleccionar $\mathcal{DDT}$ el depurador genera por completo el árbol de cómputo y lanza el proceso Java, que muestra el árbol de cómputo y un menú con diferentes opciones. Cada nodo del árbol se puede expandir o contraer según el usuario desee. En el caso de que el árbol considerado no sea muy grande puede verse completo, tal y como se muestra en la figura 6.3. La comparación entre este árbol y el APA de la figura 6.2 nos muestra que ambos árboles tienen los mismos nodos aunque utilizando una presentación diferente ya que en $\mathcal{DDT}$ los hijos en lugar de izquierda a derecha se muestran de arriba hacia abajo.

El usuario puede cambiar libremente el estado de los nodos, simplemente situando el ratón sobre el nodo deseado y pulsando el botón derecho, momento en el que aparece una lista con los posibles estados del nodo, o seleccionando el nodo con el ratón y marcando el estado deseado en la opción *Node* del menú. Los posibles estados son:

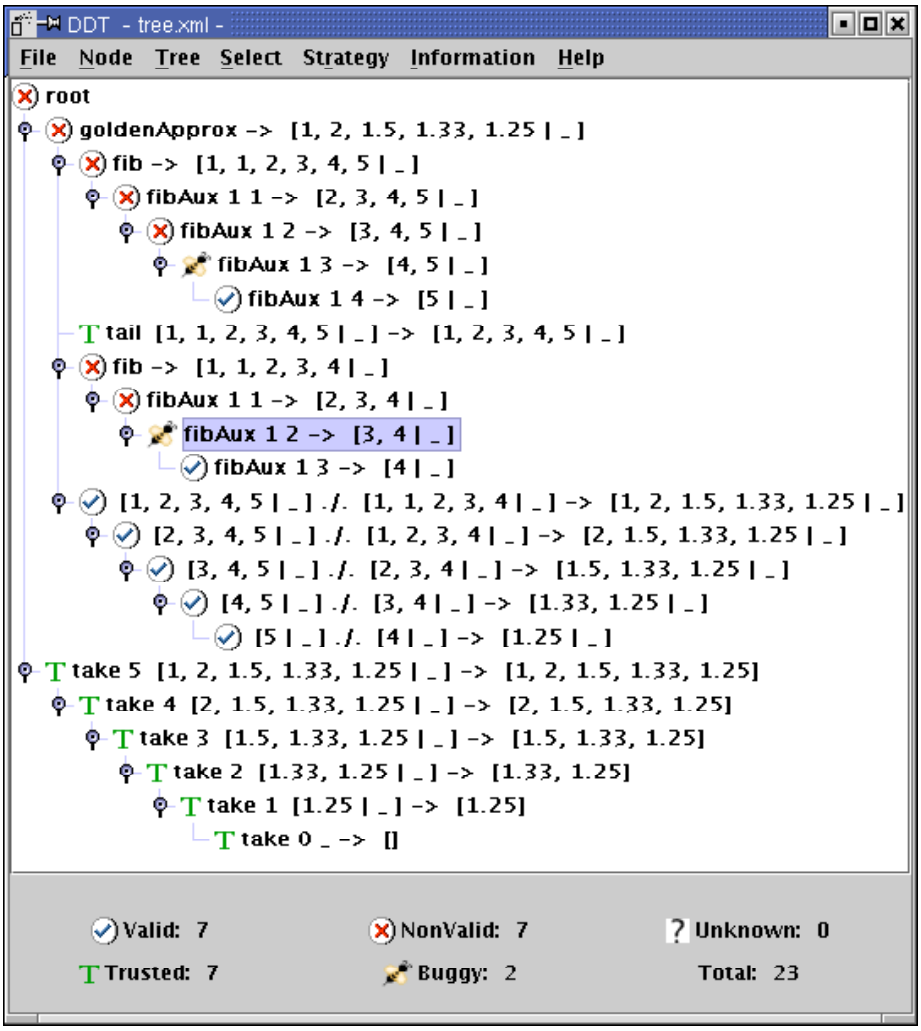


Figura 6.3: Árbol de depuración correspondiente al APA de la figura 6.2

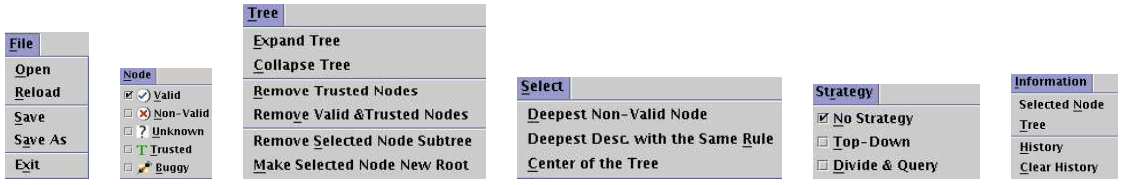


Figura 6.4: Algunas de las opciones de menú en $\mathcal{DDT}$

- *Valid*: El nodo es válido. En este caso $\mathcal{DDT}$ coloca un signo similar a una 'V' precediendo al nodo, tal y como se puede ver en la figura 6.3 para todos los nodos asociados al operador infijo ./..

- *Non-Valid*: El nodo no es válido. En este caso se marca mediante una cruz como se puede ver en los dos nodos de fib.

- *Unknown*: No se sabe si el nodo es correcto o no. Es el estado en el que se encuentran todos los nodos del árbol al comienzo de la navegación. Los nodos en este estado aparecen marcados con un símbolo '?'. En la figura 6.3 no hay ningún nodo de este tipo, ya que la imagen ha sido tomada tras marcar todos los nodos con un estado diferente a *Unknown*.

- *Trusted*: La función asociada al hecho básico es una función fiable de usuario. Estas funciones, que ya introdujimos en el apartado 5.3.1 (pág. 113) para el caso de Curry, son funciones tales que sus hechos básicos asociados son siempre válidos. Al marcar un nodo como *Trusted* el navegador marca automáticamente con el mismo estado todos los hechos básicos en los que se utiliza el mismo símbolo de función. Y a la inversa, cuando el usuario cambia el estado de un nodo de *Trusted* a otro valor, el navegador cambia el estado de todos los nodos en los que se utiliza el mismo símbolo de función de *Trusted* a *Unknown*. En la figura aparecen marcados como *Trusted* los nodos asociados a las funciones take y tail.

- *Buggy*: El nodo es crítico. El símbolo utilizado en esta ocasión es el dibujo de una abeja. Sin embargo el usuario no puede cambiar el estado de ningún nodo a *Buggy*, sino que es el propio navegador el que lo hace cuando detecta que un nodo es no válido con todos sus hijos válidos o *Trusted*. En ese momento el navegador también muestra un mensaje avisando al usuario de que un nodo crítico ha sido detectado, así como el número de orden de la regla utilizada, con respecto al orden textual de las reglas de la función a la que pertenece. En la figura están marcados los dos nodos críticos del APA.

En la parte inferior de la ventana que contiene el árbol de depuración se pueden consultar en cada momento los datos del número total de nodos en el árbol y la cantidad de nodos en cada estado.

Vamos a repasar a continuación el objetivo de cada una de las principales opciones de menú del navegador que pueden verse en la figura 6.4:

- Menú *File*. En este menú se encuentran las opciones que permiten grabar en fichero y recuperar árboles de cómputo. Al grabar se crea un fichero en formato *XML* con el contenido actual del árbol y con el nombre indicado por el usuario. La utilidad de estas opciones es poder interrumpir las sesiones de depuración, grabando su estado actual, para continuarlas más tarde. El nombre del fichero cargado actualmente aparece en la parte superior de la ventana (*tree.xml* en el caso de la figura 6.3). La opción

reload permite cargar de nuevo el fichero actual, probablemente para eliminar las modificaciones realizadas desde que fue cargado la última vez. Esta opción, combinada con las opciones de grabar (*Save* y *Save As*) permiten volver a un estado anterior de la depuración en el caso de que el usuario dude de la corrección de alguna decisión adoptada acerca del estado de los nodos.

- Menú *Node*. Tiene el doble objetivo de informar sobre el estado del nodo seleccionado en el árbol y de permitir la modificación de dicho estado.

- Menú *Tree*. Las opciones de expandir y colapsar el árbol tienen un claro significado, y no modifican el contenido del árbol. La opción *Remove Trusted Nodes* elimina del árbol todos los nodos *Trusted*. Al ser estos nodos válidos se trata de una operación segura en el sentido indicado por la proposición 5.3.1 (pág. 5.3.1), es decir que si en el árbol resultante hay algún nodo crítico, éste también lo era en el árbol inicial, mientras que si no hay ningún nodo crítico el árbol inicial tampoco tenía ninguno. La opción *Remove Valid & Trusted Nodes* es similar. Las dos últimas opciones modifican el árbol, bien eliminando un subárbol o haciendo que un subárbol sea el nuevo árbol considerado. Estas operaciones son seguras en las siguientes condiciones:

  - Eliminación de un subárbol. Si la raíz del árbol no es válida y la raíz del subárbol es válida la operación es segura en el sentido indicado anteriormente. Esto será demostrado en la sección 6.2.3.
  - Tomar un subárbol como nuevo árbol de cómputo. La operación es segura si el subárbol considerado contiene algún nodo no-válido, ya que en este caso sigue habiendo un nodo crítico en el nuevo considerado.

  El depurador permite realizar estas operaciones aún cuando no sean seguras, pero en este caso en primer lugar avisa al usuario y pide confirmación antes de llevarlas a cabo.

- Menú *Select*. Selecciona un nodo del árbol según el criterio especificado. El concepto de *centro* del árbol será presentado en la sección 6.2.3.

- Menú *Strategy*. Indica la estrategia seguida hasta el momento. Por defecto la opción marcada es *No Strategy* que permite la navegación libre. Al marcar cualquiera de las otras casillas comienza la navegación guiada, siguiendo la estrategia seleccionada. La estrategia *Top-Down* se refiere a la navegación descendente y la *Divide & Query* a la descrita en la sección 6.2.3.

- Menú *Information*. Aquí se puede obtener información sobre el nodo seleccionado (estado, posición de la regla utilizada) o del árbol completo (número de nodos, máximo número de hijos de un nodo, profundidad y un listado con todos los nodos). La opción *History* registra todos los cambios de estado durante la sesión actual, tanto los realizados por el usuario como los llevados a cabo automáticamente por el navegador.

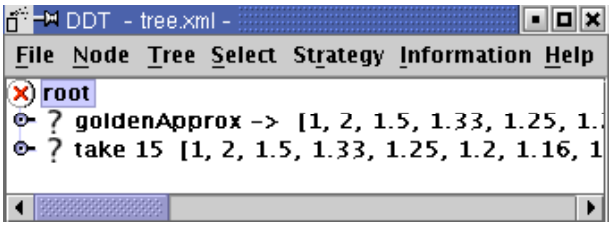
6.2.2. Navegación Libre

Consideremos de nuevo el programa de la figura 6.1, pero suponiendo que el objetivo es ahora take 15 goldenApprox == R. La respuesta computada es de nuevo errónea. Empleando el depurador textual obtenemos una sesión similar a la anterior:

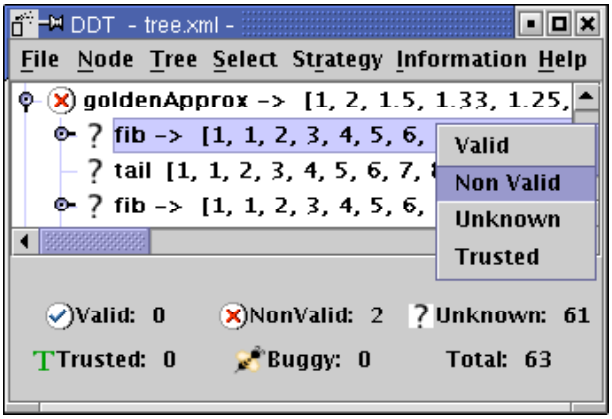
```
Consider the following facts:
1: goldenApprox → [1, 2, 1.5, 1.33, 1.25, 1.2, ... | _ ]
2: take 15 [1, 2, 1.5, 1.33, 1.25, 1.2, ... | _ ] → [1, 2, 1.5, 1.33, 1.25, 1.2, ...]
Are all of them valid? ([y]es / [n]ot) / [a]bort) n
Enter the number of a non-valid fact followed by a fullstop: 1.

Consider the following facts:
1: fib → [1, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15 | _ ]
2: tail [1, 1, 2, 3, 4, 5, ... | _ ] → [1, 2, 3, 4, 5, ... | _ ]
3: fib → [1, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14 | _ ]
4: [1, 2, 3, 4, ... | _ ] ./. [1, 1, 2, 3, ... | _ ] →
                                    [1, 2, 1.5, 1.33, 1.25, ... | _ ]
Are all of them valid? ([y]es / [n]ot) / [a]bort) n
Enter the number of a non-valid fact followed by a fullstop: 1.

Consider the following facts:
1: fibAux 1 1 → [2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15 | _ ]
Are all of them valid? ([y]es / [n]ot) / [a]bort) n

Consider the following facts:
1: fibAux 1 2 → [3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15 | _ ]

..... (12 preguntas más sobre fibAux )

Rule number 1 of the function fibAux is wrong.
```

Hemos sustituido por puntos algunos elementos de las listas por limitaciones de espacio, y suprimido 12 preguntas adicionales. Aunque estas preguntas no son difíciles de contestar la sesión de depuración se alarga y se hace más tediosa. Para objetivos que traten de obtener un número aún mayor de aproximaciones el navegador textual basado en la estrategia descendente deja de ser utilizable en la práctica. Por ejemplo para take 100 goldenApprox == R el usuario tendría que contestar 99 preguntas sobre fibAux antes de alcanzar el nodo crítico. Podría argumentarse que el usuario debería tratar de utilizar el depurador para valores de los parámetros suficientemente pequeños para acortar y simplificar las sesiones de depuración, pero esto no es siempre posible.

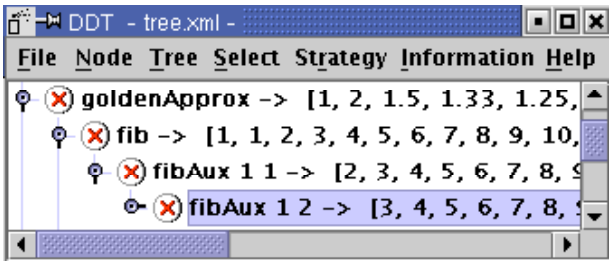
Veamos como el depurador gráfico puede ser útil en estas ocasiones, analizando paso a paso una posible sesión de depuración para el mismo objetivo con $\mathcal{DDT}$. En un principio $\mathcal{DDT}$ muestra sólo el primer nivel del árbol expandido:



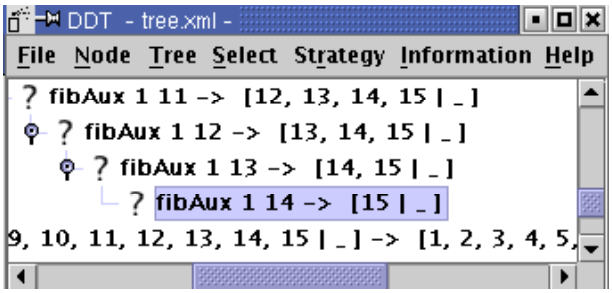
El usuario observa que el nodo asociado a goldenApprox no es válido, cambia su estado a no válido y expande el siguiente nivel para examinar sus hijos. La siguiente imagen muestra dicho siguiente nivel en el momento en el que el usuario ha pulsado el botón derecho del ratón sobre el nodo correspondiente a la función fib para cambiar su estado a no válido:



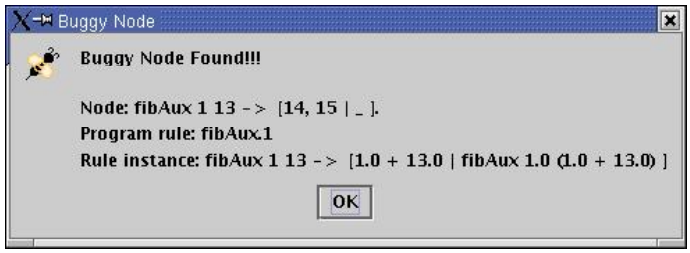
Repitiendo el proceso el usuario puede proseguir, realizando los mismos pasos hasta el momento que el navegador descendente. La siguiente figura muestra la situación tras haber descendido 4 niveles desde la raíz:



Llegado a este el usuario, al ver que parece haber varios nodos consecutivos erróneos asociados a la función fibAux, puede utilizar la opción *Deepest Descendant with the Same Rule* del menú *Select*. El navegador busca entonces por el descendiente más profundo del nodo seleccionado en ese momento en el que se haya utilizado la misma de programa. En nuestro caso localiza y selecciona el nodo con el básico fibAux 1 14 → [15 | _]:



El usuario puede comprobar fácilmente que este nodo es válido y cambiar su estado consecuentemente. Moviéndose ahora hacia arriba en el árbol, se comprueba que el padre del nodo seleccionado contiene el hecho básico fibAux 1 13 → [14,15 | _] que no es válido, ya que el segundo elemento de la lista debería ser $14 + 13 = 27$ en lugar de 15 y por tanto lo marca como no válido. En este momento $\mathcal{DDT}$ descubre que el nodo es crítico, y muestra el siguiente mensaje:



Nótese que la sesión habría sido análoga en el caso de la obtención de 100 aproximaciones de la razón aúrea.

Debido a la habitual estructura recursiva en la definición de las funciones, la opción *Deepest Descendant with the Same Rule* proporciona a menudo un nodo N cuyo hecho básico asociado es más sencillo, y por tanto más fácil de analizar, que el nodo inicial. La navegación puede continuarse entonces descendiendo si N no es válido, o ascendiendo hacia el antecesor de N en otro caso, como sucedía en este ejemplo. En ambos casos resulta sencillo probar que esta forma de proceder servirá para localizar finalmente un nodo crítico.

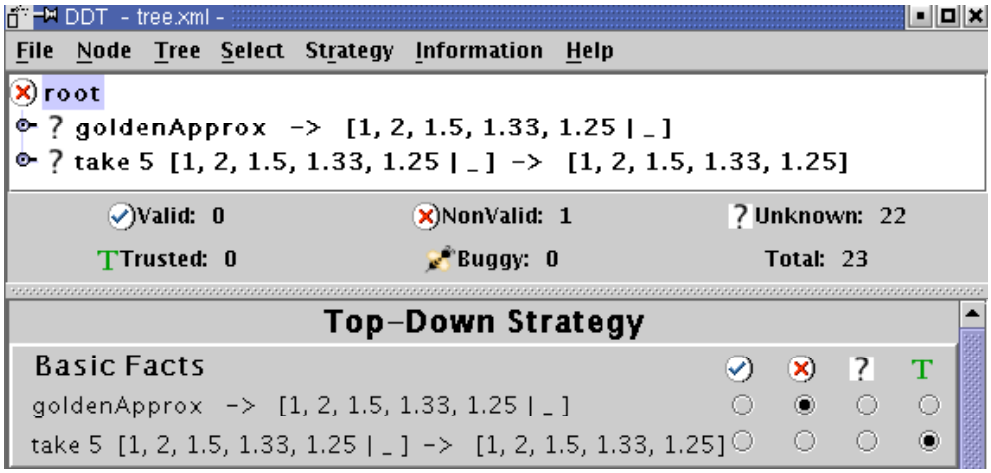
Por supuesto el depurador textual podría mejorarse incluyendo opciones más sofisticadas que permitieran una navegación más libre del árbol, pero pensamos que en cualquier caso el navegador gráfico proporciona una perspectiva más general que facilita, en muchos casos, una detección más rápida de los nodos críticos.

6.2.3. Estrategia *Pregunta y Divide*

Además del movimiento libre a través del árbol, $\mathcal{DDT}$ permite al usuario el uso de una estrategia guiada, ofreciendo la posibilidad de elegir entre la estrategia descendente ya comentada y la *pregunta y divide*.

No nos vamos a detener a comentar el funcionamiento de $\mathcal{DDT}$ cuando se utiliza la estrategia descendente por ser muy similar al del navegador textual ya descrito. La única

diferencia es que el usuario puede dar información sobre más de un nodo en cada paso, mientras que en el navegador en modo texto sólo podía seleccionar un nodo no válido. Por ejemplo en la siguiente imagen el usuario está considerando la primera pregunta realizada por el navegador guiado por la estrategia descendente:



Además de marcar el primer nodo como no válido, el usuario también indica que la función take es fiable, asegurándose así de que todos los nodos asociados a esta función quedan marcados automáticamente de la misma forma. Para que el navegador pueda proseguir tras cada paso es necesario, o bien marcar todos los nodos como válidos o al menos un nodo como no válido. Si se marca más de un nodo como no válido el navegador continuará por el primer nodo no válido de la lista.

La estrategia *pregunta y divide* fue ya presentada en [82] y ha sido también incluida en el sistema *TkCalypso* [88]. Al igual que en la estrategia descendente la depuración comienza con un árbol cuya raíz es no válida, propiedad que se conserva tras cada paso. La idea es seleccionar en cada paso un nodo N tal que coincida el número de nodos dentro y fuera del subárbol cuya raíz es N. Aunque este nodo, al que se llama el *centro* del árbol, no existe siempre, la estrategia busca el nodo que mejor aproxime esta condición. Para simplificar la explicación consideramos una función

$$dif(T, N) = (|T| - |subárbol(T, N)|) - |subárbol(T, N)|$$

que calcula la diferencia entre el número de nodos fuera y dentro del subárbol cuya raíz es N. El nodo buscado es entonces aquel $N \in T$ tal que:

$$|dif(T, N)| = min\{|dif(T, M)| : M \in T\}$$

Si el nodo N que cumple esta propiedad es no válido su subárbol será considerado en el siguiente paso, mientras que si es válido dicho subárbol será eliminado. Todas estas ideas quedan reflejadas en la siguiente descripción en pseudocódigo de la estrategia:

Navegador utilizando la estrategia pregunta y divide

Sea T un árbol y R = raíz(T), con R no válido.

- *Si $|T| = 1$*
  - *Parar, señalando a R como nodo crítico*
- *En otro caso*
  - *Sea $N \in T$ tal que $|dif(T,N)| = min\{|dif(T,M)| : M \in T\}$*
  - *Si N es válido*
    - *Sea $T' = T-$* subárbol(T,N)
    - *Repetir el proceso sobre T'*
  - *En otro caso*
    - *Repetir el proceso sobre subárbol(T,N)*

La operación $T - subárbol(T,N)$ denota la eliminación de T de todos los nodos en *subárbol(T,N)* utilizando la operación de eliminación de la definición 5.3.1 (pág. 114), que equivale a la eliminación del subárbol completo.

La siguiente proposición garantiza la corrección y completitud de la navegación basada en esta estrategia:

Proposición 6.2.1. *Sea T un árbol con raíz incorrecta. Entonces el método de navegación pregunta y divide finaliza en un número finito de pasos, señalando a un nodo crítico de T.*

Demostración

Para garantizar la terminación basta con asegurar que el nodo N seleccionado en el caso $|T| > 1$ es siempre distinto de $R =$ raíz(T), ya que en este caso el subárbol considerado al repetir el proceso tendrá un número menor de nodos que el árbol original al tiempo que no será el árbol vacío, lo que garantiza que en algún momento se alcanzará la condición de salida $|T| = 1$. Para comprobar que $N \neq R$ vamos a comprobar que para cualquier $M \in T, M \neq R$, se tiene $|dif(T,M)| < |dif(T,R)|$, de donde

$$|dif(T,R)| > min\{|dif(T,M)| : M \in T\}$$

y R no podrá ser el nodo N elegido. Sea entonces $M \in T$, $M \neq R$ (al menos un nodo de esta forma existe al estar considerando $|T| > 1$). Se cumple $|dif(T,M)| < |dif(T,R)|$ porque

- $|dif(T,R)| = |T|$, ya que $subárbol(T,R) = T$.
- $|dif(T,M)| < |T|$ ya que al ser $M \neq R$ se cumplen

$$\begin{aligned} 0 < \quad & (|T| - |subárbol(T,M)|) && < |T| \\ 0 < \quad & |subárbol(T,M)| && < |T| \end{aligned}$$

de donde

$$-|T| \ < \ (|T| - |\textit{subárbol}(T, M)|) \ - \ |\textit{subárbol}(T, M)| \ < \ |T|$$

es decir $|dif(T, M)| < |T|$.

Ahora debemos comprobar que el nodo señalado es efectivamente un nodo crítico. Para esto vamos a proceder por inducción sobre $|T|$. Si $|T| = 1$ al tener raíz no válida ésta es crítica y así lo indica la estrategia. Si $|T| > 1$ tenemos que distinguir dos casos: si el nodo seleccionado N es no válido el algoritmo considera recursivamente *subárbol(T,N)* y por hipótesis de inducción se encontrará un nodo crítico en este subárbol, que también será entonces crítico en T. Si por el contrario N es válido se elimina su subárbol y se considera el árbol resultante T' = T - *subárbol(T,N)*. T' continua teniendo la raíz no válida porque hemos probado que $N \neq R$ y un número menor de nodos, por lo que la estrategia encontrará un nodo crítico $B \in T'$. Ahora bien, debemos asegurar que este nodo B es también crítico en T. Pero para ello basta con observar que la única diferencia entre los hijos de B en T' y los hijos de B en T es que quizás en éste último caso haya un nodo más, la raíz N del subárbol eliminado, que al ser válida no altera la condición de nodo crítico de B. ■

6.2.4. Comparación de Estrategias

	Datos del Árbol			E. Descendente		E.Preg.y Div.
Ejemplo	Nodos	Hijos	Prof.	Preg.	Nodos	Nodos
B.1.1	201	2	68	68 (3)	134 (4)	9
B.1.2	194	4	14	12	45	8
B.1.3	198	3	29	27	79	8
B.1.4	206	4	18	17	49	8
B.2.1	191	2	19	19 (18)	36 (19)	8
B.2.3	198	2	15	15 (5)	29 (10)	8
B.2.4	195	2	44	6	10	7
B.2.5	191	4	14	13 (13)	24 (15)	8
B.2.8	203	4	51	50	54	9
B.3.1	200	2	42	42	83	8
B.3.2	196	2	16	12	23	8
B.3.3	197	7	31	9 (4)	39 (12)	7

Cuadro 6.1: Comparación de estrategias

El cuadro 6.1 establece una comparación experimental entre las dos estrategias incluidas en $\mathcal{DDT}$. En el cuadro hemos considerado la mayor parte de los ejemplos del apéndice B, seleccionando objetivos cuyos árboles de cómputo tengan una cantidad elevada de nodos.

En particular hemos buscado objetivos que produjeran árboles con un número de nodos similar (cercano a 200) de forma que sea posible también la comparación entre los distintos ejemplos.

La primera columna del cuadro 6.1 incluye el nombre del ejemplo en el apéndice. Las tres columnas siguientes indican respectivamente el número de nodos del árbol utilizado en la depuración, el máximo número de hijos que se alcanza dentro del árbol y su profundidad.

Las dos columnas siguientes se dedican a la estrategia descendente, indicando respectivamente el número de preguntas y de nodos considerados en la depuración del árbol (recordemos que en esta estrategia cada pregunta muestra una lista con uno o más nodos). En caso de que en alguna de las preguntas varios nodos puedan ser seleccionados como no válidos hemos seleccionado siempre el primero de la lista. Sin embargo a veces otras elecciones conducen más rápidamente al nodo crítico. En este caso se indica entre paréntesis el número menor de preguntas y nodos que conducen a un nodo crítico. Por ejemplo en el primer programa se tiene que seleccionando siempre el primer nodo no válido de la lista la estrategia descendente requiere 68 preguntas y un total de 134 hechos básicos considerados, mientras que con una selección más "hábil" la navegación se puede reducir a tan solo 3 preguntas con un total de 4 nodos.

La última columna incluye el número de preguntas requeridas por la estrategia pregunta y divide, que coincide en esta estrategia con el número de hechos básicos que el usuario debe considerar.

Los objetivos utilizados en cada programa son los siguientes:

Ejemplo	Objetivo
B.1.1 (pág. 271)	rev([67..1],L)
B.1.2 (pág. 272)	qs([12,..,1],L)
B.1.3 (pág. 274)	ordenarbur([8..1],L)
B.1.4 (pág. 275)	qsort([17,..,1],L)
B.2.1 (pág. 279)	sort [1..18] == R
B.2.3 (pág. 282)	take 13 primes == R
B.2.4 (pág. 284)	squareInt 6 == R
B.2.5 (pág. 287)	sort [11..1] == R
B.2.8 (pág. 291)	take 50 goldenApprox == R
B.3.1 (pág. 293)	permsort [40..1] == R
B.3.2 (pág. 295)	solution 11 == R
B.3.3 (pág. 297)	eval ((val 4):+:(val 4):+: (val 4):+: (val 4):+:(val 4):+:(val 4)) R == S

La estrategia pregunta y divide reduce, en promedio, el número de nodos del árbol considerado en cada paso a la mitad. Por tanto, y también en promedio, para un árbol con n nodos se requerirán $\log_2 n$ pasos hasta llegar a un sólo nodo y por tanto a la finalización de la sesión de depuración. Como cada paso equivale a una pregunta al usuario esta cantidad, $\log_2\ n$, será también el número medio de preguntas requeridas para el usuario. El cuadro 6.1 confirma este valor teórico en la práctica. En efecto, se tiene que $\log_2\ 200 = 7'64$ y la

media de nodos (y por tanto preguntas) en el cuadro para la estrategia pregunta y divide es 8, siendo este valor mejor en todos los ejemplos salvo en el primero (y esto sólo para el camino más corto hasta el nodo crítico) que el número de nodos, e incluso en casi todos los casos que el número de preguntas, requeridos por la estrategia descendente.

Esto no ocurre en los programas B.1.1 y B.3.3 para el caso de una elección adecuada de los nodos no válidos, ni en B.2.4 para el número total de preguntas. En estos casos el comportamiento de la estrategia descendente es similar o mejor al de la estrategia pregunta y divide.

Para comparar mejor ambas estrategias resulta interesante observar cómo crece el número de preguntas al aumentar el número de nodos de los árboles de cómputo en cada ejemplo, lo que se puede conseguir modificando los objetivos.

Se observa entonces que en los ejemplos B.1.1, B.3.3 y B.2.4 el número de preguntas requerido por el navegador guiado por la estrategia descendente no varía al considerar árboles con un mayor número de nodos (por ejemplo al sustituir el objetivo rev([67..1],L) para B.1.1 por rev([200..1],L)), mientras que sí aumenta el número de preguntas requerido por el navegador guiado por la estrategia pregunta y divide, aunque manteniendo la relación logarítmica con factor 1 que ya hemos mencionado.

En el resto de los ejemplos el número de preguntas aumenta con las dos estrategias al considerar objetivos análogos que generen árboles con mayor número de nodos. En estos casos, además, el número de preguntas para la estrategia descendente crece a un ritmo más rápido que en el caso de la estrategia pregunta y divide, por lo que la diferencia entre ambas estrategias va aumentando y siendo cada vez más desfavorable para la estrategia descendente.

	Árbol		Estr.		Estr.
Objetivo	Nodos	Prof.	Desc.		Preg.
S_7	98	12	15	8	7
S_{11}	196	16	12	23	8
S_{14}	292	20	15	29	8
S_{17}	403	22	18	35	9
S_{19}	488	24	20	39	9
S_{26}	850	32	27	53	10
S_{30}	1100	36	31	61	11
S_{35}	1456	40	36	71	11
S_{40}	1865	46	41	81	11
S_{45}	2321	50	46	91	11
S_{50}	2830	56	51	101	12
S_{55}	3386	60	56	111	12

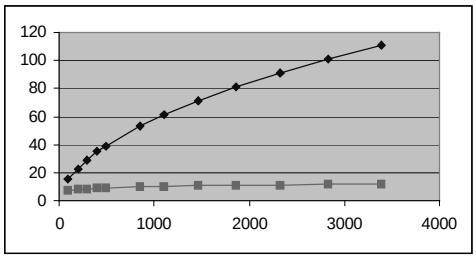


Figura 6.5: Comparación de estrategias para distintos objetivos con el programa B.3.2

Por ejemplo, el cuadro de la figura 6.5 muestra este crecimiento para distintos objetivos sobre el programa B.3.2. Cada objetivo S_i denota el objetivo solution 11 == R. Las columnas

siguientes siguen el mismo esquema que las de la figura 6.1, salvo porque hemos eliminado el máximo número de hijos de un nodo del árbol, que es en todos los casos 2. Por tanto las dos últimas columnas incluyen el número de hechos básicos que el usuario debe considerar para la estrategia descendente y pregunta y divide, respectivamente, antes de encontrar un nodo crítico. La gráfica de la derecha muestra la evolución de estos dos valores en relación con el número de nodos del árbol representado en el eje horizontal. La línea superior se refiere a los datos de la estrategia descendente mientras que la inferior corresponde a la estrategia pregunta y divide. Se puede observar como el el número de hechos básicos a considerar aumenta mucho más rápido en la estrategia descendente, llegando hasta 111 para el último objetivo, por sólo 12 hechos básicos para la estrategia pregunta y divide. En todos los ejemplos de este grupo (es decir todos los del cuadro 6.1 salvo B.2.4, así como B.1.1 B.3.3 en el caso de una elección adecuada de los nodos no válidos) se observa un comportamiento similar.

Como vemos ambas estrategias tienen un comportamiento diferente. La estrategia descendente se comporta mejor en algunos ejemplos, y peor en otros, mientras que la estrategia pregunta y divide tiene un comportamiento más "estable" en todos los casos, requiriendo un número de preguntas de aproximadamente $\log_2 n$ para árboles de n nodos. Por esta razón consideramos que es preferible la utilización de la estrategia pregunta y divide. Así, para un árbol de 65000 nodos la estrategia pregunta y divide nos asegura, al menos para todos los ejemplos que hemos considerado hasta la fecha, que necesitaremos alrededor de 16 preguntas, mientras que en el caso de la estrategia descendente para un árbol de este tamaño podemos requerir desde sólo 3 preguntas en alguno de los ejemplos considerados hasta más de 1000 en otros.

6.3. Eficiencia. Resultados Experimentales

El apartado anterior muestra como la estrategia pregunta y divide resuelve, o al menos minimiza, uno de los problemas tradicionalmente asociados a la depuración declarativa: el elevado número de preguntas que el usuario debe contestar. Por desgracia al mismo tiempo agrava otro de los problemas: el consumo de recursos, tanto en tiempo como en memoria, requeridos para la generación completa del árbol. En los dos siguientes apartados se razonará el porqué de este inconveniente, mientras que en el apartado 6.3.3 compararemos la eficiencia del navegador textual, que sólo permite la navegación descendente, con la de $\mathcal{DDT}$ que permite además la navegación siguiendo la estrategia pregunta y divide así como la navegación libre por el árbol.

6.3.1. Navegación Perezosa

Un depurador guiado por la estrategia descendente sólo necesita evaluar aquellos nodos del árbol que tiene que mostrar al usuario para determinar su evaluación. Así el árbol se puede evaluar "perezosamente" a medida que se desciende por él. Sólo los hijos del nodo marcado como no válido en un paso necesitan ser evaluados en el paso siguiente.

Por ejemplo, en el árbol de cómputo al que se refiere la última línea del cuadro de la figura 6.5 (pág. 147), se tiene que de un total de 3386 nodos un navegador siguiendo la estrategia descendente sólo 111 (el 3'28 %) son en realidad necesarios para encontrar el nodo crítico. Para árboles con mayor número de nodos la proporción se va haciendo incluso menor.

El navegador textual presentado en la sección 6.1 aprovecha que $\mathcal{TOY}$ y Curry son lenguajes que permiten la evaluación perezosa (al igual que sucede con el lenguaje funcional Haskell [75]) para reducir la evaluación del árbol de depuración únicamente a los nodos imprescindibles, con el ahorro de memoria y tiempo que veremos en el apartado 6.3.3. Para ello en lugar de comenzar la depuración con el objetivo $\mathsf{solve}^{\mathcal{T}}$ ==(true, Tree) propuesto en el teorema 4.4.7 (pág. 97), que forzaría a la evaluación completa o impaciente del árbol debido a la semántica de la igualdad estricta ==, el depurador textual comienza la depuración con un objetivo de la forma:

```
navegadorDescendente   Tree == R
                       where (Tree,true) = solve^T
```

donde navegadorDescendente es una función que implementa la especificación descrita para un navegador descendente en el apartado 6.1.1 (pág. 130). El valor de salida R no tiene importancia ya que al tratarse de una función con operaciones de entrada/salida (las preguntas al usuario y sus respuestas) se tratará de un valor de la forma <<process>> que no se mostrará al usuario.

Obsérvese que de esta manera se consigue iniciar la navegación con un árbol aún sin evaluar obtenido a través de la definición local $(\mathsf{Tree},\mathsf{true}) = \mathsf{solve}^{\mathcal{T}}$, y que irá evaluando el navegador descendente según vaya siendo necesario, evitando así la evaluación completa del árbol.

6.3.2. Navegación Impaciente: $\mathcal{DDT}$

En un primer momento puede pensarse que un razonamiento como el del punto anterior es también válido para un navegador basado en la estrategia pregunta y divide ya que esta requiere un número de preguntas que, como hemos visto, es incluso menor en mucho casos que en el navegador guiado por la estrategia descendente. Sin embargo un examen más detenido del algoritmo seguido por esta estrategia (descrita en el apartado 6.2.3, pág. 142) nos revela que ya desde el primer paso el árbol completo debe ser evaluado. En efecto, la condición a partir de la cual se obtiene el nodo N que se utilizará para dividir el árbol tras preguntar por su validez es:

$$|dif(T,N)| = min\{|dif(T,M)| : M \in T\}$$

lo que obliga a recorrer el árbol completo.

Por ello la versión actual de $\mathcal{DDT}$ evalúa el árbol de depuración completamente antes de iniciar la navegación a diferencia, como hemos visto del navegador textual descendente.

Esto también facilita el movimiento libre del usuario, el expandir o contraer partes o el árbol completo a voluntad. Como veremos en seguida esta evaluación es costosa y limita las posibilidades del navegador. Una futura versión mejorada de $\mathcal{DDT}$ debería evaluar el árbol perezosamente hasta donde fuera posible, de tal forma que si la estrategia pregunta y divide no es utilizada y el árbol completo no es expandido por el usuario pueda evitarse la evaluación de partes del árbol.

6.3.3. Resultados Experimentales

En este apartado vamos a comparar experimentalmente el consumo de memoria y tiempo requerido para diferentes sesiones de depuración mediante el navegador textual y $\mathcal{DDT}$. Al hacerlo estamos también evaluando el consumo de estos recursos para la estrategia descendente y la estrategia pregunta y divide por las razones expuestas en los puntos anteriores.

Para los experimentos hemos seleccionado cuatro ejemplos del apéndice B de entre los de mayor complejidad, tratando de incluir programas de distinto tipo. Estos son:

- Programa B.2.3. Ejemplo de programa puramente funcional que aprovecha la pereza para representar la lista infinita de los números primos. Al programa del apéndice, que se puede consultar en la página 282, le hemos añadido una regla de función:

  ```
  main N = true <== take N primes == L
  ```

  que obtiene los N primeros números primos, aunque devuelve true en lugar de mostrarlos. Esto se hace porque en las estadísticas mostradas por $\mathcal{TOY}$ el tiempo se computa desde que se lanza el objetivo hasta que se termina de escribir la respuesta, lo que en el caso de objetivos con una salida por pantalla muy extensa puede alterar significativamente el cómputo del tiempo.

- Programa B.2.6 (pág. 288). Se trata también de un programa funcional, aunque en este caso se trata de manipulación simbólica de expresiones. Al programa del apéndice se han añadido dos reglas de función para generar expresiones arbitrariamente complejas:

  ```
  buildEval N = if N== 1 then (c 0)
                    else mul  (buildEval (N-1)) (v 1)

  main N = true <== simplify (buildEval N) == R
  ```

  Un objetivo como main 3 tratará de simplificar simbólicamente la expresión mul (mul (c 0) (v 1)) (v 1) donde (v 1) representa una variable, (c 0) la constante 0 y mul la operación de multiplicación.

- Programa B.3.1 (pág. 293). Este ejemplo lógico-funcional utiliza funciones no deterministas para ordenar una lista de números enteros. Para generar objetivos que produzcan árboles con gran número de nodos añadimos dos funciones:

```
countDown N M = if N<M then []
                else if N>M then [N| countDown (N-1) M]
                     else [N]

main N = true <== permSort (countDown N 1) == L
```

  Un objetivo como main 5 intentará ordenar la lista [5,4,3,2,1].

- Programa B.3.3 (pág. 297). Es un evaluador de expresiones utilizando la técnica lógico-funcional de las extensiones [13] que hace uso de variables lógicas como argumentos de salida a través de los que se devuelven valores. Las funciones añadidas para generar expresiones complejas son:

```
builtExpr N = if N==1 then val 2
              else (val 2) :+: (builtExpr (N-1))

main N = true <== eval (builtExpr N) R1 == R2
```

  Un objetivo como main 4 trata de evaluar la suma (val 2) :+: (val 2) :+: (val 2) :+: (val 2) y de obtener su representación como cadena de caracteres.

En cada experimento hemos considerado tres magnitudes: el tiempo en milisegundos, el número de formas normales de cabeza computadas y el máximo de memoria requerido. Cada forma normal de cabeza se puede considerar un paso de cómputo en $\mathcal{TOY}$ (ver [1] para una descripción más detallada), por lo que hemos decidido incluir esta magnitud como medida del coste computacional de cada programa. Además, de los tres parámetros considerados, el número de formas normales de cabeza es el más estable, en el sentido de que al repetir los cómputos varias veces se obtiene siempre el mismo resultado. En el parámetro tiempo en cambio puede haber pequeñas diferencias (de alrededor de 10-30ms), mientras que en el valor obtenido para el máximo de memoria requerido se pueden obtener diferencias aún más significativas.

En el caso de la memoria las diferencias en los resultados obtenidos al repetir el mismo cómputo se deben a que el sistema Prolog utilizado reserva la memoria en bloques, bloques que además libera siguiendo políticas internas (según el tiempo hace que un bloque no ha sido utilizado, etc.) que el usuario no puede controlar. Por ejemplo, si realizamos un cómputo sencillo tras un cómputo complejo podemos obtener valores de consumo de memoria mucho más altos de los esperados, debido a que durante el cómputo anterior el sistema ocupó mucha memoria que todavía no ha liberado. Para minimizar este efecto y realizar unas medidas

		Tiempo en ms.			Form. Norm. Cabeza			Espacio en Ks.		
main	Nodos	P	Tex	$\mathcal{DDT}$	P	Tex	$\mathcal{DDT}$	P	Tex	$\mathcal{DDT}$
B.2.3										
10	146	20	60	260	684	1412	6118	1254	1636	2008
20	486	60	180	1080	2364	4762	24128	1255	1968	1968
40	1766	240	610	5720	8724	17462	112148	2039	5104	5111
80	6726	870	2400	34220	33444	66862	608188	2039	21823	21822
150	9992	3060	8030	195270	115204	230312	3170258	5372	33621	93343
200	?	5300	13840	†	203604	407062	†	8702	90852	†
300	?	11800	†	†	455404	†	†	14084	†	†
500	?	†	†	†	†	†	†	†	†	†
B.2.6										
10	40	20	50	110	629	1399	2669	325	812	1243
20	80	60	170	380	2384	5344	9389	663	1238	1243
40	160	250	730	1500	9194	20734	34829	1339	2014	2017
80	320	1040	3020	6390	36013	81514	133709	1255	1993	3266
100	400	1610	4470	9200	56024	126900	207149	1255	1984	5311
200	800	6470	17740	36050	222074	503854	814349	1255	1931	8590
500	2000	41150	110990	227610	1380224	3134704	5035949	2044	5098	36624
700	2800	80490	218320	442270	2702324	6138604	9850349	2046	4994	59330
900	?	133440	360980	†	4464424	10142504	†	5374	5505	†
2000	?	649870	1781910	†	22020974	50038954	†	8702	13004	†
B.3.1										
20	121	10	30	310	434	879	6632	1029	1245	1250
40	241	30	70	1060	874	1719	20472	1255	1237	2026
60	361	40	90	2250	1314	2559	41152	1253	2015	3292
80	481	40	120	3760	1754	3399	69753	2038	2006	3341
120	721	70	150	8240	2634	5079	147832	2036	1994	4654
500	3001	270	740	138920	10994	21039	2325992	5373	5191	36708
750	?	500	1090	†	16494	31539	†	5373	8246	†
10000	?	5410	14070	†	219994	420039	†	96619	93014	†
B.3.3										
10	469	40	100	3140	1015	2440	59882	2032	1984	1983
20	1654	100	260	22890	2861	7185	428237	2700	3168	8567
40	6124	330	850	178530	8651	22975	3264647	5357	8274	57705
60	?	640	1690	†	17582	47913	†	8685	13230	†
100	?	1520	4560	†	45422	127273	†	14070	34688	†
200	?	5460	†	†	171022	†	†	36881	†	†
400	?	21310	†	†	662222	†	†	96621	†	†
500	?	†	†	†	†	†	†	†	†	†

Cuadro 6.2: Comparación entre el depurador textual y $\mathcal{DDT}$

lo más precisas posibles antes de cada experimento hemos reiniciado el sistema Prolog por completo. Siguiendo esta política sí se obtienen los mismos resultados para el mismo cómputo.

El cuadro 6.2 muestra los resultados obtenidos para los ejemplos en los tres parámetros indicados. Las magnitudes encabezadas por la columna P se refieren a los datos obtenidos para el cómputo mediante el programa original, sin tener en cuenta la depuración.

Las columnas encabezadas por la palabra Tex se refieren a datos obtenidos para el depurador textual. Estos datos se refieren al coste requerido para ejecutar el programa transformado pero todavía sin comenzar a navegar el árbol. Por tanto, para el navegador textual los datos se refieren al coste mínimo. Sin embargo hay que precisar que el coste real obtenido al finalizar la depuración descendente sólo aumenta en un tanto por ciento muy pequeño con respecto a estos valores mínimos ya que, como hemos indicado, sólo unos pocos nodos, menos del 5 % en todos los casos de la tabla 6.1, son evaluados por el navegador.

Finalmente, las columnas encabezadas por las siglas $\mathcal{DDT}$ se refieren al coste computacional requerido para la evaluación completa del árbol, y han sido obtenidos justo antes de iniciar el proceso Java del navegador.

Para cada ejemplo hemos probado diversos objetivos, dando diferentes valores como parámetro a la función main que, como hemos visto, hemos incluido en todos los ejemplos. Estos valores se encuentran en la columna de la izquierda del cuadro, seguidos por el número de nodos del árbol de depuración resultante. Por ejemplo los cinco primeros datos de la primera fila del cuadro nos indican que para el ejemplo B.2.3 el objetivo main 10 == R produce un árbol de depuración con 146 nodos, requiriendo el programa original 20 ms. para resolver el objetivo, mientras el cómputo del programa transformado, sin evaluar el árbol de cómputo, lleva 50 ms. y el cómputo del programa transformado incluyendo la evaluación completa del árbol requiere 110 ms.

Todos los datos se han obtenido bajo sistema operativo Linux en un ordenador portátil Compaq Presario 1400 a 600 MHz. y 128 Mb. de memoria RAM. Los datos marcados con un símbolo † se corresponden con objetivos para los que el programa que encabeza la columna en la que se encuentra el símbolo no ha podido terminar al agotarse la memoria. Como el número de nodos del árbol utilizado lo hemos obtenido de los datos proporcionados por $\mathcal{DDT}$, en los objetivos en los que el cómputo del árbol completo agota la memoria no conocemos dicha cantidad, lo que representamos en el cuadro mediante el símbolo '?'.

Un primer análisis de los datos nos indica que, como es de esperar, el coste requerido al ejecutar el objetivo en el programa transformado (columna Tex) es mayor que para el programa normal P, y que este valor se hace aún mucho mayor si se requiere la evaluación completa del árbol. En este último caso, el que corresponde a $\mathcal{DDT}$, se tiene que el sistema se queda sin memoria para valores para los que ni el programa original ni el programa descendente tienen problema alguno, lo que muestra las limitaciones de la implementación actual del navegador gráfico. Especialmente llamativo resulta el caso del programa B.3.1 donde ya no se puede utilizar $\mathcal{DDT}$ para un objetivo como main 750 ==R mientras que tanto el programa normal como el navegador textual continúan sin alcanzar los límites de memoria para un objetivo como main 10000.

Como se ve el número de nodos máximo de los árboles para los que $\mathcal{DDT}$ funciona sin problemas de memoria es variable: entre casi 10000 nodos en el primer ejemplo, B.2.3, hasta menos de 3000 en B.2.6 o apenas esta cantidad en B.3.1. Hay dos razones para este comportamiento: en primer lugar el tamaño de los nodos que varía enormemente de un caso a otro. En el caso de B.2.3 se trata de listas de números enteros, mientras que en B.2.6 tanto los parámetros como los resultados representan expresiones de gran tamaño, con el consiguiente aumento en el consumo de memoria. En segundo lugar, el límite de memoria no siempre se alcanza durante el cómputo que produce el árbol que se depurará posteriormente, sino, debido al indeterminismo, durante alguno de los cómputos intermedios que finalmente, si no se diera el problema de memoria fallarían. Esto sucede en el programa B.3.1, que hace un extensivo uso de las funciones indeterministas.

Hay que observar que las cifras de consumo de memoria, dependen como mencionamos anteriormente de las políticas internas del sistema Prolog, lo que lleva a que estos valores sean sólo orientativos. En particular se observan algunos datos contradictorios, como en la primera fila del programa B.3.3, la correspondiente al valor 10 para el parámetro de main, donde se tiene que el consumo de memoria para el programa transformado es menor que para el programa original, tanto con evaluación perezosa como con evaluación completa del árbol. O en otros casos como en el ejemplo B.2.6 en los que se tiene que para cómputos más complejos se obtienen valores del consumo de memoria ligeramente menores que para cómputos más sencillos (con menor valor del parámetro de main.

Para hacer un estudio más detallado del coste computacional requerido por los dos navegadores podemos utilizar el cuadro 6.3, en el que hemos considerado los mismo en ejemplos que en el cuadro 6.2 sólo que mostrando solamente el factor por el que hemos de multiplicar los valores experimentales del programa original para obtener los requeridos por el navegador textual y por $\mathcal{DDT}$.

Por ejemplo los primeros datos de la primera fila indican que si consideramos el programa B.2.3 con el objetivo main 10 == R tenemos que el tiempo requerido para la ejecución del programa transformado, todavía sin evaluar el árbol, es 3 veces el tiempo del programa original, mientras que el tiempo requerido si forzamos la evaluación completa del árbol tal y como requiere $\mathcal{DDT}$ es de 13 veces el tiempo del programa original.

Insistimos en que los datos referentes al consumo en memoria son menos fiables, por lo que aquí nos vamos a limitar a interpretar los datos en el consumo de tiempo y en el número de formas normales de cabeza requeridas. Creemos que esta interpretación también es aplicable al caso del consumo de memoria pero a falta de datos experimentales más precisos esto no puede ser comprobado actualmente.

El cuadro muestra claramente como para el depurador descendente el aumento en cuanto a tiempo y a número de formas normales de cabeza es prácticamente constante para cada programa, situándose esta constante en todos los casos entre los valores 2 y 3. Sin embargo en el caso de la generación completa del árbol, y por tanto de los recursos necesarios para la utilización de $\mathcal{DDT}$ se tiene que, salvo en el programa B.2.6, el factor de crecimiento no es constante sino que aumenta al aumentar el número de nodos del árbol, llegándose a factores tan elevados como 541 para el caso del tiempo o de 377 para el consumo de formas

main	Tiempo en ms. Tex	Tiempo en ms. $\mathcal{DDT}$	Form. Norm. Cabeza Tex	Form. Norm. Cabeza $\mathcal{DDT}$	Espacio en Ks. Tex	Espacio en Ks. $\mathcal{DDT}$
B.2.3						
10	× 3'00	× 13'00	× 2'06	× 8'94	× 1'30	× 1'60
20	× 3'00	× 18'00	× 1'98	× 10'23	× 1'57	× 1'56
40	× 2'54	× 23'83	× 2'00	× 12'85	× 2'50	× 2'51
80	× 2'76	× 39'33	× 2'00	× 18'18	× 10'70	× 10'70
150	× 2'62	× 63'81	× 2'00	× 27'52	× 6'25	× 17'38
200	× 2'61	†	× 2'00	†	× 10'44	†
B.2.6						
10	× 2'50	× 5'50	× 2'22	× 4'24	× 2'49	× 3'82
20	× 2'83	× 6'33	× 2'24	× 3'94	× 1'87	× 1'87
40	× 2'92	× 6'00	× 2'25	× 3'78	× 1'59	× 1'59
80	× 2'90	× 6'14	× 2'26	× 3'71	× 1'58	× 2'60
100	× 2'77	× 5'71	× 2'26	× 3'70	× 1'58	× 4'23
200	× 2'74	× 5'57	× 2'27	× 3'67	× 1'54	× 6'84
500	× 2'69	× 5'53	× 2'27	× 3'65	× 2'49	× 17'92
700	× 2'71	× 5'49	× 2'27	× 3'65	× 2'44	× 29'29
900	× 2'70	†	× 2'27	†	× 1'03	†
2000	× 2'74	†	× 2'27	†	× 1'49	†
B.3.1						
20	× 3'00	× 31'00	× 2'03	× 15'28	× 1'21	× 1'21
40	× 2'33	× 35'33	× 1'97	× 23'42	× 0'98	× 1'61
60	× 2'25	× 56'25	× 1'95	× 31'32	× 1'60	× 2'62
80	× 3'00	× 94'00	× 1'94	× 39'77	× 0'98	× 1'64
120	× 2'14	× 117'00	× 1'93	× 56'12	× 0'98	× 2'29
500	× 2'74	× 514'00	× 1'91	× 211'57	× 0'97	× 6'83
750	× 2'18	†	× 1'91	†	× 1'53	†
10000	× 2'60	†	× 1'91	†	× 0'96	†
B.3.3						
10	× 2'50	× 78'50	× 2'40	× 59'00	× 0'97	× 0'97
20	× 2'60	× 228'90	× 2'51	× 149'68	× 1'17	× 3'17
40	× 2'57	× 541'00	× 2'66	× 377'37	× 1'54	× 10'73
60	× 2'64	†	× 2'72	†	× 1'53	†
100	× 3'00	†	× 2'80	†	× 2'47	†

Cuadro 6.3: Factores de aumento de coste en el cuadro 6.2

normales de cabeza.

Como consecuencia de estos datos podemos afirmar que el factor de crecimiento constante para el depurador textual, con una constante relativamente pequeña como 2 o 3, hace que este depurador se pueda utilizar en casos donde el uso de $\mathcal{DDT}$ es imposible, dado que se alcanza el límite de memoria del sistema. En muchos otros casos el uso de $\mathcal{DDT}$ resulta posible pero no sea realista debido a su elevado consumo de tiempo. Aún en el único ejemplo de los analizados en el que el comportamiento de la evaluación completa requerida por $\mathcal{DDT}$ tiene un comportamiento del mismo tipo que la evaluación perezosa requerida por el depurador textual, el programa B.2.6, la constante de tiempos obtenida es peor que en caso del depurador textual y el navegador agota la memoria del sistema mucho antes.

Como ya dijimos en la introducción de esta sección estas conclusiones apuntan en la dirección opuesta de los resultados comentados en el apartado 6.2.4, donde indicábamos que era preferible utilizar $\mathcal{DDT}$ con la estrategia pregunta y divide para minimizar las preguntas realizadas al usuario. Hasta el momento no hemos logrado combinar ambas ventajas, eficiencia en cuanto al consumo de recursos y en cuanto al número de preguntas al usuario, por lo que este problema queda como parte del trabajo futuro sugerido en esta tesis.

6.4. Detección Automática de la Validez de Algunos Nodos del Árbol

En esta sección vamos a plantear dos técnicas que pueden utilizarse en el depurador de $\mathcal{TOY}$ (tanto con el navegador en modo texto como en $\mathcal{DDT}$) para determinar la validez de algunos de los nodos del árbol sin necesidad de preguntar al oráculo. El primero de estos métodos define una noción decidible de consecuencia que permite deducir la validez de algunos nodos a partir de la validez de otros nodos del mismo árbol, mientras que el segundo se basa en la utilización de una especificación fiable, quizá parcial, del programa a depurar.

6.4.1. Relación de Consecuencia

Una simplificación obvia que se comenta en muchos trabajos de depuración declarativa consiste en evitar la repetición de preguntas al usuario. En efecto, a menudo hay nodos repetidos en los árboles de depuración, y cuando el usuario determina la validez de uno de estos nodos obviamente ya no es necesario preguntar de nuevo por la validez de cada una de sus repeticiones.

En [17, 19] presentamos una *relación de consecuencia* entre hechos básicos basada en el orden de aproximación $\sqsubseteq$ que introdujimos en el apartado 3.1.2 (ver definición 3.1.1, pág. 48) que amplía esta idea a nodos cuya validez o no validez es consecuencia de la de otros nodos:

Definición 6.4.1. Decimos que un hecho básico $f\ \overline{s}_n \to s$ es consecuencia de otro hecho básico $f\ \overline{t}_n \to t$, y lo escribiremos como

$$f\ \overline{t}_n \to t\ \succeq\ f\ \overline{s}_n \to s$$

cuando exista alguna sustitución total $\theta \in Subst$ tal que

$$t_1\theta \sqsubseteq s_1,\ \ldots\ , t_n\theta \sqsubseteq s_n, s \sqsubseteq t\theta$$

Por ejemplo, si consideramos los hechos básicos de la figura 6.3, puede probarse fácilmente que

$$fib \to [1,1,2,3,4,5 \mid \bot] \ \succeq\ fib \to [1,1,2,3,4 \mid \bot]$$
$$fibAux\ 1\ 2 \to [3,4,5 \mid \bot] \ \succeq\ fibAux\ 1\ 2 \to [3,4 \mid \bot]$$

con θ la sustitución identidad en ambos casos. Nótese que en el primer caso ambos hechos básicos son válidos, mientras que en el segundo caso los dos hechos básicos son no validos. Como veremos en seguida esto no es una coincidencia sino que en general dados dos hechos básicos φ, φ' tales que $\varphi\ \succeq\ \varphi'$ se tiene que

- Si φ es válido en la interpretación pretendida, φ' también lo es.
- Si φ' no es válido en la interpretación pretendida, φ tampoco lo es.

El depurador utilizará estas dos propiedades, que probaremos en el teorema 6.4.2, para ahorrar preguntas al usuario. Por ejemplo, si el usuario indica que $fibAux\ 1\ 2 \to [3,4 \mid \bot]$ no es válido, el navegador marcará automáticamente el nodo correspondiente a $fibAux\ 1\ 2 \to [3,4,5 \mid \bot]$ como no válido.

Antes de establecer el teorema que prueba que se cumplen estas propiedades hay otro punto que debemos aclarar: ¿es siempre posible determinar si se satisface o no la relación de consecuencia entre dos hechos básicos? La respuesta es positiva, tal y como asegurará el teorema 6.4.1 que veremos en seguida. Para decidir si se cumple la relación utilizaremos el siguiente algoritmo, que incluye algunas definiciones que son útiles para su descripción.

Algoritmo
Sean $f\ \overline{t}_n \to t$ y $f\ \overline{s}_n \to s$ dos hechos básicos sin variables comunes. Para comprobar si $f\ \overline{t}_n \to t\ \succeq\ f\ \overline{s}_n \to s$ vamos a definir un sistema de transformaciones similar al utilizado por Martelli y Montanari en su algoritmo de unificación [57]. Estas transformaciones se aplicarán a *configuraciones*. Cada configuración se representará mediante la notación $S \square W$, en la que S es un multiconjunto de aproximaciones $a \to b$, con $a, b \in Pat_\bot$, y W un conjunto de variables.

Para cada multiconjunto S y $\theta \in Subst$ vamos a decir que $S\theta$ se *satisface* cuando para todo $s \to t \in S$ se tenga $t\theta \sqsubseteq s\theta$. El *conjunto de soluciones* de una configuración $S \square W$ se define como el conjunto de sustituciones totales sobre las variables de W para las que se verifican todas las aproximaciones en S, es decir:

$$Sol(S \square W)\ =\ \{\theta \in Subst \mid dom(\theta) \subseteq W, ran(\theta) \subseteq Pat, S\theta \text{ se satisface } \}$$

El propósito del algoritmo es encontrar alguna solución para la configuración inicial $S_0 \Box W_0$ con $S_0 = s_1 \to t_1, \ldots, s_n \to t_n,\ t \to s$ y $W_0 = var(f\ \overline{t}_n \to t)$, con lo que indicamos que sólo las variables de $f\ \overline{t}_n \to t$ pueden ser instanciadas. Cada paso del algoritmo transformará una configuración $S_i \Box W_i$ en un nueva configuración $S_{i+1} \Box W_{i+1}$ produciendo una substitución θ_{i+1}. Esto se hace aplicando (de forma no determinista) una regla de transformación a algún elemento $a \to b$ de S_i, seleccionado también de forma no determinista. Representaremos uno de estos pasos del algoritmo de la forma:

$$\underbrace{a \to b, S}_{S_i} \Box W_i \quad \vdash_{\theta_i} \quad S_{i+1} \Box W_{i+1}$$

Para simplificar en ocasiones escribiremos K_i para representar una configuración $S_i \Box W_i$. A continuación presentamos las reglas de transformación:

Reglas de Transformación

En todas las reglas se supone que $X, Y \in Var$ con $X \in W$; $a_k, b_k, t \in Pat_\perp$; $s \in Pat$; y que $h \in DC \cup FS$. Además X_k representa siempre una variable nueva no introducida hasta el momento.

$R1$	$Y \to Y,\ S \Box W$	$\vdash_{id}$	$S \Box W$
$R2$	$t \to \perp,\ S \Box W$	$\vdash_{id}$	$S \Box W$
$R3$	$h\ \overline{a}_m \to h\ \overline{b}_m,\ S \Box W$	$\vdash_{id}$	$\ldots, a_k \to b_k, \ldots S \Box W$
$R4$	$s \to X,\ S \Box W$	$\vdash_{\{X \mapsto s\}}$	$S\{X \mapsto s\} \Box W$
$R5$	$X \to Y,\ S \Box W$	$\vdash_{\{X \mapsto Y\}}$	$S\{X \mapsto Y\} \Box W$
$R6$	$X \to h\ \overline{a}_m,\ S \Box W$	$\vdash_{\{X \mapsto h\ \overline{X}_m\}}$	$\ldots, X_k \to a_k, \ldots\ S\{X \mapsto h\ \overline{X}_m\} \Box W, \overline{X}_m$

El algoritmo finaliza cuando se llega a una configuración a la que no es posible aplicar ninguna regla. El teorema siguiente asegura que dicha configuración siempre existe y también prueba que el algoritmo sirve para determinar si se cumple la relación de consecuencia.

Teorema 6.4.1. *El algoritmo anterior finaliza tras un número finito de pasos alcanzando una configuración $S_j \Box W_j$ a la que ya no puede aplicarse transformación alguna. Más aún, la relación de consecuencia inicial $f\ \overline{t}_n \to t\ \succeq\ f\ \overline{s}_n \to s$ se cumple sii $S_j = \emptyset$.*

Demostración Ver pág. 266, apéndice A.

El interés de la relación de consecuencia para la depuración declarativa viene justificado por el siguiente resultado:

Teorema 6.4.2. *La relación de consecuencia entre hechos básicos es un preorden decidible. Además, cualquier interpretación pretendida $\mathcal{I}$ es cerrada bajo la relación de consecuencia, es decir si $f\ \overline{t}_n \to t\ \succeq\ f\ \overline{s}_n \to s$ y $(f\ \overline{t}_n \to t)\ \in \mathcal{I}$ entonces $(f\ \overline{s}_n \to s)\ \in \mathcal{I}$.*

Demostración.

El que las interpretaciones pretendidas son cerradas bajo la relación de consecuencia se deduce de la definición 6.4.1 de relación de consecuencia por las condiciones (ii), (iii) de la definición de interpretación de Herbrand (ver apartado 3.3.3, pág. 64), ya que por definición

toda interpretación pretendida debe ser de Herbrand en nuestro marco. También de la definición 6.4.1 se tiene que $\succeq$ es una relación transitiva y reflexiva, y por tanto un preorden. Para probar que $\succeq$ es decidible, consideremos dos hechos básicos cualesquiera $f\ \bar{t}_n \rightarrow t$, $f\ \bar{s}_n \rightarrow s$, así como un renombramiento de variables ρ tal que $(f\ \bar{t}_n \rightarrow t)\rho$ y $f\ \bar{s}_n \rightarrow s$ no tienen variables en común. Por la definición de relación de consecuencia se tiene ahora que los dos puntos siguientes (a) y (b) son equivalentes:

(a) $f\ \bar{t}_n \rightarrow t\ \succeq\ f\ \bar{s}_n \rightarrow s$

(b) $(f\ \bar{t}_n \rightarrow t)\rho\ \succeq\ f\ \bar{s}_n \rightarrow s$

Finalmente, el teorema 6.4.1 asegura que (b) puede decidirse aplicando el algoritmo a la configuración inicial $S_0 \Box W_0$, donde:

$$S_0 = s_1 \rightarrow t_1\rho, \ldots, s_n \rightarrow t_n\rho,\ t\rho \rightarrow s \qquad W_0 = var((f\ \bar{t}_n \rightarrow t)\rho)$$

■

Tanto el navegador descendente como $\mathcal{DDT}$ incorporan la comprobación de la relación de consecuencia:

- En el navegador descendente se lleva internamente una lista l de los nodos que el usuario ha marcado como no válidos. Antes de mostrar una secuencia de hijos se mira a ver si algún nodo N en la secuencia tiene como consecuencia un nodo en l. Si es así entonces N es no válido (por el teorema 6.4.2, al ser todos los nodos en l no válidos), y el navegador continua descendiendo por N sin necesidad de preguntar al usuario, repitiendo el proceso sobre los hijos de N.

- En el caso de $\mathcal{DDT}$, cada vez que el usuario indica que un hecho básico asociado a un nodo es válido todos los nodos cuyos hechos básicos son consecuencia suya son marcados automáticamente como válidos y viceversa, cada vez que el usuario indica que el hecho básico φ asociado a un nodo es no válido, todos los demás nodos cuyos hechos básicos tienen como consecuencia φ son marcados como no válidos.

6.4.2. Especificaciones Fiables

Como hemos visto, $\mathcal{DDT}$ permite al usuario marcar los nodos del árbol de cómputo que desee como fiables (*trusted*) durante una sesión de depuración. Al hacer esto todos los nodos cuyos hechos básicos correspondan a la misma función que a la del nodo marcado son considerados automáticamente como válidos. Vamos a ver en este apartado un sistema similar para determinar la validez de nodos en el árbol mediante *especificaciones fiables*.

Diremos que un programa $\mathcal{TOY}$ S_P es una especificación fiable si el usuario asume como válidos todos los hechos básicos que pueden deducirse de S_P con las reglas de inferencia del cálculo semántico SC de la sección 3.2.1 (pág. 55); y asume como no válidos todos los hechos básicos que no pueden deducirse de S_P con las reglas de inferencia de SC.

Sea P una programa conteniendo alguna regla incorrecta, T un árbol de depuración correspondiente a algún síntoma inicial obtenido al resolver un objetivo con respecto a P,

y S_P una especificación fiable para *algunas* de las funciones definidas en P. Entonces el depurador seguirá el siguiente procedimiento para determinar la validez de algunos nodos de T sin necesidad de preguntar al usuario:

Para cada hecho básico φ :	$f\ \bar{t}_n \rightarrow t$ *correspondiente a algún nodo* N *de* T*:*
Si válido?(S_P, φ) = sí	*eliminar* N*.*
Si válido?(S_P, φ) = no	*marcar* N *como no válido.*
Si válido?(S_P, φ) = no-sé	*no marcar* N*.*

La respuesta *válido?*(S_P, φ) = *no-sé* se corresponde con dos posibles situaciones:

- La función f no está definida en S_P.
- La función está definida en S_P pero al intentar determinar la validez de φ se ha superado un cierto tiempo límite.

El segundo punto se incluye para evitar posibles problemas de no terminación al intentar deducir la validez de φ, al tratarse de un problema no decidible. El depurador incluye un algoritmo para computar *válido?*(S_p, φ) siguiendo los siguientes pasos:

- Comprobar si la función utilizada en el hecho básico está definida en S_P. Si no es así devolver *no-sé*. Si sí está definida pasar al punto siguiente.
- Sustituir cada variable X_i de φ por una constante nueva c_i, obteniendo así un nuevo hecho básico φ' sin variables.
- Al no contener variables φ' cumple trivialmente las condiciones de admisibilidad con respecto al conjunto vacío de variables (enunciadas en el apartado 3.1.5, pág. 51) requeridas para los objetivos en $\mathcal{TOY}$. Por ello podemos utilizar el resolutor de objetivos del sistema para saber si φ' puede resolverse con respecto a S_P. Para lograr esto se ha incorporado al resolutor de objetivos del sistema la posibilidad de tratar aproximaciones conteniendo el símbolo $\perp$. Obsérvese que la única respuesta posible, de obtenerse alguna, será la respuesta identidad debido a la ausencia de variables en φ'.

  Si el sistema resolutor de objetivos es capaz de resolver φ' entonces el algoritmo devolverá *sí* y si falla sin ser capaz de resolver φ' algoritmo devolverá *no*. Si se supera un cierto límite de tiempo sin que el sistema finalice, bien fallando o con éxito, el algoritmo interrumpirá el cómputo devolviendo *no-sé*.

El razonamiento acerca de la corrección del algoritmo depende de la suposición de que el sistema de resolución de objetivos GS usado en el sistema verifica las dos siguientes propiedades:

1. GS debe ser correcto con respecto al cálculo semántico SC (definición 3.2.1, pág. 60), tal y como hemos supuesto en capítulos anteriores.

2. *GS* debe ser además *correcto con respecto al fallo finito*, siguiendo la siguiente definición:

   Un sistema de resolución GS se dice correcto con respecto al fallo finito cuando se cumple que para todo objetivo G para el que GS no es capaz de encontrar ninguna respuesta producida, fallando en tiempo finito, se cumple que no existe ninguna solución θ para G en SC.

Si asumimos que el sistema de resolución de $\mathcal{TOY}$ cumple estas dos condiciones, como pensamos que ocurre, se tiene que:

- Si el resultado de *válido?*(S_P, φ) es *sí* entonces es que el sistema resolutor de objetivos es capaz de probar φ', con φ' obtenido a partir de φ sustituyendo variables por constantes como hemos indicado. La respuesta obtenida debe ser la identidad al no haber variables en φ' y por la corrección del sistema se tiene entonces $S_P \vdash_{SC} \varphi'$. Es fácil probar que entonces se tiene $S_P \vdash_{SC} \varphi$ con una demostración con la misma estructura en la que cada paso DC de la forma $c_i \rightarrow c_i$ para una constante c_i nueva usada para reemplazar una variable X_i de φ puede ser reemplazada por un paso RR de la forma $X_i \rightarrow X_i$, y análogamente cada paso BT $c_i \rightarrow \bot$ en la prueba de $S_P \vdash_{SC} \varphi'$ se corresponde con un paso BT $X_i \rightarrow \bot$ en la prueba de $S_P \vdash_{SC} \varphi$. De aquí, al ser S_P una especificación fiable, se llega a que φ es válido en el modelo pretendido de P. Entonces φ puede ser eliminado de cualquier APA obtenido con respecto a P por el apartado 4 de la proposición 5.3.1 (pág. 114).

- Si el resultado es *no* entonces el sistema resolutor de objetivos es incapaz de resolver φ' y por la corrección con respecto al fallo se tiene que φ' no puede ser derivado de S_P en el cálculo SC. Entonces es fácil probar que tampoco puede serlo φ, ya que a partir de una prueba para $S_P \vdash_{SC} \varphi$ se puede construir otra para $S_P \vdash_{SC} \varphi'$ sustituyendo cada paso RR $X_i \rightarrow X_i$ por un paso DC de la forma $c_i \rightarrow c_i$, con c_i la constante usada para reemplazar X_i en φ', y sustituyendo cada paso BT de la forma $X_i \rightarrow \bot$ por un paso BT de la forma $c_i \rightarrow \bot$. Entonces, al ser S_p fiable, se tiene que φ no es válido en el modelo pretendido de P y marcar N como no válido es correcto.

Al comenzar cada sesión de depuración el depurador de $\mathcal{TOY}$ pregunta al usuario si dispone de alguna especificación fiable del programa. Si es así el depurador pregunta el nombre del fichero conteniendo la especificación (que debe estar ya compilada) y la utiliza durante la fase de navegación. En el caso de $\mathcal{DDT}$ al construirse previamente el árbol la simplificación se lleva a cabo antes de la navegación. En el analizador descendente antes de mostrar la lista de hijos al usuario el navegador comprueba si la validez de alguno de los hijos puede obtenerse a partir de la especificación. Si alguno puede probarse no válido, la navegación continua automáticamente por él sin consultar al usuario. Si no se encuentra ninguno no válido pero sí algunos válidos, éstos se sustituyen por sus hijos, llevando a cabo de esta forma la operación de eliminación, y se repite el proceso con el nuevo conjunto de hechos básicos.

El programa de la figura 6.6 es una especificación fiable del programa para el cómputo de la razón aúrea de la figura 6.1 (página 132). Este programa utiliza un método diferente para

```
fibN 1                  = 1
fibN 2                  = 1
fibN N                  = if N>2 then (fibN (N-1))+(fibN (N-2))

fib                     = map fibN (from 0)

goldenApprox            = (tail fib) ./. fib

infixr 20 ./.
[X | Xs]  ./.  [Y | Ys]  = [ X/Y | Xs ./. Ys]

tail [X|Xs]             = Xs

map F []                = []
map F [X|Xs]            = [F X | map F Xs]

take 0 L                = []
take N [X|Xs]           = [X| take (N-1) Xs] <== N>0

from N                  = [N | from N+1]

main R                  = true <== take 5 goldenApprox == R
```

Figura 6.6: Una especificación fiable del programa de la figura 6.1

calcular los numeros de la secuencia de Fibonacci. Podemos imaginar que este programa es una primera versión del programa de las aproximaciones de la razón aúrea. Funciona correctamente pero utiliza una generación de los números de Fibonacci muy ineficiente. En este contexto, el programa de la figura 6.1 podría ser un intento de mejorar la eficiencia de este programa. Después de probar la nueva versión el usuario puede notar que se obtiene una respuesta diferente, y decidir que el primer programa era probablemente correcto pero que ha introducido un error en la nueva versión. Entonces puede utilizarse el depurador declarativo usando la primera versión de la figura 6.6 como especificación fiable.

En el caso de $\mathcal{DDT}$ tras generar el árbol completo el navegador eliminará 12 nodos del árbol de cómputo y marcará otros 3 como no válidos. En el árbol inicial habrá entonces 11 nodos con validez desconocida, en contraste con los 23 nodos en esta situación que se tendrían si no se utiliza la especificación fiable, simplificando así la sesión de depuración.

Capítulo 7

Conclusiones y Trabajo Futuro

En este capítulo vamos a resumir los resultados obtenidos durante los capítulos anteriores, y a comentar posibles desarrollos futuros, dedicando una sección de este último capítulo a cada uno de estos aspectos.

7.1. Conclusiones

Hemos tratado en esta tesis acerca de la depuración declarativa de respuestas incorrectas en lenguajes lógico-funcionales. Nuestro propósito ha sido estudiar si esta técnica resulta adecuada tanto desde un punto de vista teórico (primera parte de la tesis) como práctico (segunda parte). Resumimos a continuación los resultados obtenidos y las conclusiones que de ellos se derivan.

7.1.1. Resultados Teóricos

El primer objetivo que planteamos en el apartado 1.2 (pág. 4) del capítulo 1 fue lograr una definición adecuada de árbol de depuración para los cómputos asociados a respuestas erróneas y la prueba formal de la corrección y completitud de la depuración declarativa basada en este árbol. En el apartado 3.3.2 (pág. 4) hemos definido los árboles que satisfacen este primer objetivo, a los que hemos llamado APAs (árboles de prueba abreviados). Estos árboles son análogos a los EDTs definidos para la depuración de la programación funcional en otras propuestas [71, 68, 78, 77]. La diferencia con estos trabajos, que representa nuestra aportación en este punto, es que en nuestro caso el árbol de prueba del que se obtiene el APA se corresponde con una demostración en un cálculo semántico adecuado (el cálculo SC del apartado 3.2.1, pág. 55), algo habitual en el caso de la programación lógica [51, 88] pero que hasta la fecha no había sucedido en relación con los árboles de depuración de lenguajes funcionales. Esta relación lógica entre el árbol de prueba y la semántica del programa depurador nos lleva a poder probar formalmente la corrección y completitud de la depuración basada en los APAs tal y como pretendíamos, como se establece en el teorema 3.3.6 (pág. 67).

Para que este árbol sea realmente útil en la práctica hemos de disponer de un método que nos permita su obtención efectiva, tal y como indica el segundo objetivo planteado al comienzo de la tesis. En su trabajo [71], H. Nilsson y J. Sparud indican dos posibilidades para la obtención de los árboles usados en la depuración de programación funcionales, que como ya hemos dicho son análogos a los APAs propuestos en nuestro caso para la programación lógico-funcional:

1. Utilizar una transformación de programas tal que todo cómputo en el programa transformado devuelva, como parte de su resultado, el árbol de prueba de su cómputo correspondiente en el programa inicial.

2. Modificar la máquina abstracta sobre la que se ejecuta el programa para que sea capaz de producir el árbol de cómputo.

La segunda opción tiene como principal ventaja la eficiencia, y ha sido seguida por H. Nilsson en posteriores trabajos como [68] para la implementación de su depurador declarativo para el lenguaje Freja[67], un subconjunto del lenguaje funcional Haskell [75]. La primera opción tiene la ventaja de su portabilidad al no depender de una máquina concreta y hacer más fácil razonar formalmente sobre su corrección, por lo que la hemos preferido en este trabajo, a costa de una menor eficiencia. Sin embargo, la transformación definida en el citado trabajo [71] sólo resultaba válida para un conjunto muy restringido de programas lo que la hacía inviable en general para programas funcionales y consecuentemente lógico-funcionales. Como indican otros autores en [65, 78] la extensión a programas generales no resulta obvia al conducir la propuesta inicial a programas con errores de tipo. Nuestra transformación resuelve este problema tal y como se discute en el apartado 4.2 (pág. 72) y permite la utilización de la transformación de programas como alternativa para la obtención de los árboles de cómputo utilizados por el depurador. Más aún, la definición formal de la transformación permite probar que efectivamente sirve para su propósito, lo que nos lleva al resultado enunciado en el teorema 4.4.7 que garantiza que el programa transformado obtendrá el cómputo en un sistema cuyo sistema de resolución de objetivos cumpla unas propiedades básicas. A partir de aquí hemos demostrado el teorema 4.4.8 que asegura que el depurador basado en esta transformación de programas será capaz de localizar una regla de programa incorrecta a partir de una respuesta incorrecta, suponiendo que el usuario sea capaz de determinar adecuadamente la validez de los nodos del árbol de cómputo.

7.1.2. Aplicación Práctica

Una vez desarrollado el método a seguir para la realización de un depurador declarativo falta por comprobar su eficiencia (tercero de los objetivos marcados en el apartado 1.2). En el capítulo hemos presentado dos depuradores declarativos desarrollados a partir de estas ideas en los sistema $\mathcal{TOY}$ [1, 54] y Curry [39] de Münster (este último desarrollado en colaboración con Wolfgang Lux y Herbert Kuchen). Resulta interesante mencionar que ambos sistemas están escritos en lenguajes diferentes ($\mathcal{TOY}$ en Prolog, Curry de Münster en Haskell) y se basan en técnicas diferentes ($\mathcal{TOY}$ produce código Prolog que es posteriormente

interpretado, mientras que el sistema Curry desarrollado por la Universidad de Münster produce código que es ejecutado posteriormente por una máquina abstracta [55]). A pesar de estas diferencias la implementación de la transformación de programas ha podido trasladarse con pocos cambios de un sistema a otro, mostrando una de las principales ventajas de esta técnica: su independencia con respecto a los detalles de la implementación. Los dos depuradores forman parte de las distribuciones actuales para estos sistemas, disponibles en

```
http://babel.dacya.ucm.es/toy
```

y

```
http://danae.uni-muenster.de/~lux/curry
```

El último objetivo marcado al inicio de la tesis consistía en investigar la eficiencia de los depuradores principalmente en dos aspectos:

- Consumo de recursos: tiempo y memoria requerida para la generación del árbol de cómputo.
- Número de preguntas que debe contestar el usuario.

De ambos aspectos nos hemos ocupado en el capítulo 6. En él hemos presentado experimentos comparando dos formas de generar el árbol: mediante generación perezosa e impaciente, o, con otras palabras, generando el árbol según lo va requiriendo el navegador o generándolo completamente antes de comenzar la navegación. Los experimentos demuestran que la generación perezosa no introduce apenas coste adicional, mientras que la generación completa del árbol sólo parece realista para programas de tamaño pequeño-medio, debido a su elevado consumo de recursos. En cuanto al número de preguntas que el usuario debe contestar antes de que el depurador llegue a localizar el error, hemos realizado experimentos comparando dos posibles estrategias: la estrategia *descendente* y la *divide y pregunta*. La mayor parte de los resultados muestran una clara ventaja para la estrategia divide y pregunta, comprobándose que el uso de esta estrategia mantiene un número de preguntas relativamente pequeño para árboles de número de nodos elevado, para los que la estrategia descendente resulta en la mayor parte de los casos inviable. Resumiendo estos experimentos, las conclusiones que se obtienen son:

- Es preferible la generación perezosa del árbol a la generación impaciente para minimizar el consumo de recursos.
- Es preferible utilizar la estrategia divide y pregunta a la estrategia descendente para minimizar el número de preguntas realizadas al oráculo.

Por tanto, lo ideal sería utilizar la estrategia divide y pregunta combinada con una generación perezosa del árbol. Sin embargo hasta el momento no hemos podido lograr los dos objetivos simultáneamente. En efecto, la estrategia divide y pregunta requiere a cada paso un recorrido por el árbol completo para determinar el nodo que mejor divide el árbol en

dos partes (los nodos de su subárbol y los nodos fuera de su subárbol). A falta de una solución futura que logre aunar generación perezosa con la estrategia divide y pregunta, o con alguna otra de similar eficiencia, en el sistema $\mathcal{TOY}$ hemos optado por disponer de dos depuradores: un depurador en modo texto que realiza generación perezosa y utiliza la estrategia descendente y otro, un navegador gráfico escrito en Java al que hemos llamado $\mathcal{DDT}$ que requiere la generación anticipada del árbol pero permite cualquiera de las dos estrategias, así como el movimiento libre por el árbol sin tener que seguir una estrategia predeterminada. El capítulo 6 muestra las posibilidades de $\mathcal{DDT}$, mientras que en el apéndice B se muestran numerosos ejemplos de programas depurados con el navegador descendente.

Hemos propuesto además dos ideas para conseguir reducir el número de preguntas realizadas al usuario (sección 6.4, pág. 156). La primera basada en una relación de implicación que permite asegurar la validez o no validez de ciertos nodos a partir de la información suministrada por el usuario para otros nodos. La segunda se basa en la utilización de una especificación fiable de todas o algunas funciones del programa que permita reemplazar como oráculo al usuario cuando sea posible. Esta última idea había sido utilizada previamente por otras técnicas de depuración pero no en depuradores declarativos similares al presentado en esta tesis.

7.2. Trabajo Futuro

El trabajo llevado a cabo en esta tesis deja lugar a desarrollos futuros, tanto de carácter teórico como práctico. En este apartado final vamos a repasar brevemente algunas de estas posibilidades. Las ideas expuestas en cada caso son tan sólo ideas iniciales pendientes de una investigación más detallada.

7.2.1. Tratamiento de programas con restricciones y depuración de respuestas perdidas

En la depuración declarativa de programas lógicos se consideran dos tipos de errores: respuestas erróneas y respuestas perdidas (véase apartado 2.1.4, pág. 10). Ambos tipos de errores tienen sentido en programación lógico-funcional, pero en esta tesis tan sólo hemos cubierto el caso de las respuestas incorrectas. Por tanto la continuación natural de este trabajo es el estudio de la depuración declarativa de respuestas perdidas y su posible incorporación al depurador.

Tal y como vimos en el apartado dedicado a los cálculos POS y NEG (pág. 13) el tratamiento de las respuestas perdidas, también llamadas *síntomas negativos*, precisa de un árbol de depuración diferente al de las respuestas incorrectas. En [88] se relaciona el problema del tratamiento de las respuestas perdidas con el de la depuración de programas admitiendo restricciones, y está idea parece también válida en el caso de la programación lógico-funcional.

7.2.2. Mejoras en el depurador declarativo del sistema $\mathcal{TOY}$

Vamos a comentar algunas posibles mejoras y extensiones que podrían incorporarse al depurador actual del sistema $\mathcal{TOY}$, así como a su navegador gráfico $\mathcal{DDT}$.

- Incluir la posibilidad de marcar módulos o funciones de $\mathcal{TOY}$ como fiables antes de la depuración. Esta posibilidad, al menos para el caso de módulos completos, sí está disponible en el depurador declarativo que hemos incluido en el sistema Curry, y se comenta en el apartado 5.3.1 (pág. 113). Las funciones correspondientes a estos módulos devuelven en el programa transformado un tipo de nodos especiales (representados por la constructora EmptyCTreeNode en el depurador de Curry). Cuando la función ctClean encuentra uno de estos nodos lo sustituye por sus hijos y de esta manera se eliminan las preguntas acerca de las funciones marcadas. En $\mathcal{TOY}$ la selección de las funciones fiables podría hacerse, por ejemplo, incorporando en el código un tipo especial de comentarios, como en el caso siguiente:

  ```
  % @trusted: take, map, filter
  ```

  El uso de esta posibilidad serviría, como sucede en Curry, para disminuir el número de preguntas realizada por el navegador al usuario.

- Tratamiento primitivas/entrada salida. Tal y como se comentó en el apartado 5.5.1 (pág. 124) el depurador actual no admite programas incluyendo operaciones de entrada salida. Una versión futura mejorada del depurador debería resolver esta limitación para ampliar el conjunto de programas a los que se puede aplicar la depuración. En el propio apartado 5.5.1 se dan algunas ideas, aún imprecisas, encaminadas a resolver este problema.

- Lograr combinar la generación perezosa con la estrategia *divide y pregunta*. Como se ha mencionado anteriormente de esta manera se conseguiría un depurador que no consume demasiados recursos y que a la vez requiere una cantidad de preguntas al usuario no demasiado elevada para localizar el error. Aunque hasta la fecha la estrategia *divide y pregunta* conlleva la evaluación completa del árbol, tal y como se hace en el navegador $\mathcal{DDT}$, podrían investigarse alternativas que puedan paliar, al menos parcialmente, esta necesidad. Una idea a este respecto se tiene al observar que la estrategia divide y pregunta no exige realmente la evaluación completa de cada nodo, sino sólo determinar el número de nodos que forman parte de cada subárbol, así como el número de nodos total. Parece posible obtener esta información sin realmente evaluar el nodo, sólo detectando su presencia. Así los nodos sólo serían evaluados realmente antes de ser mostrados. Habría que investigar si de este modo el consumo de recursos disminuye apreciablemente.

- Incorporar documentación sobre la interpretación pretendida de las funciones a los programas fuentes y utilizar esta información como ayuda a la depuración. Más concretamente, la idea consistiría en permitir en $\mathcal{TOY}$ la inclusión de comentarios similares a los utilizados en el lenguaje Java [47] y utilizados por la herramienta *javadoc*

de este lenguaje. Durante la compilación de un programa cualquiera *P.toy* el sistema generaría, además del fichero intermedio *P.tmp.out* y el fichero compilado *P.pl* que comentamos en el apartado 5.2.1 (pág. 108), un fichero adicional *P.html* recogiendo toda la información incluida en estos comentarios. Además de la evidente utilidad de esta posibilidad para la documentación de programas, también podría utilizarse durante la depuración para proporcionar información a la persona que está utilizando el depurador acerca de la interpretación pretendida de algunas o todas las funciones utilizadas durante el cómputo analizado.

Por ejemplo, supongamos que en el programa fuente se incluye una función notDiv precedida de los comentarios siguientes:

```
/**
 * A divisibility test
 * @param X an integer
 * @param Y an integer
 * @return R true if X does not divide Y and false otherwise
 */
notDiv :: int -> int -> bool
notDiv X Y = mod X Y > 0
```

Los *marcadores* @param y @return indican al sistema que el comentario se refiere a un parámetro o al valor de salida de la función, respectivamente. Con esta información el sistema podría generar automáticamente en el fichero .html una información similar a la siguiente:

Function:	notDiv.
Description:	A divisibility test.
Type:	int → int → bool
Program Arity:	2
Intended meaning:	notDiv X Y → R iff
	- X is an integer
	- Y is an integer
	- R is true if X does not divide Y and false otherwise

El depurador declarativo podría entonces permitir al usuario consultar esta información, en particular la referente a la interpretación pretendida de la función, en cualquier momento para ayudarle a determinar la validez de un hecho básico. Por ejemplo con esta información es más sencillo determinar que un hecho básico como notDiv 3 6 → true, que aparece en la sesión de depuración del ejemplo B.2.3 (apéndice B, pág. 282), es válido en la interpretación pretendida, ya que 3 sí divide a 6.

7.2.3. Utilización de aserciones

La inclusión de aserciones en relación con la depuración declarativa ya ha sido utilizada en depuradores declarativos para programación lógica y programación lógica con restricciones [30, 11, 23, 43, 49, 79, 80]. La idea es utilizar las aserciones para conocer la validez o no validez de algunos de los nodos sin necesidad de consultar al usuario.

En particular se suelen distinguir dos tipos de aserciones:

- Positivas, que indican conjuntos de hechos básicos que son válidos en la interpretación pretendida. En el caso de $\mathcal{TOY}$ una posible forma de representar estas aserciones podría ser $\varphi \Rightarrow f\ \bar{t}_n \to t$ indicando que cualquier instancia del hecho básico $f\ \bar{t}_n \to t$ correspondiente a una valoración de las variables que satisfaga la fórmula φ está en la interpretación pretendida. La definición del lenguaje aceptado para las fórmulas φ depende del tipo y generalidad de las aserciones que se quieran adoptar.

- Negativas, que indican conjuntos de hechos básicos que no son válidos en la interpretación pretendida. Con una notación análoga a la del punto anterior estas aserciones se podrían representar en $\mathcal{TOY}$ en la forma $f\ \bar{t}_n \to t \Rightarrow \varphi$, indicando así que cualquier instancia del hecho básico $f\ \bar{t}_n \to t$ correspondiente a una valoración de las variables que no satisfaga φ no está en la interpretación pretendida.

Las aserciones podrían incluirse directamente en el código del programa. Por ejemplo supongamos que nuestro lenguaje de aserciones admitiera fórmulas con igualdad, operaciones aritméticas y un operador $|\cdot|$ para representar la longitud de una lista, y consideremos la siguiente versión errónea de la función que concatena dos listas:

```
% @assert append X Y -> L => |X|+|Y| = |L|
append [] L = L
append [X|Xs] L = append Xs L
```

El marcador `@assert` indicaría aquí al sistema la presencia de una aserción, en este caso una aserción negativa. Durante la depuración un hecho básico como append [V] [U] → [U], deducible de un programa conteniendo esta función, sería reconocido por el depurador como no válido en la interpretación pretendida sin necesidad de consultar al usuario.

Nótese que las aserciones también pueden utilizarse para la localización *estática* de errores tal y como se hace en *diagnosis abstracta* [25, 4].

7.2.4. Aplicación de la depuración declarativa a SQL

Una línea de investigación diferente basada en la depuración declarativa sería la aplicación de esta técnica a otros paradigmas. Tal y como expusimos en la sección 2.2 (pág. 27), las ideas generales de la depuración declarativa son aplicables a cualquier paradigma, aunque hasta la fecha hayan sido empleadas fundamentalmente dentro del área de la programación declarativa.

En particular, pensamos que esta técnica podría ser interesante para la depuración de consultas dentro del lenguaje de consulta de bases de datos relacionales *SQL*. El resultado de una consulta *SQL* es un conjunto de filas o *tuplas* con los datos de la base de datos que satisfacen las condiciones incluidas en la consulta. Son varias las características de este lenguaje que nos llevan a sugerir la depuración declarativa como un método de depuración digno de ser investigado:

- Se trata de un lenguaje declarativo, basado en un cálculo lógico (cálculo relacional de tuplas).
- La depuración de una sentencia *SQL* no puede hacerse por mecanismo de traza usuales, ya que cada sentencia se descompone internamente como la composición de muchas operaciones más sencillas difíciles de analizar por separado.
- Se puede hablar tanto de respuestas erróneas (cuando la respuesta contiene una tupla inesperada) como de respuestas perdidas (cuando falta una tupla esperada en la respuesta).

Algunos experimentos preliminares (un pequeño depurador que detecta con ayuda del usuario errores en la cláusula *where* de sentencias *SQL* sencillas) muestran que la depuración declarativa puede aportar un enfoque nuevo para la depuración de estos lenguajes.

Apéndice A

Demostración de las Proposiciones y Teoremas

Incluimos en este apéndice las demostraciones de las proposiciones y teoremas enunciados en los capítulos 3 y 4. Estableciendo esta separación hemos pretendido facilitar la lectura de la tesis y sus resultados, dejando para este apéndice las pruebas de los resultados principales que resultaran excesivamente largas o técnicas. Algunas de las demostraciones dependen de lemas auxiliares cuyos enunciados y demostraciones también incluimos aquí.

A.1. Resultados Presentados en el Capítulo 3

A.1.1. Demostración de la Proposición 3.2.1

Esta proposición, enunciada en la página 57, establece algunas propiedades del cálculo semántico de la sección 3.2.1 (pág. 55). Probamos cada uno de sus apartados:

1. Demostramos la doble implicación procediendo por inducción estructural sobre s.

   Caso Base. 3 posibilidades:

   - $s \equiv \bot$. En este caso $t \rightarrow \bot$ se cumple siempre mediante la regla BT de SC (ninguna otra regla es aplicable), mientras que $t \sqsupseteq \bot$ se cumple por el ítem 1 de la definición 3.1.1.
   - $s \equiv X$, con $X \in Var$. En este caso tanto $t \rightarrow X$ como $t \sqsupseteq X$ se cumplen sólo si $t = s = X$, aplicando la regla RR para probar $X \rightarrow X$, la única regla SC aplicable, y el ítem 2 de la definición 3.1.1 para $X \sqsupseteq X$.
   - $s \equiv c$, $c \in DC^0$. Tanto $t \rightarrow c$ como $t \sqsupseteq c$ se cumplen sólo si $t = s = c$, en el primer caso aplicando la regla DC, de nuevo la única posibilidad, y en el segundo el ítem 2 de la definición 3.1.1.

   Caso Inductivo. Hay 2 posibilidades:

- $s \equiv c\ s_1 \ldots s_m,\ c \in DC^n, 0 \leq m \leq n$. La única posibilidad para probar tanto $t \rightarrow c\ s_1 \ldots s_m$ en SC como $t \sqsupseteq c\ s_1 \ldots s_m$ es que t sea de la forma $t \equiv c\ t_1 \ldots t_m$. En este caso tanto $c\ t_1 \ldots t_m \rightarrow c\ s_1 \ldots s_m$ se cumple si y sólo se cumplen $t_1 \rightarrow s_1, \ldots, t_m \rightarrow s_m$ aplicando la regla DC de SC. Análogamente $c\ t_1 \ldots t_m \sqsupseteq c\ s_1 \ldots s_m$ se cumple si y sólo se cumplen $t_1 \sqsupseteq s_1, \ldots, t_m \sqsupseteq s_m$ aplicando el ítem 2 de la definición 3.1.1. Pero por hipótesis de inducción se tiene que

  $$t_1 \rightarrow s_1 \Leftrightarrow t_1 \sqsupseteq s_1,\ \ldots,\ t_m \rightarrow s_m \Leftrightarrow t_m \sqsupseteq s_m$$

  por lo que $c\ t_1 \ldots t_m \rightarrow c\ s_1 \ldots s_m$ si y sólo si $c\ t_1 \ldots t_m \sqsupseteq c\ s_1 \ldots s_m$. Además, por hipótesis de inducción las pruebas en SC de cada $t_i \rightarrow s_i$ sólo requiere el uso de las reglas DC, RR y BT, por lo que sucede lo mismo con la prueba completa.
- $s \equiv f\ s_1 \ldots s_m,\ f \in DC^n, 0 \leq m < n$. Análogo al caso anterior.

2. Sea $t \in Pat_\perp$. Para probar que $P \vdash_{SC} t \rightarrow t$ bastará, por el apartado anterior con probar que $t \sqsupseteq t$, lo que se tiene al ser $\sqsubseteq$ un orden parcial y cumplir por tanto la propiedad reflexiva.

3. Para todo $t \in Pat_\perp,\ s \in Pat$: $P \vdash_{SC} t \rightarrow s$ sii $t \equiv s$. Por inducción sobre la profundidad de la demostración $t \rightarrow s$.

   Caso Base. Profundidad 0. En este caso el último paso de la demostración sólo puede haber utilizado la regla RR o la regla DC con $m = 0$, y en ambos casos $P \vdash_{SC} t \rightarrow s$ sii $t \equiv s$ (si se ha usado RR se tiene $t \equiv s \equiv X, X \in Var$, mientras que para DC debe cumplirse $t \equiv s \equiv c, c \in DC^0$).

   Caso Inductivo. Profundidad $p > 0$. La única regla que se puede haber utilizado en el último paso es DC al ser t un patrón, y por tanto t será de la forma $h\ t_1 \ldots t_m$ y s de la forma $h\ s_1 \ldots s_m$ con $t_i \in Pat_\perp,\ s_i \in Pat$ para $i = 1 \ldots m$. El último paso será entonces:

   $$\frac{s_1 \rightarrow t_1\ \ldots\ s_m \rightarrow t_m}{h\ \overline{s}_m \rightarrow h\ \overline{t}_m}$$

   Aplicando la hipótesis de inducción se llega a $s_i \equiv t_i$ para $i = 1 \ldots n$, y por tanto $s \equiv t$.

4. Sean $t,\ s \in Pat_\perp$ tales que $P \vdash_{SC} t \rightarrow s$. Aplicando el apartado 1 de esta misma proposición en el sentido "$\Rightarrow$)" se tiene que $t \sqsupseteq s$, y aplicando entonces la implicación "$\Leftarrow$)" del mismo apartado se llega al resultado.

5. En lugar del enunciado propuesto vamos a probar este otro resultado compuesto de dos partes:

   Para todo $e \in Exp_\perp$, $t \in Pat_\perp$ y $\theta,\ \theta' \in Subst_\perp$, con $\theta \sqsubseteq \theta'$, entonces:

   (a) Si $P \vdash_{SC} e\theta \rightarrow t$ entonces $P \vdash_{SC} e\theta' \rightarrow t$ con una demostración con la misma estructura y tamaño.

(b) Si $P \vdash_{SC} e\ \overline{a}_k\theta \to t$ entonces $P \vdash_{SC} e\ \overline{a}_k\theta' \to t$, con una demostración con la misma estructura y tamaño.

Probamos ambos resultados simultáneamente ya que, como veremos enseguida, ambas demostraciones están relacionadas.

Como en los casos anteriores probaremos el resultado por inducción sobre la profundidad de la demostración de $e\theta \to t$ o de $e\ \overline{a}_k\theta \to t$, suponiendo que $t \neq \bot$, ya que en el caso $t \equiv \bot$ el enunciado es cierto trivialmente.

Caso Base. Profundidad 0.
Apartado **(a)**.
Si la profundidad de la demostración es 0 tenemos que las reglas empleadas pueden haber sido RR o DC y que por tanto $e\theta$ será o bien de la forma $X \in Var$ o bien $c \in DC^0$. En ambos casos se tiene que que si $\theta \sqsubseteq \theta'$ ha de ocurrir que $e\theta' \equiv e\theta$ y por tanto ambas demostraciones serán idénticas.

Apartado **(b)**.
Para que la profundidad sea 0 debe ser $k = 0$ y el resultado se cumple trivialmente.

Caso Inductivo. Profundidad $p > 0$. Suponemos (hipótesis de inducción) que tanto **a** como **b** se cumplen para demostraciones con profundidades menores que p.

Apartado **(a)**.
La regla utilizada en el último paso de la demostración SC puede ser:

Regla DC. En este caso $t \equiv h\ \overline{t}_m$. Distinguimos 2 posibilidades, según si e es o no una variable:

(a.1) $e \equiv X$, con $X \in Var$.
En este caso debe ser $e\theta \equiv h\ \overline{s}_m$ y el paso DC final debe ser de la forma

$$\textbf{(4)} \quad \frac{s_1 \to t_1\ \ \ldots\ \ s_m \to t_m}{h\ \overline{s}_m \to h\ \overline{t}_m}$$

Como $\theta \sqsubseteq \theta'$ debe tenerse $e\theta' \equiv h\ \overline{s'}_m$ con $s'_i \sqsupseteq s_i$ para $1 \leq i \leq m$. Ahora consideramos m nuevas variables $X_1 \ldots X_m$, y $2m$ sustituciones definidas por $\theta_i = \{X_i \mapsto s_i\}$, $\theta'_i = \{X_i \mapsto s'_i\}$, para $1 \leq i \leq m$. Se tiene entonces que para todo $1 \leq i \leq m$ se cumplen:

$$\textbf{(5)} \quad P \vdash_{SC} X_i\theta_i \to t_i \quad (\text{de (4), ya que } X_i\theta_i \equiv s_i)$$

$$\textbf{(6)} \quad \theta_i \sqsubseteq \theta'_i \quad (\text{ya que } X_i\theta_i \equiv s_i \sqsubseteq t_i \equiv X_i\theta'_i)$$

Aplicando la parte (a) de hipótesis de inducción a (5) y (6) se tiene $P \vdash_{SC} X_i\theta'_i \to t_i$, es decir $P \vdash_{SC} s'_i \to t_i$ para todo i con $1 \leq i \leq m$, con una demostración con la misma estructura y tamaño que la de $P \vdash_{SC} s_i \to t_i$. Utilizando esta información se puede

probar $P \vdash_{SC} e\theta' \to t$ con una demostración cuya inferencia final también es un paso DC de la forma

$$\frac{s'_1 \to t_1 \ \ldots \ s'_m \to t_m}{h\ \overline{s'}_m \to h\ \overline{t}_m}$$

demostración que, por todo lo dicho, tiene la misma estructura y tamaño que la de $P \vdash_{SC} e\theta \to t$.

(a.2) $e \notin Var$.
Entonces e tiene que ser de la forma $e \equiv h\ \overline{e}_m$ y por tanto $e\theta \equiv h\ \overline{e}_m\theta$ con un paso final de demostración

$$\textbf{(7)} \qquad \frac{e_1\theta \to t_1 \ \ldots \ e_m\theta \to t_m}{h\ \overline{e}_m\theta \to h\ \overline{t}_m}$$

Por su parte, $e\theta'$ será $h\ \overline{e}_m\theta'$ y para probar $P \vdash_{SC} e\theta' \to t$ bastará con hacer una demostración en SC de la forma:

$$\frac{e_1\theta' \to t_1 \ \ldots \ e_m\theta' \to t_m}{h\ \overline{e}_m\theta' \to h\ \overline{t}_m}$$

para completar la demostración, recordamos que por (7) se tiene $P \vdash_{SC} e_i\theta \to t_i$ y que por tanto, aplicando el apartado (a) de la hipótesis de inducción, tendremos que $P \vdash_{SC} e_i\theta' \to t_i$ para $1 \leq i \leq m$ con una demostración con la misma estructura y tamaño.

Apartado **(b)**.
Para que el último paso de la demostración de $P \vdash_{SC} e\ \overline{a}_k\theta \to t$ sea un paso DC tiene que ser $e \equiv h\ \overline{e}_m$, con $m + k \leq n$ si $h \in DC^n$ o $m + k < n$ si $h \in FS^n$, y por tanto $t \equiv h\ \overline{t}_{m+k}$. Dicho último paso tendrá entonces la forma:

$$\textbf{(8)} \qquad \frac{e_1 \to t_1 \ \ldots \ e_m \to t_m \ \ a_1\theta \to t_{m+1} \ \ldots \ a_k\theta \to t_{m+k}}{h\ \overline{e}_m\ \overline{a}_k\theta \to h\ \overline{t}_{m+k}}$$

Entonces para probar $P \vdash_{SC} e\ \overline{a}_k\theta' \to t$ finalizaremos con un paso DC de la forma:

$$\frac{e_1 \to t_1\ \ldots\ e_m \to t_m\ \ a_1\theta' \to t_{m+1}\ \ldots\ a_k\theta' \to t_{m+k}}{h\ \overline{e}_m\ \overline{a}_k\theta' \to h\ \overline{t}_{m+k}}$$

donde:

- De (8) se tiene $P \vdash_{SC} e_i \to t_i$ para $i = 1 \ldots m$.
- Se tiene $P \vdash_{SC} a_i\theta' \to t_{m+i}$ para $i = 1 \ldots k$, con una demostración de la misma forma y tamaño que la de $P \vdash_{SC} a_i\theta \to t_{m+i}$ que se puede encontrar en (8), y aplicando la hipótesis de inducción, apartado (a).

Regla $AR + FA$.

Apartado **(a)**.

En este caso $e\theta$ debe ser la aplicación de una función f, y por tanto e debe ser de la forma $f\ \overline{e}_n\ \overline{a}_k$, con paso final de la demostración debe ser de la forma:

$$\textbf{(9)}\quad \frac{e_1\theta \to t_1\ \ldots\ e_n\theta \to t_n\ \ \dfrac{C \quad r \to s}{\boxed{f\ \overline{t}_n \to s}}\ \ s\ \overline{a}_k\theta \to t}{(f\ \overline{e}_n\ \overline{a}_k)\theta \to t}$$

Podremos entonces probar $P \vdash_{SC} e\theta' \to t$, finalizando por un paso del mismo tipo:

$$\frac{e_1\theta' \to t_1\ \ldots\ e_n\theta' \to t_n\ \ \dfrac{C\theta' \quad r\theta' \to s\theta}{\boxed{f\ \overline{t}_n \to s}}\ \ s\ \overline{a}_k\theta' \to t}{(f\ \overline{e}_n\ \overline{a}_k)\theta' \to t}$$

donde:

- Por hipótesis de inducción, apartado (a), se cumple $P \vdash_{SC} e_i\theta' \to t_i$ para todo $1 \le i \le m$, con una demostración de la misma estructura y tamaño que la de la correspondiente prueba para $P \vdash_{SC} e_i\theta \to t_i$, que se tiene por (9).
- Por hipótesis de inducción, apartado (b), se cumple $P \vdash_{SC} s\ \overline{a}_k\theta' \to t$ con una demostración de la misma estructura y tamaño que la de la correspondiente prueba para $P \vdash_{SC} s\ \overline{a}_k\theta \to t$ que existe por (9).

Apartado **(b)**.

Para que el último paso de la demostración de $P \vdash_{SC} e\ \overline{a}_k\theta \to t$ sea un paso FA, e ha de ser de la forma $e \equiv f\ \overline{e}_m$, con $m + k \ge n$ y $f \in FS^n$. Distinguimos entonces dos casos:

(b.1) $m < n$.
El último paso de la demostración será entonces:

(10)

$$\frac{e_1 \to t_1 \ \ldots \ e_m \to t_m \quad a_1\theta \to t_{m+1} \ldots a_{n-m}\theta \to t_n \quad \dfrac{C \quad r \to s}{\boxed{f\ \overline{t}_n \to s}} \quad s\ a_{n-m+1}\theta \ldots a_k\theta \to t}{f\ \ \overline{e}_m\ \overline{a}_k\theta \to t}$$

Podremos entonces probar $P \vdash_{SC} e\ \overline{a}_k\theta' \to t$ con una demostración con la misma estructura y tamaño cuyo paso final es un paso $AF + FA$ de la forma:

$$\frac{e_1 \to t_1 \ \ldots \ e_m \to t_m \quad a_1\theta' \to t_{m+1} \ldots a_{n-m}\theta' \to t_n \quad \dfrac{C \quad r \to s}{\boxed{f\ \overline{t}_n \to s}} \quad s\ a_{n-m+1}\theta' \ldots a_k\theta' \to t}{f\ \ \overline{e}_m\ \overline{a}_k\theta' \to t}$$

donde

- La demostración de $P \vdash_{SC} e_i \to t_i$ se obtiene directamente de (10), para $i = 1 \ldots m$.
- $P \vdash_{SC} a_i\theta' \to t_i$ con una demostración con la misma estructura y tamaño que la de $P \vdash_{SC} a_i\theta \to t_i$ que se puede encontrar en (10), aplicando el apartado (a) de la hipótesis de inducción.
- Las demostraciones de las condiciones C y de $r \to s$ se obtienen directamente de (10).
- $P \vdash_{SC} s\ a_{n-m+1}\theta' \ldots a_k\theta' \to t$ con una demostración con la misma estructura y tamaño que la de $P \vdash_{SC} s\ a_{n-m+1}\theta \ldots a_k\theta \to t$ que se puede encontrar en (10), aplicando la hipótesis de inducción, apartado (b).

(b.2) $m \geq n$.
Entonces el último paso de la demostración será:

$$\textbf{(11)} \quad \frac{e_1 \to t_1 \ \ldots \ e_n \to t_n \quad \dfrac{C \quad r \to s}{\boxed{f\ \overline{t}_n \to s}} \quad s\ e_{n+1} \ldots e_m\ \overline{a}_k\theta \to t}{f\ \ \overline{e}_m\ \overline{a}_k\theta \to t}$$

Podremos entonces probar $P \vdash_{SC} e\ \overline{a}_k\theta' \to t$ con una demostración con la misma estructura y tamaño cuyo último paso es un paso $AF + FA$ de la forma:

$$\frac{e_1 \to t_1 \ \ldots \ e_n \to t_n \quad \dfrac{C \quad r \to s}{\boxed{f\ \overline{t}_n \to s}} \quad s\ e_{n+1} \ldots e_m\ \overline{a}_k\theta' \to t}{f\ \ \overline{e}_m\ \overline{a}_k\theta' \to t}$$

donde

- La demostración de $P \vdash_{SC} e_i \to t_i$ se obtiene directamente (11), para $i = 1 \ldots n$.

- La demostración de $P \vdash_{SC} s\ e_{n+1} \ldots e_m\ \overline{a}_k\theta'$ tendrá la misma estructura y tamaño que la de $P \vdash_{SC} s\ e_{n+1} \ldots e_m\ \overline{a}_k\theta$ que se puede encontrar en (11). Para demostrar esto llamamos s' a $(s\ e_{n+1} \ldots e_m)$. Entonces de (11) tenemos que $P \vdash_{SC} s'\ \overline{a}_k\theta$ y aplicando la hipótesis de inducción, apartado (b), obtenemos $P \vdash_{SC} s'\ \overline{a}_k\theta'$ que es lo que pretendíamos probar.

6. Sea $e \in Exp_\perp$, $s \in Pat_\perp$ tal que $P \vdash_{SC} e \to s$, y sea $\theta \in Subst$. Veamos que entonces $P \vdash_{SC} e\theta \to s\theta$.

   Al igual que en los puntos anteriores comenzamos por separar el caso $s \equiv \perp$ en el que se cumple la propiedad de manera trivial. El resto de los casos se pueden probar, también de forma análoga a los items anteriores, por inducción sobre la profundidad de la demostración de $P \vdash_{SC} e \to s$.

   Caso Base. Profundidad 0.

   Si la profundidad de la demostración es 0 tenemos que las reglas empleadas pueden haber sido, como en los casos anteriores, RR o DC y se tiene o bien que $e \equiv s \equiv X$ con $X \in Var$ o bien $e \equiv s \equiv c$, $c \in DC^0$. En ambos casos se tendrá que $e\theta \equiv s\theta$ y por tanto $e\theta \to s\theta$ será fácilmente comprobable aplicando repetidamente las reglas DC y RR.

   Caso Inductivo. Profundidad $p > 0$.

   Distinguimos según la regla aplicada en el último paso de la demostración:

   - Regla DC. En ese caso $e \equiv h\ \overline{e}_m$ y $s \equiv h\ \overline{s}_m$ con una demostración en SC cuyo último paso es de la forma:

     $$\textbf{(12)} \quad \frac{e_1 \to s_1\ \ldots\ e_m \to s_m}{h\ \overline{e}_m \to h\ \overline{s}_m}$$

     Entonces $e\theta = h\ \overline{e}_m\theta$ y $s\theta \equiv h\ \overline{s}_m\theta$. Ahora bien, para cada i con $1 \leq i \leq m$ se tiene por (12) que $e_i \to s_i$ se cumple en SC con un árbol de prueba de profundidad menor que p, y por hipótesis de inducción se debe cumplir en SC también $e_i\theta \to s_i\theta$, de donde $e\theta \to s\theta$ puede probarse con un paso final DC de la forma:

     $$\frac{e_1\theta \to s_1\theta\ \ldots\ e_m\theta \to s_m\theta}{h\ \overline{e}_m\theta \to h\ \overline{s}_m\theta}$$

- Regla $AR + FA$. Entonces se tiene que $e \equiv f\ \overline{e}_n\ \overline{a}_k$ y el último paso para la demostración de $P \vdash_{SC} e \to s$ será:

$$\textbf{(13)}\quad \frac{e_1 \to t_1\ \ldots\ e_n \to t_n \qquad \dfrac{C \quad r \to t}{\boxed{f\ \overline{t}_n \to t}} \qquad t\ \overline{a}_k \to s}{f\ \overline{e}_n\ \overline{a}_k \to s}$$

con $(f\ \overline{t}_n \to r \Leftarrow C) \in [P]_\perp$. Podemos entonces utilizar la instancia $(f\ \overline{t}_n \to r \Leftarrow C)\theta \in [P]_\perp$ para probar $P \vdash_{SC} e\theta \to s\theta$, finalizando con una regla $AR + FA$ de la forma:

$$\frac{e_1\theta \to t_1\theta\ \ldots\ e_n\theta \to t_n\theta \qquad \dfrac{C\theta \quad r\theta \to t\theta}{\boxed{f\ \overline{t}_n\theta \to t\theta}} \qquad (t\ \overline{a}_k)\theta \to s\theta}{(f\ \overline{e}_n\ \overline{a}_k)\theta \to s\theta}$$

Ya que:

- Como estamos suponiendo $s \neq \perp$ y θ es total, se tiene que $s\theta \neq \perp$ tal y como pide la regla $AR + FA$.
- Análogamente se puede comprobar que cada aproximación $(u\theta \to v\theta) \in C\theta$ tiene demostración en SC utilizando la hipótesis de inducción y el hecho de que se tiene $P \vdash_{SC} u \to v$ para cada $(u \to v) \in C$ por (13).
- Para probar cada $a\theta == a'\theta \in C\theta$ nos fijamos en la prueba de la correspondiente $a == a' \in C$, que debe existir por (13). Esta demostración debe finalizar forzosamente por un paso JN de la forma

$$\frac{a \to w \quad a' \to w}{a == a'}$$

con $w \in Pat$. Como w y θ son totales $w\theta$ también debe serlo y se puede entonces aplicar la hipótesis de inducción para probar que se verifican $P \vdash_{SC} a\theta \to w\theta$ y $P \vdash_{SC} a'\theta \to w\theta$, y de aquí se tiene que es posible probar $P \vdash_{SC} a\theta == a'\theta$ finalizando con un paso JN de la forma:

$$\frac{a\theta \to w\theta \quad a'\theta \to w\theta}{a\theta == a'\theta}$$

- Finalmente se tiene que $P \vdash_{SC} (t\ \overline{a}_k)\theta \to s\theta$ con una demostración del mismo tamaño y estructura que la de $P \vdash_{SC} t\ \overline{a}_k \to s$ que se tiene en (13), aplicando la hipótesis de inducción.

■

A.1.2. Demostración de la Proposición 3.3.2

Probamos los diferentes puntos que forman esta proposición, definida en la página 63.

1. Si $P \vdash_{SC} e \to t$ con un árbol de prueba T y $t \sqsupseteq s$, entonces se puede probar $P \vdash_{SC} e \to s$ mediante un árbol de prueba T' de profundidad menor o igual a la de T y tal que $apa(T)$ y $apa(T')$ sólo difieren en la raíz.

   Si $s \equiv \bot$ el resultado se tiene trivialmente aplicando la regla BT de SC para probar $P \vdash_{SC} e \to s$, obteniendo una demostración con profundidad 0 y por tanto mínima. En otro caso demostraremos el resultado por inducción sobre la profundidad del árbol de prueba T de $P \vdash_{SC} e \to t$.

   Caso Base. Profundidad 0. En este caso el último paso de la demostración sólo puede haber utilizado la regla RR o la regla DC con $m = 0$ (BT no porque entonces sería $t \equiv \bot$ y en consecuencia $s \equiv \bot$ y estamos suponiendo $s \not\equiv \bot$). Como ha de ser $t \sqsupseteq s$ en ambos casos se tendrá, análogamente al caso base del resultado anterior, que forzosamente $s \equiv t$ y por tanto $P \vdash_{SC} e \to s$ con una demostración de la misma profundidad.

   Caso Inductivo. Profundidad $p > 0$. Nos fijamos en la regla SC aplicada en la raíz del árbol de prueba para $P \vdash_{SC} e \to t$. Como $p > 0$ hay dos posibilidades, que examinamos por separado.

   - Regla DC. En este caso e debe ser de la forma $e \equiv h\ \overline{e}_m$ y $t \equiv h\ \overline{t}_m$. El último paso de la prueba de $e \to t$ será entonces de la forma:

     $$\textbf{(1)} \quad \frac{e_1 \to t_1 \ \ldots \ e_m \to t_m}{h\ \overline{e}_m \to h\ \overline{t}_m}$$

     Como se tiene por hipótesis que $t \sqsupseteq s$, con $t \equiv h\ \overline{t}_m$ y $s \not\equiv \bot$, de la definición de $\sqsubseteq$ (def. 3.1.1, pág. 48) se tiene que deberán existir s_i con $1 \leq i \leq m$ tales que $s \equiv h\ \overline{s}_m$ y que $t_i \sqsupseteq s_i$ para todo $1 \leq i \leq m$. Como de **(1)** tenemos que cada $e_i \to t_i$ con $1 \leq i \leq m$ admite una prueba en SC de profundidad menor de p, aplicando la hipótesis de inducción tendremos que $P \vdash_{SC} e_i \to s_i$ para $i \leq i \leq m$ y por tanto $P \vdash_{SC} e \to s$, puede probarse con una demostración cuyo último paso consiste en la aplicación de una regla DC de la forma:

     $$\textbf{(2)} \quad \frac{e_1 \to s_1 \ \ldots \ e_m \to s_m}{h\ \overline{e}_m \to h\ \overline{s}_m}$$

     Además la profundidad de esta demostración será menor o igual a la de (1) al verificarse esta propiedad para cada una de las premisas. Por otra parte, aplicando

la definición de APA tenemos

$$apa(T) = \text{árbol}(h\ \overline{e}_m \to h\ \overline{t}_m,\ apa'(T_{e_1 \to t_1}) +\!\!+ \ \ldots\ +\!\!+\ apa'(T_{e_m \to t_m}))$$

mientras que para T':

$$apa(T') = \text{árbol}(h\ \overline{e}_m \to h\ \overline{s}_m,\ apa'(T'_{e_1 \to s_1}) +\!\!+ \ \ldots\ +\!\!+\ apa'(T'_{e_m \to s_m}))$$

donde $T'_{e_i \to s_i}$ representa el árbol de prueba de $e_i \to s_i$ para $i = 1 \ldots n$. Ahora bien, ninguna aproximación $e_i \to s_i$ con $i = 1 \ldots n$ corresponde a la conclusión de un paso FA, por lo que

$$\begin{array}{ll} apa(T') & = \text{árbol}(h\ \overline{e}_m \to h\ \overline{s}_m, \\ & \quad \text{subárboles}(apa(T'_{e_1 \to s_1})) +\!\!+ \ \ldots\ +\!\!+ \text{subárboles}(apa'(T'_{e_m \to s_m}))) \end{array}$$

por hipótesis de inducción se tiene $apa(T'_{e_i \to s_i})$ y $apa(T_{e_i \to t_i})$ sólo difieren en la raíz, y por tanto $\text{subárboles}(apa(T'_{e_i \to s_i})) = \text{subárboles}(apa(T_{e_i \to t_i}))$ para $i = 1 \ldots n$, es decir

$$\begin{array}{ll} apa(T') & = \text{árbol}(h\ \overline{e}_m \to h\ \overline{s}_m, \\ & \quad \text{subárboles}(apa(T_{e_1 \to t_1})) +\!\!+ \ \ldots\ +\!\!+ \text{subárboles}(apa'(T_{e_m \to t_m}))) \end{array}$$

y como ninguna aproximación $e_i \to t_i$ con $i = 1 \ldots n$ corresponde a la conclusión de un paso FA se tiene que $\text{subárboles}(apa(T_{e_i \to t_i})) = apa'((T_{e_i \to t_i})$ para $i = 1 \ldots n$ y

$$apa(T') = \text{árbol}(h\ \overline{e}_m \to h\ \overline{s}_m,\ apa'(T_{e_1 \to t_1}) +\!\!+ \ \ldots\ +\!\!+\ apa'(T_{e_m \to t_m}))$$

que coincide con $apa(T)$ salvo por la raíz.

- Regla $AR + FA$. El último paso de la prueba de $e \to t$ será de la forma:

$$\textbf{(3)}\quad \frac{e_1 \to t_1\ \ldots\ e_n \to t_n \quad \dfrac{C \quad r \to s'}{\boxed{f\ \overline{t}_n \to s'}} \quad s'\ \overline{a}_k \to t}{f\ \overline{e}_n\ \overline{a}_k \to t}$$

y por tanto $e \equiv f\ \overline{e}_n\ \overline{a}_k$. Entonces podemos probar $P \vdash_{SC} e \to s$ finalizando con otro paso $AR + FA$ de la forma:

$$\textbf{(4)}\quad \frac{e_1 \to t_1\ \ldots\ e_n \to t_n \quad \dfrac{C \quad r \to s'}{\boxed{f\ \overline{t}_n \to s'}} \quad s'\ \overline{a}_k \to s}{f\ \overline{e}_n\ \overline{a}_k \to s}$$

Para completar la demostración basta con observar que:

- Por (3) se tiene que $e_i \to t_i$ tendrá una prueba en SC, para todo $i = 1 \ldots n$. Igual sucederá tanto con $r \to t$ como con todas las condiciones atómicas de C.

- También de (3) se tiene que $P \vdash_{SC} s'\ \overline{a}_k \to t$ con un árbol de profundidad menor que p, y por hipótesis de inducción se verificará entonces $P \vdash_{SC} s'\ \overline{a}_k \to s$.

Además la profundidad de esta demostración será menor o igual a la de (3) al verificarse esta propiedad para cada una de las premisas. En cuanto a los APAs de (3) y (4) la única diferencia, aparte de la raíz, podría venir al calcular $apa'(T'_{(s'\ \overline{a}_k \to s)})$ para el APA de T' en lugar de $apa'(T_{(s'\ \overline{a}_k \to t)})$ para el de T, donde $T'_{(s'\ \overline{a}_k \to s)}$ representa el subárbol de prueba de $(s'\ \overline{a}_k \to s)$ en (4) y $T_{(s'\ \overline{a}_k \to t)}$ el subárbol de prueba de $(s'\ \overline{a}_k \to T)$ en (3). Pero como ni $s'\ \overline{a}_k \to s$ ni $s'\ \overline{a}_k \to t$ corresponden a conclusiones de pasos FA (la conclusión es aquí $f\ \overline{t}_n \to s'$), se verifica:

$$\begin{aligned} apa'(T'_{(s'\ \overline{a}_k \to s)}) \quad &= \text{subárboles}(apa(T'_{(s'\ \overline{a}_k \to s)})) = \text{ hip. ind.} = \\ &\text{subárboles}(apa(T_{(s'\ \overline{a}_k \to t)})) = apa'(T_{(s'\ \overline{a}_k \to t)}) \end{aligned}$$

lo que completa la demostración de este punto.

2. $P \vdash_{SC} e\ e' \to t$ sii existe un $s \in Pat_\perp$ tal que $P \vdash_{SC} (e\ s \to t, e' \to s)$. Además ambas pruebas admiten APAs que sólo difieren en la raíz.

   El caso $t \equiv \perp$ se verifica trivialmente en ambos sentidos, tomando $\perp$ como s para la implicación "$\Rightarrow$)". Además los APAs de las dos demostraciones sólo constan de la raíz por lo que se cumple el enunciado.

   Para $t \not\equiv \perp$ probamos ambos sentidos de la implicación por separado:

   $\Rightarrow$) Por inducción completa sobre la profundidad de la derivación de $P \vdash_{SC} e\ e' \to t$.

   Nos fijamos para ello en la regla utilizada en el último paso. Esta regla no puede ser BT ya que estamos suponiendo $t \not\equiv \perp$, ni JN al estar tratando una aproximación y no con una igualdad estricta, ni tampoco RR porque $(e\ e')$ no puede ser una variable. Sólo nos quedan las reglas DC y $AR + FA$. En ambos casos e debe ser de la forma $h\ \overline{e}_m$ con $h \in FS \cup DC$ y $e_i \in Exp_\perp$ para cada i, $1 \leq i \leq m$. Tratamos las dos posibilidades por separado.

   - Regla DC. En este caso tenemos que $h\ \overline{e}_m\ e'$ es una expresión pasiva y por tanto t es de la forma $h\ \overline{t}_{m+1}$. El último paso de la demostración en SC será entonces de la forma:

$$\textbf{(1)} \quad \frac{e_1 \to t_1\ \ldots\ e_m \to t_m\ \ e' \to t_{m+1}}{h\ \overline{e}_m\ e' \to h\ \overline{t}_{m+1}}$$

   Entonces el patrón s buscado es t_{m+1} ya que se cumple:

   - $P \vdash_{SC} e' \to s$ por (1).

- $P \vdash_{SC} e\ s \to t$ puede probarse con una demostración cuyo último paso corresponde a la aplicación de una regla DC de la forma:

$$\frac{e_1 \to t_1\ \ldots\ e_m \to t_m\ \ t_{m+1} \to t_{m+1}}{h\ \overline{e}_m\ t_{m+1} \to h\ \overline{t}_{m+1}}$$

  donde todas las premisas pueden probarse en SC por ser premisas de (1) excepto $t_{m+1} \to t_{m+1}$ que se cumple por el apartado 2 de la proposición 3.2.1 (pág. 57).

Sea T un árbol de prueba correspondiente a (1). Entonces, si utilizamos la notación $T_{(e \to t)}$ para referirnos al subárbol de prueba en T de cada premisa $e \to t$, podemos escribir:

$$\begin{aligned} apa(T) = \quad & \mathit{árbol}(h\ \overline{e}_m\ e' \to h\ \overline{t}_{m+1}, \\ & \quad apa'(T_{(e_1 \to t_1)})\ +\!\!+\ \ldots\ +\!\!+\ apa'(T_{(e_m \to t_m)})\ +\!\!+ \\ & \quad +\!\!+\ apa'(T_{(e' \to t_{m+1})})\ ) \end{aligned}$$

Por otra parte y por la misma definición el APA de $P \vdash_{SC} (e\ s \to t, e' \to s)$ será:

$$\begin{aligned} & \mathit{árbol}(h\ \overline{e}_m\ t_{m+1} \to h\ \overline{t}_{m+1}, e' \to t_{m+1}\ , \\ & \quad apa'(T_{(e_1 \to t_1)})\ +\!\!+\ \ldots\ +\!\!+\ apa'(T_{(e_m \to t_m)})\ +\!\!+ \\ & \quad +\!\!+\ apa'(T_{(t_{m+1} \to t_{m+1})})\ +\!\!+\ apa'(T_{(e' \to t_{m+1})})\ ) \end{aligned}$$

que coincide con $apa(T)$ salvo por la raíz al tenerse $apa'(T_{(t_{m+1} \to t_{m+1})}) = [\,]$ ya que en la prueba de $t_{m+1} \to t_{m+1}$ no se usará ningún paso $AR + FA$ por ser una aproximación entre patrones.

- Regla $AR + FA$.
  En este caso $h\ \overline{e}_m\ e'$ es una expresión activa, y por tanto $h \in FS^n$ con $n \leq m+1$.

  Si $n = m + 1$ el último paso será:

$$\textbf{(2)}\quad \frac{e_1 \to t_1\ \ldots\ e_m \to t_m\ \ e' \to t_n\ \ \boxed{\dfrac{C\quad r \to u}{h\ \overline{t}_n \to u}}\ \ u \to t}{h\ \overline{e}_m\ e' \to t}$$

  y podemos tomar t_n como el s que requiere el resultado. En efecto:
  - $P \vdash_{SC} e' \to s$ por (2).
  - $P \vdash_{SC} e\ s \to t$ puede probarse con una demostración cuyo último paso corresponde a la aplicación de una regla FA análoga a la de (2) pero sustituyendo la premisa $e' \to t_n$ por $s \to t_n$, y notando, como en el caso anterior, que al ser $s \equiv t_n$ esta aproximación siempre puede probarse según establece el apartado 2 de esta la proposición.

Si en cambio $n < m + 1$ el último paso de la demostración sera de la forma:

$$\textbf{(3)}\quad \frac{e_1 \to t_1 \ \ldots \ e_n \to t_n \quad \dfrac{C \quad r \to u}{\boxed{h\ \overline{t}_n \to u}} \quad u\ e_{n+1} \ldots e_m\ e' \to t}{h\ \overline{e}_m\ e' \to t}$$

Aplicamos la hipótesis de inducción a la premisa de (3) $u\ e_{n+1} \ldots e_m\ e' \to t$ cuya demostración en SC tiene una profundidad menor que la de $e\ e' \to t$, y se tiene que debe existir un s tal que

$$P \vdash_{SC} \quad e' \to s \qquad \textbf{(4)}$$

$$P \vdash_{SC} \quad u\ e_{n+1} \ldots e_m\ s \to t \qquad \textbf{(5)}$$

Este s es el indicado por el resultado, ya que cumple:

- $P \vdash_{SC} e' \to s$ por (4).
- $P \vdash_{SC} e\ s \to t$ puede probarse con una demostración cuyo último paso corresponde a la aplicación de una regla $AR + FA$ análoga a la de (3) pero sustituyendo la premisa $u\ e_{n+1} \ldots e_m\ e' \to t$ por $u\ e_{n+1} \ldots e_m\ s \to t$, que también tiene demostración en SC por (5).

En cualquiera de los dos casos se comprueba fácilmente que la relación entre los APAs es la deseada. En particular el único descendiente directo de la raíz es $h\ \overline{t}_n \to u$.

$\Leftarrow$) Vamos a hacer la demostración, utilizando inducción completa sobre la profundidad de la demostración de $P \vdash_{SC} e\ s \to t$. La demostración es similar al otro sentido de la implicación visto anteriormente. En particular la discusión acerca de la relación entre los APAs del apartado anterior es aplicable a este por lo que no la repetimos.

También en este caso por ser $t \neq \bot$ se tiene que e debe ser de la forma $h\ \overline{e}_m$ con $h \in FS \cup DC$ y $e_i \in Exp_\bot$ para cada i, $1 \leq i \leq m$, y el último paso de la demostración corresponderá a la aplicación de una regla DC o $AR + FA$.

- Regla DC. Al igual que sucedía en el caso correspondiente a la regla DC en el otro sentido de la aplicación, t ha de ser de la forma $h\ \overline{t}_{m+1}$ y el último paso:

$$\textbf{(6)}\quad \frac{e_1 \to t_1 \ \ldots \ e_m \to t_m \quad s \to t_{m+1}}{h\ \overline{e}_m\ s \to h\ \overline{t}_{m+1}}$$

En este caso se puede probar $e\ e' \to t$ con una demostración cuyo último paso corresponde igualmente a la aplicación de una regla DC:

$$\frac{e_1 \to t_1\ \ldots\ e_m \to t_m\ \ e' \to t_{m+1}}{h\ \overline{e}_m\ e' \to h\ \overline{t}_{m+1}}$$

Donde las premisas $e_i \to t_i$ tienen demostración en SC para todo i, $1 \leq i \leq m$ por (19), mientras que $e' \to t_{m+1}$ tiene una demostración en SC porque:

- Por (6) se tiene que $P \vdash_{SC} s \to t_{m+1}$. Como $s, t_{m+1} \in Pat_\perp$ se tiene por el apartado 1 de la proposición 3.2.1 que $s \sqsupseteq t_{m+1}$.
- Al tenerse $P \vdash_{SC} e' \to s$ por hipótesis, y $s \sqsupseteq t_{m+1}$ por el punto anterior, el apartado 1 de esta proposición asegura que $P \vdash_{SC} e' \to t_{m+1}$.

- Regla $AR + FA$. En este caso $h\ \overline{e}_m\ s$ es una expresión activa, es decir $h \in FS^n$ con $n \leq m + 1$. Distinguimos de nuevo los casos $n = m + 1$ y $n < m + 1$. Si $n = m + 1$ el último paso será de la forma:

$$\textbf{(7)}\quad \frac{e_1 \to t_1\ \ldots\ e_m \to t_m\ \ s \to t_n\ \ \dfrac{C \quad r \to u}{\boxed{h\ \overline{t}_n \to u}}\ \ u \to t}{h\ \overline{e}_m\ s \to t}$$

con $u \in Pat_\perp$ y $(h\ \overline{t}_n \to r \Leftarrow C) \in [P]_\perp$.
Entonces podemos probar $e\ e' \to t$ con una demostración cuyo último paso corresponde a una aplicación de la regla $AR + FA$ análoga:

$$\frac{e_1 \to t_1\ \ldots\ e_m \to t_m\ \ e' \to t_n\ \ \dfrac{C \quad r \to u}{\boxed{h\ \overline{t}_n \to u}}\ \ u \to t}{h\ \overline{e}_m\ e' \to t}$$

Las demostración de las premisas se tiene al ser también premisas de (7), salvo la de $e' \to t_n$, que se deduce de los apartados 1 y 4 de la proposición 3.2.1 por una razonamiento análogo al hecho en el caso de la regla DC.

Si en cambio $m > n$ el último paso tendrá el aspecto:

$$\textbf{(8)}\quad \frac{e_1 \to t_1\ \ldots\ e_n \to t_n\ \ \dfrac{C \quad r \to u}{\boxed{h\ \overline{t}_n \to u}}\ \ u\ e_{n+1} \ldots e_m\ s \to t}{h\ \overline{e}_m\ s \to t}$$

con $(h\ \overline{t}_n \to r \Leftarrow C) \in [P]_\perp$. Igual que en caso anterior podemos sustituir entonces s por e' y obtener una demostración para $e\ e' \to t$ cuyo último paso

corresponde a una aplicación de la regla $AR+FA$ con el mismo aspecto al de (8) pero reemplazando $u\ e_{n+1} \ldots e_m\ s \to t$ por $u\ e_{n+1} \ldots e_m\ e' \to t$ que se tiene a partir de la hipótesis de inducción, al tener de (8) que $P \vdash_{SC} u\ e_{n+1} \ldots e_m\ s \to t$ con una demostración de profundidad menor que la de $P \vdash_{SC} e\ s \to t$ y verificarse por hipótesis $P \vdash_{SC} e' \to s$.

3. $P \vdash_{SC} e\ e' \to t$ sii existe un $s \in Pat_\perp$ tal que $P \vdash_{SC} (e \to s,\ s\ e' \to t)$. Además ambas pruebas admiten APAs que sólo difieren en la raíz.

   Muy similar a la del punto anterior, por lo que la omitimos.

■

A.1.3. Demostración del Teorema 3.3.3

Para probar este teorema, enunciado en la página 66 necesitaremos algunos lemas auxiliares que enunciamos y probamos a continuación:

Lema A.1.1. *Dada una interpretación de Herbrand $\mathcal{I}$, se verifica para el cálculo $SC_\mathcal{I}$ un resultado análogo a 3.2.1 (pág. 57):*

1. *Para todo $t,\ s \in Pat_\perp$: $t \to s$ es válida en $\mathcal{I}$ sii $t \sqsupseteq s$.*
2. *Para todo $e \in Exp_\perp$, $t,\ s \in Pat_\perp$: si $e \to t$ es válida en $\mathcal{I}$ y $t \sqsupseteq s$, entonces $e \to s$ también es válida en $\mathcal{I}$.*
3. *Para todo $e \in Exp_\perp$, $t \in Pat_\perp$ y $\theta,\ \theta' \in Subst_\perp$ tales que $e\theta \to t$ es válida en $\mathcal{I}$ y $\theta \sqsubseteq \theta'$, la aproximación $e\theta' \to t$ también es válida en $\mathcal{I}$ con una demostración $SC_\mathcal{I}$ con la misma estructura y tamaño.*
4. *Para todo $e \in Exp_\perp$ y $s \in Pat_\perp$ tales que $e \to s$ es válida en $\mathcal{I}$, se verifica que la aproximación $e\theta \to s\theta$ también es válida en $\mathcal{I}$ para toda sustitución* total *$\theta \in Subst$.*

Idea de la demostración Este resultado puede probarse por inducción sobre la longitud de las derivaciones $SC_\mathcal{I}$, de forma similar a la demostración de los apartados análogos de la proposición 3.2.1 que puede encontrarse en la pág. 171. □

Lema A.1.2. *Sea $\mathcal{I}$ una interpretación de Herbrand, $e \in Exp_\perp$ una expresión parcial y $t \in Pat_\perp$ un patrón parcial. Entonces $[\![e]\!]^\mathcal{I} \supseteq [\![t]\!]^\mathcal{I}$ sii $t \in [\![e]\!]^\mathcal{I}$.*

Demostración.
Supongamos primero $[\![e]\!]^\mathcal{I} \supseteq [\![t]\!]^\mathcal{I}$. Por el lema A.1.1, $t \to t$ es válida en $\mathcal{I}$. Entonces $t \in [\![t]\!]^\mathcal{I}$ y por tanto $t \in [\![e]\!]^\mathcal{I}$.
Supongamos ahora $t \in [\![e]\!]^\mathcal{I}$. Entonces $e \to t$ es válida en $\mathcal{I}$, y para todo $s \in [\![t]\!]^\mathcal{I}$, se tiene que $t \to s$ también será válida en $\mathcal{I}$. Entonces del lema A.1.1 se sigue que $e \to s$ es válida en $\mathcal{I}$ para todo $s \in [\![t]\!]^\mathcal{I}$, lo que quiere decir que $[\![e]\!]^\mathcal{I} \supseteq [\![t]\!]^\mathcal{I}$. □

Lema A.1.3. *Sea $\mathcal{I}$ una interpretación de Herbrand y $f\ \bar{t}_n \to s$ un hecho básico. Entonces $f\ \bar{t}_n \to s$ es válido en $\mathcal{I}$ sii $(f\ \bar{t}_n \to s) \in \mathcal{I}$.*

Demostración.
Si $s = \bot$ el resultado es cierto al pertenecer $f\,\bar{t}_n \to \bot$ a toda interpretación de Herbrand y ser además válido en $\mathcal{I}$ gracias a la regla BT de $SC_{\mathcal{I}}$. Por tanto en el resto de la demostración supondremos $s \neq \bot$.
Si $(f\ \bar{t}_n \to s) \in \mathcal{I}$ entonces $f\ \bar{t}_n \to s$ es válido en $\mathcal{I}$, como prueba la siguiente derivación $SC_{\mathcal{I}}$, derivación que finaliza en una paso $FA_{\mathcal{I}}$ de la forma:

$$\frac{t_1 \to t_1\ \ldots\ t_n \to t_n \quad s \to s}{f\ \bar{t}_n \to s} \quad (f\ \bar{t}_n \to s) \in \mathcal{I}$$

La derivación puede completarse utilizando porque el lema A.1.1 asegura que todas las premisas $t_i \to t_i$ y $s \to s$ son válidas en $\mathcal{I}$.
Al contrario, si $f\ \bar{t}_n \to s$ es válido en $\mathcal{I}$, entonces debe existir una prueba en $SC_{\mathcal{I}}$ para $f\ \bar{t}_n \to s$, que debe finalizar en un paso $FA_{\mathcal{I}}$ de la siguiente forma:

$$\frac{t_1 \to t'_1\ \ldots\ t_n \to t'_n \quad s' \to s}{f\ \bar{t}_n \to s} \quad (f\ \overline{t'}_n \to s') \in \mathcal{I}$$

Y por el lema A.1.1 podemos concluir que $t'_1 \sqsubseteq t_1,\ \ldots,\ t'_n \sqsubseteq t_n$ y $s \sqsubseteq s'$. Como además $(f\ \overline{t'}_n \to s') \in \mathcal{I}$, del ítem 2 de la definición de interpretación de Herbrand (def. 3.3.1, pág. 64), se sigue que $(f\ \bar{t}_n \to s) \in \mathcal{I}$. □

Lema A.1.4. *Sea P un programa y $\mathcal{M}_P = \{f\ \bar{t}_n \to s \mid P \vdash_{SC} f\ \bar{t}_n \to s\}$. Entonces $\mathcal{M}_P$ es una interpretación de Herbrand.*

Demostración M_P debe satisfacer las tres condiciones de las interpretaciones de Herbrand (enunciadas en la def. 3.3.1, pág. 64):

1. $(f\,\bar{t}_n \to \bot) \in \mathcal{M}_P$.
   Esta propiedad se cumple ya que $f\,\bar{t}_n \to \bot$ puede probarse en SC mediante la regla BT.

2. Si $(f\ \bar{t}_n \to s) \in \mathcal{M}_P$, $t_i \sqsubseteq t'_i, s \sqsupseteq s'$ entonces $(f\,\overline{t'}_n \to s') \in \mathcal{M}_P$.
   Esto se sigue inmediatamente de la definición de $\mathcal{M}_P$ y de la proposición 3.2.1.

3. Si $(f\ \bar{t}_n \to s) \in \mathcal{M}_P$ y $\theta \in Subst$ es una *sustitución total*, entonces $(f\ \bar{t}_n \to s)\theta \in \mathcal{M}_P$. Esto es, de nuevo, una consecuencia directa de la proposición 3.2.1 y de la construcción de $\mathcal{M}_P$.

□

Una vez probados estos lemas podemos pasar a la demostración del teorema. Primero probaremos el apartado (a), después el (c) y finalmente el (b).

(a) Sea G un objetivo tal que $P \vdash_{SC} G$ y sea $\mathcal{I}$ un modelo de Herbrand para P. Sea $\varphi \in G$ una aproximación o una igualdad estricta. Sea T un árbol de prueba para φ en SC (que debe existir al poder probarse G en SC). Vamos a construir un árbol de prueba T' para φ en $SC_{\mathcal{I}}$, demostrando de esta forma que G es válido en $\mathcal{I}$ por la definición 3.3.2, pág. 65. La prueba de que tal T' puede construirse la hacemos por inducción sobre la profundidad de T.

Caso Base. Profundidad(T)= 0.
Entonces φ es el único nodo de T y se corresponde con la aplicación de una regla BT, DC o RR. Como estas reglas también forman parte de $SC_{\mathcal{I}}$ basta con tomar $T' = T$.

Caso Inductivo. Profundidad(T)= p, $p > 0$.
Distinguimos distintos casos según la regla SC aplicada en la raíz de T:

Regla DC:

En este caso φ debe ser de la forma $h\ \overline{e}_m \rightarrow h\ \overline{t}_m$ y el paso DC aplicado en la raíz de T debe ser de la forma:

$$\frac{e_1 \rightarrow t_1\ \ldots\ e_m \rightarrow t_m}{h\ \overline{e}_m \rightarrow h\ \overline{t}_m}$$

Ahora bien, como la regla DC existe también en $SC_{\mathcal{I}}$, podemos construir T' con la misma raíz que T y con los mismos nodos hijos para dicha raíz. Además, por la hipótesis de inducción se tiene que todos los $e_i \rightarrow t_i$ son válidos $\mathcal{I}$, por lo que tendrán árboles de prueba en $SC_{\mathcal{I}}$ que nos permitirán completar la construcción de T'.

Regla JN:

φ es entonces de la forma $e == e'$ y el paso de inferencia asociado a la raíz de T debe ser:

$$\frac{e \rightarrow t \qquad e' \rightarrow t}{e\ ==\ e'}$$

con t un patrón total. Como la regla JN existe también en $SC_{\mathcal{I}}$ podemos construir T', de manera análoga al caso anterior, partiendo de la misma raíz y de los mismos nodos hijos que en T. Además, por la hipótesis de inducción se tendrá que tanto $e \rightarrow t$ como $e' \rightarrow t$ son válidas en $\mathcal{I}$ y por consiguiente tienen sus correspondientes árboles de prueba en $SC_{\mathcal{I}}$ que podemos utilizar para completar la demostración de T'.

Regla AR + FA:

En este caso $\varphi \equiv f\ \overline{e}_n\ \overline{a}_k \rightarrow t$ y la regla SC aplicada en la raíz de T debe ser:

$$\frac{e_1 \rightarrow t_1\ \ldots\ e_n \rightarrow t_n \qquad \dfrac{C \quad r \rightarrow s}{\boxed{f\ \overline{t}_n \rightarrow s}} \qquad s\ \overline{a}_k \rightarrow t}{f\ \overline{e}_n\ \overline{a}_k \rightarrow t}$$

Entonces construiremos T' usando en la raíz una inferencia $FA_{\mathcal{I}}$ de la forma:

$$\frac{e_1 \to t_1 \ \ldots \ e_n \to t_n \quad s\ \overline{a}_k \to t}{f\ \overline{e}_n\, \overline{a}_k \to t} \quad (f\ \overline{t}_n \to s) \in \mathcal{I}$$

y completando la demostración mediante árboles de prueba en $SC_{\mathcal{I}}$ para las aproximaciones $e_i \to t_i$ y $s\ \overline{a}_k \to t$, las cuales deben existir por hipótesis de inducción al tener todas estas aproximaciones árboles de prueba en SC con una profundidad menor a p. Pero también debemos comprobar que se verifican las condiciones requeridas por $FA_{\mathcal{I}}$. En primer lugar, t es un patrón diferente de $\perp$ al ser esta condición requerida por la regla $AR + FA$ de SC. Para comprobar que $f\ \overline{t}_n \to s$ pertenece a $\mathcal{I}$ observamos que $f\ t_1 \ldots t_n \to t \Leftarrow C$ debe ser la instancia de alguna regla de programa de P. Además, $\mathcal{I}$ satisface C por hipótesis de inducción. Por tanto, al ser $\mathcal{I}$ un modelo de P, podemos concluir que $[\![f\ t_1 \ldots t_n]\!]^{\mathcal{I}} \supseteq [\![r]\!]^{\mathcal{I}}$. También por hipótesis de inducción sabemos que $r \to s$ es válida en $\mathcal{I}$. De aquí se sigue que $s \in [\![r]\!]^{\mathcal{I}} \subseteq [\![ft_1 \ldots t_n]\!]^{\mathcal{I}}$ y por tanto $s \in [\![ft_1 \ldots t_n]\!]^{\mathcal{I}}$, tal y como necesitamos.

(c) $\mathcal{M}_P$ es una interpretación de Herbrand como muestra el lema A.1.4. Supongamos que para todo $\varphi \in G$ se cumple que φ es válido en $\mathcal{M}_P$ con un árbol de prueba T en $SC_{\mathcal{M}_P}$. Vamos a probar por inducción sobre la profundidad de T que podemos construir un árbol T' para φ en SC, con lo que habremos probado $P \vdash_{SC} G$.

Caso Base. Profundidad(T)=0.
Las únicas inferencias posibles que se pueden aplicar en la raíz de T serán BT, RR or DC. Como las 3 reglas pertenecen también a SC podemos tomar $T' = T$.

Caso Inductivo. Profundidad(T) = p, $p > 0$.
Si la profundidad es mayor que 0, entonces se ha aplicado DC, JN o $FA_{\mathcal{M}_P}$ en la raíz de T. En los casos correspondientes a las reglas DC y JN, se puede aplicar la misma inferencia a la raíz de T', teniéndose, por hipótesis de inducción, árboles de prueba en SC para todos los hijos T_i, lo que permite completar el árbol de prueba T'.

En el caso de la regla $FA_{\mathcal{M}_P}$, la inferencia en la raíz de T debe tener la forma:

$$\frac{e_1 \to t_1 \ \ldots \ e_n \to t_n \quad s\ \overline{a}_k \to t}{f\ \overline{e}_n\, \overline{a}_k \to t} \quad t \text{ patrón},\ t \neq \perp,\ (f\ \overline{t}_n \to s) \in \mathcal{M}_P$$

(esta es la forma abreviada de la regla AR+FA que indicamos en la sección 3.2 Como $(f\ \overline{t}_n \to s) \in \mathcal{M}_P$ debe existir un árbol de prueba en SC para $f\ \overline{t}_n \to s$. Este árbol tendrá una inferencia $AR + FA$ en la raíz, de la forma:

$$\frac{t_1 \to t'_1 \ \ldots \ t_n \to t'_n \quad \dfrac{C \quad r \to s}{\boxed{f\ \overline{t'}_n \to s}}}{f\ \overline{t}_n \to s} \quad (f\ \overline{t'}_n \to r \Leftarrow C) \in [P]_{\perp},\ s \text{ patrón},\ s \neq \perp$$

Por tanto las aproximaciones $t_i \to t'_i$, C y $r \to s$ han de tener sus correspondientes árboles de prueba en SC. Entonces podemos construir T' comenzando con una inferencia FA de la siguiente forma:

$$\frac{e_1 \to t'_1 \ \ldots \ e_n \to t'_n \qquad \dfrac{C \quad r \to s}{\boxed{f\ \overline{t'}_n \to s}} \qquad s\ \overline{a}_k \to t}{f\ \overline{e}_n\ \overline{a}_k \to t} \qquad \begin{array}{l} s \text{ pattern} \\ f\ \overline{t'}_n \to r \Leftarrow C \in [P]_\perp, \\ t \text{ patrón}, \ \ t \neq \perp \end{array}$$

Para completar T' necesitamos aún poder asegurar que las aproximaciones $e_i \to t'_i$ tienen árboles de prueba en SC. Pero esto se obtiene fácilmente a partir de la proposición 3.2.1, ya que cada $e_i \to t_i$ pueden probarse en SC por hipótesis de inducción, y hemos visto que también las aproximaciones $t_i \to t'_i$ pueden probarse en SC.

(b). A partir del lema A.1.3, la construcción de $\mathcal{M}_P$ y el ítem (a) de este mismo teorema, se deduce fácilmente que $\mathcal{M}_P$ está incluido en cualquier modelo de Herbrand de P. Además, sabemos por el lema A.1.4 que $\mathcal{M}_P$ es una interpretación de Herbrand. Para ver que $\mathcal{M}_P$ es un modelo de P, debemos probar que $\mathcal{M}_P$ satisface toda instancia de regla de programa. Para ver esto consideramos una instancia cualquiera $(f\ \bar{t}_n \to r \Leftarrow C) \in [P]_\perp$. En primer lugar $\mathcal{M}_P$ satisface la instancia de manera trivial si no satisface la condición C (ver la definición 3.3.4, pág. 65). Si por el contrario $\mathcal{M}_P$ satisface C, tendremos que probar que $[\![r]\!]^{\mathcal{M}_P} \subseteq [\![f\ \bar{t}_n]\!]^{\mathcal{M}_P}$. Esto significa que cualquier $t \in Pat_\perp$ tal que $r \to t$ es válida $\mathcal{M}_P$ debe verificar que $f\ \bar{t}_n \to t$ también debe ser válida en $\mathcal{M}_P$. Por el apartado (c) de este mismo teorema y por la construcción de $\mathcal{M}_P$, basta con probar que $P \vdash_{SC} f\ \bar{t}_n \to t$ bajo el supuesto $P \vdash_{SC} r \to t$. Si $t = \perp$ esto es cierto trivialmente. En otro caso podemos construir el siguiente árbol de prueba SC, comenzando con una inferencia $AR + FA$ de la forma:

$$\frac{t_1 \to t_1 \ \ldots \ t_n \to t_n \qquad \dfrac{C \quad r \to t}{\boxed{f\ \bar{t}_n \to t}}}{f\ \bar{t}_n \to t} \qquad f\ \bar{t}_n \to r \Leftarrow C \in [P]_\perp,\ t \neq \perp$$

Este árbol puede ser completado porque:

- Se cumple $P \vdash_{SC} t_i \to t_i$ por la proposición 3.2.1.
- Del apartado (c) del teorema se deduce que toda aproximación φ' de C es válida en $\mathcal{M}_P$, y por tanto verifica $P \vdash_{SC} \varphi'$.
- $P \vdash_{SC} r \to t$ por hipótesis.

■

A.1.4. Demostración de la Proposición 3.3.5

Comprobamos este resultado, cuyo enunciado se puede encontrar en la página 67.

Sea T el árbol de prueba del que se ha obtenido *apa*, r la raíz de ambos árboles (que coincide), P el programa que se ha utilizado para la prueba de la que T es testigo y $\mathcal{I}$ la interpretación pretendida de P. Si r es correcta (i.e. válida en $\mathcal{I}$), el resultado se cumple trivialmente; por definición todo nodo crítico debe ser erróneo. Nos fijamos por tanto en el caso en que r es incorrecta. Vamos a probar un resultado auxiliar:

Sea T un AP con raíz r incorrecta. Entonces r tiene en T un descendiente n tal que

1. *n es conclusión de una regla FA.*
2. *n es incorrecto.*
3. *No existe ningún nodo p conclusión de una regla FA tal que p sea a la vez antecesor de n y descendiente de r.*

Del resultado se deduce que el nodo incorrecto n será hijo de r en el APA (se tiene $r \neq n$ porque r no puede ser conclusión de una regla FA), y por tanto r no será crítico.

Sin perdida de generalidad suponemos que r es un objetivo atómico. En otro caso al ser incorrecto debe contener una componente atómica φ incorrecta cuyos descendientes son descendientes de r, por lo que seleccionaríamos el árbol de prueba de φ como T y el propio φ como r.

Hacemos la prueba por inducción completa sobre el tamaño de T, distinguiendo casos sobre la regla SC aplicada en la raíz del árbol. Nótese que r no puede ser de la forma $X \to X$ con $X \in Var$, ni $c \to c$ con $c \in DC^0$ ni $e \to \bot$, porque en cualquiera de estos casos la raíz r sería correcta. Nos quedan por tanto tres posibilidades:

- Regla DC con $r \equiv h\ e_1 \dots e_n \to h\ t_1 \dots t_n$ y $n > 0$. El paso asociado a la raíz es entonces de la forma:

$$\frac{e_1 \to t_1\ \dots\ e_n \to t_n}{h\ \overline{e}_n \to h\ \overline{t}_n}$$

  Ahora bien, alguno de los $e_i \to t_i$ con $1 \leq i \leq n$ debe ser erróneo. Si todos fueran válidos, por la definición 3.3.2 (pág. 65) podrían probarse en el cálculo $SC_{\mathcal{I}}$, y aplicando un paso DC como este la raíz r también podría probarse en $SC_{\mathcal{I}}$, por lo que sería válida en contra de lo que estamos suponiendo.

  Sea $e_j \to t_j$ erróneo para algún $1 \leq j \leq n$. El resultado entonces se tiene por hipótesis de inducción. En particular obsérvese que $e_j \to t_j$ no corresponde a la conclusión de un paso FA y por tanto la propiedad c) se mantiene al pasar a r.

- Regla JN, con $r \equiv e == e'$, con $e, e' \in Exp_{\bot}$ y el paso asociado a la raíz de la forma:

$$\frac{e \to t \quad e' \to t}{e == e'}$$

para algún $t \in Pat$. Análogo al caso anterior: se tiene que alguna de las dos premisas, $e \to t$ o $e' \to t$ debe ser errónea, ya que de otra forma por la definición 3.3.2 (pág. 65) podríamos aplicar un paso JN como el anterior pero en el cálculo $SC_{\mathcal{I}}$, y llegar a que r es válida en el modelo pretendido, en contra de lo que estamos suponiendo. Sea $e \to t$ errónea (análogo para $e' \to t$). Entonces se tiene el resultado por hipótesis de inducción, notando que $e \to t$ no corresponde a la conclusión de un paso FA y por tanto la propiedad c) se mantiene al pasar a r.

- Regla $AR + FA$, con $r \equiv f\ \overline{e}_n\ \overline{a}_k \to t$, $t \neq \bot$ y un paso asociado a la raíz de la forma:

$$\frac{e_1 \to t_1\ \ldots\ e_n \to t_n \qquad \dfrac{C \quad r \to s}{\boxed{f\ \overline{t}_n \to s}} \qquad s\ \overline{a}_k \to t}{f\ \overline{e}_n\ \overline{a}_k \to t}$$

Como en el caso anterior o bien alguna de las premisas $e_i \to t_i$ para $1 \leq i \leq n$, o bien $f\ \overline{t}_n \to s$, o $s\ \overline{a}_k \to t$ debe ser incorrecta en $\mathcal{I}$ porque si no la raíz podría probarse en $SC_{\mathcal{I}}$ acabando con una aplicación de la regla $FA_{\mathcal{I}}$:

$$\textsf{FA}_{\mathcal{I}} \qquad \frac{e_1 \to t_1\ \ldots\ e_n \to t_n \quad s\ \overline{a}_k \to t}{f\ \overline{e}_n\, \overline{a}_k \to t} \qquad \begin{array}{l} t \text{ patrón}, t \neq \bot, s \text{ patrón} \\ (f\ \overline{t}_n\ \to\ s) \in \mathcal{I} \end{array}$$

Si la premisa errónea es $e_j \to t_j$ para algún $1 \leq j \leq n$, o $s\ \overline{a}_k \to t$, el resultado se tiene como en el apartado anterior por hipótesis de inducción. Si la premisa errónea es $f\ \overline{t}_n \to s$ el resultado se obtiene tomando $n \equiv f\ \overline{t}_n \to s$, que es un nodo incorrecto conclusión de una regla FA, además de verificar trivialmente el apartado c) del enunciado que estamos demostrando.

■

A.2. Resultados Presentados en el Capítulo 4

A.2.1. Demostración de la Proposición 4.3.1

Para probar este resultado, enunciado en la página 78, vamos a establecer primero un resultado auxiliar que nos será igualmente de utilidad en la prueba de otros resultados de la misma sección.

Lema A.2.1. *Sean e, e' expresiones y C condición tales que $e \Rightarrow_{\mathcal{A}} (e'; C)$. Entonces:*

(a) e' es un patrón.

(b) C es de la forma $e_1 \to X_1, \ldots\ e_n \to X_n$, con $X_i \in Var$ y e_i expresión plana para todo $i = 1 \ldots n$. Además, las X_i se corresponden con las variables nuevas introducidas mediante las reglas (PL_4) y (PL_6) y cumplen $X_i \neq X_j$ para $i \neq j$ y $X_i \notin var(e_j)$ para $j \leq i$.

(c) Para cada $e_i \to X_i \in C$, $var(e_i) \subseteq var(e) \cup \{X_1, \ldots, X_{i-1}\}$.

(d) $var(e') \subseteq var(e) \cup \{X_1, \ldots, X_n\}$.

Demostración
Demostramos los cuatro resultados simultáneamente por inducción completa sobre la estructura de la demostración de $e \Rightarrow_{\mathcal{A}} (e'; C)$, razonando que si todas las premisas de una regla verifican la propiedad la conclusión también lo hace. Nótese que en la derivación $e \Rightarrow_{\mathcal{A}} (e'; C)$ sólo pueden haberse utilizado las reglas (PL_2)-(PL_6) de la figura 4.1 (pág. 77). Distinguimos casos según la regla aplicada en último lugar:

($\mathbf{PL}_2$)-($\mathbf{PL}_3$). Obvio; tanto $\perp$ como X son patrones (ítem (a)) y la condición vacía siempre verifica trivialmente los ítems (b) y (c). Además en este caso $var(e') = var(e)$ (ítem (d)).

($\mathbf{PL}_4$). En este caso u es un patrón por hipótesis de inducción (ítem (a)).

Para el ítem (b) observamos en primer lugar que C_1, C provienen del aplanamiento de expresiones y por tanto verifican la hipótesis de inducción. En cuanto a la aproximación $X\ u_1 \to R$, R es una variable nueva (que no puede aparece en C_1 ni en $X\ u_1$). Además, al ser u_1 un patrón por el ítem (a) se tiene que $X\ u_1$ es una expresión plana según la definición 4.3.1 (pág. 75), con lo que también verifica el resultado. Finalmente resulta inmediato comprobar que la conjunción de condiciones verificando (b) también verifica este apartado (lo mismo con el apartado (c)).

Para los apartados (c) y (d) vamos a suponer que C_1 es de la forma $C_1 \equiv e_1 \to X_1,\ \ldots,\ e_m \to X_m$, y $C \equiv e_{m+2} \to X_{m+2},\ \ldots,\ e_n \to X_n$. Llamando X_{m+1} a R y e_{m+1} a $X\ u_1$ podemos escribir la condición $(C_1,\ X\ u_1 \to R, C)$ como $(e_1 \to X_1,\ \ldots,\ e_n \to X_n)$.

Veamos primero el apartado (c). Distinguimos tres posibilidades:

- $e_i \to X_i \in C_1$. En este caso por h.i. $var(e_i) \subseteq var(a_1) \cup \{X_1, \ldots, X_{i-1}\}$, donde $\{X_1, \ldots, X_{i-1}\}$ es el conjunto formado por los lados derechos de las $i-1$ primeras aproximaciones en C_1 que son también las primeras $i-1$ aproximaciones en $(C_1,\ X\ u_1 \to R, C)$. Al ser $var(a_1) \subseteq var(X\ \overline{a}_k)$ se tiene el resultado deseado $var(e_i) \subseteq var(X\ \overline{a}_k) \cup \{X_1, \ldots, X_{i-1}\}$.
- $e_i \to X_i \equiv X\ u_1 \to R$, es decir $i = m+1$. Por la h.i., apartado (d) del lema, se cumple $var(u_1) \subseteq var(a_1) \cup \{X_1, \ldots X_m\}$. Por tanto $var(X\ u_1) \subseteq var(X\ \overline{a}_k) \cup \{X_1, \ldots X_m\}$.
- $e_i \to X_i \in C$. La h.i. es entonces

$$var(e_i) \subseteq var(R\ a_2 \ldots a_k) \cup \{X_{m+2}, \ldots, X_{i-1}\}$$

lo que, recordando que $R \equiv X_{m+1}$, se puede reescribir como

$$var(e_i) \subseteq var((a_2, \ldots, a_k)) \cup \{X_{m+1}, \ldots, X_i\}$$

de donde se llega al resultado deseado $var(e_i) \subseteq var(X\ \overline{a}_k) \cup \{X_1 \ldots X_{i-1}\}$, al ser $var((a_2, \ldots, a_k)) \subseteq var(X\ \overline{a}_k)$ y $\{X_{m+1}, \ldots, X_i\} \subseteq \{X_1, \ldots, X_i\}$.

Para el ítem (d) basta con observar que, por hipótesis de inducción

$$var(u) \subseteq var(R\ a_2 \ldots a_k) \cup \{X_{m+2}, \ldots, \ldots X_n\}$$

que podemos escribir también como

$$var(u) \subseteq var((a_2, \ldots, a_k)) \cup \{X_{m+1}, \ldots, \ldots X_n\}$$

al ser $R \equiv X_{m+1}$. De nuevo recordamos que se cumplen $var((a_2, \ldots, a_k)) \subseteq var(X\ \overline{a}_k)$ y $\{X_{m+1}, \ldots, \ldots X_n\} \subseteq \{X_1, \ldots, \ldots X_n\}$, de donde se deduce el resultado deseado:

$$var(u) \subseteq var(X\ \overline{a}_k) \cup \{X_1, \ldots, \ldots X_n\}$$

($\mathbf{PL}_5$). Para el ítem (a): Por hipótesis $h\ \overline{e}_m$ es pasivo y por tanto $h\ \overline{u}_m$ será un patrón, al ser $u_1 \ldots u_m$ patrones por hipótesis de inducción. Los apartados (b), (c) y (d) se tienen a partir de la hipótesis de inducción, teniendo en cuenta las mismas observaciones del apartado anterior.

($\mathbf{PL}_6$). El ítem (a) se tiene por hipótesis de inducción. Para el (b) tenemos que en $f\ \overline{u}_n \rightarrow S$ la variable S es nueva (i.e. no aparece en $C_1 \ldots C_n$ ni en $f\ \overline{u}_n$) y los u_i son patrones por el ítem (a), de donde $f\ \overline{u}_n$ es una expresión plana. Por otra parte $C_1, \ldots C_n, D$ verifican el resultado por hipótesis de inducción como en los casos anteriores. Los apartados (c) y (d) se siguen por un razonamiento análogo al usado en la regla (PL_4).

□

Ahora resulta sencillo probar la proposición:

(a) Observando la regla (PL_1) se tiene que $r_{\mathcal{A}}$ proviene del aplanamiento de r por lo que, a partir del apartado (a) del lema anterior, es un patrón.

(b) El resultado para las igualdades estrictas se obtiene comprobando que las únicas condiciones de este tipo en el programa transformado son de la forma $u_1 == u_2$, donde u_1, u_2 provienen del aplanamiento de expresiones l y r (regla (PL_8)), y del resultado (a) del lema anterior se tiene que u_1, u_2 son necesariamente patrones.

De la regla (PL_1) se deduce que las aproximaciones en la condición de la regla plana pueden venir bien del aplanamiento del lado derecho de la regla, en cuyo caso el resultado queda garantizado por el apartado (b) del lema anterior, o bien del aplanamiento de las condiciones. Probamos por inducción completa sobre la profundidad de la derivación de $C \Rightarrow_{\mathcal{A}} C_{\mathcal{A}}$ en (PL_1) que las aproximaciones contenidas en $C_{\mathcal{A}}$ son de la forma indicada. Para ello nos fijamos en la primera regla utilizada en la demostración, que puede ser:

($\mathbf{PL}_7$). Se cumple el resultado directamente por hipótesis de inducción (es fácil ver que si dos o más condiciones verifican el resultado su conjunción también lo hace).

($\mathbf{PL}_8$). Por el apartado (b) del lema A.2.1 todas las aproximaciones en C y en D son de la forma $e \rightarrow X$, con e plana y X variable nueva introducida por (PL_4) o (PL_6).

($\mathbf{PL}_9$). En C se verifica el resultado por el apartado b) lema A.2.1, como en el apartado anterior. En el caso de la aproximación $u \rightarrow s$ se tiene que s proviene de una aproximación del conjunto de condiciones del programa original y que u es un patrón por el apartado (a) del mismo lema, lo que coincide con el segundo tipo de posibles aproximaciones descritas en la proposición.

($\mathbf{PL}_{10}$). La única aproximación que compone la condición aplanada corresponde en esta regla a una aproximación del programa original con lado izquierdo plano, lo que coincide con el tercero de los posibles tipos de aproximaciones descritos en la proposición.

■

A.2.2. Demostración de la Proposición 4.3.2

Este resultado, enunciado en la página 79, puede deducirse fácilmente a partir de los resultados auxiliares que presentamos a continuación.

Lema A.2.2. *Sea $e, e' \in Exp_\perp$ y C una condición tales que $e \Rightarrow_{\mathcal{A}} (e' \,;\, C)$. Entonces:*

a) Si $e \in Pat_\perp$ entonces $e' \equiv e$ y C es la condición vacía.

b) Si $e \notin Pat_\perp$ entonces $e' \neq e$ y C no es la condición vacía.

Demostración
Vamos aprobar el resultado utilizando inducción estructural sobre e.

- $e \equiv \perp$. Para el apartado a) observamos que $\perp \Rightarrow_{\mathcal{A}} (\perp \,;\,)$ ya que sólo se puede aplicar la regla (PL_2). El apartado b) se cumple trivialmente al no cumplirse su premisa porque $\perp \in Pat_\perp$.

- $e \in Var$. Análogo al anterior pero utilizando la regla (PL_3).

- e pasiva de la forma $e \equiv h\ \overline{e}_m$, donde o bien $h \in DC^n$ con $n \geq m$ o $h \in FS^n$ con $n > m$. El aplanamiento del e debe concluir necesariamente con una regla (PL_5) de la forma:

$$\frac{e_1 \Rightarrow_{\mathcal{A}} (u_1 \,;\, C_1) \quad \ldots \quad e_m \Rightarrow_{\mathcal{A}} (u_m \,;\, C_m)}{h\ \overline{e}_m \ \Rightarrow_{\mathcal{A}}\ (h\ \overline{u}_m \,;\, C_1, \ldots, C_m)}$$

a) Suponemos que $e \in Pat_\perp$, es decir que $e_i \in Pat_\perp$ para $i = 1 \ldots m$. Por hipótesis de inducción $e_i \Rightarrow_{\mathcal{A}} (e_i\ ;\ )$ para $i = 1 \ldots m$, de donde $u_i \equiv e_i$ y todos los C_i representan la condición vacía. Por tanto el resultado del aplanamiento $(h\ \overline{u}_m\ ; C_1, \ldots, C_m)$ se puede reescribir como $(h\ \overline{e}_m\ ;\ )$, que es resultado deseado.

b) A la inversa, si algún $e_i \notin Pat_\perp$ entonces por hipótesis de inducción $u_i \neq e_i$ y C_i no es vacía, por lo que $(C_1, \ldots, C_m)$ no es vacía y $h\ \overline{e}_m \neq h\ \overline{u}_m$.

- e activa o flexible. En este caso a) se cumple trivialmente al no ser e un patrón. Para b) observamos en primer lugar que $e' \neq e$ ya que por el lema A.2.1 (pág. 191) e' es un patrón y sabemos que en este caso e no lo es.

  Veamos que además C no es vacía. Si e es flexible el último paso de la demostración de aplanamiento corresponderá necesariamente a un paso (PL_4) que introduce al menos una aproximación en C, mientras que si e es activa habrá que emplear la regla (PL_6) que también incluye al menos una aproximación en C.

□

Lema A.2.3. *Sea $C \equiv (c_1, \ldots, c_m)$ una condición compuesta por m condiciones atómicas, $m \geq 1$. Sea C' una condición tal que $C \Rightarrow_{\mathcal{A}} C'$. Entonces $C' \equiv (C'_1, \ldots, C'_m)$ donde para $i = 1 \ldots m$ se cumple que C'_i es una condición (no necesariamente atómica) tal que $c_i \Rightarrow_{\mathcal{A}} C'_i$.*

Demostración
Por inducción sobre m:
Caso Base $m = 1$. Obvio al ser C en este caso una condición atómica.

Caso Inductivo $m > 1$. Entonces $C \equiv (c_1, \ldots, c_m)$. Consideramos la condición $B \equiv (c_1, \ldots, c_{m-1})$. De la hipótesis de inducción se deduce que se verifica $B \Rightarrow_{\mathcal{A}} B'$, donde $B' \equiv (C'_1, \ldots, C'_{m-1})$ con $c_i \Rightarrow_{\mathcal{A}} C'_i$ para cada $i = 1 \ldots m-1$. Debido a que la conjunción (representada aquí por el símbolo ',') es asociativa basta con escribir C como $C \equiv (B, c_m)$ y aplicar la regla (PL_7) (con B como C_1 y c_m como C_2) para tener el resultado deseado.

□

Lema A.2.4. *Sean C dos condiciones. Entonces $C \Rightarrow_{\mathcal{A}} C$ sii C es una condición plana.*

Demostración
$\Rightarrow$) Por reducción al absurdo, probamos que si C no es plana no se tiene $C \Rightarrow_{\mathcal{A}} C$. Si C no es plana es que ocurre una de las siguientes situaciones:

- Existe $e \to s \in C$ tal que e no es una expresión plana. Aplicando reiteradamente la regla (PL_7) llegaremos al aplanamiento de $e \to s$ para lo que se aplicará entonces la regla (PL_9), que indica que si $e \Rightarrow_{\mathcal{A}} (u\,;\, C')$ para cierta expresión u y condición C', en la condición aplanada aparecerán todas las condiciones atómicas de C', además de la aproximación $u \to s$. Por el lema anterior (lema A.2.2) se verifica que C' no es la condición vacía ya que al no ser e plana no puede en particular ser un patrón, y por el lema 4.3.1, las aproximaciones en C' tienen en su lado derecho variables nuevas, por lo que no podían formar parte de C, lo que prueba el resultado.

- Existe $e == e' \in C$ donde bien e o e' no son patrones. Entonces se aplicará la regla (PL_8), que por un razonamiento análogo al del caso anterior introducirá en la condición aplanada una o más aproximaciones nuevas y por tanto no se verificará $C \Rightarrow_{\mathcal{A}} C$.

$\Leftarrow$) Por el lema A.2.3 se tiene que bastará con probar que para toda condición atómica $c_i \in C$ se cumple $c_i \Rightarrow_{\mathcal{A}} c_i$. De aquí que es suficiente probar los dos siguientes apartados:

- Para toda $e \to t \in C$ se cumple $(e \to t) \Rightarrow_{\mathcal{A}} (e \to t)$. Esto es cierto porque al estar suponiendo que C es plana e debe serlo y se aplicará la regla (PL_{10}).
- Para toda $e == e' \in C$ se cumple $(e == e') \Rightarrow_{\mathcal{A}} (e == e')$. Para aplanar igualdades estrictas se utiliza la regla (PL_8), que transforma $(e == e')$ en C_1, C_2, $u_1 == u_2$, donde $e \Rightarrow_{\mathcal{A}} (u_1 \,;\, C_1)$ y $e' \Rightarrow_{\mathcal{A}} (u_2 \,;\, C_2)$. Pero al ser C plana $e, e' \in Pat_\perp$ y por el apartado a) del lema A.2.2, se cumple que C_1 y C_2 son condiciones vacías y $u_1 \equiv e$, $u_2 \equiv e'$, por lo que C_1, C_2, $u_1 == u_2$ es simplemente $e == e'$ como queríamos demostrar.

□

Ahora resulta sencillo probar las dos implicaciones de la proposición:
$\Rightarrow$) Si P es plano entonces $P \Rightarrow_{\mathcal{A}} P$. Para esto basta con ver que para toda regla de programa $(f\ \bar{t}_n \to r \Leftarrow C) \in P$ se cumple

$$(f\ \bar{t}_n \to r \Leftarrow C) \Rightarrow_{\mathcal{A}} (f\ \bar{t}_n \to r \Leftarrow C)$$

Por la regla (PL_1) tenemos que

$$(f\ \bar{t}_n \to r \Leftarrow C) \Rightarrow_{\mathcal{A}} (f\ \bar{t}_n \to u \Leftarrow C_{\mathcal{A}}.C_r)$$

donde $r \Rightarrow_{\mathcal{A}} (u \,;\, C_r)$ y $C \Rightarrow_{\mathcal{A}} C_{\mathcal{A}}$.

Pero al ser P plano r debe ser un patrón y por el lema A.2.2 se tiene que $u \equiv r$ y C_r es la condición vacía, mientras que del lema A.2.4 se tiene que $C_{\mathcal{A}} \equiv C$.
$\Leftarrow$) Si $P \Rightarrow_{\mathcal{A}} P$ entonces P es plano. Sea P tal que $P \Rightarrow_{\mathcal{A}} P$. Como el aplanamiento se obtiene transformando cada regla de función según la regla (PL_1) debe cumplirse que para cada regla $(f\ \bar{t}_n \to r \Leftarrow C)$ se tiene

$$(f\ \bar{t}_n \to r \Leftarrow C) \Rightarrow_{\mathcal{A}} (f\ \bar{t}_n \to r \Leftarrow C)$$

de donde por la regla (PL_1) se debe cumplir

- $r \Rightarrow_{\mathcal{A}} (r;\ )$, por lo que del lema A.2.2 se tiene que r es un patrón.
- $C \Rightarrow_{\mathcal{A}} C$ y por el lema A.2.4 esto significa que C es una condición plana.

De la definición 4.3.1 (pág. 75) se tiene entonces que la regla $(f\ \bar{t}_n \to r \Leftarrow C)$ es a su vez plana. Al suceder esto para todas las reglas de P se tiene que P es un programa plano. ■

A.2.3. Demostración del Teorema 4.3.3

Vamos a probar en primer lugar algunos resultados auxiliares previos que nos servirán en la demostración de este teorema, enunciado en la página 80.

El primer resultado asegura que dado un contexto T utilizado para dar tipo a una expresión e con respecto a una signatura Σ, es posible encontrar otro contexto T' que permita dar tipo a la expresión aplanada y a sus condiciones asociadas con respecto a la misma signatura Σ. Para ello el lema prueba, por inducción sobre la estructura del aplanamiento de e, que T' puede obtenerse extendiendo T para dar el tipo adecuado a las variables nuevas.

Lema A.2.5. *Sean e una expresión, Σ una signatura, T un contexto y τ un tipo tales que $(\Sigma, T) \vdash_{WT} e :: \tau$. Supongamos que $e \Rightarrow_{\mathcal{A}} (e' ;\ e_1 \to X_1, \ldots, e_p \to X_p)$. Entonces existe un contexto $T' = T \uplus \{X_1 :: \mu_1, \ldots, X_p :: \mu_p\}$ tal que se cumple $(\Sigma, T') \vdash_{WT} e' :: \tau$ y $(\Sigma, T') \vdash_{WT} X_i :: \mu_i :: e_i$ para $i = 1 \ldots p$.*

Demostración
Nótese en primer lugar que escribimos $e \Rightarrow_{\mathcal{A}} (e' ;\ e_1 \to X_1, \ldots, e_p \to X_p)$ en lugar de $e \Rightarrow_{\mathcal{A}} (e' ;\ C')$ ya que el lema A.2.1 (pág. 191) nos asegura que las condiciones obtenidas al aplanar una expresión son de la forma $(e_1 \to X_1, \ldots, e_p \to X_p)$.

Vamos a demostrar el lema por inducción completa sobre la estructura de la prueba de $e \Rightarrow_{\mathcal{A}} (e' ;\ e_1 \to X_1, \ldots, e_p \to X_p)$. Para ello nos fijamos en el último paso de esta prueba y distinguimos casos según la regla aplicada (ver las reglas en la figura 4.1, pág. 77).

($\mathbf{PL_2}$), ($\mathbf{PL_3}$). Entonces $e \Rightarrow_{\mathcal{A}} (e\,;\,)$, por el apartado a) del lema A.2.2 (pág. 194), y el resultado se cumple para $T' = T$.

($\mathbf{PL_4}$). Si se ha aplicado esta regla debe ser $e \equiv X\ \overline{a}_k$. Como sabemos, $X\ \overline{a}_k$ abrevia $((\ldots(X\ a_1)\ \ldots)\ a_k)$, y aplicando k veces regla de tipo (AP) de la sección 3.1.4 (pág. 50) si $(\Sigma, T) \vdash_{WT} X\ \overline{a}_k :: \tau$ se tiene que deben existir tipos $\overline{\tau}_k$ tales que $(\Sigma, T) \vdash_{WT} \overline{a}_k :: \overline{\tau}_k$, y que $(\Sigma, T) \vdash_{WT} X :: \tau_1 \to\ \ldots\ \to \tau_k \to \tau$.

Nos fijamos ahora en la premisa $a_1 \Rightarrow_{\mathcal{A}} (u_1\ ;\ C_1)$ de (PL_4), donde C_1 (por el lema A.2.1) debe ser de la forma $(e_1 \to X_1, \ldots, e_l \to X_l)$ para cierto $l \geq 0$. Al tenerse $(\Sigma, T) \vdash_{WT} a_1 :: \tau_1$, por hipótesis de inducción debe existir $T_1 = T \uplus \{X_1 :: \mu_1, \ldots, X_l :: \mu_l\}$ tal que

(1) $(\Sigma, T_1) \vdash_{WT} u_1 :: \tau_1$ y $(\Sigma, T_1) \vdash_{WT} X_i :: \mu_i :: e_i$ para todo $i = 1 \ldots l$

Pasamos ahora a la segunda premisa de la regla: $R\ a_2 \ldots a_k \Rightarrow_{\mathcal{A}} (u; C)$, donde podemos suponer que C es de la forma $(e_{l+1} \to X_{l+1}, \ldots, e_m \to X_m)$ para cierto $m \geq l$ (si $m = l$ tenemos que C es la condición vacía). Definimos ahora $T_{aux} = T \uplus \{R :: \tau_2 \to\ \ldots\ \to \tau_k \to \tau\}$. Es fácil ver que entonces se tiene $(\Sigma, T_{aux}) \vdash_{WT} R\ a_2 \ldots a_k :: \tau$, y por hipótesis de inducción debe existir $T_2 = T_{aux} \uplus \{X_{l+1} \to \mu_{l+1}, \ldots, X_m \to \mu_m\}$ tal que

(2) $(\Sigma, T_2) \vdash_{WT} u :: \tau$ y $(\Sigma, T_2) \vdash_{WT} X_i :: \mu_i :: e_i$ para todo $i = l+1 \ldots m$

Sabemos por el lema A.2.1 que las variables X_i con $i = 1 \ldots m$ son distintas entre sí, y distintas de la variable nueva R. Podemos entonces definir el contexto

$$T' = T \uplus \{X_1 :: \mu_1,\ \ldots,\ X_l :: \mu_l,\ R :: \tau_2 \to \cdots \to \tau_k \to \tau,\\ X_{l+1} \to \mu_{l+1},\ \ldots,\ X_m \to \mu_m\}$$

T' es de la forma indicada por el lema, ya que es una extensión de T que da tipo a las variables nuevas de la condición formada por la conjunción de las aproximaciones contenidas en C_1, C y la nueva aproximación $X\ u_1 \to R$. Además:

- $(\Sigma, T') \vdash_{WT} u :: \tau$ de (2) al ser T' una extensión de T_2.
- $(\Sigma, T') \vdash_{WT} X_i :: \mu_i :: e_i$ para todo $i = 1 \ldots m$, de (1) y de (2) al ser T' extensión tanto de T_1 como de T_2. Esto verifica la propiedad para todas las condiciones contenidas en C_1, C.
- $(\Sigma, T') \vdash_{WT} X\ u_1 :: \tau_2 \to \ldots \tau_k \to \tau :: R$.

Por lo que T' cumple las condiciones requeridas por el lema.

($\mathbf{PL_5}$). En esta regla partimos de una expresión e de la forma $h\ \overline{e}_m$ por lo que de la premisa $(\Sigma, T) \vdash_{WT} e :: \tau$ se deduce que debe suceder $(\Sigma, T) \vdash_{WT} \overline{e}_m :: \overline{\tau}_m$ y $(\Sigma, T) \vdash_{WT} h :: \tau_1 \to \ \ldots\ \to \tau_k \to \tau$. Por hipótesis de inducción tenemos que para cada premisa de (PL_5) de la forma $e_i \Rightarrow_{\mathcal{A}} (u_i\ ;\ C_i)$ debe existir un contexto $T_i = T \uplus V_i$, donde V_i es el contexto que da tipo a las variables del lado derecho de las aproximaciones en C_i, y que verifica $(\Sigma, T_i) \vdash_{WT} u_i :: \tau_i$. Como cada $dom(V_i)$ coincide con el conjunto de variables nuevas utilizadas en cada premisa (lema A.2.1) se verifica que para todo $i \neq j$, $1 \leq i, j \leq m$, se tiene $dom(V_i) \cap dom(V_j) = \emptyset$. Podemos definir entonces $T' = T \uplus (T_1 \uplus \ \ldots \uplus \ T_m)$ y resulta sencillo comprobar que este nuevo contexto verifica la propiedad para la conclusión de la regla.

($\mathbf{PL_6}$). En este caso e es de la forma $f\ \overline{e}_n\ \overline{a}_k$, con $f \in FS^n$. De la hipótesis $(\Sigma, T) \vdash_{WT} e :: \tau$ se tiene que deben existir tipos $\overline{\tau}_n$, $\overline{\nu}_k$ tales que:

- $(\Sigma, T) \vdash_{WT} \overline{e}_n :: \overline{\tau}_n$
- $(\Sigma, T) \vdash_{WT} \overline{a}_k :: \overline{\nu}_k$.
- $(\Sigma, T) \vdash_{WT} f :: \tau_1 \to \ \cdots \to \tau_n\ \to \nu_1 \to \cdots \to \nu_k \to \tau$.

Aplicando la hipótesis de inducción tenemos que para cada premisa de la regla de la forma $e_i \Rightarrow_{\mathcal{A}} (u_i\ ;\ C_i)$, con $i = 1 \ldots n$, debe existir un cierto contexto con las características indicadas en el lema $T_i = T \uplus V_i$, donde V_i es el contexto que da tipo a las variables del lado derecho de las aproximaciones en C_i. En particular cada T_i debe verificar $(\Sigma, T_i) \vdash_{WT} u_i :: \tau_i$.

Para la premisa $S\ \overline{a}_k \Rightarrow_{\mathcal{A}} (u\ ;\ D)$ definimos un contexto apropiado

$$T_{aux}\ =\ T \biguplus \{S :: \nu_1 \to \cdots \to \nu_k \to \tau\}$$

en el que se verifica $(\Sigma, T_{aux}) \vdash_{WT} S\ \overline{a}_k\ ::\ \tau$, y por hipótesis de inducción tenemos que debe existir un cierto contexto $T_{n+1} = T_{aux} \uplus V_{n+1}$ tal que

$$\textbf{(3)} \qquad (\Sigma, T_{n+1}) \vdash_{WT} u\ ::\ \tau$$

además de dar tipo mediante V_{n+1} a las variables nuevas de los lados derechos de las aproximaciones en D, de forma que éstas queden bien tipadas. Igual que sucedía en los casos anteriores, al ser $dom(V_i)$ las variables nuevas introducidas en cada premisa se verifica que distintas entre sí y distintas de S, por S variable nueva de la regla. Es decir, se cumple que para todo $i \neq j$, $1 \leq i, j \leq n+1$, $dom(V_i) \cap dom(V_j) = \emptyset$ lo que nos permite definir un nuevo contexto

$$T' = T \uplus (V_1 \uplus \ldots \uplus V_n \uplus \{S :: \nu_1 \to \cdots \to \nu_k \to \tau\} \uplus V_{n+1})$$

que verifica:

- $(\Sigma, T') \vdash_{WT}\ u :: \tau$ de (3) al ser T' una extensión de T_{n+1}.
- Para cada $b \to X$ contenida en algún C_i $(i \leq n)$ o en D se cumple $(\Sigma, T') \vdash_{WT} X :: \mu :: b$ para algún tipo μ, ya que T' es una extension de $T_1, \ldots, T_{n+1}$ y según hemos visto se cumple $(\Sigma, T_i) \vdash_{WT}\ X :: \mu :: b$.
- $(\Sigma, T') \vdash_{WT}\ f\ \overline{u}_n :: \nu_1 \to \ldots \nu_k \to \tau :: S$, por la definición de T' para S y porque $(\Sigma, T') \vdash_{WT}\ f :: \tau_1 \to \cdots \to \tau_n \to \nu_1 \to \cdots \to \nu_k \to \tau$ y $(\Sigma, T') \vdash_{WT} u_i :: \tau_i$ para $i = 1 \ldots n$, al ser T' extensión de T y de T_i para $i = 1 \ldots n$, respectivamente.

□

El siguiente lema extiende el resultado anterior a las condiciones.

Lema A.2.6. *Sea C una condición, Σ una signatura y T un contexto tal que*

a) Para cada $(e == e') \in C$ existe algún $\mu \in Type$ tal que $(\Sigma, T) \vdash_{WT} e\ ::\ \mu\ ::\ e'$.

b) Para cada $(d \to s) \in C$ existe algún $\mu \in Type$ que verifica $(\Sigma, T) \vdash_{WT} d\ ::\ \mu\ ::\ s$.

Sea C' una condición tal que $C \Rightarrow_{\mathcal{A}} C'$. Entonces existe un contexto T_V, que da tipo a las variables nuevas introducidas durante el aplanamiento de C, tal que $T' = T \uplus T_V$ permite dar tipo a las condiciones atómicas de C' en el sentido de los ítems a) y b) anteriores.

Demostración
Por inducción completa sobre la estructura de la prueba $C \Rightarrow_{\mathcal{A}} C'$. Nos fijamos en el paso dado en último lugar y razonamos dependiendo de la regla aplicada, que puede ser (PL_7), (PL_8), (PL_9) o (PL_{10}).

(PL$_7$). En este caso $C \equiv C_1, C_2$. Es claro que T permite dar tipo a las condiciones atómicas de C_1 y C_2, al estar éstas en C. Como $C_1 \Rightarrow_{\mathcal{A}} D_1$ y $C_2 \Rightarrow_{\mathcal{A}} D_2$, por hipótesis de inducción deben existir T_{V1} y T_{V2} tales que $T_1 = T \uplus T_{V1}$ permite dar tipo a las condiciones atómicas en C_1' y $T_2 = T \uplus T_{V2}$ permite dar tipo las de C_2'. Además, al incluir supuestos de tipo sólo para las variables nuevas tenemos que $T_{V1} \cap T_{V2} = \emptyset$. Definimos entonces $T' = T \uplus T_{V1} \uplus T_{V2}$, que permitirá dar tipo a todas las condiciones atómicas en D_1, D_2.

(PL$_8$). Esta regla trata las igualdades estrictas y por tanto $C \equiv l == r$. Entonces por hipótesis $(\Sigma, T) \vdash_{WT} l :: \mu :: r$ para algún tipo $\mu \in Type$. Al verificarse $l \Rightarrow_{\mathcal{A}} (u_1 \,;\, C_1)$ por el lema A.2.5 existe un tipo $T_1 = T \uplus T_{V1}$ tal que $(\Sigma, T_1) \vdash_{WT} u_1 :: \mu$ y tal que T_1 permite dar tipo a las aproximaciones contenidas en C_1. Análogamente al cumplirse $r \Rightarrow_{\mathcal{A}} (u_2 \,;\, C_2)$ debe existir por el lema un tipo $T_2 = T \uplus T_{V2}$ tal que $(\Sigma, T_2) \vdash_{WT} u_2 :: \mu$ y tal que T_2 permite dar tipo a las aproximaciones contenidas en C_2. Además $T_{V1} \cap T_{V2} = \emptyset$ al contener supuestos de tipo para conjuntos disjuntos de variables. Definimos, igual que en la regla anterior, $T' = T \uplus T_{V1} \uplus T_{V2}$, que permitirá dar tipo a todas las condiciones atómicas en $(C_1, C_2, u_1 == u_2)$.

(PL$_9$). Si se ha aplicado esta regla la condición está formada por una única aproximación $e \to s$, donde por hipótesis $(\Sigma, T) \vdash_{WT} e :: \mu :: s$ para algún tipo $\mu \in Type$. Como $e \to (u, C')$ aplicando el lema A.2.5 se tiene que existe $T' = T \uplus T_V$ tal que $(\Sigma, T') \vdash_{WT} u :: \mu$, y que además permite dar tipo a las aproximaciones contenidas en C'. Por tanto T' permite dar tipo a $C', u \to s$, tal y como se requiere en este resultado.

(PL$_{10}$). En este caso basta con tomar $T' = T$, ya que la condición aplanada coincide con la condición inicial.

□

Además de probar que las condiciones y las expresiones del programa aplanado tienen un tipo adecuado, para asegurar que una regla de programa está bien tipada necesitamos comprobar que su condición es admisible según la definición dada en la sección 3.1.5 (pág. 51). De esto se encarga el lema siguiente.

Lema A.2.7. *Sea P un programa con signatura Σ y sea (R) una regla de programa $f\ \bar{t}_n \to r \Leftarrow C$. Sea C' una condición tal que $C \Rightarrow_{\mathcal{A}} C'$. Entonces C' es una secuencia de condiciones atómicas admisible con respecto a $var(f\ \bar{t}_n)$.*

Demostración
Consideramos una reordenación C_o de las aproximaciones e igualdades estrictas de C en la forma:

$$C_o \equiv\ e_1 \to s_1,\ \ldots,\ e_m \to s_m,\ l_1 == r_1,\ \ldots,\ l_k == r_k$$

donde las $e_1 \to s_1, \ldots,\ e_m \to s_m$ son todas las aproximaciones de C situadas en el orden requerido por las condiciones de admisibilidad enunciadas en la sección 3.1.5 (pág. 51),

y $l_1 == r_1,\ \ldots,\ l_k == r_k$ todas las igualdades estrictas en C. Sea entonces C'_o tal que $C_o \Rightarrow_{\mathcal{A}} C'_o$. A partir del lema A.2.3 (pág. 195) se tiene que C'_o será una reordenación de C' por serlo C_o de C, y de aquí y de la definición de admisibilidad (una condición es admisible si existe alguna reordenación de sus condiciones atómicas que cumple ciertas propiedades), bastará con probar que C'_o es admisible con respecto a $var(f\ \bar{t}_n)$, para tener que C' también lo es.

Del mismo lema A.2.3 tenemos que C'_o es de la forma:

$$C'_o \equiv\ CA_1, \ldots, CA_m, CI_1, \ldots, CI_k$$

donde $(e_i \rightarrow s_i) \Rightarrow_{\mathcal{A}} CA_i$ para $i = 1 \ldots m$, y $(l_j == s_j) \Rightarrow_{\mathcal{A}} CI_j$ para $j = 1 \ldots m$. Concretando más la forma de cada CA_i y CI_j y observando las reglas de aplanamiento de condiciones atómicas (PL_8), (PL_9) y (PL_{10}) se deduce que C'_o debe ser de la forma:

$$C'_o \equiv\ E_1, u_1 \rightarrow s_1,\ \ldots,\ E_m, u_m \rightarrow s_m,\ L_1, R_1, a_1 == b_1,\ \ldots,\ L_k, R_k, a_k == b_k$$

donde $CA_i \equiv (E_i,\ u_i \rightarrow s_i)$ para $i = 1 \ldots m$ y $CI_j \equiv (L_j,\ R_j,\ a_j == b_j)$, con $l_j \Rightarrow_{\mathcal{A}} (a_j\ ;\ L_j)$ y $r_j \Rightarrow_{\mathcal{A}} (b_j\ ;\ R_j)$ para $j = 1 \ldots k$. Observamos que en C'_o hay dos tipos de aproximaciones:

(1) Las de la forma $u_i \rightarrow s_i$, $1 \leq i \leq m$, cuya parte derecha coincide con la de una aproximación de C de la que proviene por aplanamiento. Dentro de estas aproximaciones podemos distinguir:

  (1.1) $u_i \rightarrow s_i$ proviene del aplanamiento de una aproximación $e_i \rightarrow s_i \in C_o$ verificándose que $e_i \Rightarrow_{\mathcal{A}} (u_i\ ;\ E_i)$. En este caso se ha utilizado la regla (PL_9) para aplanar $e_i \rightarrow s_i$.

  (1.2) $u_i \rightarrow s_i \in C_o$, en cuyo caso se ha utilizado la regla (PL_{10}) en el aplanamiento.

(2) Las incluidas en los conjuntos E_i, L_j y R_j, con $1 \leq i \leq m$, $1 \leq j \leq k$. Estas provienen del aplanamiento de expresiones y por apartado (b) del lema A.2.1 (pág. 191), son de la forma $e \rightarrow X$ donde X es una variable nueva. Conviene precisar el significado de "nueva"en este contexto: significa que la variable X sólo puede aparecer en la parte izquierda de otras aproximaciones obtenidas a partir de la misma condición atómica que $e \rightarrow X$, y que aparezcan situadas tras ella. En particular si $e \rightarrow X \in E_i$ para algún i entonces X sólo puede aparecer en la parte izquierda de otra aproximación de E_i (tal como especifica el apartado (b) del lema A.2.1) o en u_i. Análogamente si $e \rightarrow \in L_j$ o $e \rightarrow \in R_j$ para algún j entonces X sólo puede aparecer en la parte izquierda de otra aproximación de L_j (R_j respectivamente) o en a_j (respectivamente b_j).

Veamos que C'_o cumple las tres condiciones de admisibilidad:

(a) Para todo $e \rightarrow t \in C'_o$, $var(t) \cap \mathcal{X} = \emptyset$, donde $\mathcal{X} = var(f\ \bar{t}_n)$.
Siguiendo la clasificación de las aproximaciones en C'_o establecida más arriba, si $e \rightarrow t$ pertenece al tipo (1), i.e. si $e \rightarrow t \equiv u_i \rightarrow s_i$ para algún i, $1 \leq i \leq m$, se tiene t es la parte derecha de una aproximación de C y la propiedad se cumple al ser C admisible por hipótesis. Por el contrario si la aproximación pertenece al tipo (2) t es una variable nueva y $var(t) \cap \mathcal{X} = \emptyset$.

(b) Para todo $e \to t \in C'_o$, t es lineal y si $e' \to t' \in C'_o$, con $e \to t \neq e' \to t'$, $var(t) \cap var(t') = \emptyset$.

Para la primera parte: si t proviene de una aproximación de C es lineal por hipótesis. En otro caso (tipo (2)) es una variable y por tanto lineal. Para la segunda parte, si t o t' (o ambos) corresponden a aproximaciones del tipo (2) se trata de variables nuevas que no pueden aparecer en la parte derecha de ninguna otra aproximación, y el resultado se cumple directamente. Si ambos tanto t como t' corresponden a partes derechas de aproximaciones en C, el resultado se tiene por hipótesis.

(c) Sean $e \to t,\ e' \to t' \in C'_o$, tales que o bien $e \to t$ antecede a $e' \to t'$ en C'_o o bien $e' \to t' \equiv e \to t$. Entonces $var(t') \cap var(e) = \emptyset$.

Distinguimos casos según el tipo al que pertenezca la aproximación $e' \to t'$:

(1) Entonces $e' \to t' \equiv u_i \to s_i$ con s_i la parte derecha de una condición $e_i \to s_i \in C_o$. Distinguimos ahora los posibles tipos de $e \to t$, separando los casos (1.1), (1.2) y (2).

(1.1) Debe ser $e \to t \equiv u_j \to s_j$ y como $e \to t$ antecede a (o es) $e' \to t'$ debe cumplirse $1 \leq j \leq i \leq m$. Al ser $u_j \to s_j$ de tipo (1,1) ha de existir en C una aproximación de la forma $e_j \to s_j$ tal que $e_j \Rightarrow_{\mathcal{A}} (u_j\ ;\ C_j)$ para ciertas condiciones C_j.
Al ser C_o admisible y respetarse el orden de las aproximaciones en C'_o se tiene que $var(s_i) \cap var(e_j) = \emptyset$. Además por el apartado (d) del lema A.2.1 (pág. 191) $var(u_j) \subseteq var(e_j) \cup \{X_1, \ldots, X_k\}$, donde $\{X_1, \ldots, X_k\}$ son variables nuevas que por tanto no pueden aparecer en s_i (que es la parte derecha de una condición en C_o), de donde se tiene $var(s_i) \cap var(u_j) = \emptyset$, o lo que es lo mismo $var(t') \cap var(e) = \emptyset$.

(1.2) Entonces $e \to t \in C_o$ antecede (o coincide) en C_o a la aproximación de la que proviene por aplanamiento $u_i \to s_i$ y se cumple al ser C_o admisible $var(s_i) \cap var(e) = \emptyset$.

(2) Por anteceder a $e' \to t'$ debe ser $e \to t \in E_j$ para $1 \leq j \leq i \leq m$. Nos fijamos ahora en la aproximación $u_j \to s_j$, que debe ser de tipo (1.1) (si fuera de tipo (1.2) E_j sería la condición vacía y no podría cumplirse $e \to t \in E_j$), cuyo lado derecho se corresponde con el de una aproximación $e_j \to s_j \in C_o$ tal que $e_j \Rightarrow_{\mathcal{A}} (u_j\ ;\ E_j)$. Por el apartado (c) del lema A.2.1 (pág. 191) se tiene $var(e) \subseteq var(e_j) \cup V$, donde V representa un conjunto de variables nuevas. Ahora bien, al ser C_o admisible y $j \leq i$, también se cumple $var(e_j) \cap var(s_i) = \emptyset$ de donde $(var(e_j) \cup V) \cap var(s_i) = \emptyset$, por ser V variables que no pueden estar en s_i y por tanto $var(e) \cap var(s_i) = \emptyset$, o lo que es lo mismo (ya que $s_i = t'$) $var(t') \cap var(e) = \emptyset$, como queríamos demostrar.

- Tipo (2). Entonces $e' \to t' \in E_i$ para algún i, $1 \leq i \leq m$, o $e' \to t' \in L_i$ para $1 \leq i \leq k$ o $e' \to t' \in R_i$ para $1 \leq i \leq k$. En los tres casos se tiene que t' debe ser una variable nueva X. De nuevo distinguimos casos según el tipo de $e \to t$.

(1) Entonces $e \to t \equiv u_j \to s_j$ para algún j, $1 \leq j \leq m$. Si $e' \to t' \in E_i$ debe ser $j < i$ (no puede ser $j = i$ ya que $u_i \to s_i$ *sigue* y no antecede a E_i en C'_o). Entonces E_i y $u_j \to s_j$ provienen de diferentes condiciones atómicas en C_o y por lo expuesto durante la discusión del tipo (2) al comienzo de la demostración, $var(t) \cap var(u_j) = \emptyset$ ya que t' una variable nueva.
El mismo razonamiento es aplicable para $e' \to t' \in L_i$ y $e' \to t' \in R_i$.

(2) En este caso debemos tener en cuenta el conjunto E_j, L_j o R_j que contiene a $e \to t$. Si no es el mismo que el $e' \to t'$ entonces proviene de una condición atómica diferente y no puede contener a la variable t', por lo que se cumple el resultado. Pero si pertenece al mismo conjunto, o bien antecede a $e' \to t'$ o bien se trata del propio $e' \to t'$. Ambos casos están contemplados (con la condición $j \leq i$) en el apartado (b) del lema A.2.1 que asegura que $t' \notin var(e)$ o lo que es lo mismo $var(t') \cap var(e) = \emptyset$.

□

A partir de estos resultados la demostración del teorema queda:

Partimos de dos reglas de programa (R) $f\ \bar{t}_n \to r \Leftarrow C$ y (R') $f\ \bar{t}_n \to u \Leftarrow C_{\mathcal{A}}, C_r$ tales que $(R) \Rightarrow_{\mathcal{A}} (R')$. Queremos probar que (R') es una regla bien tipada con tipo principal $\overline{\tau}_n \to \tau$ sabiendo que:

(1) $f\ \bar{t}_n \to r \Leftarrow C$ es una regla bien tipada con tipo principal $\overline{\tau}_n \to \tau$.

(2) $r \Rightarrow_{\mathcal{A}} (u\ ;\ C_r)$.

(3) $C \Rightarrow_{\mathcal{A}} C_{\mathcal{A}}$.

Las condiciones que deben cumplir las reglas bien tipadas están definidas en la sección 3.1.5 (pág. 51). Vamos a repasarlas probándolas una por una:

(i) $t_1 \ldots t_n$ debe ser una secuencia lineal de patrones transparentes, y u debe ser una expresión.

La primera parte se tiene por **(1)** ya que el lado izquierdo de la regla no varía. La segunda parte, u es una expresión se tiene a partir de **(2)** ya que el apartado (a) del lema A.2.1 (pág. 191) dice que es en particular un patrón.

(ii) La *condición* $C_{\mathcal{A}}, C_r$ será una secuencia de *condiciones atómicas* $C_1, \ldots, C_k$, donde cada C_i puede ser o bien una *igualdad estricta* de la forma $e == e'$, con $e, e' \in Exp$, o bien una *aproximación* de la forma $d \to s$, con $d \in Exp$ y $s \in Pat$.

En cuanto a $C_{\mathcal{A}}$, el lema A.2.7 establece que se trata de una secuencia de condiciones atómicas. En cuanto a C_r, el apartado (b) del lema A.2.1 describe su estructura indicando que se trata de una secuencia de aproximaciones cuyos lados derechos son variables (por tanto patrones).

(iii) Además la condición $C_{\mathcal{A}}, C_r$ debe ser *admisible* respecto al conjunto de variables $\mathcal{X} =_{def} var(f\ \bar{t}_n)$.

La admisibilidad de $C_{\mathcal{A}}$ se establece de **(3)** y del lema A.2.7, al ser C admisible, mientras que la de C_r se deduce del apartado (b) del lema A.2.1. Para comprobar la admisibilidad de la conjunción de ambas conjunciones supongamos que tenemos ambas secuencias ya ordenadas según los criterios de admisibilidad. Sean entonces $e \to t \in C_{\mathcal{A}}$ y $e' \to X \in C_r$ cualesquiera. Veamos que se cumplen las condiciones (ii) y (iii) de admisibilidad descritas en la sección 3.1.5 (pág. 51) (la condición (i) no se ve alterada al hacer la conjunción de dos condiciones admisibles).

Para que se cumpla (ii) se debe tener (aparte de que la linealidad de t y X, ya comprobada al ser admisibles $C_{\mathcal{A}}$ y C_r) que $var(t) \cap var(X) = \emptyset$, cosa que ocurre al ser X una variable nueva obtenida en el aplanamiento de r y que por tanto no puede aparecer fuera de C_r. Para (iii) debería tenerse $var(X) \cap var(e) = \emptyset$ al ser $e' \to X$ posterior en la secuencia a $e \to t$. Y esto sucede por la misma razón en el caso de (ii).

(iv) Existe un contexto T' con dominio $var(R')$, que permite tipar correctamente la regla de programa.

Para esto partimos de que por **(1)** existe un contexto T que permite dar tipo a la regla en las condiciones expresadas en la sección 3.1.5 (pág. 51). En particular $(\Sigma, T) \vdash_{WT} r\ ::\ \tau$, por lo que de **(2)** y del lema A.2.5 (pág. 197) se tiene que debe existir $T_1 = T \biguplus T_{V1}$ tal que $(\Sigma, T_1) \vdash_{WT} u\ ::\ \tau$ y tal que además T_1 permita dar tipo a la condición C_r. Del mismo modo, a partir de **(3)** y del lema A.2.6 tenemos que debe existir un contexto $T_2 = T \biguplus T_{V2}$ que permite tipar a la condición $C_{\mathcal{A}}$. Además $T_{V1} \cap T_{V2} = \emptyset$. Definimos entonces T' como

$$T' = T \bigbiguplus T_{V1} \bigbiguplus T_{V2}$$

Veamos que T' cumple condiciones necesarias:

a) Para todo $1 \leq i \leq n$: $(\Sigma, T') \vdash_{WT} t_i\ ::\ \tau_i$.

Esto se tiene al ser T' una extensión de T y no haber cambiado el lado izquierdo de la regla.

b) $(\Sigma, T') \vdash_{WT} u\ ::\ \tau$.

Cierto porque $(\Sigma, T_1) \vdash_{WT} u\ ::\ \tau$ y T' es una extensión de T_1.

c) Para cada $(e == e') \in C_{\mathcal{A}}, C_r$ existe algún $\mu \in Type$
tal que $(\Sigma, T') \vdash_{WT} e\ ::\ \mu\ ::\ e'$.

Cierto por ser T' una extensión tanto de T_1 como de T_2; T_2 permite tipar las condiciones atómicas en $C_{\mathcal{A}}$ y T_1 las de C_r.

d) Para cada $(d \to s) \in C_{\mathcal{A}}, C_r$ existe algún $\mu \in Type$
que verifica $(\Sigma, T) \vdash_{WT} d\ ::\ \mu\ ::\ s$.

Análogo al anterior.

Como este razonamiento se puede aplicar a todas las reglas del programa transformado, ya que todas provienen, por definición, de una regla del programa original, tenemos que el programa transformado es un programa bien tipado, tal y como queríamos probar. Para finalizar observamos que en el programa transformado no se ha introducido ningún símbolo de función ni constructora de datos. Además el tipo principal de las funciones, como acabamos de comprobar, no varía. De aquí que la signatura del programa transformado sea la misma del programa original, como establecía el teorema.

■

A.2.4. Demostración del Teorema 4.3.4

Antes de demostrar el teorema, enunciado en la página 80, necesitamos algunos resultados auxiliares que presentamos y probamos a continuación.

Los siguientes lemas establecen algunas propiedades del cálculo SC descrito en la sección 3.2.1 (pág. 55) que nos serán útiles para la demostración del teorema. La idea detrás de estos resultados es que una aproximación $(v\ e_1 \dots e_k \to s)$ puede probarse en SC sii pueden probarse todas las aproximaciones del siguiente conjunto $S = \{v\ e_1 \to v_1,\ v_1\ e_2 \to v_3, \dots,\ v_{k-1}\ e_k \to v_k,\ v_k \to s\}$, y que, además, cualquier APA para la prueba de $(v\ e_1 \dots e_k \to s)$ puede obtenerse, salvo por la raíz (que es el propio $(v\ e_1 \dots e_k \to s)$) a partir de APAs para las pruebas de los elementos de S y viceversa. La utilidad de estos resultados estriba en que las reglas de aplanamiento van precisamente a realizar este cambio de una sola aproximación, aplicada a múltiples argumentos, por múltiples aproximaciones sobre elementos con un argumento.

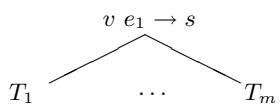
Lema A.2.8. *Sea P un programa, e una expresión $e \equiv v\ \overline{e}_k$ con $v \in Pat_\perp$, $e_i \in Exp_\perp$ para $i = 1 \dots k$, con $k > 0$, y sea $s \in Pat_\perp$ un patrón, $s \neq \perp$.*

Entonces se verifica $P \vdash_{SC} e \to s$ sii existen patrones $\overline{v}_k$, tales que (llamando v_0 a v) se cumple $P \vdash_{SC} (v\ e_1 \to v_1,\ v_1\ e_2 \to v_3, \dots,\ v_{k-1}\ e_k \to v_k,\ v_k \to s)$, Además ambas pruebas admiten APAs que sólo difieren en la raíz.

Demostración
Separamos las dos implicaciones que componen el resultado.

$\Rightarrow$) Partimos de $P \vdash_{SC} e \to s$, y hay que probar que existen los patrones $\overline{v}_k$ tales que $P \vdash_{SC} (v\ e_1 \to v_1,\ v_1\ e_2 \to v_3, \dots,\ v_{k-1}\ e_k \to v_k,\ v_k \to s)$, y que dado un APA para la primera demostración existe un APA para la segunda que sólo difiere en la raíz. Por inducción sobre $k \geq 1$.

Caso Base: $k = 1$. Partimos de que se cumple $P \vdash_{SC} v\ e_1 \to s$ con un cierto árbol de prueba T de la forma



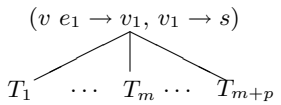
para ciertos subárboles $T_1, \ldots, T_m$. Aplicando la definición de APA (apartado 3.3.2, pág. 61),

$$\begin{aligned} apa(T) &= \mathit{árbol}(\text{raíz}(T),\, apa'(T_1) \mathbin{+\!\!+} \ldots \mathbin{+\!\!+} apa'(T_m)\,) = \\ &= \mathit{árbol}(v\ e_1 \to s,\, apa'(T_1) \mathbin{+\!\!+} \ldots \mathbin{+\!\!+} apa'(T_m)\,) \end{aligned}$$

donde

$$apa'(T) = \begin{cases} [apa(T)] & \text{si } \mathit{raíz}(T) \text{ es la concl. de un paso FA no trivial} \\ \mathit{subárboles}(apa(T)) & \text{en otro caso} \end{cases}$$

Queremos probar que existe v_1 tal que es posible probar $P \vdash_{SC} (v\ e_1 \to v_1,\, v_1 \to s)$. Obviamente basta con tomar $v_1 \equiv s$, ya que entonces se tiene por hipótesis $P \vdash_{SC} v\ e_1 \to v_1$ con un árbol de prueba T. Además por el apartado 2 de la proposición 3.2.1 (pág. 57), se cumple $P \vdash_{SC} v_1 \to s$. El árbol T' de prueba de $P \vdash_{SC} (v\ e_1 \to v_1,\, v_1 \to s)$ será de la forma:



Donde $T_{m+1}, \ldots, T_{m+p}$ son los subárboles hijos de una prueba de $P \vdash_{SC} v_1 \to s$, es decir de $P \vdash_{SC} s \to s$. Ahora bien, en esta prueba no se utiliza ningún paso $AR+FA$ al tratarse de una aproximación entre patrones por lo que, a partir de la definición de apa' es fácil probar que $apa'(T_{m+1}) = \cdots = apa'(T_{m+p}) = [\,]$. Entonces:

$$\begin{aligned} apa(T') &= \mathit{árbol}(\text{raíz}(T'),\, apa'(T_1) \mathbin{+\!\!+} \ldots \mathbin{+\!\!+} apa'(T_{m+p})\,) = \\ &= \mathit{árbol}((v\ e_1 \to v_1,\, v_1 \to s),\, apa'(T_1) \mathbin{+\!\!+} \ldots \mathbin{+\!\!+} apa'(T_m)\,) \end{aligned}$$

y por tanto $apa(T)$ y $apa'(T')$ sólo difieren en la raíz, como indica el enunciado.

Caso inductivo: Suponemos cierto cierto para $k-1$ y lo probamos para k. Escribiendo $v\ \overline{e}_k$ como $(v\ \overline{e}_{k-1})\ e_k$, por el apartado 3 de la proposición 3.3.2 (pág. 63) tenemos que existe un patrón s' tal que

$$\textbf{(1)} \quad P \vdash_{SC} v\ \overline{e}_{k-1} \to s'$$

$$\textbf{(2)} \quad P \vdash_{SC} s'\ e_k \to s$$

y tal que $P \vdash_{SC} (v\ \overline{e}_{k-1} \to s', s'\ e_k \to s)$ admite un APA que sólo difiere de cierto APA para $P \vdash_{SC} v\ \overline{e}_k \to s$ en la raíz.

Aplicando la hipótesis de inducción a (1) se deduce que existen patrones $\overline{v}_{k-1}$ tales que para todo $i = 1 \ldots k-1$ se cumple $P \vdash_{SC} v_{i-1}\ e_i \to v_i$, donde $v_0 = v$, y además

$$\textbf{(3)} \qquad P \vdash_{SC} v_{k-1} \to s'$$

Veamos que tomando $v_k = s$ completamos el resultado. La hipótesis de inducción se traduce entonces en la existencia de una prueba para

$$P \vdash_{SC} (v_0\ e_1 \to v_1, \ldots, v_{k-2}\ e_{k-1} \to v_{k-1},\ v_{k-1} \to s')$$

tal que admite un APA que sólo difiere de algún APA para (1) en la raíz, por lo que, añadiendo la aproximación de (3) llegamos a que se puede probar

$$\textbf{(4)} \qquad P \vdash_{SC} (v_0\ e_1 \to v_1, \ldots, v_{k-2}\ e_{k-1} \to v_{k-1},\ v_{k-1} \to s', s'\ e_k \to v_k)$$

y que esta prueba admite un APA que sólo difiere de algún APA de $P \vdash_{SC} v\ \overline{e}_k \to s$ en la raíz. Finalmente, combinando (2) y (3), es decir las dos últimas aproximaciones de (4), se tiene por el apartado 3 de la proposición 3.3.2 (pág. 63, utilizando en esta ocasión la implicación "$\Leftarrow$") que se cumple $P \vdash_{SC} v_{k-1}\ e_k \to s$, es decir $P \vdash_{SC} v_{k-1}\ e_k \to v_k$ y esta prueba admite un APA que sólo varía con el de $P \vdash_{SC} (\ v_{k-1} \to s', s'\ e_k \to v_k)$ en la raíz (y en el orden de los subárboles hijos, pero por la observación 3.3.1 de la pág. 63 admitimos que se trata del mismo APA). Utilizando esta información en (4) llegamos a que existe una prueba para

$$\textbf{(5)} \qquad P \vdash_{SC} (v_0\ e_1 \to v_1, \ldots, v_{k-2}\ e_{k-1} \to v_{k-1},\ v_{k-1}\ e_k \to v_k)$$

que admite un APA que sólo difiere de algún APA de $P \vdash_{SC} v\ \overline{e}_k \to s$ en la raíz.

Finalmente la aproximación $v_k \to s$ se puede probar para cualquier programa P por el apartado 2 de la proposición 3.2.1 al ser $v_k \equiv s$ por lo que se puede añadir esta prueba a (5) llegando a:

$$P \vdash_{SC} (v_0\ e_1 \to v_1, \ldots, v_{k-2}\ e_{k-1} \to v_{k-1},\ v_{k-1}\ e_k \to v_k, v_k \to v_k)$$

y, al tratarse $v_k \to s$ (i.e. $v_k \to v_k$) de una aproximación entre patrones no afecta al APA de la prueba, manteniéndose que esta demostración admite un APA que sólo difiere de algún APA de $P \vdash_{SC} v\ \overline{e}_k \to s$ en la raíz.

$\Leftarrow$) Suponemos ahora que existen patrones $\overline{v}_k$ tales que

$$\textbf{(6)} \quad P \vdash_{SC} v_{i-1}\ e_i \to v_i \quad \text{para } i = 1 \ldots k$$
$$\textbf{(7)} \quad P \vdash_{SC} v_k \to s$$

con $v_0 \equiv v$, y queremos probar $P \vdash_{SC} v\ \overline{e}_k \to s$. De nuevo utilizamos inducción sobre k.

Caso Base: $k = 1$. De (7) se tiene por el apartado 1 de la proposición, 3.2.1 que $v_1 \sqsupseteq s$ y combinando este resultado con (6) (para i=1), por el apartado 1 de la proposición 3.3.2 (pág. 63) se llega al resultado deseado $P \vdash_{SC} v\ e_1 \to s$. Además la prueba de (7) no aporta ningún nodo al APA al tratarse de una aproximación entre patrones lo que, a partir del apartado 1 de la proposición 3.3.2 (pág. 63) se tiene que los APAs de (6) (para $i = 1$) y de $P \vdash_{SC} v\ e_1 \to s$ coinciden salvo por la raíz.

Caso Inductivo: Suponemos el resultado cierto para $k-1$. De (6) para $i = 1 \ldots k-1$, y al cumplirse por el apartado 2 de la proposición 3.2.1 $P \vdash_{SC} v_{k-1} \rightarrow v_{k-1}$ tenemos por hipótesis de inducción:

$$\textbf{(8)} \qquad P \vdash_{SC} v\ \overline{e}_{k-1} \rightarrow v_{k-1}$$

De nuevo de (6) pero para $i = k$:

$$\textbf{(9)} \qquad P \vdash_{SC} v_{k-1}\ e_k \rightarrow v_k$$

Aplicando el apartado 1 de la proposición 3.2.1 (pág. 57) a (7), tenemos $v_k \sqsupseteq s$, lo que por el apartado 1 de la proposición 3.3.2 (pág. 63) y por (7) nos lleva a

$$P \vdash_{SC} v_{k-1}\ e_k \rightarrow s$$

Y uniendo este resultado a (8), por el apartado 3 de la proposición 3.3.2 (pág. 63) llegamos a $P \vdash_{SC} v\ \overline{e}_{k-1}\ e_k \rightarrow s$, o lo que es lo mismo

$$P \vdash_{SC} v\ \overline{e}_k \rightarrow s$$

como queríamos demostrar. La relación entre los APAs se obtiene de forma análoga al caso inductivo de la implicación contraria, al ser el apartado 3 de la proposición 3.3.2 de la forma "si y sólo si".

□

El teorema va a ser consecuencia del siguiente lema:

Lema A.2.9. *Sean P, $P_{\mathcal{A}}$ programas tales que $P \Rightarrow_{\mathcal{A}} P_{\mathcal{A}}$, c una condición atómica y C una condición tales que $c \Rightarrow_{\mathcal{A}} C$. Sea $\theta \in Subst_{\perp}$ una sustitución y sea $V_{\mathcal{A}}$ el conjunto de variables nuevas introducidas durante el aplanamiento de c, que podemos suponer disjuntas con las variables de θ, es decir*

$$V_{\mathcal{A}} \cap (dom(\theta) \cup ran(\theta)) = \emptyset$$

Entonces $P \vdash_{SC} c\theta$ sii existe una sustitución ν con $dom(\nu) \subseteq V_{\mathcal{A}}$ tal que $P_{\mathcal{A}} \vdash_{SC} C(\theta \biguplus \nu)$. Además ambas demostraciones admiten APAs que sólo difieren en la raíz.

Demostración
Separamos ambas implicaciones:

$\Rightarrow$) Suponemos $P \vdash_{SC} c\theta$ y llamamos $V_{\mathcal{A}}$ al conjunto de variables nuevas generado durante el aplanamiento de c. Queremos probar que existe una sustitución ν con $dom(\nu) \subseteq V_{\mathcal{A}}$ tal que $P_{\mathcal{A}} \vdash_{SC} C(\theta \biguplus \nu)$. Utilizaremos inducción completa sobre la profundidad de la demostración de $P \vdash_{SC} c\theta$.

Para ello distinguimos 6 casos según la forma de c.

1. $c \equiv e == e'$ para $e, e' \in Exp_\perp$.

   Por la regla (PL_8) se tiene que

   $$e == e' \ \Rightarrow_{\mathcal{A}} \ C_1,\ C_2,\ e_{\mathcal{A}} == e'_{\mathcal{A}}$$

   donde $e \Rightarrow_{\mathcal{A}} (e_{\mathcal{A}}\,;\, C_1)$, $e' \Rightarrow_{\mathcal{A}} (e'_{\mathcal{A}}\,;\, C_2)$. La hipótesis es en este caso que $P \ \vdash_{SC} e\theta == e'\theta$. El último paso de esta demostración será una aplicación de la regla JN de la forma:

   $$\textbf{(1)} \quad \frac{e\theta \to t \qquad e'\theta \to t}{e\theta == e'\theta}$$

   Por hipótesis de inducción existen dos sustituciones ν_1 y ν_2 con dominios disjuntos formados por variables nuevas tales que

   $$\begin{array}{ll} P_{\mathcal{A}} \vdash_{SC} & e_{\mathcal{A}}(\theta \uplus \nu_1) \to t \\ P_{\mathcal{A}} \vdash_{SC} & C_1(\theta \uplus \nu_1) \\ P_{\mathcal{A}} \vdash_{SC} & e'_{\mathcal{A}}(\theta \uplus \nu_2) \to t \\ P_{\mathcal{A}} \vdash_{SC} & C_2(\theta \uplus \nu_2) \end{array}$$

   que, definiendo $\nu = \nu_1 \uplus \nu_2$ se pueden reescribir como:

   $$\begin{array}{lll} \textbf{(2)} & P_{\mathcal{A}} \vdash_{SC} & e_{\mathcal{A}}(\theta \uplus \nu) \to t \\ \textbf{(3)} & P_{\mathcal{A}} \vdash_{SC} & C_1(\theta \uplus \nu) \\ \textbf{(4)} & P_{\mathcal{A}} \vdash_{SC} & e'_{\mathcal{A}}(\theta \uplus \nu) \to t \\ \textbf{(5)} & P_{\mathcal{A}} \vdash_{SC} & C_2(\theta \uplus \nu) \end{array}$$

   Entonces se cumple $P_{\mathcal{A}} \vdash_{SC} C(\theta \uplus \nu)$ porque:

   - $P_{\mathcal{A}} \vdash_{SC} C_1(\theta \uplus \nu)$ por (3).
   - $P_{\mathcal{A}} \vdash_{SC} C_2(\theta \uplus \nu)$ por (5).
   - $P_{\mathcal{A}} \vdash_{SC} e_{\mathcal{A}}(\theta \uplus \nu) == e'_{\mathcal{A}}(\theta \uplus \nu)$ con una demostración cuyo último paso es de la forma

     $$\textbf{(6)} \quad \frac{e_{\mathcal{A}}(\theta \uplus \nu) \to t \qquad e'_{\mathcal{A}}(\theta \uplus \nu) \to t}{e_{\mathcal{A}}(\theta \uplus \nu) == e'_{\mathcal{A}}(\theta \uplus \nu)}$$

     con t el mismo patrón que aparece en (1) y donde las premisas tienen demostración en $P_{\mathcal{A}}$ por (2) y por (4).

   Por otra parte, si llamamos T al árbol de prueba de (1) y $T_{(e\theta \to t)}$ $T_{(e\theta \to t)}$ a los de sus premisas, aplicando la definición de APA tenemos

   $$apa(T) = \text{árbol}(e\theta == e'\theta, apa'(T_{(e\theta \to t)}) \ +\!\!+ apa'(T_{(e'\theta \to t)}))$$

y como ni $(e\theta \to t)$ ni $(e'\theta \to t)$ corresponden a conclusiones de pasos FA:

$$\begin{aligned} apa(T) \quad &= \text{árbol}(e\theta == e'\theta, \\ &\qquad \text{subárboles}(apa(T_{(e\theta \to t)})) \; +\!\!+ \text{subárboles}(apa(T_{(e'\theta \to t)}))\,) \end{aligned}$$

De la hipótesis de inducción se tiene

$$apa(T_{(e\theta \to t)}) = apa(T'_{(e_{\mathcal{A}}(\theta \uplus \nu) \to t)})$$
$$apa(T_{(e'\theta \to t)}) = apa(T'_{(e'_{\mathcal{A}}(\theta \uplus \nu) \to t)})$$

con T' el árbol de prueba de (6) y $T'_{(e_{\mathcal{A}}(\theta \uplus \nu) \to t)}$, $T'_{(e'_{\mathcal{A}}(\theta \uplus \nu) \to t)}$ los subárboles de prueba de sus premisas. Reemplazando estas dos igualdades en $apa(T)$ se llega a:

$$\begin{aligned} apa(T) \quad &= \text{árbol}(e\theta == e'\theta, \\ &\qquad \text{subárboles}(apa(T'_{(e_{\mathcal{A}}(\theta \uplus \nu) \to t)})) \; +\!\!+ \text{subárboles}(apa(T'_{(e'_{\mathcal{A}}(\theta \uplus \nu) \to t)}))\,) \end{aligned}$$

y como ni $(e_{\mathcal{A}}(\theta \biguplus \nu) \to t)$ ni $(e'_{\mathcal{A}}(\theta \biguplus \nu) \to t)$ corresponden a la conclusión de pasos FA, por la definición de apa' se tiene que:

$$\text{subárboles}(apa(T'_{(e_{\mathcal{A}}(\theta \uplus \nu) \to t)})) = apa'(T'_{(e_{\mathcal{A}}(\theta \uplus \nu) \to t)})$$
$$\text{subárboles}(apa(T'_{(e'_{\mathcal{A}}(\theta \uplus \nu) \to t)})) = apa'(T'_{(e'_{\mathcal{A}}(\theta \uplus \nu) \to t)})$$

lo que, sustituyendo en $apa(T)$ nos lleva a

$$apa(T) = \text{árbol}(e\theta == e'\theta, apa'(T'_{(e_{\mathcal{A}}(\theta \uplus \nu) \to t)}) \; +\!\!+ apa'(T'_{(e'_{\mathcal{A}}(\theta \uplus \nu) \to t)})\,)$$

árbol que coincide con $apa(T')$ (i.e. el APA de la prueba de (6)) salvo por la raíz, tal y como queríamos demostrar.

2. $c \equiv e \to t$ con $t \equiv \bot$.

Si e es plana aplicando la regla (PL_{10}) se tiene que $C \equiv c$, y el resultado se cumple con ν la identidad, ya que $C(\theta \biguplus \nu) = e\theta \to \bot$, y esta aproximación puede probarse en cualquier programa. Si e no es plana se aplica la regla (PL_9):

$$(e \to \bot) \Rightarrow_{\mathcal{A}} C_{\mathcal{A}}, e_{\mathcal{A}} \to \bot$$

Definimos una sustitución $\nu(X_i) = \bot$ para todo $X_i \in V_{\mathcal{A}}$. Entonces:

- $P_{\mathcal{A}} \vdash_{SC} \; e_{\mathcal{A}}(\theta \biguplus \nu) \to \bot$. Esto es cierto trivialmente, con una demostración formada por un único paso en el que se aplica la regla BT.
- $P_{\mathcal{A}} \vdash_{SC} \; C_{\mathcal{A}}(\theta \biguplus \nu)$. Sabemos por el apartado b) del lema A.2.1 (pág. 191) que $C_{\mathcal{A}}$, al provenir del aplanamiento de una expresión, está formado únicamente por aproximaciones de la forma $b_i \to X_i$, con $X_i \in V_{\mathcal{A}}$ y por tanto

  $$(b_i \to X_i)(\theta \biguplus \nu) = b_i(\theta \biguplus \nu) \to X_i\nu = b_i(\theta \biguplus \nu) \to \bot$$

  y estas aproximaciones pueden probarse en SC en un paso aplicando la regla BT.

Además en todos los casos los APAs guardan la relación establecida en el enunciado al constar sólo de la raíz.

3. $c \equiv e \to t$ con $t \neq \bot$, e plana.

   En este caso se aplica la regla (PL_{10}) y se cumple que $e \to t \Rightarrow_{\mathcal{A}} e \to t$, es decir que $c \equiv C$ con $V_{\mathcal{A}} = \emptyset$, por lo que la única sustitución ν con dominio $V_{\mathcal{A}}$ es $\nu = id$ y

   $$C(\theta \biguplus \nu) = C\theta = e\theta \to t\theta = c\theta$$

   y lo que hay que probar es que se cumple $P_{\mathcal{A}} \vdash_{SC} e\theta \to t\theta$. Distinguimos 3 posibilidades:

   - e es un patrón. Entonces $e\theta$ también será un patrón, por lo que del apartado 4 de la proposición 3.2.1 (pág. 57), se tiene que se cumple $P_{\mathcal{A}} \vdash_{SC} e\theta \to t\theta$ al hacerlo $P \vdash_{SC} e\theta \to t\theta$. Además los APAs en ambos casos constarán únicamente de la raíz al tratarse de aproximaciones entre patrones en cuya prueba no puede intervenir la regla $AR + FA$ del cálculo SC, verificándose así el enunciado.
   - $e \equiv X\ t$ con $X \in Var$, $t \in Pat_{\bot}$. A su vez aquí distinguimos dos casos, según si $e\theta$ es o no activa. Si $e\theta$ es activa debe cumplirse $X\theta = f\ \overline{t}_{n-1}$, con $f \in FS^n$, y en este caso la demostración es análoga a la del apartado $e \equiv f\ \overline{s}_n$ que se verá a continuación.

     Si $e\theta$ no es activa al ser $X\theta$, $t\theta$ necesariamente patrones, $e\theta$ es un patrón y la aproximación $e\theta \to t\theta$ se cumple en $P_{\mathcal{A}}$ por el apartado 4 de la proposición 3.2.1, verificándose además la relación entre los APAs de forma trivial al constar estos de un único nodo como en el apartado anterior.
   - $e \equiv f\ \overline{s}_n$ con $f \in FS^n$, $s_i \in Pat_{\bot}$ para cada $i = 1 \ldots n$. En este caso la prueba de $P \vdash_{SC} (f\ \overline{s}_n \to t)\theta$ finalizará con un paso $AR + FA$ de la forma:

     $$\textbf{(7)} \quad \frac{s_1\theta \to t_1\ \ldots\ s_n\theta \to t_n \qquad \dfrac{C \quad r \to s}{\boxed{f\ \overline{t}_n \to s}} \qquad s\ \to t\theta}{f\ \overline{s}_n\theta\ \to t\theta}$$

     donde se tiene que

     $$(f\ \overline{t}_n \to r \Leftarrow C) = (f\ \overline{t'}_n \to r' \Leftarrow C')\sigma$$

     con $(f\ \overline{t'}_n \to r' \Leftarrow C') \in P$. Podemos suponer además que el dominio de σ está formado únicamente por variables de la regla de programa y que para ésta se ha tomado una variante con variables nuevas.

     Entonces por (7) se cumple $P \vdash_{SC} C$, es decir $P \vdash_{SC} C'\sigma$ al ser $C'\sigma = C$. Si suponemos C' compuesto por l condiciones atómicas $C' \equiv c'_1 \ldots c'_l$ esto significa que $P \vdash_{SC} c'_i\sigma$ para $i = 1 \ldots l$, donde cada una de estas demostraciones tiene una

profundidad menor que la de $c\theta$. Esto significa que, por hipótesis de inducción, existe una sustitución σ_i para cada $i = 1 \dots l$ tal que $P_{\mathcal{A}} \vdash_{SC} (C'_i)(\sigma \uplus \sigma_i)$ donde $c'_i \Rightarrow_{\mathcal{A}} (C'_i)$, con el dominio de σ_i formado por las variables nuevas introducidas durante el aplanamiento, que podemos suponer diferentes de todas las variables aparecidas hasta el momento. Por el lema A.2.3 (pág. 195) si $C' \Rightarrow_{\mathcal{A}} C_{\mathcal{A}}$ se tiene que $C_{\mathcal{A}} \equiv C'_1, \dots C'_l$, y basta con definir

$$\sigma_c = \sigma_1 \biguplus \dots \biguplus \sigma_l$$

para tener

$$\textbf{(8)} \quad P_{\mathcal{A}} \vdash_{SC} C_{\mathcal{A}}(\sigma \biguplus \sigma_c)$$

donde esta prueba admite un APA que sólo difiere de un APA para $P \vdash_{SC} C$ en la raíz.

También de (7) se tiene

$$P \vdash_{SC} r'\sigma \to t\theta$$

definiendo una sustitución $\theta' = \sigma \uplus \{Y \mapsto t\theta\}$ donde Y es una variable nueva se cumple $P \vdash_{SC} (r' \to Y)\theta'$ al ser

$$(r' \to Y)\theta' = r'\sigma \to t\theta = r \to t\theta$$

Además la profundidad de esta demostración es menor que la de $c\theta$, por lo que podemos aplicar la hipótesis de inducción para tener que existe una sustitución, a la que podemos llamar μ_r, con dominio formado por variables nuevas introducidas durante el aplanamiento $(r' \to Y) \Rightarrow_{\mathcal{A}} A_r$ tal que $P_{\mathcal{A}} \vdash_{SC} A_r(\theta' \uplus \mu_r)$, con una prueba que admite un APA que coincide con el de $P \vdash_{SC} r'\sigma \to t\theta$ salvo por la raíz. Vamos a comprobar que entonces existe una sustitución σ_r tal que

$$\textbf{(9)} \quad P_{\mathcal{A}} \vdash_{SC} r_{\mathcal{A}}(\sigma \uplus \sigma_r) \to t\theta$$
$$\textbf{(10)} \quad P_{\mathcal{A}} \vdash_{SC} C_r(\sigma \uplus \sigma_r)$$

donde $r' \Rightarrow_{\mathcal{A}} (r_{\mathcal{A}} \,;\, C_r)$ y $dom(\sigma_r) \subseteq V_r$ con V_r las variables nuevas producidas durante el aplanamiento de r.

Para ello observamos que existen dos posibilidades para A_r según si en el aplanamiento se ha utilizado la regla (PL_9) o (PL_{10}):

- Si r' no es plana por la regla PL_9 se tiene que $A_r = C_r, r_{\mathcal{A}} \to Y$ con $r' \Rightarrow_{\mathcal{A}} (r_{\mathcal{A}} \,;\, C_r)$, y las variables nuevas introducidas durante el aplanamiento de la aproximación coincidirán con las introducidas durante el aplanamiento de la expresión a_i. Basta con tomar entonces σ_r como μ_r. Se cumple

$$\begin{aligned} (r_{\mathcal{A}} \to Y)(\theta' \uplus \sigma_r) &= r_{\mathcal{A}}(\sigma \uplus \sigma_r) \to t\theta \\ C_r(\theta' \uplus \sigma_r) &= C_r(\sigma \uplus \sigma_r) \end{aligned}$$

ya que σ_r no puede contener en su dominio a Y, ni Y puede aparecer entre las variables de C_r ni de $r_{\mathcal{A}}$. De aquí, al tenerse $P_{\mathcal{A}} \vdash_{SC} A_r(\theta' \uplus \mu_r)$ separando las dos condiciones de A_r se llega a (9) y (10), y ' al provenir A_r

- Si r' ya es plana se ha utilizado PL_{10} y se tiene $A_r \equiv r' \to Y$, con $\mu_r = id$ y por tanto $P_{\mathcal{A}} \vdash_{SC} A_r(\theta' \uplus \mu_r)$ queda

$$P_{\mathcal{A}} \vdash_{SC} r'\sigma \to t\theta$$

Ahora bien, si r' es un patrón por el lema A.2.2 (pág. 194) se tiene que en $r' \Rightarrow_{\mathcal{A}} (r_{\mathcal{A}}\,;\, C_r)$, C_r es la condición vacía y $r_{\mathcal{A}} \equiv r'$, por lo que con $\sigma_r = id$ se cumplen (9) y (10). Si por el contrario r' no es un patrón, al ser una expresión plana debe ser, por la definición 4.3.1 (pág. 75), o bien de la forma $r' \equiv g\ \overline{u}_m$ con $g \in FS^m$ y $u_i \in Pat_\perp$ para $i = 1 \ldots m$, o bien $r' \equiv X\ t$, con $X \in Var, t \in Pat_\perp$. En cualquier caso aplicando bien la regla (PL_4) o la regla (PL_6) se llega a que $r_{\mathcal{A}} \equiv Z$, $C_r = r' \to Z$, con Z una variable nueva. Entonces definimos $\sigma_r = \{Z \mapsto t\theta\}$ y se tiene que se cumple (9) por el apartado 2 de la proposición 3.2.1 (pág. 57) al ser $r_{\mathcal{A}}(\theta' \uplus \sigma_r) = t\theta$, y (10) por (7), al ser $C_r(\theta' \uplus \sigma_r) = r \to t\theta$.

Entonces consideramos el resultado de aplanar la regla utilizada en (7) mediante (PL_1):

$$(f\ \overline{t}'_n \to r' \Leftarrow C') \Rightarrow_{\mathcal{A}} (f\ \overline{t}'_n \to r_{\mathcal{A}} \Leftarrow C_r, C_{\mathcal{A}})$$

donde $r' \Rightarrow_{\mathcal{A}} (r_{\mathcal{A}}\,; C_r)$ y $C \Rightarrow_{\mathcal{A}} C_{\mathcal{A}}$. Podemos probar entonces $P_{\mathcal{A}} \vdash_{SC} (f\ \overline{s}_n \to t)\theta$ acabando la demostración con una regla $AR + FA$ en la que se utiliza la instancia de esta regla dada por $\sigma' = \sigma \uplus \sigma_c \uplus \sigma_r$, y donde se cumple $t'_i\sigma' = t'_i\sigma = t_i$:

$$\frac{s_1\theta \to t_1\ \ \ldots\ \ s_n\theta \to t_n \qquad \boxed{f\ \overline{t}_n \to s} \qquad s\ \to t\theta}{f\ \overline{s}_n\theta\ \to t\theta} \qquad \frac{C_{\mathcal{A}}\sigma',\ C_r\sigma'\ \ r_{\mathcal{A}}\sigma' \to s}{\boxed{f\ \overline{t}_n \to s}}$$

Las premisas se cumplen en $P_{\mathcal{A}}$ bien por ser premisas de (7), aplicando el apartado 4 de la proposición 3.2.1, o bien por (8), (9) o (10), al ser $C_{\mathcal{A}}\sigma' = C_{\mathcal{A}}(\sigma \uplus \sigma_c)$, $r_{\mathcal{A}}\sigma' = r_{\mathcal{A}}(\sigma \uplus \sigma_r)$ y $C_r\sigma' = C_r(\sigma \uplus \sigma_r)$, respectivamente. Además el APA de (7) y de esta prueba coinciden salvo por la raíz ya que tienen las mismas premisas salvo aquellas que como hemos visto se obtienen a partir de premisas de (7) a las que se puede aplicar la hipótesis de inducción.

4. $c \equiv e \to t$ con $t \neq \perp$, e no plana, $e \equiv X\ \overline{a}_k$, $a_i \in Exp_\perp$ para $i = 1 \ldots k$, $k > 1$ y por tanto e flexible.

   En este caso es fácil ver que, aplicando repetidamente (PL_4) y finalmente (PL_3) se tiene

$$X\ \overline{a}_k \Rightarrow_{\mathcal{A}} (X_k\,;\, C_1,\, X\ u_1 \to X_1,\, C_2,\, X_1\ u_2 \to X_2,\, \ldots,\, C_k,\, X_{k-1}\ u_k \to X_k)$$

donde $a_i \Rightarrow_{\mathcal{A}} (u_i\,;\, C_i)$ para todo $1 \leq i \leq k$. Llamando X_0 a X podemos escribir:

$$\begin{aligned} e_{\mathcal{A}} \equiv &\quad X_k \\ C_{\mathcal{A}} \equiv &\quad C_1,\, X_0\ u_1 \to X_1,\, \ldots,\, C_k,\, X_{k-1}\ u_k \to X_k \end{aligned}$$

de donde

$$\textbf{(11)} \qquad X\ \overline{a}_k \to t \Rightarrow_{\mathcal{A}} C_1,\, X_0\ u_1 \to X_1,\, \ldots,\, C_k,\, X_{k-1}\ u_k \to X_k,\, X_k \to t$$

En primer lugar tratamos el caso en el que $t\theta \equiv \bot$. En este caso basta con definir $\nu(Y) = \bot$ para toda variable Y introducida durante el aplanamiento. Recordando que los lados derechos de las condiciones C_i están formados por variables nuevas (lema A.2.1, pág. 191) se cumplirá entonces

$$P_{\mathcal{A}} \vdash_{SC} (C_1,\, X_0\ u_1 \to X_1,\, \ldots,\, C_k,\, X_{k-1}\ u_k \to X_k,\, X_k \to t)(\theta \biguplus \nu)$$

ya que todos los lados derechos de las aproximaciones serán $\bot$ al aplicar ν. Además la relación entre los APAs se verificará trivialmente al constar éstos de un único nodo.

Para la demostración se $t\theta \neq \bot$ vamos a seguir los siguientes pasos:

a) Partimos de $P \vdash_{SC} (X\ \overline{a}_k \to t)\theta$, o lo que es lo mismo $P \vdash_{SC} \theta(X)\ \overline{a}_k\theta \to t\theta$. Llamamos v a $\theta(X)$, e_i a $a_i\theta$ para $i = 1 \ldots k$ y s a $t\theta$. Entonces podemos aplicar el lema A.2.8 (pág. 205), de donde tenemos que existen patrones $\overline{v}_k$ tales que (llamando v_0 a $v \equiv \theta(X)$):

$$\textbf{(12)} \quad P \vdash_{SC} v_{i-1}\ a_i\theta \to v_i \quad \text{para } i = 1 \ldots k$$

$$\textbf{(13)} \quad P \vdash_{SC} v_k \to t\theta$$

Además existe una prueba para

$$P \vdash_{SC} (\theta(X)\ a_1\theta \to v_1, \ldots, v_{k-1}\ a_k\theta \to v_k)$$

cuyo APA coincide salvo por la raíz con el de $P \vdash_{SC} \theta(X)\ \overline{a}_k\theta \to t\theta$. Nótese que no incluimos la aproximación $v_k \to t\theta$ ya que al ser una aproximación entre patrones no contribuye al APA.

b) A partir de (12) probaremos que se verifica para $i = 1 \ldots k$ que existe una sustitución ν_i tal que

$$\textbf{(14)} \qquad P_{\mathcal{A}} \vdash_{SC} (C_i(\theta \biguplus \nu_i),\, v_{i-1}\ u_i(\theta \biguplus \nu_i) \to v_i)$$

donde $dom(\nu_i) \subseteq V_i$, con V_i el conjunto de variables nuevas introducidas durante el aplanamiento $a_i \Rightarrow_{\mathcal{A}} (u_i\,;\, C_i)$. Veremos además que

$$\begin{aligned} P_{\mathcal{A}} \vdash_{SC} \quad & (\ C_1(\theta \uplus \nu_1),\, \theta(X)\ u_1(\theta \uplus \nu_1) \to v_1, \ldots, \\ & C_k(\theta \uplus \nu_k),\, v_{k-1}\ u_k(\theta \uplus \nu_k) \to v_k\ ) \end{aligned}$$

puede probarse con un APA que difiere sólo en la raíz del de $P \vdash_{SC} \theta(X)\ \overline{a}_k\theta \to t\theta$ (en realidad también difiere en el orden de los hijos, pero no tenemos esta diferencia en cuenta, tal y como indicamos en la observación 3.3.1, pág. 63).

c) De (13), por el apartado 4 de la proposición 3.2.1 al ser v_k un patrón se tiene que al tratarse de una aproximación entre patrones:

$$\textbf{(15)} \qquad P_{\mathcal{A}} \vdash_{SC} v_k \to t\theta$$

d) Definimos ahora

$$\nu = \nu_1 \biguplus \ldots \biguplus \nu_k \biguplus \{X_1 \mapsto v_1, \ldots, X_k \mapsto v_k\}$$

Se verifica que el dominio de ν está formado sólo por variables nuevas introducidas durante el aplanamiento (11). Veamos que además ν satisface las condiciones del teorema.

- $P_{\mathcal{A}} \vdash_{SC} (C_i,\ X_{i-1}\ u_i \to X_i)(\theta \biguplus \nu)$ para $i = 1 \ldots k$. Se cumple por (12) al ser
$$\begin{aligned} (C_i,\ X_{i-1}\ u_i \to X_i)(\theta \biguplus \nu) \quad &= \\ (C_i(\theta \biguplus \nu),\ v_{i-1}\ u_i(\theta \biguplus \nu) \to v_i) \quad &= \\ (C_i(\theta \biguplus \nu_i),\ v_{i-1}\ u_i(\theta \biguplus \nu_i) \to v_i) & \end{aligned}$$
que se puede probar en $P_{\mathcal{A}}$ por (14). La igualdad $X_{i-1}(\theta \biguplus \nu) = v_{i-1}$ se cumple tanto si $i > 1$ (por la definición de ν) como cuando $i = 1$. En este último caso $X_0(\theta \biguplus \nu) = \theta(X) = v_0$, ya que hemos llamado v_0 a v y definido v como $\theta(X)$.
- $P_{\mathcal{A}} \vdash_{SC} (X_k \to t)(\theta \biguplus \nu)$. Se cumple por (15) al ser $(X_k \to t)(\theta \biguplus \nu) = v_k \to t\theta$.

Además por la observación asociada a (14) el APA correspondiente a la prueba de todas las condiciones reunidas difiere tan sólo en la raíz del de $P \vdash_{SC} \theta(X)\ \overline{a}_k\theta \to t\theta$. Nótese que en dicha observación no se incluye $(X_k \to t)(\theta \biguplus \nu)$ pero que la demostración de esta observación no contribuye al APA al ser una aproximación entre patrones.

Para que esta demostración sea correcta debemos aún probar el paso b), es decir el paso de la fórmula (12) a (14).

Observamos en primer lugar que en (12) debe ser $v_i \neq \bot$ ya que en otro caso sería fácil probar por inducción que $v_j \equiv \bot$ para todo $j > i$ y por (13) se llegaría a $t\theta \equiv \bot$ en contra de lo que estamos suponiendo.

Nos fijamos en el último paso de la demostración (12) y distinguimos dos casos, según la regla *SC* aplicada:

- Regla DC. En este caso será $v_{i-1} = h\ \overline{s}_{n-1}$ para $h \in FS \cup DC$ y $s_i \in Pat_\perp$ para $i = 1 \ldots n-1$, mientras que $v_i \equiv h\ \overline{t}_n$ con $t_i \in Pat_\perp$ para $i = 1 \ldots n$, y el último paso de la demostración será de la forma:

$$\textbf{(16)} \quad \frac{s_1 \to t_1\ \ \ldots\ \ s_{n-1} \to t_{n-1}\ \ \ a_i\theta \to t_n}{h\ \overline{s}_{n-1}\ \ a_i\theta \to h\ \overline{t}_n}$$

De (16) se tiene que $P \vdash_{SC} a_i\theta \to t_n$, y aplicando un razonamiento análogo al que justifica (9) y (10) se llega a:

$$\textbf{(17)} \quad P_{\mathcal{A}} \vdash_{SC} C_i(\theta \biguplus \nu_i)$$
$$\textbf{(18)} \quad P_{\mathcal{A}} \vdash_{SC} u_i(\theta \biguplus \nu_i) \to t_n$$

con $dom(\nu_i) \subseteq V_{a_i}$ y V_{a_i} las variables nuevas introducidas durante el aplanamiento de a_i. Entonces podemos probar que se cumple

$$\textbf{(19)} \quad P_{\mathcal{A}} \vdash_{SC} v_{i-1}\ u_i(\theta \biguplus \nu_i) \to v_i$$

con una demostración SC que finaliza en un paso DC análogo a (16) pero sustituyendo $a_i\theta$ por $u_i(\theta \biguplus \nu_i)$ tanto en las premisas como en la conclusión. Las premisas $s_i \to t_i$ para $i = 1 \ldots m$ se cumplen en $P_{\mathcal{A}}$ al hacerlo en P por (16), debido al apartado 4 de la proposición 3.2.1 (pág. 57), mientras que $u_i(\theta \biguplus \nu_i) \to t_{m+1}$ se cumple por (18). Uniendo (17) y (19) obtenemos (14), el resultado buscado.

- Regla $AR{+}FA$. Si se ha aplicado esta regla debe ser $v_{i-1} \equiv f\ \overline{s}_{n-1}$ con $f \in FS^n$, ya que v_{i-1} es un patrón pero $v_{i-1}\ a_i\theta$ es activa. El último paso de la demostración de $c\theta$ será entonces:

$$\textbf{(20)} \quad \frac{s_1 \to t_1\ \ \ldots\ \ s_{n-1} \to t_{n-1}\ \ \ a_i\theta \to t_n\ \ \boxed{\dfrac{C \quad r \to s}{f\ \overline{t}_n \to s}}\ \ \ s \to v_i}{f\ \overline{s}_{n-1}\ a_i\theta \to v_i}$$

Donde $f\ \overline{t}_n \to r \Leftarrow C$ es la instancia de programa utilizada, es decir

$$(f\ \overline{t}_n \to r \Leftarrow C) = (f\ \overline{t}'_n \to r' \Leftarrow C')\sigma$$

para alguna $(f\ \overline{t}'_n \to r' \Leftarrow C') \in P$ que podemos suponer con sus variables renombradas y nuevas con respecto a las variables de $c\theta$. También podemos suponer, sin pérdida de generalidad, que las variables del dominio de σ son variables de la regla de programa y por tanto distintas de las variables del dominio de θ y de las de $c\theta$.

Entonces:

- Al tenerse $a_i\theta \to t_n$ como premisa (20) podemos aplicar aplicar un razonamiento análogo al que justifica (9) y (10) para llegar a que debe existir una sustitución ν_i cuyo dominio son las variables nuevas definidas durante el aplanamiento $a_i \Rightarrow_{\mathcal{A}} (u_i\,;\, C_i)$ tal que se verifican (17) y (18).
- Análogamente a como se obtuvieron las fórmulas (8), (9) y (10) a partir de (7) se pueden obtener a partir de (20) las fórmulas:

$$\textbf{(21)} \quad P_{\mathcal{A}} \vdash_{SC} C_{\mathcal{A}}(\sigma \uplus \sigma_c)$$

$$\textbf{(22)} \quad P_{\mathcal{A}} \vdash_{SC} r_{\mathcal{A}}(\sigma \uplus \sigma_r) \to s$$

$$\textbf{(23)} \quad P_{\mathcal{A}} \vdash_{SC} C_r(\sigma \uplus \sigma_r)$$

donde $r' \Rightarrow_{\mathcal{A}} (r_{\mathcal{A}}\,;\, C_r)$ y $dom(\sigma_r) \subseteq V_r$ con V_r las variables nuevas producidas durante el aplanamiento de r, y $C' \Rightarrow_{\mathcal{A}} C_{\mathcal{A}}$ y $dom(\sigma_c) \subseteq V_c$, con V_c las variables nuevas introducidas durante el aplanamiento de C'.

Consideramos el resultado de aplanar la regla de programa utilizada en (20), $(f\ \bar{t}'_n \to r' \Leftarrow C') \in P$. Por la regla de aplanamiento (PL_{10}):

$$(f\ \bar{t}'_n \to r' \Leftarrow C') \Rightarrow_{\mathcal{A}} (f\ \bar{t}'_n \to r_{\mathcal{A}} \Leftarrow C_{\mathcal{A}}, C_r)$$

Definimos una sustitución $\sigma' = \sigma \uplus \sigma_c \uplus \sigma_r$, y comprobamos que entonces la instancia $(f\ \bar{t}'_n \to r_{\mathcal{A}} \Leftarrow C_r, C_{\mathcal{A}})\sigma'$ puede utilizarse para probar $v_{i-1}\ u_i(\theta \uplus \nu_i) \to v_i$

$$\dfrac{s_1 \to t'_1\sigma'\ \ldots\ s_{n-1} \to t'_{n-1}\sigma'\ \ u_i(\theta \uplus \nu_i) \to t'_n\sigma' \qquad \dfrac{C_r\sigma'\ C_{\mathcal{A}}\sigma'\ r_{\mathcal{A}}\sigma' \to s}{\boxed{f\ \bar{t}_n \to s}} \qquad s\ \to v_i}{f\ \bar{s}_{n-1}\ u_i(\theta \uplus \nu_i) \to v_i}$$

Donde las premisas $t_i \to t'_i\sigma'$ se cumple en $P_{\mathcal{A}}$ por el apartado 2 de la proposición 3.2.1 al ser $t'_i\sigma' = t_i$, $u_i(\theta \uplus \nu_i) \to t'_n\sigma'$ se puede probar por (18), y $C_r\sigma'$, $C_{\mathcal{A}}\sigma'$ y $r_{\mathcal{A}}\sigma' \to s$ se deducen respectivamente de (21), (23) y (22) al ser σ' extensión de las sustituciones empleadas en cada una de estas fórmulas. Finalmente $s \to v_i$ por el apartado 4 de la proposición 3.2.1, al ser premisa de (20).

Por tanto $P_{\mathcal{A}} \vdash_{SC} v_{i-1}\ u_i(\theta \uplus \nu_i) \to v_i$, lo que completa, junto con (17), el resultado (14).

Continuamos la demostración para el resto de las posibles formas de c.

5. $c \equiv e \to t$ con $t \neq \bot$, e no plana y $e \equiv h\ \bar{e}_m$, $e_i \in Exp_\bot$ para $i = 1 \ldots m$, $m \geq 0$, e no activa.

   Debe ser entonces $t \equiv h\ \bar{t}_m$ y la demostración $P \vdash_{SC} (e \to t)\theta$ acabará con una aplicación de la regla DC de la forma:

$$\textbf{(24)}\quad \frac{e_1\theta \to t_1\theta \ \ldots \ e_m \to t_m\theta}{h\ \overline{e}_m\theta \to h\ \overline{t}_m\theta}$$

Al no ser e plana para el aplanamiento de $e \to t$ se utilizará la regla (PL_9). A su vez para el aplanamiento de e se empleará (PL_5), por lo que:

$$(h\ \overline{e}_m \to t) \Rightarrow_{\mathcal{A}} (C_1, \ldots, C_m, h\ \overline{u}_m \to t)$$

es decir, $C \equiv (C_1, \ldots, C_m, h\ \overline{u}_m \to t)$ con $e_i \Rightarrow_{\mathcal{A}} (u_i\,;\, C_i)$ para $i = 1 \ldots m$. Por hipótesis de inducción sobre las premisas de (24) se verifica que debe existir un μ_i para cada $i = 1 \ldots m$ tal que $P_{\mathcal{A}} \vdash_{SC} A_i(\theta \uplus \mu_i)$, con $(e_i \to t_i) \Rightarrow_{\mathcal{A}} A_i$, admitiendo esta prueba un APA que sólo difiere del de $P \vdash_{SC} e_i\theta \to t_i\theta$ en la raíz.

Fijándonos en las posibles formas de A_i, según si e_i es plana o no, es posible hacer un razonamiento similar al que justifica las fórmulas (9) y (10) y llegar a que existen sustituciones ν_i cuyo dominio está incluido en el conjunto de variables nuevas introducidas durante el aplanamiento $e_i \Rightarrow_{\mathcal{A}} (u_i\,;\, C_i)$ tales que para $i = 1 \ldots m$:

$$\textbf{(25)}\quad P_{\mathcal{A}} \vdash_{SC} u_i(\theta \uplus \nu_i) \to t_i\theta$$

$$\textbf{(26)}\quad P_{\mathcal{A}} \vdash_{SC} C_i(\theta \uplus \nu_i)$$

Como en casos anteriores definimos $\nu = \nu_1 \uplus \ldots \uplus \nu_m$ y probamos $P_{\mathcal{A}} \vdash_{SC} C(\theta \uplus \nu)$ examinando cada componente de C.

- $P_{\mathcal{A}} \vdash_{SC} C_i(\theta \uplus \nu)$. Cierto por (26) al ser ν extensión de toda ν_i con $i = 1 \ldots m$.
- $P_{\mathcal{A}} \vdash_{SC} (h\ \overline{u}_m \to h\ \overline{t}_m)(\theta \uplus \nu)$. Esto se puede probar con una demostración que finaliza en un paso DC y donde las premisas $P_{\mathcal{A}} \vdash_{SC} (u_i \to t_i)(\theta \uplus \nu)$ se tienen de (25), ya que $(u_i \to t_i)(\theta \uplus \nu) = u_i(\theta \uplus \nu) \to t_i\theta$, por estar el dominio de ν formado por variables nuevas y no poder por tanto afectar a t_i. Además el APA de esta prueba coincide con el (24) salvo por la raíz, al suceder lo mismo con sus premisas por hipótesis de inducción y no corresponder éstas con la conclusión de un paso $AR + FA$.

6. $c \equiv e \to t$ con $t \neq \bot$, e no plana y $e \equiv f\ \overline{e}_n\ \overline{a}_k$, $e_i, a_j \in Exp_\bot$ para $i = 1 \ldots m$, $j = 1 \ldots k$ respectivamente, $k, m \geq 0$, e activa.

   La demostración SC de $c\theta$ en P debe finalizar entonces con una aplicación de la regla $AR + FA$ de la forma:

$$\textbf{(27)}\quad \frac{e_1\theta \to t_1\theta \ldots e_n\theta \to t_n\theta \qquad \dfrac{C \quad r \to s}{\boxed{f\ \overline{t}_n \to s}}\ s\ \overline{a}_k\theta \to t\theta}{f\ \overline{e}_n\theta\ \overline{a}_k\theta \to t\theta}$$

El razonamiento es similar al del apartado dedicado a c flexible y $c\theta$ activa, por lo que mostramos resumidos los pasos a seguir:

- Suponemos que hemos utilizado una instancia de la regla de programa $(f\ \overline{t}'_n \to r' \Leftarrow C') \in P$ y que σ es la sustitución utilizada para obtener la instancia $(f\ \overline{t}_n \to r \Leftarrow C)$ empleada en la regla. Además suponemos que el resultado de aplanar esta regla de programa es $(f\ \overline{t}'_n \to r_{\mathcal{A}} \Leftarrow C_r, C_{\mathcal{A}}) \in P_{\mathcal{A}}$. Definimos $\theta' = \theta \uplus \sigma \uplus \{Y \mapsto s\}$.
- Las premisas $e_i\theta \to t_i$ se pueden reescribir como $(e_i \to t'_i)\theta'$, por lo que se puede aplicar la hipótesis de inducción. Por una razonamiento análogo al que se utiliza para (9) y (10) se llega a que debe existir una sustitución ν_i para cada $i = 1 \ldots m$ tal que

  (28) $P_{\mathcal{A}} \vdash_{SC} u_i(\theta \uplus \nu_i) \to t_i$

  (29) $P_{\mathcal{A}} \vdash_{SC} C_i(\theta \uplus \nu_i)$

  Además cada prueba

  $$P_{\mathcal{A}} \vdash_{SC} (u_i(\theta \bigcupplus \nu_i) \to t_i, C_i(\theta \bigcupplus \nu_i))$$

  para $i = 1 \ldots m$ admite un APA que sólo difiere en la raíz de un APA para $P \vdash_{SC} e_i\theta \to t_i$.
- De las premisas C y $r \to s$ se deducen análogamente las fórmulas (21), (22) y (23).
- Finalmente la premisa $s\ \overline{a}_k\theta \to t\theta$, puede reescribirse como $(Y\ \overline{a}_k \to t)\theta$, y suponiendo $k > 0$ (en otro caso este apartado no es necesario) por hipótesis de inducción se tiene que existe ν_s tal que

  $$P_{\mathcal{A}} \vdash_{SC} (C'_1,\ Y\ u'_1 \to Y_2,\ \ldots, C'_k,\ Y_{k-1}\ u'_k \to Y_k, Y_k \to t)(\theta \bigcupplus \nu_s)$$

  donde $Y_1, \ldots Y_k$ son variables nuevas y $a_i \Rightarrow_{\mathcal{A}} (u'_i ; C'_i)$ para $i = 1 \ldots k$. Si llamamos V_i a las variables definidas en el aplanamiento de a_i podemos definir entonces para $i = 1 \ldots k$ podemos definir $\nu'_i = \nu_s \restriction V_i$. Si además llamamos Y_0 a Y y definimos $\nu' s = \nu_s \uplus \{Y_0 \mapsto s\}$ podemos comprobar que se verifican para $i = 1 \ldots k$:

  (30) $P_{\mathcal{A}} \vdash_{SC} \nu'_s(Y_{i-1})\ u'_i(\theta \uplus \nu'_i) \to \nu'_s(Y_i)$

  (31) $P_{\mathcal{A}} \vdash_{SC} C'_i(\theta \uplus \nu'_i)$

  y además

  (32) $P_{\mathcal{A}} \vdash_{SC} \nu'_s(Y_k) \to t\theta$

El aplanamiento de c es, aplicando las reglas (PL_9) y (PL_6):

$$C \equiv C_1, \ldots C_n, f\ \overline{u}_n \to S, D, u \to t$$

con $e_i \Rightarrow_{\mathcal{A}} (u_i\,;\, C_i)$ para $i = 1 \ldots n$ y $S\ \overline{a}_k \Rightarrow_{\mathcal{A}} (u\,;\, D)$.

Si $k = 0$ definimos

$$\nu = \nu_1 \biguplus \ldots \biguplus \nu_m \biguplus \{S \mapsto s\}$$

ya que el aplanamiento de $S\ \overline{a}_k$ no introduce ninguna variable nueva y $u \equiv S$. Si por el contrario $k > 0$ tenemos que debe ser:

$$S\ \overline{a}_k \Rightarrow_{\mathcal{A}} (Z_k\,;\ C'_1, S\ u'_1 \rightarrow Z_1, \ldots C_k, Z_{k-1}\ u'_k \rightarrow Z_k)$$

con $u \equiv Z_k$, $Z_1 \ldots Z_k$ variables nuevas y $a_i \Rightarrow_{\mathcal{A}} (u'_i\,;\, C'_i)$ para $i = 1 \ldots k$. En este caso definimos

$$\begin{aligned}\nu = \quad & \nu_1 \biguplus \ldots \biguplus \nu_m \biguplus \{S \mapsto s\} \biguplus \nu'_1 \biguplus \ldots \biguplus \nu'_k \biguplus \\ & \{Z_1 \mapsto \nu_s(Y_1),\ \ldots,\ Z_k \mapsto \nu_s(Y_k)\}\end{aligned}$$

que, al igual que en el caso $k = 0$ da valores a todas las variables nuevas producidas durante el aplanamiento de c. Comprobamos ahora que $P_{\mathcal{A}} \vdash_{SC} C(\theta \biguplus \nu)$, examinando cada componente de C:

- $P_{\mathcal{A}} \vdash_{SC} C_i(\theta \biguplus \nu)$ para $i = 1 \ldots n$. De (29).
- $P_{\mathcal{A}} \vdash_{SC} (f\ \overline{u}_n \rightarrow S)(\theta \biguplus \nu)$. Por la definición de ν hay que probar: $P_{\mathcal{A}} \vdash_{SC} (f\ \overline{u}_n(\theta \biguplus \nu) \rightarrow s)$, para lo que utilizamos la instancia que se obtiene al aplicar la sustitución $\sigma' = \sigma \biguplus \sigma_c \biguplus \sigma_r$ al resultado de aplanar la regla de programa usada en (27), es decir $(f\ \overline{t}'_n \rightarrow r_{\mathcal{A}} \Leftarrow C_r, C_{\mathcal{A}})$. Entonces la demostración acaba en un paso $AR + FA$ como el siguiente:

$$\frac{u_1(\theta \biguplus \nu) \rightarrow t'_1\sigma' \ldots u_n(\theta \biguplus \nu) \rightarrow t'_n\sigma' \qquad \dfrac{C_{\mathcal{A}}\sigma'\ \ C_r\sigma'\ r_{\mathcal{A}}\sigma' \rightarrow s}{\boxed{f\ \overline{t'}_n\sigma \rightarrow s}} \qquad s\ \rightarrow s}{f\ \overline{u}_n(\theta \biguplus \nu)\ \rightarrow s}$$

  y las premisas se pueden probar a partir de (28) para $u_i(\theta \biguplus \nu) \rightarrow t'_i\sigma'$ (al ser $t'_i\sigma' = t_i$ y ν extensión de ν_i), de (17), (19) y (18) para $C_{\mathcal{A}}\sigma'$, $C_r\sigma'$ y $r_{\mathcal{A}}\sigma' \rightarrow s$ respectivamente al ser σ' extensión de la sustitución que aparece en cada una de estas fórmulas, y finalmente $s \rightarrow s$ es siempre válida, tal y como establece el apartado 2 de la proposición 3.2.1.
- $P_{\mathcal{A}} \vdash_{SC} C'_i(\theta \biguplus \nu)$ para $i = 1 \ldots k$ con $k > 0$. De (31).
- $P_{\mathcal{A}} \vdash_{SC} (Z_{i-1}\ u'_i \rightarrow Z_i)(\theta \biguplus \nu)$ para $i = 1 \ldots k$ con $k > 0$, llamando Z_0 a S. El resultado se tiene de (30) al tenerse por definición de ν que para $i = 0 \ldots k$ se cumple $Z_i(\theta \biguplus \nu) = \nu'_s(Y_i)$ y por tanto

$$(Z_{i-1}\ u'_i \rightarrow Z_i)(\theta \biguplus \nu) = \nu'_s(Y_{i-1})\ u'_i(\theta \biguplus \nu) \rightarrow \nu'_s(Y_i)$$

  y ser ν extensión de ν'_i para $i = 1 \ldots k$.

- $P_{\mathcal{A}} \vdash_{SC} (u \to t)(\theta \uplus \nu)$. Si $k = 0$ entonces $u \equiv S$ y $(u \to t)(\theta \uplus \nu) = s \to t\theta$, y está aproximación se cumple en $P_{\mathcal{A}}$ al hacerlo en P (es premisa de (27) para $k = 0$), por el apartado 4 de la proposición 3.2.1.

  Si $k > 0$ se cumple por (32), al ser $u \equiv Z_k$ y $Z_k(\theta \uplus \nu) = \nu'_s(Y_k)$.

$\Leftarrow$) La hipótesis es ahora $P_{\mathcal{A}} \vdash_{SC} C(\theta \uplus \nu)$ para algún ν tal que $dom(\nu) \subseteq V_{\mathcal{A}}$, donde $V_{\mathcal{A}}$ es el conjunto de variables nuevas introducidas en el aplanamiento $c \Rightarrow_{\mathcal{A}} C$, y queremos probar que se verifica $P \vdash_{SC} c\theta$ y que esta prueba admite un APA que sólo difiere en la raíz de un APA para $P_{\mathcal{A}} \vdash_{SC} C(\theta \uplus \nu)$.

Si $P_{\mathcal{A}} \vdash_{SC} C(\theta \uplus \nu)$ es que existe una demostración en SC para cada una de las condiciones atómicas en $C(\theta \uplus \nu)$. Sean $T_1, \ldots T_m$ los árboles representando dichas demostraciones. Vamos a hacer la demostración por inducción completa sobre la expresión $n(T_1) + \cdots + n(T_m)$, donde la función $n(T)$ representa el número de nodos del árbol T.

Como C proviene del aplanamiento de c distinguimos casos según la forma de c, tal y como hicimos para la otra implicación. En los casos en los que el razonamiento sea muy similar a la de implicación en sentido contrario éste se mostrará abreviado, omitiendo los detalles repetidos. En particular se ha evitado la repetición del proceso de aplanamiento de c y la relación entre los APAs.

1. $c \equiv e == e'$ para $e, e' \in Exp_{\perp}$.

   La demostración es similar a la de la implicación contraria. Partimos de que se cumplen

   $$P_{\mathcal{A}} \vdash_{SC} C_1(\theta \uplus \nu) \quad P_{\mathcal{A}} \vdash_{SC} C_2(\theta \uplus \nu) \quad P_{\mathcal{A}} \vdash_{SC} e_{\mathcal{A}}(\theta \uplus \nu) == e'_{\mathcal{A}}(\theta \uplus \nu)$$

   donde $e \Rightarrow_{\mathcal{A}} (e_{\mathcal{A}}\,;\, C_1)$ y $e' \Rightarrow_{\mathcal{A}} (e'_{\mathcal{A}}\,;\, C_2)$. Al tratarse de una igualdad estricta, la demostración en SC de $e_{\mathcal{A}}(\theta \uplus \nu) == e'_{\mathcal{A}}(\theta \uplus \nu)$ acabará en un paso JN como el descrito en (6). Podemos entonces aplicar la hipótesis de inducción a las premisas de dicho paso JN y obtener la prueba en P de $e\theta \to t$ y de $e'\theta \to t$, que a su vez servirán de premisas para probar $c\theta$ en P finalizando en un paso como el descrito en la fórmula (1).

2. $c \equiv e \to t$ con $t \equiv \perp$ o bien $t\theta \equiv \perp$.

   Entonces $P \vdash_{SC} (e \to t)\theta$ se cumple trivialmente.

3. $c \equiv e \to t$ con $t \neq \perp$ y e plana.

   Si e es un patrón o $e = X\ t$ con $e\theta$ pasiva, se verifica el resultado por el mismo argumento que el utilizado para el mismo caso de la implicación en el otro sentido.

   La otra posibilidad es $e\theta \equiv f\ \overline{s}_n\theta$ con $f \in FS^n$ y $s_i \in Pat_{\perp}$. Nos fijamos en el último paso de la demostración SC, que corresponderá a un paso $AR + FA$ de la forma:

$$\textbf{(33)} \quad \frac{s_1\theta \to t_1 \ \ldots \ s_n\theta \to t_n \quad \dfrac{C \quad r \to s}{\boxed{f\ \overline{t}_n \to s}} \quad s \ \to t\theta}{f\ \overline{s}_n\theta \ \to t\theta}$$

Sabemos que cada regla en $P_{\mathcal{A}}$ tiene la estructura $(f\ \overline{t}'_n \to r_{\mathcal{A}} \Leftarrow C_r, C_{\mathcal{A}}) \in P_{\mathcal{A}}$ al corresponderse con el aplanamiento de alguna regla en P de la forma $(f\ \overline{t}'_n \to r' \Leftarrow C') \in P$ tal que $r' \Rightarrow_{\mathcal{A}} (r_{\mathcal{A}}\,;\, C_r)$ y $C' \Rightarrow_{\mathcal{A}} C_{\mathcal{A}}$. Así pues debe existir una sustitución σ utiliza para obtener la instancia de regla utiliza en (33) tal que

- $t_i = t'_i\sigma$ para $i = 1 \ldots n$.
- $C = (C_r\sigma, C_{\mathcal{A}}\sigma)$.
- $r = r_{\mathcal{A}}\sigma$.

Sea $V_\sigma = dom(\sigma)$, y sean $V_{r'}$, $V_{C'}$ dos conjuntos conteniendo las variables introducidas durante al aplanamiento de r' y de C' respectivamente.

Se verifica trivialmente que $\sigma = \sigma \restriction (V_\sigma - V_{C'}) \uplus \sigma \restriction V_{C'}$ de donde también se cumple

$$P_{\mathcal{A}} \vdash_{SC} C_{\mathcal{A}}(\sigma \restriction (V_\sigma - V_{C'}) \ \biguplus \ \sigma \restriction V_{C'})$$

Podemos aplicar la hipótesis de inducción, al ser obviamente $dom(\sigma \restriction V_{C'}) \subseteq V_{C'}$, y tener el árbol de la demostración de $C_{\mathcal{A}}$ menor número de nodos que el de (33) al ser un súbarbol de éste. Aplicando la hipótesis de inducción:

$$P \vdash_{SC} C'(\sigma \restriction (V_\sigma - V_{C'}))$$

Y teniendo en cuenta que las variables de $dom(V_{C'})$ no pueden ser variables de C':

$$\textbf{(34)} \qquad P \vdash_{SC} C'\sigma$$

Veamos ahora que se cumple igualmente

$$\textbf{(35)} \qquad P \vdash_{SC} r'\sigma \to t\theta$$

Para ello distinguimos tres posibilidades, según la forma de r':

- Si r' no es plana se tiene que aplicando la regla (PL_9):

  $$r' \to t\theta \Rightarrow_{\mathcal{A}} C_r,\ r_{\mathcal{A}} \to t\theta$$

  Y por un argumento similar al aplicado para obtener (34):

  $$\begin{array}{lll} \sigma & = \sigma \restriction (V_\sigma - V_{r'}) \ \biguplus \ \sigma \restriction V_{r'} & \text{de donde:} \\ P_{\mathcal{A}} & \vdash_{SC} (r_{\mathcal{A}} \to t\theta, C_r)(\sigma \restriction (V_\sigma - V_{r'}) \ \biguplus \ \sigma \restriction V_{r'}) & \text{y por h.i.:} \\ P & \vdash_{SC} (r' \to t\theta)(\sigma \restriction (V_\sigma - V_{r'})) & \end{array}$$

  que lleva directamente a (35), ya que σ no puede afectar a las variables de $t\theta$ y $r'(\sigma \restriction (V_\sigma - V_{r'})) = r'\sigma$

- Si r' es una patrón el resultado se cumple trivialmente al ser por el lema A.2.2 (pág. 194) $r_{\mathcal{A}} \equiv r'$ y C_r la condición vacía. La condición vacía se cumple siempre, y al poder probar $r_{\mathcal{A}}\sigma \to t\theta$ en $P_{\mathcal{A}}$ con $r_{\mathcal{A}}\sigma$ necesariamente un patrón, también será posible hacerlo en P por el apartado 4 de la proposición 3.2.1 (pág. 57).
- Por último si r' es plana pero no un patrón, aplicando bien la regla (PL_4) o (PL_6), según la forma de r' se tiene que en $r' \Rightarrow_{\mathcal{A}} (r_{\mathcal{A}}\,;\, C_r)$ se cumplen $r_{\mathcal{A}} \equiv Y$ y $C_r \equiv r' \to Y$ donde Y es una variable nueva, la única introducida durante el aplanamiento de r', es decir $V_{r'} = \{Y\}$. Ahora, al ser C_r y $r \to t\theta$ premisas de (33) se tiene entonces que

  $$\textbf{(36)} \quad P_{\mathcal{A}} \vdash_{SC} (r' \to Y)\sigma$$
  $$\textbf{(37)} \quad P_{\mathcal{A}} \vdash_{SC} Y\sigma \to t\theta$$

  Al ser r' plana se tiene por (PL_{10}) que: $(r' \to Y) \Rightarrow_{\mathcal{A}} (r' \to Y)$, aplanamiento en el que no se ha generado ninguna variable nueva. Entonces de (36), y por hipótesis de inducción tomando $\nu \equiv id$ se tiene:

  $$\textbf{(38)} \qquad P \vdash_{SC} (r' \to Y)\sigma$$

  Por otro lado de (37) por el apartado 4 de la proposición 3.2.1 se tiene $P \vdash_{SC} Y\sigma \to t\theta$ que por el apartado 1 de la proposición 3.2.1 (pág. 57) significa que se tiene $Y\sigma \sqsupseteq t\theta$, lo que combinado con (38) y aplicando el apartado 1 de la proposición 3.3.2 (pág. 63) llegamos a (35).

Ahora utilizando la instancia $(f\ \overline{t}'_n \to r' \Leftarrow C')\sigma$ de $(f\ \overline{t}'_n \to r' \Leftarrow C') \in P$ probar $P \vdash_{SC} f\ \overline{s}_n\theta\ \to t\theta$ con una demostración cuyo último paso es una aplicación de la regla $AR + FA$ de la forma:

$$\textbf{(39)} \quad \frac{s_1\theta \to t'_1\sigma\ \ \ldots\ \ s_n\theta \to t'_n\sigma \qquad \dfrac{C'\sigma \quad r'\sigma \to s}{\boxed{f\ \overline{t}'_n\sigma \to s}} \qquad s\ \to t\theta}{f\ \overline{s}_n\theta\ \to t\theta}$$

ya que las premisas se cumplen por (33) (recordando que $t_i = t'_i\sigma$) gracias al apartado 4 de la proposición 3.2.1 (pág. 57), o por (34) y (35) para las premisas $r'\sigma \to t\theta$ y $C'\sigma$, respectivamente.

4. $c \equiv e \to t$ con $t \neq \bot$, $t\theta \neq \bot$, e no plana y $e \equiv X\ \overline{a}_k$, $a_i \in Exp_\bot$ para $i = 1 \ldots k$, $k > 1$ y por tanto e flexible.

   Partimos ahora de que para algún $\nu \in Subst_\bot$ con $dom(\nu) \subseteq V_{\mathcal{A}}$ se verifican

   $$\textbf{(40)} \quad P_{\mathcal{A}} \vdash_{SC} (X_{i-1}\ u_i \to X_i)(\theta \biguplus \nu) \qquad \text{para } i = 1 \ldots k$$
   $$\textbf{(41)} \quad P_{\mathcal{A}} \vdash_{SC} C_i(\theta \biguplus \nu) \qquad \text{para } i = 1 \ldots k$$
   $$\textbf{(42)} \quad P_{\mathcal{A}} \vdash_{SC} (X_k \to t)(\theta \biguplus \nu)$$

con $a_i \Rightarrow_{\mathcal{A}} (u_i\,;\, C_i)$ para $i = 1 \ldots k$, y tenemos que comprobar que entonces

$$P \vdash_{SC} \ (X\ \overline{a}_k\ \rightarrow t)\theta$$

Vamos a hacer la demostración siguiendo estos 3 pasos:

a) Probaremos que $P \vdash_{SC} \ \theta(X)\ \ a_1\theta \rightarrow \nu(X_1)$ y que para $i = 2 \ldots k$ se verifica $P \vdash_{SC} \ \nu(X_{i-1})\ \ a_i\theta \rightarrow \nu(X_i)$.

b) A partir de (42), por el apartado 1 de la proposición 3.2.1 (pág. 57), tenemos $\nu(X_k) \sqsupseteq t\theta$ y del apartado 1 de la proposición 3.3.2 (pág. 63) se deduce que $P \vdash_{SC} \nu(X_k) \rightarrow t\theta$.

c) Combinando los dos puntos anteriores y por el lema A.2.8 (pág. 205), llamando v a $\theta(X)$, v_i a $\nu(X_i)$ para $i = 1 \ldots k$, e_i a $a_i\theta$ para $i = 1 \ldots k$, y s a $t\theta$, tendremos que se verifica $P \vdash_{SC} v\ \overline{e}_k \rightarrow s$, o lo que es lo mismo $P \vdash_{SC} (X\ \overline{a}_k \rightarrow t)\theta$ como queríamos demostrar.

Por tanto basta con probar el primer punto, que podemos abreviar diciendo que para $i = 1 \ldots k$ se debe cumplir

$$\textbf{(43)} \qquad P \vdash_{SC} (X_{i-1}\ a_i \rightarrow X_i)(\theta \biguplus \nu)$$

ya que $(X_i)(\theta \biguplus \nu) = X\theta$ si $i = 0$ y $(X_i)(\theta \biguplus \nu) = \nu(X_i)$ para $i > 0$. Por tanto una forma de escribir (43) es también:

$$P \vdash_{SC} X_{i-1}(\theta \biguplus \nu)\ a_i\theta \rightarrow \nu(X_i)$$

Consideramos un i cualquiera. Combinando (40), (41) y (42) se cumple por hipótesis:

$$\textbf{(44)} \qquad P \vdash_{SC} (X_{i-1}\ u_i \rightarrow X_i,\ X_k \rightarrow t)(\theta \biguplus \nu)$$

Nos fijamos en el último paso de la demostración (40) y distinguimos dos casos según la regla aplicada, teniendo en cuenta que el lado derecho $X_i(\theta \biguplus \nu)$ no puede ser $\perp$ ya que en ese caso sería fácil probar por inducción que $X_j(\theta \biguplus \nu) = \perp$ para $j \geq i$ y en particular que $X_k(\theta \biguplus \nu) = \perp$, lo que por (42) nos llevaría a $t\theta = \perp$, caso que ya ha sido tratado.

- Regla *DC*. En este caso debe ser $X_{i-1}(\theta \biguplus \nu) = h\ \overline{s}_m$ y y $X_i(\theta \biguplus \nu) = h\ \overline{t}_{m+1}$. El último paso será de la forma:

$$\textbf{(45)} \quad \frac{s_1 \rightarrow t_1\ \ \ldots\ \ s_m \rightarrow t_m\ \ \ u_i(\theta \biguplus \nu) \rightarrow t_{m+1}}{h\ \overline{s}_m\ u_i(\theta \biguplus \nu) \rightarrow h\ \overline{t}_{m+1}}$$

De (45) se tiene $P_{\mathcal{A}} \vdash_{SC} u_i(\theta \biguplus \nu) \to t_{m+1}$ que, definiendo una sustitución θ' como $\theta \biguplus \{Y \mapsto t_{m+1}\}$ con Y una variable nueva se puede reescribir como $P_{\mathcal{A}} \vdash_{SC} (u_i \to Y)(\theta' \biguplus \nu)$. De (41), al ser Y nueva se puede escribir: $P_{\mathcal{A}} \vdash_{SC} C_i(\theta' \biguplus \nu)$, y combinando ambas:

$$\textbf{(46)} \qquad P_{\mathcal{A}} \vdash_{SC} (C_i, u_i \to Y)(\theta' \bigbiguplus \nu)$$

El número total de nodos en las demostraciones de (46) es menor que el número total de nodos de (44), al faltar en (46) la aproximación $X_1 \to t$, aparecer C_i en ambas y ser el árbol de la demostración de $(u_i \to Y)(\theta' \biguplus \nu)$ subárbol de $(X_{i-1}\ u_i \to X_i)(\theta \biguplus \nu)$ por (47). Además se tiene que

$$(a_i \to Y) \ \Rightarrow_{\mathcal{A}} (C_i, u_i \to Y)$$

lo que se obtiene aplicando la regla (PL_9) si a_i no es un patrón, o (PL_{10}) si $a_i \in Pat_\perp$, ya que en este caso por el lema A.2.2 (pág. 194) $a_i \Rightarrow_{\mathcal{A}} (a_i\,;\,)$, y por tanto $(C_i, u_i \to Y) = a_i \to Y$ tal y como indica la regla (PL_{10}). Por tanto, y por hipótesis de inducción se cumple $P \vdash_{SC} (a_i \to Y)\theta'$, o lo que es lo mismo por la definición de θ':

$$\textbf{(47)} \qquad P \vdash_{SC} a_i\theta \to t_{m+1}$$

los que nos permite probar

$$P \vdash_{SC} (X_{i-1}\ a_i \to X_i)(\theta \bigbiguplus \nu)$$

con una demostración cuyo último paso corresponde a una aplicación de la regla DC de la forma:

$$\textbf{(48)} \quad \frac{s_1 \to t_1 \ \ldots \ s_m \to t_m \quad a_i\theta \to t_{m+1}}{h\ \overline{s}_m\ a_i\theta \to h\ \overline{t}_{m+1}}$$

donde las premisas se pueden probar en P, bien por corresponder a aproximaciones entre patrones que ya eran premisas de (45), aplicando el apartado 1 de la proposición 3.3.2 (pág. 63), o por (47) en el caso de $a_i\theta \to t_{m+1}$.

- Regla $AR + FA$. Si se ha aplicado esta regla para finalizar (40) es que se cumple $X_{i-1}(\theta \biguplus \nu) = f\overline{s}_{n-1}$ con $f \in FS^n$, y el último paso será de la forma:

$$\textbf{(49)} \quad \frac{s_1 \to t_1 \ \ldots \ s_{n-1} \to t_{n-1} \quad u_i(\theta \biguplus \nu) \to t_n \quad \dfrac{\dfrac{C \quad r \to s}{f\ \overline{t}_n \to s}}{} \quad s \ \to \nu(X_i)}{f\ \overline{s}_{n-1} u_i(\theta \biguplus \nu) \ \to \nu(X_i)}$$

Entonces se puede probar en P la aproximación

$$(X_{i-1}\ a_i \to X_i)(\theta \bigbiguplus \nu) = f\ \overline{s}_{n-1}\ a_i\theta \to \nu(X_i)$$

con una demostración que acaba en una aplicación de la regla AR+FA:

$$\frac{s_1 \to t'_1\sigma \ \ldots \ s_{n-1} \to t'_{n-1}\sigma \ \ a_i\theta \to t'_n\sigma \ \ \boxed{\begin{array}{c}\dfrac{C'\sigma \quad r'\sigma \to s}{f\ \overline{t'}_n\sigma \to s)}\end{array}} \ \ s \ \to \nu(X_i)}{f\ \overline{s}_{n-1}\ a_i\theta \to \nu(X_i)}$$

donde la elección y justificación de la instancia de regla utilizada $(f\ \overline{t}'_n \to r' \Leftarrow C')\sigma$, se ha hecho siguiendo un razonamiento idéntico al que conduce de (33) a (39) y que no repetimos. Dicho razonamiento prueba además que las premisas $s_i \to t'_i\sigma$ para $i = 1 \ldots n-1$, así como $C'\sigma$, $r'\sigma \to s$ y $s \to \nu(X_i)$ tienen una demostración SC en el programa P (en (33) y (39) aparece $t\theta$ en lugar de $\nu(X_i)$, pero esto no cambia el razonamiento al ser ambas expresiones patrones).

Finalmente podemos probar que se cumple $P \vdash_{SC} a_i\theta \to t'_n\sigma$ recordando que $t'_n\sigma = t_n$, que se tiene de (49) que $P_{\mathcal{A}} \vdash_{SC} u_i(\theta \biguplus \nu) \to t_n$, y aplicando un razonamiento análogo al que conduce de (45) a (47).

5. $c \equiv e \to t$ con $t \neq \bot$, $t\theta \neq \bot$, e no plana y $e \equiv h\ \overline{e}_m$, $e_i \in Exp_\bot$ para $i = 1 \ldots m$, $m \geq 0$, e no activa.

   Partimos en esta ocasión de que se cumple

   $$\textbf{(50)} \qquad P_{\mathcal{A}} \vdash_{SC} (C_1, \ldots C_m, h\ \overline{u}_m \to t)(\theta \biguplus \nu)$$

   y queremos llegar a

   $$\textbf{(51)} \qquad P_{\mathcal{A}} \vdash_{SC} (h\ \overline{e}_m \to t)(\theta \biguplus \nu)$$

   Al ser e no activa será o bien $h \in DC^n$ con $n \geq m$ o $h \in FS^n$ con $n > m$. En cualquier caso $h\ \overline{u}_m$ será también pasiva. Nos fijamos en el último paso de la demostración de $P_{\mathcal{A}} \vdash_{SC} (h\ \overline{u}_m \to t)(\theta \biguplus \nu)$, que al ser $t(\theta \biguplus \nu) = t\theta \neq \bot$ ha de corresponder a un paso DC de la forma:

   $$\textbf{(52)} \quad \frac{u_1(\theta \biguplus \nu) \to t_1 \ \ldots \ u_m(\theta \biguplus \nu) \to t_m}{(h\ \overline{u}_m)(\theta \biguplus \nu) \to h\ \overline{t}_m}$$

   donde $t(\theta \biguplus \nu) = t\theta = h\ \overline{t}_m$. A partir de (52) $P_{\mathcal{A}} \vdash_{SC} u_i(\theta \biguplus \nu) \to t_i$ para $i = 1 \ldots m$, y como además por (50) $P_{\mathcal{A}} \vdash_{SC} C_i(\theta \biguplus \nu)$ podemos realizar un razonamiento análogo al que lleva de (45) a (47) para obtener $P_{\mathcal{A}} \vdash_{SC} e_i\theta \to t_i$ para $i = 1 \ldots m$, lo que permitirá probar (51) acabando con la aplicación de una regla DC:

   $$\frac{e_1\theta \to t_1 \ \ldots \ e_m\theta \to t_m}{(h\ \overline{e}_m)\theta \to h\ \overline{t}_m}$$

6. $c \equiv e \to t$ con $t \neq \bot$, $t\theta \neq \bot$, e no plana y $e \equiv f\ \overline{e}_n\ \overline{a}_k$, $e_i, a_j \in Exp_\bot$ para $i = 1 \ldots m$, $j = 1 \ldots k$ respectivamente, $k, m \geq 0$, e activa.

   Partimos de que para un cierto ν se cumple:

$$\textbf{(53)} \qquad P_{\mathcal{A}} \vdash_{SC} (C_1, \ldots C_n, f\ \overline{u}_n \to S, D, u \to t)(\theta \biguplus \nu)$$

   con $e_i \Rightarrow_{\mathcal{A}} (u_i\,;\, C_i)$ para $i = 1 \ldots n$ y $S\ \overline{a}_k \Rightarrow_{\mathcal{A}} (u\,;\, D)$, que es el resultado de aplanar $f\ \overline{e}_n\ \overline{a}_k \to t$ según las reglas de aplanamiento de la figura 4.1 (pág. 77), y queremos probar

$$\textbf{(54)} \qquad P \vdash_{SC} (f\ \overline{e}_n\ \overline{a}_k \to t)\theta$$

   Llamando s a $S(\theta \biguplus \nu) = \nu(S)$ tenemos que el último paso de la demostración de $P_{\mathcal{A}} \vdash_{SC} (f\ \overline{u}_n \to S)(\theta \biguplus \nu)$ debe ser de la forma:

$$\textbf{(55)} \quad \frac{u_1(\theta \biguplus \nu) \to t'_1\sigma \ldots u_n(\theta \biguplus \nu) \to t'_n\sigma \qquad \dfrac{C_{\mathcal{A}}\sigma \quad C_r\sigma\ r_{\mathcal{A}}\sigma \to s'}{\boxed{f\ \overline{t'}_n\sigma \to s'}} \qquad s' \to s}{f\ \overline{u}_n(\theta \biguplus \nu) \to s}$$

   donde se ha utilizado la instancia $(f\ \overline{t}'_n \to r_{\mathcal{A}} \Leftarrow C_r, C_{\mathcal{A}})\sigma$ de la regla de programa $(f\ \overline{t}'_n \to r_{\mathcal{A}} \Leftarrow C_r, C_{\mathcal{A}}) \in P_{\mathcal{A}}$ obtenida al aplanar la regla $(f\ \overline{t}'_n \to r' \Leftarrow C') \in P$. Obsérvese que para que (55) sea correcta hay que asegurar que $s \neq \bot$, lo que haremos más adelante.

   Suponiendo cierto (55) podremos probar

$$\textbf{(56)} \qquad P \vdash_{SC} f\ \overline{e}_n\theta \to s$$

   acabando con una aplicación de la regla $AR + FA$ que utiliza la instancia $(f\ \overline{t}'_n \to r' \Leftarrow C')\sigma$ y que tiene la forma:

$$\frac{e_1\theta \to t'_1\sigma\ \ldots\ e_n\theta \to t'_n\sigma \quad \dfrac{C'\sigma \quad r'\sigma \to s'}{\boxed{f\ \overline{t'}_n\sigma \to s'}} \quad s' \to s}{f\ \overline{e}_n\theta \to s}$$

   La justificación de las premisas se obtiene por razonamientos ya utilizados en casos anteriores y que resumimos a continuación:

   - $P \vdash_{SC} e_i\theta \to t'_i\sigma$ para $i = 1 \ldots n$. Como para $i = 1 \ldots n$ se cumplen

$$\begin{array}{ll} P_{\mathcal{A}} \vdash_{SC} u_i(\theta \biguplus \nu) \to t'_i\sigma & \text{(por (55)} \\ P_{\mathcal{A}} \vdash_{SC} C_i(\theta \biguplus \nu) & \text{(por (53))} \end{array}$$

     Por el mismo argumento que lleva de (45) a (47) obtenemos la justificación de $e_i\theta \to t'_i\sigma$ en P.

- $P \vdash_{SC} (C'\sigma,\, r'\sigma \to s')$. La justificación es idéntica a la que se ha utilizado para deducir (34) y (35) a partir de (33).
- $P \vdash_{SC} s' \to s$. Cierto por ser s' un patrón y tenerse de (55) $P_{\mathcal{A}} \vdash_{SC} s' \to s$, aplicando el apartado 1 de la proposición 3.3.2 (pág. 63).

Para finalizar la demostración distinguimos tres posibilidades:

- $k = 0$. Entonces (53) se puede reescribir como:

  $$P_{\mathcal{A}} \vdash_{SC} (C_1,\, \ldots,\, C_n,\, f\ \overline{u}_n \to S, S \to t)(\theta \biguplus \nu)$$

  Ya que por el lema A.2.2 (pág. 194) se tiene $S \Rightarrow_{\mathcal{A}} (S\,;\,)$, es decir en (53) D es la condición vacía y $u \equiv S$. Aplicando las sustituciones a cada componente y recordando que hemos llamado s a $S(\theta \biguplus \nu)$, tenemos:

  $$P_{\mathcal{A}} \vdash_{SC} (C_1(\theta \biguplus \nu),\, \ldots,\, C_n(\theta \biguplus \nu),\, f\ \overline{u}_n(\theta \biguplus \nu) \to s, s \to t\theta)$$

  En particular nos fijamos ahora en

  $$\textbf{(57)} \qquad P_{\mathcal{A}} \vdash_{SC} s \to t\theta$$

  de donde se deduce $s \neq \bot$, ya que en otro caso $t\theta = \bot$, en contra de lo que estamos suponiendo. Esto justifica la aplicación de la regla $AR + FA$ en (55).

  Por otra parte de (57) se tiene por el apartado 1 de la proposición 3.2.1 $s \sqsupseteq t\theta$, lo que combinado con (56), por el apartado 1 de la proposición 3.3.2 (pág. 63) nos lleva a $P \vdash f\ \overline{e}_\theta \to t\theta$, o, lo que es lo mismo, a (54) para el caso $k = 0$, es decir al resultado buscado.

- $k > 0$. y $S\ \overline{a}_k$ no plana. Se tiene que

  $$S\ \overline{a}_k \Rightarrow_{\mathcal{A}} (Z_k\,;\ C'_1, S\ u'_1 \to Z_1, \ldots C'_k, Z_{k-1}\ u'_k \to Z_k)$$

  con $u \equiv Z_k$, $Z_1 \ldots Z_k$ variables nuevas y $a_i \Rightarrow_{\mathcal{A}} (u'_i\,;\, C'_i)$ para $i = 1 \ldots k$, lo que nos permite escribir la hipótesis (53) como:

  $$\textbf{(58)} \quad P_{\mathcal{A}} \vdash_{SC} (\ \begin{array}{l} C_1, \ldots C_n,\, f\ \overline{u}_n \to S, \\ C'_1,\, S\ u'_1 \to Z_1,\, \ldots,\, C'_k,\, Z_{k-1}\ u'_k \to Z_k, \\ Z_k \to t)(\theta \biguplus \nu) \end{array}$$

  Si s es $\bot$, es decir si $S(\theta \biguplus \nu) = \bot$, resulta fácil demostrar por inducción que $Z_i(\theta \biguplus \nu)$ para $i = 1 \ldots k$, y de aquí por la última aproximación que $t\theta = \bot$ en contra de lo que estamos suponiendo, por lo que $s \neq \bot$.

  Probamos a continuación que se verifica:

  $$\textbf{(59)} \qquad P \vdash_{SC} s\ \overline{a}_k\theta \to t\theta$$

Lo que junto a (56), y por apartado 3 de la proposición 3.3.2 nos lleva al resultado deseado establecido en (54).

Para probar (59) recordamos que estamos suponiendo que $S\ \overline{a}_k$ no es plana, por lo que se cumple:

$$(S\ \overline{a}_k \to t) \Rightarrow_{\mathcal{A}} (C'_1,\ S\ u'_1 \to Z_1,\ \ldots,\ C'_k,\ Z_{k-1}\ u'_k \to Z_k,\ Z_k \to t)$$

donde podemos suponer sin perdida de generalidad que las variables nuevas introducidas coinciden con las de (58) (en otro caso es posible aplicar un renombramiento y el razonamiento que sigue podría realizarse igualmente). Estas condiciones son parte de (58), es decir se verifica:

$$\textbf{(60)} \qquad P_{\mathcal{A}} \vdash_{SC} \quad (C'_1,\ S\ u'_1 \to Z_1,\ \ldots,\ C'_k,\ Z_{k-1}\ u'_k \to Z_k,\ Z_k \to t)(\theta \biguplus \nu)$$

Ahora bien, en ν están todas las variables obtenidas durante el aplanamiento de $S\ \overline{a}_k \to t$, pero "sobra"S, ya que $S \in dom(\nu)$ y S no es una variable nueva obtenida durante este aplanamiento. Definimos entonces

$$\begin{aligned} \theta' &= \theta \biguplus \{S \mapsto \nu(S)\} \\ \nu' &= \nu \restriction (dom(\nu) - \{S\}) \end{aligned}$$

Como evidentemente $\theta \biguplus \nu = \theta' \biguplus \nu'$ podemos reescribir (60) como

$$\textbf{(61)} \qquad P_{\mathcal{A}} \vdash_{SC} \quad (C'_1,\ S\ u'_1 \to Z_1,\ \ldots,\ C'_k,\ Z_{k-1}\ u'_k \to Z_k,\ Z_k \to t)(\theta' \biguplus \nu')$$

y ahora sí que el dominio de ν' está formado únicamente por las variables nuevas introducidas durante el aplanamiento de $S\ \overline{a}_k \to t$. Además en (58) existen necesariamente condiciones que no están entre las del aplanamiento, por ejemplo $f\ \overline{u}_n \to S$, por lo que la cantidad total de nodos en los árboles de prueba de (60) (y por tanto de (61)) será menor que la de (58) y podemos aplicar la hipótesis de inducción, que nos lleva a: $P \vdash (S\ \overline{a}_k \to t)\theta'$, que aplicando la definición de θ' se convierte en (59).

- $k > 0$ y $S\ \overline{a}_k$ es plana entonces $k = 1$ y $a_1 \in Pat_\perp$. La hipótesis (53) se escribe en este caso como (58) aplicado al caso $k = 1$:

$$P_{\mathcal{A}} \vdash_{SC} (C_1,\ \ldots C_n,\ f\ \overline{u}_n \to S,\ C'_1,\ S\ u'_1 \to Z_1,\ Z_1 \to t)(\theta \bigl\lfloor\!\biguplus\!\bigr\rfloor \nu)$$

O lo que es lo mismo, teniendo en cuenta que $a_1 \Rightarrow_{\mathcal{A}} (a_1\,;\ )$ por el lema A.2.2 y que por tanto $u'_1 \equiv a_1$ y C'_1 es la condición vacía:

$$\textbf{(62)} \qquad P_{\mathcal{A}} \vdash_{SC} (C_1,\ \ldots C_n,\ f\ \overline{u}_n \to S,\ S\ a_1 \to Z_1,\ Z_1 \to t)(\theta \bigl\lfloor\!\biguplus\!\bigr\rfloor \nu)$$

Separamos las dos últimas aproximaciones:

$$\textbf{(63)} \quad P_{\mathcal{A}} \vdash_{SC} (S\ a_1 \to Z_1)(\theta \biguplus \nu)$$
$$\textbf{(64)} \quad P_{\mathcal{A}} \vdash_{SC} (Z_1 \to t)(\theta \biguplus \nu)$$

En este caso también se cumple que $s \neq \bot$ como se requería para (55), utilizando el mismo argumento que en caso anterior, porque en otro caso de (63) se tendría $\nu(Z_1) \equiv \bot$ y entonces de (64) se deduciría $t\theta \equiv \bot$ en contra de lo que estamos suponiendo.

Llamando θ' a $\theta \biguplus \nu$ se puede reescribir (63) como

$$\textbf{(65)} \qquad P_{\mathcal{A}} \vdash_{SC} (S\ a_1 \to Z_1)(\theta' \biguplus id)$$

y al cumplirse además $S\ a_1 \to Z_1 \Rightarrow_{\mathcal{A}} S\ a_1 \to Z_1$ aplicando la regla (PL_{10}) al ser $S\ a_1$ plana, por lo que durante este aplanamiento no se introduce ninguna variable nueva. Por otra parte, el número de nodos del árbol representando la prueba SC (63), que es obviamente el de (65), será menor que la suma de los nodos de todos los árboles de prueba SC en (62), ya que la aproximación de (63) está en (62) y además en (62) hay otros árboles de prueba a considerar, al menos el de $P_{\mathcal{A}} \vdash_{SC} f\ \overline{u}_n \to S$. Podemos entonces aplicar la hipótesis de inducción y obtenemos $P_{\mathcal{A}} \vdash_{SC} (S\ a_1 \to Z_1)\theta'$ que aplicando la definición de θ' se transforma en:

$$\textbf{(66)} \qquad s\ a_1\theta \to \nu(Z_1)$$

De (63) por el apartado 1 de la proposición 3.2.1 (pág. 57) se cumple $\nu(Z_1) \sqsupseteq t\theta$, y de aquí y de (66), y por el apartado 1 de la proposición 3.3.2 (pág. 63)

$$P \vdash_{SC} s\ a_1\theta \to t\theta$$

lo que a su vez combinado con (56), y por el apartado 3 de la proposición 3.3.2 nos lleva a concluir

$$P \vdash_{SC} f\ \overline{e}_n\theta\ a_1\theta \to t\theta$$

es decir (54) para el caso $k = 1$, como deseábamos.

□

La demostración del teorema es sencilla a partir del lema anterior:

a) Distinguimos dos casos:

- Si e es plana pero no es un patrón. Entonces, por la definición 4.3.1 (pág. 75) hay dos posibilidades: $e \equiv X\,t$ con $X \in Var$, $t \in Pat_\bot$, o $e \equiv f\ \overline{t}_n$ con $f \in FS^n$ y $t_i \in Pat_\bot$ para $i = 1 \ldots n$.

Recordando que por el lema A.2.2 (pág. 194) se tiene $t \Rightarrow_{\mathcal{A}} (t\,;\,)$ para todo patrón t, es sencillo comprobar que de las reglas de aplanamiento de la figura 4.1 (pág. 77) y en particular de la regla (PL_4) si $e \equiv X\ t$ o (PL_6) si $e \equiv f\ \bar{t}_n$, se tiene que $e \Rightarrow_{\mathcal{A}} (Y\,;\, e \to Y)$, con Y la única variable nueva producida en el aplanamiento de e. Por tanto $e_{\mathcal{A}} \equiv Y$ y $C_{\mathcal{A}} \equiv e \to Y$. Probamos las dos implicaciones por separado:

$\Rightarrow$) Partimos de $P \vdash_{SC}\ e\theta \to t$ y queremos probar $P_{\mathcal{A}} \vdash_{SC}\ (e_{\mathcal{A}}(\theta \biguplus \nu) \to t,\ C_{\mathcal{A}}(\theta \biguplus \nu))$.

Definimos $\theta' \equiv \theta \biguplus \{Y \mapsto t\}$. Entonces

$$P \vdash_{SC}\ C_{\mathcal{A}}\theta'$$

al ser $C_{\mathcal{A}}\theta' = e\theta \to t$. Ahora bien, $C_{\mathcal{A}}$ es una condición atómica, por lo que podemos aplicar el lema A.2.9 a $P \vdash_{SC}\ C_{\mathcal{A}}\theta'$. Además al ser e plana se tiene por la regla (PL_{10}) que $C_{\mathcal{A}} \Rightarrow_{\mathcal{A}} C_{\mathcal{A}}$, sin que en este aplanamiento se produzca ninguna variable nueva, por lo que del citado lema se deduce:

$$P_{\mathcal{A}} \vdash_{SC}\ C_{\mathcal{A}}(\theta' \bigbiguplus id)$$

al ser la sustitución identidad (id) la única con dominio el conjunto vacío.

Evidentemente $(\theta' \biguplus id) = \theta'$, por lo que se cumple

$$P_{\mathcal{A}} \vdash_{SC}\ C_{\mathcal{A}}\theta'$$

y llamando ν a la sustitución $\{Y \mapsto t\}$ tenemos por la definición de θ'

$$P_{\mathcal{A}} \vdash_{SC}\ C_{\mathcal{A}}(\theta \bigbiguplus \nu)$$

Además, según indica el lema, es posible encontrar para esta demostración un APA que sólo difiere de un APA para $P \vdash_{SC}\ C_{\mathcal{A}}\theta'$, es decir de un APA para $P \vdash_{SC}\ e\theta \to t$, en la raíz.

Por otra parte se tiene

$$P_{\mathcal{A}} \vdash_{SC}\ (e_{\mathcal{A}}(\theta \bigbiguplus \nu) \to t$$

ya que $e_{\mathcal{A}}(\theta \biguplus \nu) = Y(\theta \biguplus \nu) = t$ y $t \to t$ siempre se puede probar (apartado 2 de la proposición 3.2.1, pág. 57). Se cumple entonces

$$P_{\mathcal{A}} \vdash_{SC}\ (e_{\mathcal{A}}(\theta \bigbiguplus \nu) \to t,\ C_{\mathcal{A}}(\theta \bigbiguplus \nu))$$

Además la prueba de $(e_{\mathcal{A}}(\theta \biguplus \nu) \to t$ (es decir de $t \to t$) no contribuye al APA de esta demostración ya que no puede usar la regla $AR + FA$ por tratarse de una aplicación entre patrones. Por tanto cualquier APA de esta prueba sólo diferirá del de $P_{\mathcal{A}} \vdash_{SC}\ C_{\mathcal{A}}(\theta \biguplus \nu)$ en la raíz, y como hemos visto es último sólo difiere de un APA para $P \vdash_{SC}\ e\theta \to t$, en la raíz, completando el resultado.

$\Leftarrow$) Partimos ahora de $P_{\mathcal{A}} \vdash_{SC} (e_{\mathcal{A}}(\theta \uplus \nu) \to t, C_{\mathcal{A}}(\theta \uplus \nu))$ con $\nu \subseteq \{Y\}$. Además $Y \notin dom(\theta)$ al ser Y una variable nueva. Al ser $e_{\mathcal{A}} = Y$ se cumple $e_{\mathcal{A}}(\theta \uplus \nu) \to t = \nu(Y) \to t$ y por el apartado 1 de la proposición 3.2.1 al poder probarse en P esta aproximación se tiene $\nu(Y) \sqsupseteq t$. Análogamente, $C_{\mathcal{A}}(\theta \uplus \nu) = e\theta \to \nu(Y)$ y de ambas por el apartado 1 de la proposición 3.3.2 (pág. 63)

$$P_{\mathcal{A}} \vdash_{SC} e\theta \to t$$

Definiendo $\theta' = \theta \uplus \{Z \mapsto t\}$, con Z una variable nueva, se tiene

$$P_{\mathcal{A}} \vdash_{SC} (e \to Z)(\theta' \bigcupplus id)$$

y al ser $e \to Z \Rightarrow_{\mathcal{A}} e \to Z$ podemos aplicar el lema A.2.9 en sentido inverso para deducir $P \vdash_{SC} (e \to Z)\theta'$ es decir

$$P \vdash_{SC} e\theta \to t$$

con la relación de los APAs que queríamos probar.

- Si e no es plana o e es un patrón. Probamos ambas implicaciones simultáneamente.

  Sea Y una variable nueva. Definimos, como en el caso anterior, una sustitución $\theta' \equiv \theta \uplus \{Y \mapsto t\}$. El aplanamiento de $e \to Y$ será:

  $$e \to Y \Rightarrow_{\mathcal{A}} (C_{\mathcal{A}}, e_{\mathcal{A}} \to Y)$$

  donde $e \Rightarrow_{\mathcal{A}} (e_{\mathcal{A}}\,;\, C_{\mathcal{A}})$. Esto es cierto si e no es plana aplicando la regla (PL_9), pero también si es plana y en particular un patrón, ya que por el lema A.2.2 (pág. 194) en este caso $e \Rightarrow_{\mathcal{A}} (e\,;\,)$ y por tanto

  $$(C_{\mathcal{A}}, e_{\mathcal{A}} \to Y) = e \to Y$$

  es decir que se cumple $e \to Y \Rightarrow_{\mathcal{A}} e \to Y$ tal y como pide la regla de aplanamiento (PL_{10}) que se encarga de las aproximaciones con lado izquierdo plano.

  Entonces $P \vdash_{SC} e\theta \to t$ sii $P \vdash_{SC} (e \to Y)\theta'$ sii, por el lema A.2.9 aplicado a la condición atómica $(e \to Y)\theta'$, existe una sustitución ν con $dom(\nu) \subseteq V_{\mathcal{A}}$ (ya que las variables nuevas introducidas durante el aplanamiento de $e \to Y$ coinciden con las del aplanamiento de e) tal que se verifica

  $$P_{\mathcal{A}} \vdash_{SC} (C_{\mathcal{A}},\ e_{\mathcal{A}} \to Y)(\theta' \bigcupplus \nu)$$

  que es el resultado deseado, ya que ni $C_{\mathcal{A}}$ ni $e_{\mathcal{A}}$ pueden contener la variable Y, al provenir ambos del aplanamiento de e y ser Y nueva, y por tanto se cumplen $C_{\mathcal{A}}(\theta' \uplus \nu) = C_{\mathcal{A}}(\theta \uplus \nu)$, $(e_{\mathcal{A}} \to Y)(\theta' \uplus \nu) = e_{\mathcal{A}}(\theta \uplus \nu) \to t$. Además el lema nos indica que la relación entre ciertos APAs para $P \vdash_{SC} e\theta \to t$ y $P_{\mathcal{A}} \vdash_{SC} (C_{\mathcal{A}},\ e_{\mathcal{A}} \to Y)(\theta' \uplus \nu)$ es la que indica el enunciado.

b) Demostramos las dos implicaciones por separado:

$\Rightarrow$) Suponemos $P \vdash_{SC} C\theta$. C debe ser de la forma $c_1, \ldots c_m$, donde c_i es una condición atómica para cada i tal que $1 \leq i \leq m$. Entonces se debe cumplir $P \vdash_{SC} c_i\theta$ para cada $c_i \in C$. Por el lema A.2.9 (pág. 208) existe una substitución ν_i tal que $P_{\mathcal{A}} \vdash_{SC} C_i(\theta \biguplus \nu_i)$ donde $c_i \Rightarrow_{\mathcal{A}} C_i$ para $i = 1 \ldots m$ y ambas demostraciones admiten APAs que sólo difieren en la raíz. Además para todo $1 \leq i, j \leq m$ tales que $i \neq j$ se cumple $dom(\nu_i) \cap dom(\nu_j) = \emptyset$ al corresponder el dominio de cada sustitución a las variables nuevas introducidas durante el aplanamiento de su correspondiente condición atómica.

Definimos ahora

$$\nu = \nu_1 \biguplus \ldots \biguplus \nu_m$$

que verifica $C_i(\theta \biguplus \nu) = C_i(\theta \biguplus \nu_i)$ para $i = 1 \ldots m$. Entonces se cumple la conclusión del teorema $P_{\mathcal{A}} \vdash_{SC} C_{\mathcal{A}}(\theta \biguplus \nu)$, ya que a partir del lema A.2.3 (pág. 195) se tiene $C_{\mathcal{A}} \equiv C_1 \ldots C_m$. Además cualquier APA de $P_{\mathcal{A}} \vdash_{SC} C_{\mathcal{A}}(\theta \biguplus \nu)$ tendrá como raíz $C_{\mathcal{A}}(\theta \biguplus \nu)$ y como subárboles hijos los de los APAs de $C_i(\theta \biguplus \nu)$ por lo que se cumple la relación entre los APAs indicada en el enunciado.

$\Leftarrow$) Partimos de que existe una sustitución ν cuyo dominio sólo incluye las variables nuevas producidas durante el aplanamiento de C y que verifica

$$P_{\mathcal{A}} \vdash_{SC} C_{\mathcal{A}}(\theta \biguplus \nu)$$

Por el lema A.2.3 (pág. 195) tenemos que, al igual que en el caso anterior $C_{\mathcal{A}} \equiv C_1 \ldots C_m$, donde $C \equiv c_1 \ldots c_m$ y $c_i \Rightarrow_{\mathcal{A}} C_i$ para todo $i = 1 \ldots m$. Por tanto $P_{\mathcal{A}} \vdash_{SC} C_i(\theta \biguplus \nu)$ para $i = 1 \ldots m$, y por el lema A.2.9 se verifica entonces $P \vdash_{SC} c_i\theta$ para $i = 1 \ldots m$, o lo que es lo mismo $P \vdash_{SC} C\theta$. Del lema se tiene que además un APA para $P \vdash_{SC} C\theta$ se obtiene, salvo por la raíz, a partir de los APAs de $P_{\mathcal{A}} \vdash_{SC} C_i(\theta \biguplus \nu)$ para $i = 1 \ldots m$.

■

A.2.5. Demostración del Teorema 4.3.5

Comprobamos este resultado, cuyo enunciado se puede encontrar en la página 81.

Partimos de $C \equiv c_1, \ldots c_n$, con c_i condiciones atómicas. Entonces basta con comprobar que para un i cualquiera $1 \leq i \leq n$ se verifica

$$P \vdash_{SC} c_i \qquad \text{sii} \qquad P_{\mathcal{A}} \vdash_{SC} c_i$$

.

Para ello distinguimos el caso en que c_i es una aproximación y cuando es una igualdad estricta.

a) c_i es una aproximación $e \rightarrow t$. Demostramos cada una de las implicaciones que componen el resultado

$\Rightarrow$) Utilizamos inducción completa sobre la profundidad de la demostración de $P \vdash_{SC} e \rightarrow t$. Nos fijamos en el último paso de la demostración que puede corresponder a una regla:

- BT, RR o DC. En cualquiera de estos casos se puede construir una demostración acabando con un paso equivalente para $P_{\mathcal{A}} \vdash_{SC} e \to t$. Nótese que en el caso de DC las premisas tendrán una prueba en $P_{\mathcal{A}}$ al tenerla en P, por hípótesis de inducción, lo que permitirá completar la demostración.

- $AR + FA$. Entonces $e \equiv f\ \overline{e}_n\ \overline{a}_k$, con un último paso es de la forma:

$$\textbf{(1)}\quad \frac{e_1 \to t_1\theta\ \ldots\ e_n \to t_n\theta \qquad \dfrac{C\theta \quad r\theta \to s}{\boxed{f\ \overline{t}_n \to s}} \qquad s\ \overline{a}_k \to t}{f\ \overline{e}_n\ \overline{a}_k \to t}$$

con $(f\ \overline{t}_n \to r \Leftarrow C) \in P$. Obsérvese que hemos hecho explícita la sustitución θ que proporciona la instancia de la regla en (1).

Consideramos ahora el resultado de aplanar la regla de programa utilizada en el paso anterior mediante la regla de aplanamiento (PL_1):

$$\textbf{(2)}\qquad (f\ \overline{t}_n \to u \Leftarrow C_r, C_{\mathcal{A}}) \in P_{\mathcal{A}}$$

con $r \Rightarrow_{\mathcal{A}} (u\,;\, C_r)$ y $C \Rightarrow_{\mathcal{A}} C_{\mathcal{A}}$. Ahora bien:

- Al tenerse $P \vdash_{SC} r\theta \to s$ por (1) y ser $r \Rightarrow_{\mathcal{A}} (u\,;\, C_r)$ podemos aplicar el apartado a) del teorema 4.3.4 (pág. 80), que nos garantiza la existencia de una substitución ν_1 cuyo dominio son únicamente las variables nuevas introducidas en el aplanamiento de r y tal que se verifican:

$$\textbf{(3)}\quad P_{\mathcal{A}} \vdash_{SC} u(\theta \uplus \nu_1) \to s$$
$$\textbf{(4)}\quad P_{\mathcal{A}} \vdash_{SC} C_r(\theta \uplus \nu_1)$$

- También por (1) tenemos que se cumple $P \vdash_{SC} C\theta$ y como $C \Rightarrow_{\mathcal{A}} C_{\mathcal{A}}$, podemos aplicar el apartado b) del mismo teorema 4.3.4 por lo que existirá una substitución ν_2 con dominio está formado únicamente por las variables nuevas introducidas durante el aplanamiento de C y tal que:

$$\textbf{(5)}\qquad P_{\mathcal{A}} \vdash_{SC} C_{\mathcal{A}}(\theta \uplus \nu_2)$$

Definimos ahora $\nu = \nu_1 \uplus \nu_2$. Entonces podemos construir una demostración para $P_{\mathcal{A}} \vdash_{SC} e \to t$ acabando también con una aplicación de regla $AR + FA$. En este caso la regla utilizada será la regla aplanada (2), y la instancia dada por la sustitución $(\theta \uplus \nu)$:

$$\textbf{(6)}\quad \frac{e_1 \to t_1(\theta \uplus \nu)\ \ldots\ e_n \to t_n(\theta \uplus \nu) \qquad \dfrac{(C_r, C_{\mathcal{A}})(\theta \uplus \nu) \quad u(\theta \uplus \nu) \to s}{\boxed{f\ \overline{t}_n \to s}} \qquad s\ \overline{a}_k \to t}{f\ \overline{e}_n\ \overline{a}_k \to t}$$

Veamos que todas las premisas pueden probarse a partir de $P_{\mathcal{A}}$:

- $P_{\mathcal{A}} \vdash_{SC} e_i \to t_i(\theta \biguplus \nu)$ para $i = 1 \ldots n$. Esto es cierto a partir de (1) por hipótesis de inducción, porque el dominio de ν está compuesto únicamente por variables nuevas y por tanto $t_i(\theta \biguplus \nu) = t_i\theta$.
- $P_{\mathcal{A}} \vdash_{SC} (C_r, C_{\mathcal{A}})(\theta \biguplus \nu)$. En este caso hay que recordar que de todas las variables de ν sólo las de ν_1 pueden afectar a C_r y sólo las de ν_2 a $C_{\mathcal{A}}$ por lo que el resultado se tiene sólo si se cumple que:

  $$P_{\mathcal{A}} \vdash_{SC} \quad C_r(\theta \biguplus \nu_1)$$
  $$P_{\mathcal{A}} \vdash_{SC} \quad C_{\mathcal{A}}(\theta \biguplus \nu_2)$$

  que corresponden, respectivamente, a (4) y (5).
- $P_{\mathcal{A}} \vdash_{SC} u(\theta \biguplus \nu) \to s$. Por un razonamiento análogo se tiene que

  $$u(\theta \bigcupplus \nu) = u(\theta \bigcupplus \nu_1)$$

  y que por tanto el resultado se cumple al coincidir con (3).
- $P_{\mathcal{A}} \vdash_{SC} s\ \overline{a}_k \to t$. Cierto por hipótesis de inducción, al ser esta aproximación premisa de (1).

$\Leftarrow$) Por inducción sobre la profundidad de la demostración de $P_{\mathcal{A}} \vdash_{SC} e \to t$. Nos fijamos en el último paso de la demostración SC y distinguimos casos según la regla aplicada:

- BT, RR o DC. Análogamente al la implicación en el sentido contrario, en cualquiera de estos casos se puede repetir la misma demostración en P, utilizando en el caso de DC la hipótesis de inducción.
- $AR + FA$. Entonces tenemos un último paso de la forma:

$$\textbf{(7)} \quad \frac{e_1 \to t_1\sigma \ \ldots\ e_n \to t_n\sigma \qquad \boxed{f\ \overline{t}_n \to s} \qquad s\ \overline{a}_k \to t}{f\ \overline{e}_n\ \overline{a}_k \to t} \qquad \frac{(C_r, C_{\mathcal{A}})\sigma \quad u\sigma \to s}{\boxed{f\ \overline{t}_n \to s}}$$

  donde σ representa la instancia de la regla de programa

  $$(f\ \overline{t}_n \to u \Leftarrow C_r, C_{\mathcal{A}}) \in P_{\mathcal{A}}$$

  utilizada en (7) que sabemos debe provenir del aplanamiento de una regla $(f\ \overline{t}_n \to r \Leftarrow C) \in P$, con $r \Rightarrow_{\mathcal{A}} (u\,;\, C_r)$ y $C \Rightarrow_{\mathcal{A}} C_{\mathcal{A}}$.

  Sea V_r el conjunto de variables nuevas introducidas durante el aplanamiento de r y V_c el conjunto de variables nuevas introducidas durante el aplanamiento de C. Sea $V_{\mathcal{A}} = V_r \cup V_c$. Llamando θ a $\sigma \restriction (dom(\sigma) - V_{\mathcal{A}})$ y ν a $\sigma \restriction V_{\mathcal{A}}$ Podemos escribir σ como $\theta \biguplus \nu$, lo que nos permite reescribir (7) como (6).

  Entonces:

- Se tiene $P \vdash_{SC} e_i \rightarrow t_i\theta$ para $i = 1 \ldots n$, por hipótesis de inducción al tenerse de (6) $P_{\mathcal{A}} \vdash_{SC} e_i \rightarrow t_i(\theta \uplus \nu)$ y ser $t_i(\theta \uplus \nu) = t_i\theta$ para $i = 1 \ldots n$.
- También por hipótesis de inducción $P \vdash_{SC} s\ \overline{a}_k \rightarrow t$.
- Se tiene $P \vdash_{SC} C_{\mathcal{A}}\theta$ por el apartado b) del teorema 4.3.4 (pág. 80) al tenerse de (6) $P_{\mathcal{A}} \vdash_{SC} C_{\mathcal{A}}(\theta \uplus \nu)$ y verificarse por la definición de ν que $C_{\mathcal{A}}(\theta \uplus \nu) = C_{\mathcal{A}}(\theta\ \uplus (\nu \restriction V_c))$, siendo $dom(\nu \restriction V_c) \subseteq V_c$.
- Análogamente se deduce $P \vdash_{SC} r\theta \rightarrow s$.

y con toda esta información podemos probar $f\ \overline{e}_n\ \overline{a}_k \rightarrow t$ en P con una demostración cuyo último paso coincide con el representado en (1).

b) Si c_i es una igualdad estricta, $c_i \equiv l == r$ tendremos que el último paso de la prueba de $P \vdash_{SC} l == r$ debe corresponder a una aplicación de la regla JN de la forma:

$$\textbf{(8)} \quad \frac{l \rightarrow t \qquad r \rightarrow t}{l == r}$$

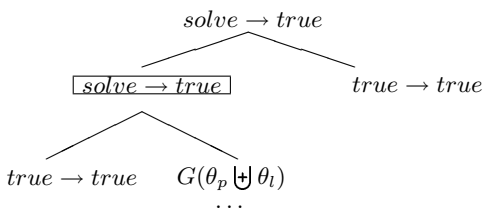
de donde se tiene $P \vdash_{SC} l \rightarrow t$, y $P \vdash_{SC} r \rightarrow t$. Por el apartado a) esto ocurre si y sólo si se tiene igualmente $P_{\mathcal{A}} \vdash_{SC} l \rightarrow t$, y $P_{\mathcal{A}} \vdash_{SC} r \rightarrow t$, lo que nos permite probar $P_{\mathcal{A}} \vdash_{SC} l == r$ con una demostración acabada en una paso JN idéntico a (8), pero en $P_{\mathcal{A}}$. Este razonamiento es válido en ambos sentidos y prueba las dos implicaciones del apartado. ■

A.2.6. Demostración de la proposición 4.4.1

Vamos a probar esta proposición, (enunciada en la página 90).

(i) Al ser GS correcto y tenerse por hipótesis $G \Vdash_{GS,P} \theta$ se tiene $P \vdash_{SC} G(\theta_p \uplus \theta_l)$. Entonces se cumple igualmente $P_{solve} \vdash_{SC} G(\theta_p \uplus \theta_l)$, ya que P_{solve} sólo se diferencia de P por la regla adicional de la función $solve$, que no influye en la demostración de $G(\theta_p \uplus \theta_l)$ ($solve$ es un símbolo nuevo con respecto a la signatura de P).

Entonces $P_{solve} \vdash_{SC} solve \rightarrow true$ se puede probar con una demostración del aspecto:

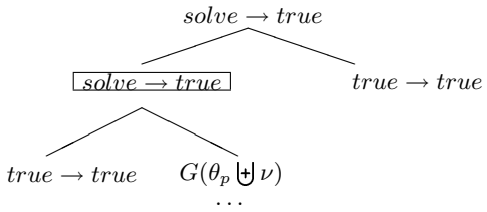


donde en el primer paso se ha utilizado la regla $AR{+}FA$ con una instancia dada por la sustitución θ_l de la regla para solve, es decir $(solve = true \Leftarrow G\theta_p)\theta_l$. Para aceptar que

esta es realmente la instancia utiliza hay que aclarar que $(G\theta_p)\theta_l = G(\theta_p \biguplus \theta_l)$. Esto se cumple porque por definición de solución $dom(\theta_p) \cap dom(\theta_l) = \emptyset$ y $ran(\theta_p) \cap dom(\theta_l) = \emptyset$.

Además, los dos nodos $true \rightarrow true$ pueden probarse aplicando la regla DC. Finalmente $P \vdash_{SC} G(\theta_p \biguplus \theta_l)$ se tiene, como hemos visto, de la corrección de GS. Esto prueba en efecto $P_{solve} \vdash_{SC} solve \rightarrow true$.

(ii) Sea ahora T a cualquier árbol de prueba para $P_{solve} \vdash_{SC} solve \rightarrow true$ cumpliendo que la sustitución ν utilizada para obtener la instancia cumple $dom(\nu) \subseteq def(G)$. Este árbol será de la forma



Ya que $G\theta_p\nu = G(\theta_p \biguplus \nu)$. Esto es cierto al verificarse que $dom(\nu) = dom(\theta_l) = def(G)$ y al tenerse que $G\theta_p\theta_l = G(\theta_p \biguplus \theta_l)$. Al estar la prueba de $G(\theta_p \biguplus \nu)$ en T, se tiene que $\theta' \equiv (\theta_p \biguplus \nu)$ es una solución para G. Vemos que además la relación entre los APAs es la indicada.

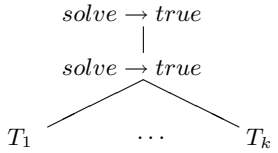
Vamos a representar por $T_{(n)}$ el subárbol de T cuya raíz es n. Vamos también a suponer que $T_{(G\theta')}$ tiene m árboles hijos $U_1, \ldots, U_m$, y que se verifica que

$$apa'(U_1) +\!+ \cdots +\!+ apa'(U_m) = [T_1, \ldots, T_k]$$

para ciertos árboles $T_1, \ldots, T_k$, con apa' definido tal como se muestra en la sección 3.3.2, pág. 61), y que por tanto, y a partir de esta misma definición, $apa'(T_{(G\theta')}) = [T_1, \ldots, T_k]$ ya que $G\theta'$ no puede corresponder con la conclusión de un paso FA. Entonces:

$$\begin{aligned}
\text{apa(T)} &= \text{árbol(raíz(T), apa'}(T_{(solve \rightarrow true)}) +\!+ \text{apa'}(T_{(true \rightarrow true)})\,) && = \\
&= \text{árbol}(solve \rightarrow true, \text{apa'}(T_{(solve \rightarrow true)})) && = \\
&= \text{árbol}(solve \rightarrow true, \text{árbol}(solve \rightarrow true, \\
&\qquad \text{apa'}(T_{(true \rightarrow true)}) +\!+ \text{apa'}(T_{G\theta'}))) && = \\
&= \text{árbol}(solve \rightarrow true, \text{árbol}(solve \rightarrow true, [T_1, \ldots, T_k]))
\end{aligned}$$

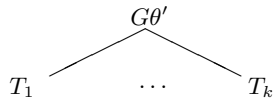
que corresponde con el APA



tal y como indica el enunciado. Por otra parte se cumple

apa($T_{G\theta'}$) = árbol[1]($G\theta'$, apa'(U_1) ++ ... ++ $apa'(U_m)$) = árbol($G\theta'$, $[T_1, \ldots, T_k]$)

que corresponde al árbol



lo que finaliza la prueba.

■

A.2.7. Demostración del Teorema 4.4.5

Como se indica tras el enunciado de este teorema (pág. 97), gracias al teorema 4.3.3 (pág. 80) basta con probar este otro resultado:

Seam P, $P_{\mathcal{A}}$ y $P^{\mathcal{T}}$ programas tales que $P \Rightarrow_{\mathcal{A}} P_{\mathcal{A}}$ y $P_{\mathcal{A}} \Rightarrow^{\mathcal{T}} P^{\mathcal{T}}$. Entonces si $P_{\mathcal{A}}$ está bien tipado $P^{\mathcal{T}}$ es un programa bien tipado.

Antes de probar este resultado vamos a establecer algunos resultados auxiliares, similares a los establecidos para la demostración del teorema 4.3.3 (demostración que se puede consultar en la página 197).

El siguiente lema prueba que si existe un contexto T que permite dar tipo a una patrón t con respecto a una signatura Σ, entonces el mismo contexto permite dar tipo a la expresión $t^{\mathcal{T}}$ con respecto a la signatura $\Sigma^{\mathcal{T}}$.

Lema A.2.10. *Sean t un patrón, Σ una signatura, T un contexto y τ un tipo tales que $(\Sigma, T) \vdash_{WT} t :: \tau$. Supongamos que $t \Rightarrow^{\mathcal{T}} t^{\mathcal{T}}$. Entonces existe un contexto T' tal que cumple $(\Sigma^{\mathcal{T}}, T') \vdash_{WT} t^{\mathcal{T}} :: \tau^{\mathcal{T}}$. Además $dom(T') = dom(T)$, con $T'(X) = (T(X))^{\mathcal{T}}$ para cada $X \in dom(T')$.*

Demostración
Sea T' definido tal y como indica el enunciado. Vamos a hacer la demostración por inducción estructural sobre la forma del patrón t, distinguiendo las posibles formas que puede tener un patrón parcial según la definición dada en el apartado 3.1.2 (pág. 48), así como la regla de transformación aplicada en cada caso de las definidas en la figura 4.2 (pág. 84).

- $t \equiv \bot$ y $t \equiv X, X \in Var$. En ambos casos se tiene $t \Rightarrow^{\mathcal{T}} t$ (reglas (TR_2) y (TR_3), respectivamente) y el resultado se cumple por la construcción de T'.

- $t \equiv h\ \bar{t}_m$, $h \in DC^n$.

  Distinguimos dos casos:

[1]En realidad si $G\theta'$ no es atómico tendríamos un *bosque* y no un árbol para $G\theta'$ pero cometiendo un abuso del lenguaje utilizamos la misma notación para simplificar el razonamiento.

- $m = n$. Entonces deben existir tipos $\tau, \overline{\tau}_m \in \mathit{Type}$ tales que:

$$\begin{array}{ll} \textbf{(1)} & (\Sigma, T) \vdash_{WT} \quad h\ \overline{t}_m :: \tau \\ \textbf{(2)} & (\Sigma, T) \vdash_{WT} \quad h :: \tau_1 \to \cdots \to \tau_m \to \tau \\ \textbf{(3)} & (\Sigma, T) \vdash_{WT} \quad t_i :: \tau_i \quad \text{para } i = 1 \ldots m \end{array}$$

  Por hipótesis de inducción, para cada $s_i \in \mathit{Pat}_\perp$ tal que $t_i \Rightarrow^{\mathcal{T}} s_i$, con $1 \leq i \leq m$, se tiene de (3) que $(\Sigma^{\mathcal{T}}, T') \vdash_{WT} s_i :: \tau_i^{\mathcal{T}}$. Como las constructoras no admiten tipos funcionales en sus argumentos (ni en su resultado) se tiene de la transformación de tipos del apartado 4.4.2 (pág. 82) que $\tau_i^{\mathcal{T}} = \tau_i$, por lo que

$$\textbf{(4)} \qquad (\Sigma^{\mathcal{T}}, T') \vdash_{WT} s_i :: \tau_i$$

  Por otra parte, al ser $n = m$ por (TR_5) se cumple $h\ \overline{t}_m \Rightarrow^{\mathcal{T}} h\ \overline{s}_m$, lo que unido a (1) nos indica que lo que debemos probar es que se cumple

$$(\Sigma^{\mathcal{T}}, T') \vdash_{WT} h\ \overline{s}_m :: \tau^{\mathcal{T}}$$

  Como h es una constructora τ no incluye tipos funcionales y se tiene $\tau^{\mathcal{T}} = \tau$ por lo que la relación anterior queda

$$\textbf{(5)} \qquad (\Sigma^{\mathcal{T}}, T') \vdash_{WT} h\ \overline{s}_m :: \tau$$

  Ahora bien, en la definición de $\Sigma^{\mathcal{T}}$ del apartado 4.4.2 (pág. 82) se tiene que las constructoras conservan su tipo en el programa transformado, por lo que de (2) se tiene

$$(\Sigma^{\mathcal{T}}, T') \vdash_{WT} \quad h :: \tau_1 \to \cdots \to \tau_m \to \tau$$

  lo que combinado con (4) nos lleva a como queríamos a (5), aplicando reiteradamente la regla de inferencia de tipos *AP* (pág. 50) a la expresión $h\ \overline{s}_m$ que como sabemos se puede escribir como $(((h\ s_1)\ s_2) \ldots s_m)$.

- $m < n$. En este caso deben existir tipos $\overline{\tau}_n, \tau \in \mathit{Type}$ tales que se cumpla:

$$\begin{array}{ll} \textbf{(6)} & (\Sigma, T) \vdash_{WT} \quad h :: \tau_1 \to \cdots \to \tau_m \to \tau_{m+1} \to \cdots \to \tau_n \to \tau \\ \textbf{(7)} & (\Sigma, T) \vdash_{WT} \quad h\ \overline{t}_m :: \tau_{m+1} \to \cdots \to \tau_n \to \tau \end{array}$$

  y aplicando la regla (TR_4) y se tiene $h\ \overline{t}_m \Rightarrow^{\mathcal{T}} h_m^{\mathcal{T}}\ \overline{s}_m$. Por tanto en este apartado hay que probar que se cumple:

$$\textbf{(8)} \qquad (\Sigma^{\mathcal{T}}, T') \vdash_{WT} h_m^{\mathcal{T}}\ \overline{s}_m :: (\tau_{m+1} \to \cdots \to \tau_n \to \tau)^{\mathcal{T}}$$

aplicando la definición de la transformación de tipos y recordando que τ_{m+1} no puede contener el símbolo $\rightarrow$ al ser parte del tipo de una constructora se tiene:

$$\begin{array}{ll}(\tau_{m+1} \rightarrow \cdots \rightarrow \tau_n \rightarrow \tau)^{\mathcal{T}} & = \\ \tau_{m+1}^{\mathcal{T}} \rightarrow ((\tau_{m+2} \rightarrow \cdots \rightarrow \tau_n)^{\mathcal{T}}, cTree) & = \\ \tau_{m+1} \rightarrow ((\tau_{m+2} \rightarrow \cdots \rightarrow \tau_n)^{\mathcal{T}}, cTree) & \end{array}$$

Por lo que (8) se convierte en

$$\textbf{(9)} \qquad (\Sigma^{\mathcal{T}}, T') \ \vdash_{WT} \ h_m^{\mathcal{T}} \ \overline{s}_m :: \tau_{m+1} \rightarrow ((\tau_{m+2} \rightarrow \cdots \rightarrow \tau_n \rightarrow \tau)^{\mathcal{T}}, cTree)$$

Del apartado 4.4.2 (pág. 82) sabemos que

$$h_m^{\mathcal{T}} \ :: \ \tau_1 \rightarrow \cdots \rightarrow \tau_{m+1} \rightarrow ((\tau_{m+2} \rightarrow \cdots \rightarrow \tau_n \rightarrow \tau)^{\mathcal{T}}, cTree)$$

lo que unido a (4) (que se sigue cumpliendo en este apartado) lleva a (9) como deseamos, aplicando como en el caso anterior la regla AP de inferencia de tipos.

- $t \equiv f \ \overline{t}_m$, $f \in FS^n$. Como en el apartado anterior conviene distinguir dos casos, teniendo en cuenta que debe cumplirse $m < n$ (en otro caso t no sería un patrón).

  - $m = n - 1$. Entonces deben existir tipos $\tau, \overline{\tau}_n \in \mathit{Type}$ tales que:

$$\begin{array}{ll}\textbf{(10)} & (\Sigma, T) \vdash_{WT} \quad f :: \tau_1 \rightarrow \cdots \rightarrow \tau_{n-1} \rightarrow \tau_n \rightarrow \tau \\ \textbf{(11)} & (\Sigma, T) \vdash_{WT} \quad t_i :: \tau_i \quad \text{para } i = 1 \ldots n-1 \\ \textbf{(12)} & (\Sigma, T) \vdash_{WT} \quad f \ \overline{t}_{n-1} :: \tau_n \rightarrow \tau \end{array}$$

    Según la regla de transformación (TR_5) se tiene que $f \ \overline{t}_{n-1} \ \Rightarrow^{\mathcal{T}} f \ \overline{s}_{n-1}$ y lo que debemos probar entonces es, por (12), que se cumple

$$(\Sigma^{\mathcal{T}}, T') \ \vdash_{WT} \ f \ \overline{s}_{n-1} :: (\tau_n \rightarrow \tau)^{\mathcal{T}}$$

    es decir

$$\textbf{(13)} \qquad (\Sigma^{\mathcal{T}}, T') \ \vdash_{WT} \ f \ \overline{s}_{n-1} :: \tau_n^{\mathcal{T}} \rightarrow (\tau^{\mathcal{T}}, \ cTree)$$

    ya que $(\tau_n \rightarrow \tau)^{\mathcal{T}} = \tau_n^{\mathcal{T}} \rightarrow (\tau^{\mathcal{T}}, \ cTree)$. Y (13) se tiene porque tal y como se indica en apartado 4.4.2 (pág. 82) el tipo principal de f en el programa transformado es

$$f \ :: \ \tau_1^{\mathcal{T}} \rightarrow \cdots \rightarrow \tau_n^{\mathcal{T}} \rightarrow (\tau^{\mathcal{T}}, \ cTree)$$

    y por hipótesis de inducción se tiene

$$\textbf{(14)} \qquad (\Sigma^{\mathcal{T}}, T') \ \vdash_{WT} \ s_i :: \tau_i^{\mathcal{T}}$$

    de (11) y al ser $t_i \Rightarrow^{\mathcal{T}} s_i$ para cada $i = 1 \ldots m$, por lo que llegaríamos a (13) aplicando reiteramente la regla de inferencia de tipos AP a la expresión $f \ \overline{s}_{n-1}$.

- $n \leq n-2$. De nuevo deben existir tipos $\tau, \overline{\tau}_n \in \mathit{Type}$ tales que se cumplan (10) y (11) como en el caso anterior y además:

$$\textbf{(15)} \qquad (\Sigma, T) \vdash_{WT} \ f\ \overline{t}_m :: \tau_{m+1} \to \cdots \to \tau_n \to \tau$$

y como, aplicando la regla (TR_4), $f\ \overline{t}_m \ \Rightarrow^{\mathcal{T}} f_m^{\mathcal{T}}\ \overline{s}_m$, lo que habrá que probar en esta ocasión es

$$\textbf{(16)} \quad (\Sigma^{\mathcal{T}}, T') \ \vdash_{WT} \ f_m^{\mathcal{T}}\ \overline{s}_m :: \tau_{m+1}^{\mathcal{T}} \to ((\tau_{m+2} \to \cdots \to \tau_n \to \tau)^{\mathcal{T}}, cTree)$$

ya que

$$(\tau_{m+1} \to \cdots \to \tau_n \to \tau)^{\mathcal{T}} = \tau_{m+1}^{\mathcal{T}} \to ((\tau_{m+2} \cdots \to \tau_n \to \tau)^{\mathcal{T}}, cTree)$$

Para ver que (16) se cumple hay que observar en primer lugar que la fórmula (14) sigue siendo válida en este apartado, y que por el apartado 4.4.2 tenemos que el tipo principal de $f_m^{\mathcal{T}}$ es

$$f_m^{\mathcal{T}} \ :: \ \tau_1^{\mathcal{T}} \to \cdots \to \tau_{m+1}^{\mathcal{T}} \to ((\tau_{m+2} \to \cdots \to \tau_n \to \tau)^{\mathcal{T}}, \ cTree)$$

por lo que (16) se obtiene aplicando reiteradamente la regla de inferencia de tipos AP hasta determinar el tipo de $f_m^{\mathcal{T}}\ \overline{s}_m$.

□

El siguiente lema aplica la misma idea a condiciones completas.

Lema A.2.11. *Sea $C \equiv (A, B)$ una condición plana (def. 4.3.1, pág. 75), Σ una signatura y T un contexto tal que*

a) Para cada $(e == e') \in C$ existe algún $\mu \in$ Type tal que $(\Sigma, T) \vdash_{WT} e \ :: \ \mu \ :: \ e'$.

b) Para cada $(d \to s) \in C$ existe algún $\mu \in Type$ que verifica $(\Sigma, T) \vdash_{WT} d \ :: \ \mu \ :: \ s$.

Sea T_{aux} un contexto tal que $dom(T_{aux}) = dom(T)$ y $T_{aux}(X) = T(X)^{\mathcal{T}}$ para todo $X \in dom(T)$. Sea $C^{\mathcal{T}}$ una condición tal que $C \Rightarrow^{\mathcal{T}} (C^{\mathcal{T}}; trees)$ para cierto valor trees . Entonces existe un contexto T_V, que da tipo cTree a las variables nuevas introducidas durante la transformación de C, tal que $T' = T_{aux} \uplus T_V$ permite dar tipo a las condiciones atómicas de $C^{\mathcal{T}}$ en el sentido de los ítems a) y b) anteriores.

Demostración
Por inducción completa sobre la estructura de la prueba $C \Rightarrow^{\mathcal{T}} (C^{\mathcal{T}}; trees)$. Nos fijamos en el paso dado en último lugar y razonamos dependiendo de la regla aplicada, que puede ser (TR_7), (TR_8), (TR_9), (TR_{10}) o (TR_{11}).

(TR$_7$). En este caso $C \equiv A, B$, con A, B también condiciones planas. Es claro que T permite dar tipo a las condiciones atómicas de A y B, al estar éstas en C. Por la estructura de la regla se tiene que entonces $C^{\mathcal{T}} \equiv (A^{\mathcal{T}}, B^{\mathcal{T}}; trees_A, trees_B)$, con $A \Rightarrow^{\mathcal{T}} (A^{\mathcal{T}}; trees_A)$ y $B \Rightarrow^{\mathcal{T}} (B^{\mathcal{T}}; trees_B)$ las premisas de la regla. Aplicando a estas premisas la hipótesis de inducción tenemos que deben existir T_{VA} y T_{VB} tales que $T_A = T_{aux} \uplus T_{VA}$ permite dar tipo a las condiciones atómicas en $A^{\mathcal{T}}$ y $T_B = T_{aux} \uplus T_{VB}$ permite dar tipo las de $B^{\mathcal{T}}$. Además, al incluir supuestos de tipo sólo para las variables nuevas tenemos que $T_{VA} \cap T_{VB} = \emptyset$. Definimos entonces $T' = T_{aux} \uplus T_{VA} \uplus T_{VB}$, que tiene la forma requerida por el lema y permite dar tipo a todas las condiciones atómicas en $(A^{\mathcal{T}}, B^{\mathcal{T}})$.

(TR$_8$). En este caso $C \equiv l == r$, con $l, r \in Pat_\perp$ (en otro caso C no sería plana). Por hipótesis se tiene $(\Sigma, T) \vdash_{WT} l :: \tau :: r$ para cierto tipo τ, y del lema A.2.10 se tiene que existen contextos T_1 y T_2 tales que $(\Sigma^{\mathcal{T}}, T_1) \vdash_{WT} u :: \tau^{\mathcal{T}}$, con $l \Rightarrow^{\mathcal{T}} u$ y $(\Sigma^{\mathcal{T}}, T_2) \vdash_{WT} v :: \tau^{\mathcal{T}}$ con $r \Rightarrow^{\mathcal{T}} v$. En ninguna de las dos transformaciones se introducen variables nuevas al ser l y r patrones (estamos partiendo de un programa plano). Además por la construcción del contexto del lema se tiene que $T_1 \equiv T_2 \equiv T_{aux}$ y por tanto, llamando T' a T_{aux} tenemos que T' tiene la forma requerida por el resultado y verifica $(\Sigma^{\mathcal{T}}, T') \vdash_{WT} u :: \tau^{\mathcal{T}} :: v$, por lo que T' permite dar tipo a $u == v$ en el programa transformado.

(TR$_9$). Entonces $C \equiv e \to t$, con $e, t \in Pat_\perp$. Análogo al caso anterior.

(TR$_{10}$). Si se ha aplicado esta regla se tiene $C \equiv X\ s \to t$, $X \in Var$, $s, t \in Pat_\perp$, y $C^{\mathcal{T}} \equiv (X\ u \to v; (v, Y))$ con $s \Rightarrow^{\mathcal{T}} u$ y $t \Rightarrow^{\mathcal{T}} v$, e Y una nueva variable. Por hipótesis se tiene $(\Sigma, T) \vdash_{WT} (X\ s) :: \tau :: t$ para algún tipo τ y debemos probar que entonces $(\Sigma^{\mathcal{T}}, T') \vdash_{WT} (X\ u) :: \tau^{\mathcal{T}} :: (v, Y)$ con $T' \equiv T_{aux} \uplus T_V$. Al ser s, t patrones, Y es la única variable nueva, por lo que $T_V = \{Y :: cTree\}$.

Para que $(\Sigma, T) \vdash_{WT} (X\ s) :: \tau$ debe ser $(\Sigma, T) \vdash_{WT} s :: \gamma$ para un cierto $\gamma \in$ *Type* y entonces $T(X) = \gamma \to \tau$ según exige la regla AP. Por la definición de T' se tendrá $T'(X) = T_{aux}(X) = \gamma^{\mathcal{T}} \to (\tau^{\mathcal{T}}, cTree)$ mientras que por el lema A.2.10 se tiene $(\Sigma^{\mathcal{T}}, T') \vdash_{WT} u :: \gamma^{\mathcal{T}}$, de donde por la regla AP se llega a $(\Sigma^{\mathcal{T}}, T') \vdash_{WT} X\ u :: (\tau^{\mathcal{T}}, cTree)$.

Por otra parte, y de nuevo por el lema A.2.10 se tiene $(\Sigma^{\mathcal{T}}, T') \vdash_{WT} v :: \tau^{\mathcal{T}}$ al ser $(\Sigma, T) \vdash_{WT} t :: \tau$ y $t \Rightarrow^{\mathcal{T}} v$. Entonces aplicando AP con la constructora de tuplas tup_2 como functor y v e Y como argumentos se prueba $(\Sigma^{\mathcal{T}}, T') \vdash_{WT} (v, Y) :: (\tau^{\mathcal{T}}, cTree)$, que al coincidir con el tipo de $X\ u$ en el programa transformado completa la prueba.

(TR$_{11}$). En este caso $C \equiv f\ \bar{t}_n \to t$, $C^{\mathcal{T}} \equiv (f^{\mathcal{T}}\ \bar{s}_n \to (v, Y)\,;\,(v, Y))$, con $f \in FS^n$, Y variable nueva, $t_i \Rightarrow^{\mathcal{T}} s_i$ para $i = 1 \ldots n$ y $t \Rightarrow^{\mathcal{T}} v$.

Por hipótesis se debe tener que existen tipos $\bar{t}_n, \tau \in$ *Type* tales que se cumplen

$$(\Sigma, T) \vdash_{WT} t_i :: \tau_i \quad \text{para } i = 1 \ldots n$$
$$(\Sigma, T) \vdash_{WT} f :: \tau_1 \to \cdots \to \tau_n \to \tau$$

y por tanto se tendrá $(\Sigma, T) \vdash_{WT} f\ \bar{t}_n :: \tau :: t$.

Por la definición de la transformación de programas se tiene en el programa transformado

$$(\Sigma^{\mathcal{T}}, T') \vdash_{WT} f^{\mathcal{T}} :: \tau_1^{\mathcal{T}} \to \cdots \to \tau_n^{\mathcal{T}} \to (\tau^{\mathcal{T}}, cTree)$$

y por el lema A.2.10 se cumple $(\Sigma^{\mathcal{T}}, T') \vdash_{WT} s_i :: \tau_i^{\mathcal{T}}$ para $i = 1 \ldots n$ por lo que, aplicando de nuevo por la regla AP, se tiene

$$(\Sigma^{\mathcal{T}}, T') \vdash_{WT} f^{\mathcal{T}}\ \bar{s}_n :: (\tau^{\mathcal{T}}, cTree)$$

Para completar la prueba debemos comprobar que el lado derecho de la aproximación tiene el mismo tipo, es decir que se cumple

$$(\Sigma^{\mathcal{T}}, T') \vdash_{WT} (v, Y) :: (\tau^{\mathcal{T}}, cTree)$$

para lo que habrá que ver si se cumplen:

1. $(\Sigma^{\mathcal{T}}, T') \vdash_{WT} v :: \tau^{\mathcal{T}}$. Cierto por el A.2.10 lema al ser $(\Sigma, T) \vdash_{WT} t :: \tau$ y $t \Rightarrow^{\mathcal{T}} v$.
2. $(\Sigma^{\mathcal{T}}, T') \vdash_{WT} Y :: cTree$. Cierto ya que $T'(Y) = T_V(Y) = cTree$ al tratarse de una variable nueva, por la definición de T'.

□

Vamos a demostrar un lema muy sencillo que indica que la transformación de un patrón no altera su conjunto de variables.

Lema A.2.12. *Sean $t, s \in Pat_\perp$ tales que $t \Rightarrow^{\mathcal{T}} s$. Entonces*

1. *$var(t) = var(s)$*
2. *Si t es lineal entonces s es lineal.*
3. *Si t es transparente entonces s es transparente.*

Demostración

Demostramos los tres resultados simultáneamente por inducción completa sobre la estructura de la prueba $t \Rightarrow^{\mathcal{T}} s$. Nos fijamos en el paso dado en último lugar y razonamos dependiendo de la regla aplicada, que al ser un patrón sólo puede ser una de las siguientes:

- (TR_2). Evidente ya que en este caso $t \equiv s \equiv \perp$, $var(t) = var(s) = \emptyset$, y t, s son necesariamente lineales y transparentes.
- (TR_3) En este caso $t \equiv s \equiv X$ para algún $X \in Var$ y por tanto $var(t) = var(s) = \{X\}$ y de nuevo estamos ante dos patrones lineales y transparentes.

- (TR_5) En este caso $t \equiv h\ \bar{t}_m$ y $s \equiv h^{\mathcal{T}}_m\ \bar{s}_m$. Como $t_i \Rightarrow^{\mathcal{T}} s_i$ para $i = 1 \ldots m$ se tiene por hipótesis de inducción que $var(t_i) = var(s_i)$ para $i = 1 \ldots m$. Entonces

$$\begin{aligned} var(h\ \bar{t}_m) &= var(t_1)\ \cup\ \ldots\ \cup\ var(t_m) = \\ &= var(s_1)\ \cup\ \ldots\ \cup\ var(s_m) =\ var(h^{\mathcal{T}}_m\ \bar{s}_m) \end{aligned}$$

  Para ver que si t es lineal también debe serlo s vamos a probar que si s no es lineal t tampoco lo es. Supongamos entonces que existe alguna variable repetida en s. Como $h^{\mathcal{T}}_m \notin Var$ dicha repetición debe estar en dos valores s_i, s_j para ciertos $1 \leq i, j \leq m$ (donde puede que $i = j$). Entonces, como acabamos de ver $var(t_i) = var(s_i)$, $var(t_j) = var(s_j)$, por lo que la repetición la tendremos igualmente en t, que no será lineal.

  Por último, si t es transparente entonces cada t_i, $i = 1 \ldots m$, debe ser transparente y por hipótesis de inducción también lo será cada s_i, lo que hará a su vez que s sea a su vez transparente.
- (TR_4, TR_6) Análogos al anterior.

□

Este lema nos será útil para probar la admisibilidad de las condiciones transformadas.

Lema A.2.13. *Sean $d \to s$ una aproximación en una condición plana y $d' \to s'$ tal que $(d \to s) \Rightarrow^{\mathcal{T}} (d' \to s')$. Entonces*

1. *$var(d') = var(d)$.*
2. *$var(s') = var(s) \uplus V$, donde V es un conjunto de variables nuevas introducidas por las reglas de transformación.*

Demostración

Para ver que el enunciado se cumple nos fijamos en el último paso de la transformación $(d \to s) \Rightarrow^{\mathcal{T}} (d' \to s')$ que al tratarse de la transformación de una aproximación sólo puede ser (TR_9), (TR_{10}) o (TR_{11}), y probamos ambos puntos simultáneamente:

- (TR_9). En este caso de las premisas de la regla se tiene $d \Rightarrow^{\mathcal{T}} d'$ y $s \Rightarrow^{\mathcal{T}} s'$ (donde en la regla d sería e, s sería t, d' está representado como u y s' sería v). Al ser todos los valores patrones el lema A.2.12 nos asegura que $var(d') = var(d)$ y $var(s') = var(s)$.
- (TR_{10}) En este caso $d \equiv X\ u$ y $d' \equiv X\ u'$ para ciertos patrones u, u' tales que $u \Rightarrow^{\mathcal{T}} u'$, por lo que del lema A.2.12 se tiene $var(u') = var(u)$ y por tanto $var(d') = var(X\ u) = var(X\ u') = var(d)$, lo que prueba el primer punto del enunciado. Para el segundo hay que observar que $s' \equiv (v, T)$ con T variable nueva y v patrón, por lo que $var(s') = var(v) \uplus \{T\}$, y al tenerse $(d \to s) \Rightarrow^{\mathcal{T}} (d' \to s')$ la regla de transformación indica que será $s \Rightarrow^{\mathcal{T}} v$, por lo que del lema A.2.12 tenemos $var(v) = var(s)$, y por tanto $var(s') = var(s) \uplus \{T\}$.

- (TR_{11}) Análoga a la regla anterior.

□

El siguiente lema asegura que las condiciones del programa transformado son admisibles.

Lema A.2.14. *Sea P un programa plano con signatura Σ y sea (R) una regla de programa $f\ \bar{t}_n \to r \Leftarrow C$. Sea $f^{\mathcal{T}}\ \bar{s}_n \to r^{\mathcal{T}} \Leftarrow C'$ tal que*

$$(f\ \bar{t}_n \to r \Leftarrow C) \Rightarrow^{\mathcal{T}} (f^{\mathcal{T}}\ \overline{t'}_n \to r^{\mathcal{T}} \Leftarrow C')$$

Entonces C' es una secuencia de condiciones atómicas admisible con respecto al conjunto $var(f^{\mathcal{T}}\ \overline{t'}_n)$.

Demostración

En primer lugar observamos que $var(f\ \bar{t}_n) = var(f^{\mathcal{T}}\ \overline{t'}_n)$. Por la regla ($\mathrm{TR}_1$) (figura 4.2 pág. 84) tenemos que $t_i \Rightarrow^{\mathcal{T}} t'_i$ para cada $i = 1 \ldots n$ (los t'_i aparecen nombrados como s_i en la regla), por lo que:

$$\begin{aligned} var(f\ \bar{t}_n) &= var(t_1)\ \cup\ \ldots\ \cup\ var(t_n)\ = (\text{lema A.2.12}) = \\ &= var(t'_1)\ \cup\ \ldots\ \cup\ var(t'_n)\ = var(f^{\mathcal{T}}\ \overline{t'}_n) \end{aligned}$$

Por tanto bastará con probar que C' es admisible con respecto a $var(f\ \bar{t}_n)$.

Nos fijamos a continuación en la forma de C' que según la citada regla (TR_1) es de la forma:

$$\begin{aligned} &C^{\mathcal{T}},\ ctNode\ \text{"}f.k\text{"}\ \ [dVal\, s_1, \ldots,\ dVal\, s_n] \\ &\qquad\qquad (dVal\, s)\ (ctClean\ [(dVal\, r_1, T_1), \ldots,\ (dVal\, r_m, T_m)]) \to T \end{aligned}$$

Con $C \Rightarrow^{\mathcal{T}} (C^{\mathcal{T}}\ ;\ (r_1, T_1), \ldots, (r_m, T_m))$. Como C es admisible con respecto a $var(f\,\bar{t}_n)$, suponemos sin perdida de generalidad que ya está ordenado acuerdo a las condiciones establecidas en la definición de admisible (ver punto (iii) de la definición de regla bien tipada en el apartado 3.1.5, pág. 51).

Sea entonces $d'_1 \to s'_1, \cdots, d'_m \to s'_m, d'_{m+1} \to s'_{m+1}$ la secuencia de aproximaciones en C' en su orden de aparición. La última aproximación será entonces

$$\begin{aligned} d'_{m+1} \to s'_{m+1}\ \equiv\ &ctNode\ \text{"}f.k\text{"}\ \ [dVal\, s_1, \ldots,\ dVal\, s_n] \\ &\qquad\qquad (dVal\, s)\ (ctClean\ [(dVal\, r_1, T_1), \ldots,\ (dVal\, r_m, T_m)]) \to T \end{aligned}$$

mientras que $d'_1 \to s'_1, \cdots, d'_m \to s'_m$ serán las aproximaciones de $C^{\mathcal{T}}$ en su orden de aparición. Cada una de las aproximaciones en $C^{\mathcal{T}}$ debe provenir de la transformación de la correspondiente aproximación en C, ya que las reglas de transformación

(TR_7)-(TR_{11}) transforman cada condición atómica en una y sólo condición atómica, respetando el orden de aparición en la condición (lo que se tiene por la regla (TR_7)).

Por tanto las aproximaciones en C serán $d_1 \to s_1, \cdots, d_m \to s_m$ verificándose que $(d_i \to s_i) \Rightarrow^{\mathcal{T}} ((d'_i \to s'_i); tree_i)$ para ciertos valores $tree_i$ y para $i = 1 \ldots m$.

Veamos entonces que C' cumple las condiciones indicadas en la definición de condición admisible:

1. Para todo $1 \leq i \leq m+1$: $var(s'_i) \cap var(f\ \bar{t}_n) = \emptyset$.
   En el caso $i = m+1$ estamos hablando de la última aproximación en $C^{\mathcal{T}}$ y por tanto $s'_i \equiv T$ que tal y como indica la regla es una variable nueva por lo que $\{T\} \cap var(f\ \bar{t}_n) = \emptyset$.
   En el resto de los casos, $1 \leq i \leq m$, el resultado se deduce del lema A.2.13, que nos indica que $var(s'_i) = var(s) \biguplus V$ con V formado únicamente por variables nuevas por lo que $var(s'_i) \cap var(f\ \bar{t}_n) = \emptyset$ sii $var(s_i) \cap var(f\ \bar{t}_n) = \emptyset$, lo que se tiene por hipótesis.
2. Para todo $1 \leq i \leq m+1$, s'_i es lineal y para todo $1 \leq j \leq m+1$ con $i \neq j$ $var(s'_i) \cap var(s'_j) = \emptyset$.

   Comenzamos por ver que s'_i es lineal para cada $i = 1 \ldots m+1$. El caso $i = m+1$ es de nuevo directo al ser $s_{i+1} = T \in Var$. Comprobamos entonces los valores $1 \leq i \leq m$, distinguiendo casos según la regla aplicada en el último paso de la prueba $(d_i \to s_i) \Rightarrow^{\mathcal{T}} ((d'_i \to s'_i); tree_i)$:

   - (TR_9). En este caso se tiene $s_i \Rightarrow^{\mathcal{T}} s'_i$ y por el lema A.2.12 s'_i es lineal al serlo s_i.
   - (TR_{10}) Entonces $s'_i \equiv (v, T)$ con T variable nueva, por lo que s'_i será linea sii lo es v. Y al tenerse $(d_i \to s_i) \Rightarrow^{\mathcal{T}} (d'_i \to s'_i)$ la regla de transformación indica que se cumple $s_i \Rightarrow^{\mathcal{T}} v$, por lo que al ser s_i lineal por hipótesis, el lema A.2.12 asegura que v también lo es.
   - (TR_{11}) Análogo al caso anterior.

   Falta ahora por comprobar que dado un i cualquiera se verifica que para todo $1 \leq j \leq m+1$ con $i \neq j$ $var(s'_i) \cap var(s'_j) = \emptyset$. De nuevo el caso $i = m+1$ se comprueba directamente al ser en este caso $s'_i = T$ una variable nueva y tenerse por tanto $\{T\} \cap var(s'_j) = \emptyset$ para todo $j \neq i$. Lo mismo sucede si $j = m+1$, por lo que sólo debemos considerar $1 \leq i, j \leq m$ con $i \neq j$. En estos casos el lema A.2.13 nos indica que $var(s'_i) \cap var(s'_j) = \emptyset$ sii

   $$(var(s_i) \biguplus V) \cap (var(s_j) \biguplus V') = \emptyset$$

   con V, V' conjuntos de variables nuevas, que por tanto no pueden tener variables comunes entre sí ni con s_i ni s_j, por lo que la igualdad anterior se cumple sii $var(s_i) \cap var(s_j) = \emptyset$, y esto último lo tenemos por hipótesis al ser C admisible.

3. Para todo $1 \leq i \leq m+1, 1 \leq j \leq i$: $var(s'_i) \cap var(d'_j) = \emptyset$.
   El caso $i = m+1$ lo tenemos de nuevo por ser $s_{m+1} = T$ una variable nueva. Por tanto probamos el resultado para $1 \leq i \leq m, 1 \leq j \leq i$. En estos casos el lema A.2.13 resulta aplicable, indicando que $var(s'_i) \cap var(d'_j) = \emptyset$ sii $(var(s_i) \uplus V) \cap var(d_j) = \emptyset$, y al ser V un conjunto de variables nuevas, disjuntas por tanto a las variables de d_j, la igualdad anterior es cierta si y sólo si se cumple $var(s_i) \cap var(d_j) = \emptyset$, lo que se tiene por la hipótesis de admisibilidad de C.

□

Con todos estos resultados podemos demostrar el teorema. Para ello, según se indica en el apartado 3.1.5 (pág. 51) tenemos que probar que es posible dar tipo a todas las reglas del programa transformado. Para probar esto distinguimos dos tipos de reglas: las funciones transformadas según las reglas (TR_1)-(TR_{11}) de la figura 4.2 (pág. 84) y las funciones auxiliares introducidas durante el proceso de transformación tal y como se explica en el apartado 4.4.3 (pág. 83).

Funciones Transformadas

Sea $f\ \bar{t}_n \rightarrow r \Leftarrow C$ una regla de un programa aplanado $P_{\mathcal{A}}$ bien tipada para cierto contexto T y sea $f\ ::\ \tau_1 \rightarrow \cdots \rightarrow \tau_n \rightarrow \tau$ el tipo principal de f en la signatura de $P_{\mathcal{A}}$. Sea $f^{\mathcal{T}}\ \bar{s}_n\ \rightarrow (s, A) \Leftarrow C'$ tal que

$$(f\ \bar{t}_n \rightarrow r \Leftarrow C)\ \Rightarrow^{\mathcal{T}}\ (f^{\mathcal{T}}\ \bar{s}_n\ \rightarrow (s, A) \Leftarrow C')$$

es decir la correspondiente regla en $P^{\mathcal{T}}$. Entonces vamos a probar que el contexto T' definido como $T' = T_{aux} \uplus T_V$ permite dar tipo a la regla transformada, donde

- $dom(T_{aux}) = dom(T)$ y $T_{aux}(X) = T(X)^{\mathcal{T}}$ para cada $X \in dom(T)$.
- T_V es un contexto que da tipo $cTree$ a todas las variables nuevas introducidas por las reglas de transformación.

y donde el tipo principal de la función $f^{\mathcal{T}}$ es

$$f^{\mathcal{T}}\ ::\ \tau_1^{\mathcal{T}} \rightarrow \cdots \rightarrow \tau_n^{\mathcal{T}} \rightarrow (\tau^{\mathcal{T}}, cTree)$$

Para ver que esto sucede debemos comprobar las condiciones del apartado 3.1.5 para las reglas bien tipadas que reproducimos a continuación con la notación adaptada a la de la regla transformada:

(i) $s_1 \ldots s_n$ debe ser una secuencia lineal de patrones transparentes, y (s, A) debe ser una expresión.

Sabemos que $t_1 \ldots t_n$ es por hipótesis una secuencia lineal de patrones transparentes, por lo que el patrón $(t_1, \ldots, t_n)$ será linea por transparente. La transformación de este patrón es: $(t_1, \ldots, t_n) \Rightarrow^{\mathcal{T}} (s_1, \ldots, s_n)$, y por tanto el lema A.2.12 nos asegura que $(s_1, \ldots, s_n)$ es lineal y transparente, de donde $s_1 \ldots s_n$ es una secuencia lineal de patrones transparentes. Por otro lado (s, A) es obviamente una expresión (s es un patrón y A una variable).

(ii) La *condición* C' será una secuencia de *condiciones atómicas* $C_1, \ldots, C_k$, donde cada C_i puede ser o bien una *igualdad estricta* de la forma $e == e'$, con $e, e' \in Exp$, o bien una *aproximación* de la forma $d \to s$, con $d \in Exp$ y $s \in Pat$.

Tal y como indica (TR_1) la condición C' es de la forma:

$$C^{\mathcal{T}},\ ctNode\ "f.k"\ \ [dVal\, s_1, \ldots,\ dVal\, s_n]$$
$$(dVal\, s)\ (ctClean\ [(dVal\, r_1, T_1), \ldots,\ (dVal\, r_m, T_m)]) \to A$$

donde se verifica $C \Rightarrow^{\mathcal{T}} C^{\mathcal{T}}$. Es fácil comprobar que las reglas (TR_7)-(TR_{11}) transforman cada condición atómica en otra condición atómica del mismo tipo (aproximaciones en aproximaciones e igualdades estrictas en igualdades estrictas) con las características indicadas, lo que garantiza que todas las condiciones atómicas $C^{\mathcal{T}}$ son de la forma indicada. En cuanto a la última condición atómica

$$C^{\mathcal{T}},\ ctNode\ "f.k"\ \ [dVal\, s_1, \ldots,\ dVal\, s_n]$$
$$(dVal\, s)\ (ctClean\ [(dVal\, r_1, T_1), \ldots,\ (dVal\, r_m, T_m)]) \to A$$

se trata de una aproximación cuyo lado derecho es un patrón (en particular una variable) y su lado izquierdo una expresión, en concreto la aplicación de la constructora $ctNode$.

(iii) Además la condición C debe ser *admisible* respecto al conjunto de variables $\mathcal{X} =_{def} var(f^{\mathcal{T}}\ \overline{s}_n)$. Probado en el lema A.2.14.

(iv) Existe un contexto T' con dominio las variables de la regla, que permite tipar correctamente la regla de programa cumpliendo las siguientes condiciones:

a) Para todo $1 \leq i \leq n$: $(\Sigma^{\mathcal{T}}, T') \vdash_{WT} s_i\ ::\ \tau_i^{\mathcal{T}}$.
Esto es cierto al ser T' una extensión del contexto del que se habla en el lema A.2.10, y verificarse $t_i \Rightarrow^{\mathcal{T}} s_i$ con $(\Sigma, T) \vdash_{WT} t_i\ ::\ \tau_i^{\mathcal{T}}$ (por estar la regla original bien tipada con respecto a T) para todo $1 \leq i \leq n$, tal y como requieren las hipótesis del lema.

b) $(\Sigma^{\mathcal{T}}, T') \vdash_{WT} (s, A)\ ::\ (\tau^{\mathcal{T}},\ cTree)$.
Del nuevo del lema A.2.10 al ser $r \Rightarrow^{\mathcal{T}} s$ con $(\Sigma, T) \vdash_{WT} r\ ::\ \tau$ y T' una extensión del contexto definido en el lema se tiene $(\Sigma^{\mathcal{T}}, T') \vdash_{WT} s)\ ::\ \tau^{\mathcal{T}}$. Por otra parte de la construcción de T', al ser A una variable nueva se tiene $T'(A) = cTree$, por lo que este punto también se cumple.

c) Para cada $(e == e') \in C'$ existe algún $\mu \in Type$
tal que $(\Sigma^{\mathcal{T}}, T') \vdash_{WT} e\ ::\ \mu\ ::\ e'$.
Las únicas igualdades estrictas de C' deben estar en $C^{\mathcal{T}}$ (ya que se tiene $C' \equiv C^{\mathcal{T}}, cNode \cdots \to A$) y el resultado se tiene como consecuencia del lema A.2.11 al ser T' una extensión del contexto allí considerado.

d) Para cada $(d \to s) \in C'$ existe algún $\mu \in Type$
que verifica $(\Sigma^{\mathcal{T}}, T') \vdash_{WT} d \ :: \ \mu \ :: \ s$.
Para las aproximaciones de $C^{\mathcal{T}}$ el resultado se tiene como en el caso anterior del lema A.2.11. Debemos comprobar que esto también sucede para la última aproximación

$$ctNode\ "f.k"\ \ [dVal\, s_1, \ldots, dVal\, s_n]$$
$$(dVal\, s)\ (ctClean\ [(dVal\, r_1, T_1), \ldots, (dVal\, r_m, T_m)]) \to A$$

Por definición de T' al ser A una variable nueva se tiene $T'(A) = cTree$ por lo que habrá que probar que

$$(\Sigma^{\mathcal{T}}, T') \vdash_{WT} ctNode\ "f.k"\ \ [dVal\, s_1, \ldots, dVal\, s_n]$$
$$(dVal\, s)\ (ctClean\ [(dVal\, r_1, T_1), \ldots, (dVal\, r_m, T_m)]) \ :: A$$

Comprobamos a continuación algunas partes de esta prueba que utilizan las reglas de inferencia de tipos definidas en la pág. 50:

1) Cada T_i con $i = 1 \ldots m$ es una variable nueva y por tanto $T'(T_i) = cTree$ de donde por la regla de inferencia de tipos VR se llega a $(\Sigma^{\mathcal{T}}, T') \vdash_{WT} T_i :: cTree$ para $i = 1 \ldots m$.
2) Cada r_i corresponde con un patrón, parte del lado derecho de una aproximación introducida en la regla (TR_{10}) o (TR_{11}) que forma parte de las condiciones, por lo que sabemos que está bien tipado por T' y por tanto $(\Sigma^{\mathcal{T}}, T') \vdash_{WT} r_i :: \tau_i$ para algún $\tau_i \in Type$.
3) El tipo principal de $dVal$ en $\Sigma^{\mathcal{T}}$ es $dVal \ :: \ A \to pVal$. Definiendo una sustitución de tipos $\sigma_t = \{A \mapsto \tau_i\}$ podemos aplicar la regla de inferencia de tipos ID de donde $(\Sigma^{\mathcal{T}}, T') \vdash_{WT} dVal :: \tau_i \to pVal$ para cada $i = 1 \ldots m$.
4) De los dos puntos anteriores, por la regla de inferencia de tipos AP, se cumple que $(\Sigma^{\mathcal{T}}, T') \vdash_{WT} dVal\ r_i :: pVal$ para cada $i = 1 \ldots m$.
5) Aplicando AP a los puntos 1 y 4 tenemos que se cumple

$$(\Sigma^{\mathcal{T}}, T') \vdash_{WT} (dVal\ r_i, T_i) :: (pVal, cTree)$$

para cada $i = 1 \ldots m$.

Procediendo así se llega a que el tipo de la expresión completa es $cTree$ como deseamos (en particular obsérvese que este es el tipo del resultado de la constructora más externa $ctNode$ en su definición de tipo del apartado 4.4.2).

Funciones Auxiliares

El conjunto de las funciones auxiliares incluye a las funciones $f_m^{\mathcal{T}}$ y $c_m^{\mathcal{T}}$ introducidas para la sustitución de aplicaciones parciales así como la función auxiliar $ctClean$. La comprobación de la validez desde el punto de vista de los tipos de $ctClean$ es un sencillo ejercicio y no vamos a incluirla en este apartado.

En cuanto a las funciones $f_m^{\mathcal{T}}$ y $c_m^{\mathcal{T}}$, comprobamos a continuación que las funciones $f_m^{\mathcal{T}}$ están bien tipadas en la signatura $\Sigma^{\mathcal{T}}$ (la demostración para las funciones $c_m^{\mathcal{T}}$ sería análoga).

El tipo principal de cada función f_i con $i = 0 \ldots n-2$ es

$$f_i^{\mathcal{T}} \;::\; \tau_1^{\mathcal{T}} \to \cdots \to \tau_{i+1}^{\mathcal{T}} \to ((\tau_{i+2} \to \cdots \to \tau_n \to \tau)^{\mathcal{T}},\, cTree)$$

y su definición (que puede consultarse en el apartado 4.4.3, pág. 83):

$$\begin{array}{lll} f_i^{\mathcal{T}}\, \overline{X}_{i+1} & \to & (f_{i+1}^{\mathcal{T}}\, \overline{X}_{i+1},\ void) \quad \text{para } i = 0 \ldots n-3 \\ f_{n-2}^{\mathcal{T}}\, \overline{X}_{n-1} & \to & (f\ \overline{X}_{n-1},\ void) \end{array}$$

Definimos entonces un contexto T' con $dom(T') = \{X_1, \ldots, X_{n-1}\}$ y $T'(X_i) = \tau_i^{\mathcal{T}}$. Vamos a comprobar que se verifican las condiciones, definidas el apartado 3.1.5, que deben cumplir las reglas bien tipadas para T' en la signatura $\Sigma^{\mathcal{T}}$:

(i) Se tiene que las secuencias de variables (que son los parámetros de estas funciones auxiliares) siempre forman secuencia lineal de patrones transparentes. Así mismo se comprueba fácilmente que los lados derechos son expresiones.

(ii) La *condición* C será una secuencia de *condiciones atómicas* $C_1, \ldots, C_k$, donde cada C_i puede ser o bien una *igualdad estricta* de la forma $e == e'$, con $e, e' \in Exp$, o bien una *aproximación* de la forma $d \to s$, con $d \in Exp$ y $s \in Pat$.

Estas funciones tiene condiciones vacías, por lo que esta condición se cumple trivialmente.

(iii) Además la condición C debe ser *admisible* respecto al conjunto de variables $\mathcal{X} =_{def} var(f\ \bar{t}_n)$.

Como en el caso anterior la condición se cumple trivialmente.

(iv) T' permite dar tipo correctamente a la regla de programa cumpliendo las siguientes condiciones:

a) Para todo argumento X_i se cumple $(\Sigma^{\mathcal{T}}, T') \vdash_{WT} X_i \;::\; \tau_i^{\mathcal{T}}$, cierto por la definición de T'.

b) Hay que probar que el tipo del lado derecho coincide con el tipo del valor de salida de la función. Distinguimos dos casos:

- $1 \leq i \leq n-3$. Hay que probar que

$$(\Sigma^{\mathcal{T}}, T') \vdash_{WT} (f_{i+1}^{\mathcal{T}}\, \overline{X}_{i+1},\ void) :: ((\tau_{i+2} \to \cdots \to \tau_n \to \tau)^{\mathcal{T}},\, cTree)$$

El tipo principal de f_{i+1} es:

$$f_{i+1}^{\mathcal{T}} \;::\; \tau_1^{\mathcal{T}} \to \cdots \to \tau_{i+2}^{\mathcal{T}} \to ((\tau_{i+3} \to \cdots \to \tau_n \to \tau)^{\mathcal{T}},\, cTree)$$

y como $T'(X_j) = \tau_j^{\mathcal{T}}$ para $j = 1 \ldots i+1$ tenemos

$$(\Sigma^{\mathcal{T}}, T') \vdash_{WT} f_{i+1}^{\mathcal{T}}\, \overline{X}_{i+1} :: \tau_{i+2}^{\mathcal{T}} \to ((\tau_{i+3} \to \cdots \to \tau_n \to \tau)^{\mathcal{T}}, cTree)$$

Por su parte el tipo principal de *void* en $\Sigma^{\mathcal{T}}$ es $void :: cTree$, de donde

$$(\Sigma^{\mathcal{T}}, T') \vdash_{WT}$$
$$(f_{i+1}^{\mathcal{T}}\, \overline{X}_{i+1}, void) :: (\tau_{i+2}^{\mathcal{T}} \to ((\tau_{i+3} \to \cdots \to \tau_n \to \tau)^{\mathcal{T}}, cTree), cTree)$$

que es el tipo esperado al verificarse

$$(\tau_{i+2} \to \cdots \to \tau_n \to \tau)^{\mathcal{T}} =$$
$$\tau_{i+2}^{\mathcal{T}} \to ((\tau_{n+3} \to \cdots \to \tau_n \to \tau)^{\mathcal{T}}, cTree)$$

- $i = n-2$. Entonces hay que probar:

$$(\Sigma^{\mathcal{T}}, T') \vdash_{WT} (f\overline{X}_{n-1},\ void) :: ((\tau_n \to \tau)^{\mathcal{T}}, cTree)$$

Como el tipo principal de f es

$$f\ ::\ \tau_1^{\mathcal{T}} \to \cdots \to \tau_n^{\mathcal{T}} \to (\tau^{\mathcal{T}}, cTree)$$

y $T'(X_i) = \tau_i^{\mathcal{T}}$ para $i = 1 \ldots n-1$ tenemos que

$$(\Sigma^{\mathcal{T}}, T') \vdash_{WT} f\overline{X}_{n-1} :: \tau_n^{\mathcal{T}} \to (\tau^{\mathcal{T}}, cTree)$$

y al ser el tipo de *void*, $void : cTree$, se llega a

$$(\Sigma^{\mathcal{T}}, T') \vdash_{WT} (f\overline{X}_{n-1},\ void) :: (\tau_n^{\mathcal{T}} \to (\tau^{\mathcal{T}}, cTree),\ cTree)$$

que coincide con el tipo que debíamos deducir ya que por la definición de la transformación de tipos se tiene que $(\tau_n \to \tau)^{\mathcal{T}} = \tau_n^{\mathcal{T}} \to (\tau^{\mathcal{T}}, cTree)$.

c) Para cada $(e == e') \in C$ se cumple $(\Sigma, T) \vdash_{WT} e\ ::\ \mu\ :\ e'$ para algún tipo $\mu \in Type$. Trivial para C vacía.

d) Para cada $(d \to s) \in C$ se verifica $(\Sigma, T) \vdash_{WT} d\ ::\ \mu\ ::\ s$ para algún tipo $\mu \in Type$. Trivial para C vacía.

■

A.2.8. Demostración del Teorema 4.4.6

Comenzamos la prueba de este teorema, enunciado en la página 97, probando que una aproximación o igualdad estricta es deducible en SC a partir de $P_{\mathcal{A}}$ si y sólo si su transformada es deducible en $dValSC$ a partir de $P^{\mathcal{T}}$.

Lema A.2.15. *Sean $P_{\mathcal{A}}, P^{\mathcal{T}}$ tales que $P_{\mathcal{A}} \Rightarrow^{\mathcal{T}} P^{\mathcal{T}}$, y sea $Pat_{\perp}$ el conjunto de patrones parciales sobre la signatura de $P_{\mathcal{A}}$. Entonces para todo $t, s \in Pat_{\perp}$:*

1. *Se tiene $P_{\mathcal{A}} \vdash_{SC} s \to t$ sii $P^{\mathcal{T}} \vdash_{dValSC} s^{\mathcal{T}} \to t^{\mathcal{T}}$.*
2. *Se tiene $P_{\mathcal{A}} \vdash_{SC} s == t$ sii $P^{\mathcal{T}} \vdash_{dValSC} s^{\mathcal{T}} = t^{\mathcal{T}}$.*

Demostración

En primer lugar obsérvese que al ser t, s patrones también lo son $t^{\mathcal{T}}$ y $s^{\mathcal{T}}$ y que por tanto la regla DV no puede utilizarse en la demostración en ninguno de los dos apartados, al igual que tampoco se utilizará la regla $AR + FA$. Por tanto basta con probar el resultado sobre el cálculo SC en lugar de sobre $dValSC$, es decir:

1. $P_{\mathcal{A}} \vdash_{SC} s \to t$ sii $P^{\mathcal{T}} \vdash_{SC} s^{\mathcal{T}} \to t^{\mathcal{T}}$.
2. $P_{\mathcal{A}} \vdash_{SC} s == t$ sii $P^{\mathcal{T}} \vdash_{SC} s^{\mathcal{T}} = t^{\mathcal{T}}$.

Vemos cada uno de los apartados por separado:

1. $P_{\mathcal{A}} \vdash_{SC} s \to t$ sii $P^{\mathcal{T}} \vdash_{SC} s^{\mathcal{T}} \to t^{\mathcal{T}}$. Razonamos por inducción completa sobre el tamaño del árbol de prueba, distinguiendo casos según la forma de s (para la implicación de izquierda o derecha) o $s^{\mathcal{T}}$ (para la implicación de derecha a izquierda). Comprobamos ambas implicaciones simultáneamente.

   - $s \equiv \perp$. Esto sucede sii $t \equiv \perp$, sii $s^{\mathcal{T}} \equiv t^{\mathcal{T}} \equiv \perp$. En este caso ambas demostraciones constan de un único paso BT.
   - Si $s \equiv X$, $X \in Var$, entonces tiene que ser $t \equiv X$, y esto sii $s^{\mathcal{T}} \equiv t^{\mathcal{T}} \equiv X$. Ambas demostraciones constan de un sólo paso RR.
   - $s \equiv h\ \overline{s}_m$ con $h \in DC^n$ para $n > m$ o $h \in FS^n$ para $n > m + 1$. En este caso t debe ser $t \equiv h\ \overline{t}_m$ y esto ocurre sii (regla (TR_5)) $s^{\mathcal{T}} \equiv h^{\mathcal{T}}_m\ \overline{s}^{\mathcal{T}}_m\ t^{\mathcal{T}} \equiv h^{\mathcal{T}}_m\ \overline{t}^{\mathcal{T}}_m$.
     La demostración de $P_{\mathcal{A}} \vdash_{SC} s \to t$, si existe, debe acabar con un paso DC de la forma:

     $$\frac{s_1 \to t_1\ \ldots\ s_m \to t_m}{h\ \overline{s}_m \to h\ \overline{t}_m}$$

     mientras que la de $P_{\mathcal{A}} \vdash_{SC} s^{\mathcal{T}} \to t^{\mathcal{T}}$, si existe, acabará con un paso DC de la forma:

     $$\frac{s^{\mathcal{T}}_1 \to t^{\mathcal{T}}_1\ \ldots\ s^{\mathcal{T}}_m \to t^{\mathcal{T}}_m}{h^{\mathcal{T}}\ \overline{s^{\mathcal{T}}}_m \to h^{\mathcal{T}}\ \overline{t^{\mathcal{T}}}_m}$$

     Pero por hipótesis de inducción se tiene que para $i = 1 \ldots m$ se cumple $P_{\mathcal{A}} \vdash_{SC} s_i \to t_i$ sii $P_{\mathcal{A}} \vdash_{SC} s^{\mathcal{T}}_i \to t^{\mathcal{T}}_i$, y por tanto la existencia de cualquiera de las dos demostraciones asegura la de la otra, como queríamos probar.

- $s \equiv h\ \overline{s}_m$ con $h \in DC^m$. En este caso t debe ser $t \equiv h\ \overline{t}_m$ y esto ocurre sii $s^{\mathcal{T}} \equiv h\ \overline{s}^{\mathcal{T}}_m$, $t^{\mathcal{T}} \equiv h\ \overline{t}^{\mathcal{T}}_m$ (regla (TR_4)), y el resultado se tiene de forma análoga al punto anterior.
- $s \equiv f\ \overline{s}_m$ con $f \in FS^{m+1}$. En este caso t debe ser $t \equiv f\ \overline{t}_m$ y esto ocurre sii $s^{\mathcal{T}} \equiv f^{\mathcal{T}}\ \overline{s}^{\mathcal{T}}_m$, $t^{\mathcal{T}} \equiv f^{\mathcal{T}}\ \overline{t}^{\mathcal{T}}_m$ (regla (TR_6)), y el resultado se tiene de forma análoga a los puntos anteriores, al tratarse tanto $f^{\mathcal{T}}\ \overline{s}^{\mathcal{T}}_m$ como $f^{\mathcal{T}}\ \overline{t}^{\mathcal{T}}_m$ de aplicaciones parciales ($f^{\mathcal{T}}$ tiene aridad m+1) y por tanto de patrones.

2. $P_{\mathcal{A}} \vdash_{SC} s == t$ sii $P^{\mathcal{T}} \vdash_{SC} s^{\mathcal{T}} = t^{\mathcal{T}}$. La demostración de $P_{\mathcal{A}} \vdash_{SC} s == t$ puede realizarse sii existe algún patrón $u \in Pat$ tal que $P_{\mathcal{A}} \vdash_{SC} s \to u$, $P_{\mathcal{A}} \vdash_{SC} t \to u$, ya que en este caso la prueba puede concluir con un paso DC de la forma:

$$\frac{s \to u \quad t \to u}{s == t}$$

y DC es la única regla del cálculo SC aplicable a una igualdad estricta. Análogamente, la prueba de $P^{\mathcal{T}} \vdash_{SC} s^{\mathcal{T}} == t^{\mathcal{T}}$ depende de la existencia de un u tal que $P^{\mathcal{T}} \vdash_{SC} s^{\mathcal{T}} \to u$, $P^{\mathcal{T}} \vdash_{SC} t^{\mathcal{T}} \to u$. Y como por el apartado anterior $P_{\mathcal{A}} \vdash_{SC} s \to u$ sii $P^{\mathcal{T}} \vdash_{SC} s^{\mathcal{T}} \to u$ y $P_{\mathcal{A}} \vdash_{SC} t \to u$ sii $P^{\mathcal{T}} \vdash_{SC} t^{\mathcal{T}} \to u$, llegamos a que $P_{\mathcal{A}} \vdash_{SC} s == t$ sii $P_{\mathcal{A}} \vdash_{SC} s^{\mathcal{T}} = t^{\mathcal{T}}$.

□

El siguiente lema indica la relación entre la transformación $^{\mathcal{T}}$ y patrones afectados por sustituciones:

Lema A.2.16. *Sean $P_{\mathcal{A}}, P^{\mathcal{T}}$ tales que $P_{\mathcal{A}} \Rightarrow^{\mathcal{T}} P^{\mathcal{T}}$. Sea $\Sigma = \langle TC,\ DC,\ FS\rangle$ la signatura de $P_{\mathcal{A}}$ y $Pat_\perp$ y $Subst_\perp$, respectivamente, el conjunto de patrones parciales y de sustituciones parciales sobre la Σ. Sean $Pat^{\mathcal{T}}_\perp$, $Subst^{\mathcal{T}}_\perp$, los conjuntos de patrones y de sustituciones parciales sobre la signatura de $P^{\mathcal{T}}$, respectivamente. Entonces para todo $t \in Pat_\perp$, se cumple*

1. *Dado $\theta \in Subst_\perp$ se tiene $(t\theta)^{\mathcal{T}} = t^{\mathcal{T}}\mu$, donde $\mu \in Subst_{\mathrm{inv}}$ es tal que:*
   a) *$dom(\mu) = dom(\theta)$.*
   b) *$\mu(X) = \theta(X)^{\mathcal{T}}$ para toda $X \in dom(\mu)$.*
2. *Dado $\mu \in Subst_{\mathrm{inv}}$ entonces $t^{\mathcal{T}}\mu = (t\theta)^{\mathcal{T}}$, donde $\theta \in Subst_\perp$ es tal que:*
   a) *$dom(\theta) = dom(\mu)$.*
   b) *$\theta(X) = \mu(X)^{\mathcal{T}^{-1}}$ para toda $X \in dom(\theta)$.*

Demostración

1. De la definición de μ se tiene que efectivamente $\mu \in Subst_{inv}$ ya que para cada $X \in dom(\mu)$, $\mu(X) = \theta(X)^{\mathcal{T}}$, con $\theta(X) \in Pat_{\perp}$ y $t^{\mathcal{T}} \in Pat_{inv}$ para todo $t \in Pat_{\perp}$ por el apartado 1) de la proposición 4.4.2 (pág. 93).

   Comprobamos inducción estructural sobre t que $(t\theta)^{\mathcal{T}} = t^{\mathcal{T}}\mu$:

   - $t \equiv \perp$.
   $$\begin{aligned}(t\theta)^{\mathcal{T}} &= \perp^{\mathcal{T}} = \ (\mathrm{TR}_2) \ = \perp \\ t^{\mathcal{T}}\mu &= \ (\mathrm{TR}_2) \ = \perp \mu = \perp\end{aligned}$$
   - $t \equiv X$, $X \in Var$.
   $$\begin{aligned}(t\theta)^{\mathcal{T}} &= \theta(X)^{\mathcal{T}} \\ t^{\mathcal{T}}\mu &= \ (\mathrm{TR}_3) \ = X\mu = \theta(X)^{\mathcal{T}}\end{aligned}$$
   - $t \equiv h\ \bar{t}_m$, con $t_i \in Pat_{\perp}$ para $i = 1 \ldots m$, $h \in DC^m$.
   $$\begin{aligned}(t\theta)^{\mathcal{T}} &= (h\ \bar{t}_m\theta)^{\mathcal{T}} = \ (\mathrm{TR}_4) \ = h\ (\bar{t}_m\theta)^{\mathcal{T}} = \ \text{h. i.} \ = h\ \bar{t}_m^{\mathcal{T}}\mu \\ t^{\mathcal{T}}\mu &= \ (\mathrm{TR}_4) \ = (h\ \bar{t}_m^{\mathcal{T}})\mu = h\ \bar{t}_m^{\mathcal{T}}\mu\end{aligned}$$
   - $t \equiv h\ \bar{t}_m$, con $t_i \in Pat_{\perp}$ para $i = 1 \ldots m$, $h \in DC^n$, $n > m$ o $h \in FS^n$, $n > m+1$.
   $$\begin{aligned}(t\theta)^{\mathcal{T}} &= (h\ \bar{t}_m\theta)^{\mathcal{T}} = \ (\mathrm{TR}_5) \ = h_m^{\mathcal{T}}\ (\bar{t}_m\theta)^{\mathcal{T}} = \ \text{h. i.} \ = h_m^{\mathcal{T}}\ \bar{t}_m^{\mathcal{T}}\mu \\ t^{\mathcal{T}}\mu &= \ (\mathrm{TR}_5) \ = (h_m^{\mathcal{T}}\ \bar{t}_m^{\mathcal{T}})\mu = h_m^{\mathcal{T}}\ \bar{t}_m^{\mathcal{T}}\mu\end{aligned}$$
   - $t \equiv f\ \bar{t}_m$, con $t_i \in Pat_{\perp}$ para $i = 1 \ldots m$, $f \in FS^{m+1}$.
   $$\begin{aligned}(t\theta)^{\mathcal{T}} &= (f\ \bar{t}_m\theta)^{\mathcal{T}} = \ (\mathrm{TR}_6) \ = f^{\mathcal{T}}\ (\bar{t}_m\theta)^{\mathcal{T}} = \ \text{hip. ind.} \ = f^{\mathcal{T}}\ \bar{t}_m^{\mathcal{T}}\mu \\ t^{\mathcal{T}}\mu &= \ (\mathrm{TR}_6) \ = (f^{\mathcal{T}}\ \bar{t}_m^{\mathcal{T}})\mu = f^{\mathcal{T}}\ \bar{t}_m^{\mathcal{T}}\mu\end{aligned}$$

2. De la definición de θ se tiene que efectivamente $\theta \in Subst_{\perp}$ ya que para cada $X \in dom(\theta)$, $\theta(X) = \mu(X)^{\mathcal{T}^{-1}}$ y para todo $t \in Pat_{inv}$ se cumple $t^{\mathcal{T}^{-1}} \in Pat_{\perp}$ por el apartado 2) de la proposición 4.4.2 (pág. 93).

   Comprobamos inducción estructural sobre t que $(t\theta)^{\mathcal{T}} = t^{\mathcal{T}}\mu$:

   - $t \equiv \perp$.
   $$\begin{aligned}(t\theta)^{\mathcal{T}} &= \perp^{\mathcal{T}} = \ (\mathrm{TR}_2) \ = \perp \\ t^{\mathcal{T}}\mu &= \ (\mathrm{TR}_2) \ = \perp \mu = \perp\end{aligned}$$
   - $t \equiv X$, $X \in Var$.
   $$\begin{aligned}(t\theta)^{\mathcal{T}} &= \theta(X)^{\mathcal{T}} = (\mu(X)^{\mathcal{T}^{-1}})^{\mathcal{T}} = \ (\text{prop. 4.4.2, ap.2}) == \mu(X) \\ t^{\mathcal{T}}\mu &= \ (\mathrm{TR}_3) \ = X\mu = \mu(X)\end{aligned}$$

- $t \equiv h\ \bar{t}_m$, con $t_i \in Pat_\perp$ para $i = 1 \ldots m$, $h \in DC^m$. Idéntico a la demostración de este mismo caso para del apartado 1 de este mismo lema.
- $t \equiv h\ \bar{t}_m$, con $t_i \in Pat_\perp$ para $i = 1 \ldots m$, $h \in DC^n$, $n > m$ o $h \in FS^n$, $n > m+1$. Idéntico a la demostración de este mismo caso para del apartado 1 de este mismo lema.
- $t \equiv f\ \bar{t}_m$, con $t_i \in Pat_\perp$ para $i = 1 \ldots m$, $f \in FS^{m+1}$. Idéntico a la demostración de este mismo caso para del apartado 1 de este mismo lema.

□

Corolario A.2.17. *Sea $P_\mathcal{A}$, $P^\mathcal{T}$, $Pat_\perp$, $Pat^\mathcal{T}_\perp$, $Subst_\perp$, θ y μ como en el lema anterior. Sean $a, b \in Pat_\perp$ y $a^\mathcal{T}, b^\mathcal{T} \in Pat^\mathcal{T}_\perp$ tales que $a \Rightarrow^\mathcal{T} a^\mathcal{T}$ y $b \Rightarrow^\mathcal{T} b^\mathcal{T}$. Entonces se cumplen:*

- $P_\mathcal{A} \vdash_{SC} a \rightarrow b\theta$ *sii* $P^\mathcal{T} \vdash_{dValSC} a^\mathcal{T} \rightarrow b^\mathcal{T}\mu$.
- $P_\mathcal{A} \vdash_{SC} a\theta \rightarrow b$ *sii* $P^\mathcal{T} \vdash_{dValSC} a^\mathcal{T}\mu \rightarrow b^\mathcal{T}$.

Demostración

- Por el lema A.2.15 $P_\mathcal{A} \vdash_{SC} a \rightarrow b\theta$ sii $P^\mathcal{T} \vdash_{SC} a^\mathcal{T} \rightarrow (b\theta)^\mathcal{T}$, y por el lema A.2.16 $(b\theta)^\mathcal{T} = b^\mathcal{T}\mu$, de donde se tiene el resultado.
- Análogo: por el lema A.2.15 $P_\mathcal{A} \vdash_{SC} a\theta \rightarrow b$ sii $P^\mathcal{T} \vdash_{SC} (a\theta)^\mathcal{T} \rightarrow b^\mathcal{T}$, y por el lema A.2.16 $(a\theta)^\mathcal{T} = a^\mathcal{T}\mu$, de donde se tiene el resultado.

□

El teorema va a ser consecuencia del siguiente lema, que establece un resultado análogo pero sobre el programa aplanado.

Lema A.2.18. *Sean $P_\mathcal{A}$ un programa plano y f una función de aridad n definida en $P_\mathcal{A}$. Sea $P^\mathcal{T}$ un programa tal que $P_\mathcal{A} \Rightarrow^\mathcal{T} P$. Sean además $\bar{t}_n$, patrones parciales cualesquiera sobre la signatura de $P_\mathcal{A}$ y t un patrón sobre la misma signatura, $t \neq \perp$. Entonces:*

(i) Si $P_\mathcal{A} \vdash_{SC} f\ \bar{t}_n \rightarrow t$ y apa es un APA para esta demostración, que por la proposición 3.3.1 (pág. 63) debe tener la forma $\begin{array}{l} f\ \bar{t}_n \rightarrow t \\ \mid \\ T \end{array}$ *para cierto árbol T.*

Entonces se cumple $P^\mathcal{T} \vdash_{dValSC} f^\mathcal{T}\ \overline{t^\mathcal{T}_n} \rightarrow (t^\mathcal{T}, ct)$, donde ct :: $cTree$ es un patrón total representando T.

(ii) Si $P^\mathcal{T} \vdash_{dValSC} f^\mathcal{T}\ \overline{t^\mathcal{T}_n} \rightarrow (t^\mathcal{T}, ct)$ entonces $P_\mathcal{A} \vdash_{SC} f\ \bar{t}_n \rightarrow t$. Además si ct es un patrón total, entonces representa un árbol T tal que $\begin{array}{l} f\ \bar{t}_n \rightarrow t \\ \mid \\ T \end{array}$ *es un APA para $P_\mathcal{A} \vdash_{SC} f\ \bar{t}_n \rightarrow t$.*

Demostración

En la demostración vamos a representar el conjunto de los patrones parciales, de constructoras y de símbolos de función sobre la signatura de $P_{\mathcal{A}}$ respectivamente por $Pat_{\perp}$, DC y FS.

Comenzamos por establecer una relación entre ambos apartados que nos permitirá probarlos simultáneamente:

(i) Si se cumple $P_{\mathcal{A}} \vdash_{SC} f\ \overline{t}_n \to t$, tal y como indica (i), el último paso de esta demostración que debe ser necesariamente un paso $AR + FA$ (al ser $t \neq \perp$) de la forma:

$$\textbf{(1)}\quad \frac{t_1 \to t'_1\theta\ \ldots\ t_n \to t'_n\theta \quad \boxed{f\ \overline{t'}_n\theta \to u} \quad u\ \to t}{f\ \overline{t}_n\ \to t} \qquad \frac{C'\theta \quad r'\theta \to u}{\boxed{f\ \overline{t'}_n\theta \to u}}$$

donde $(f\ \overline{t'}_n \to r' \Leftarrow C') \in P_{\mathcal{A}}$ y $\theta \in Subst_{\perp}$. Supongamos que

$$\textbf{(2)}\qquad (f\ \overline{t'}_n \to r' \Leftarrow C') \Rightarrow^{\mathcal{T}} (f^{\mathcal{T}}\ \overline{s}_n \to (s,T) \Leftarrow C)$$

es decir que $(f^{\mathcal{T}}\ \overline{s}_n \to (s,T) \Leftarrow C)$ es la transformada de la regla aplanada utilizada en este último paso. Entonces vamos a probar que $P^{\mathcal{T}} \vdash_{dValSC} f^{\mathcal{T}}\ \overline{t}^{\mathcal{T}}_n \to (t^{\mathcal{T}}, ct)$ se puede probar con un último paso de la forma:

(3)

$$\frac{t_1^{\mathcal{T}} \to s_1(\mu \biguplus \nu)\ \ldots\ t_n^{\mathcal{T}} \to s_n(\mu \biguplus \nu) \quad \boxed{f^{\mathcal{T}}\ \overline{s}_n(\mu \biguplus \nu) \to (u^{\mathcal{T}}, ct)} \quad (u^{\mathcal{T}}, ct)\ \to (t^{\mathcal{T}}, ct)}{f^{\mathcal{T}}\ \overline{t}^{\mathcal{T}}_n\ \to (t^{\mathcal{T}}, ct)} \qquad \frac{C(\mu \biguplus \nu) \quad (s,T)(\mu \biguplus \nu) \to (u^{\mathcal{T}}, ct)}{\boxed{f^{\mathcal{T}}\ \overline{s}_n(\mu \biguplus \nu) \to (u^{\mathcal{T}}, ct)}}$$

en el que se ha usado una instancia

$$(f^{\mathcal{T}}\ \overline{s}_n \to (s,T) \Leftarrow C)(\mu \bigcup\!\!\biguplus \nu)$$

donde

- $\mu \in Subst_{inv}$ tal que $dom(\mu) = dom(\theta)$, con θ la sustitución empleada en (1) y $\mu(X) = \theta(X)^{\mathcal{T}}$ para cada $X \in dom(\theta)$.
- $\nu \in Subst_{\perp}$ tal que $dom(\nu) = \{T_1, \ldots, T_m, T\}$, con $T_1, \ldots, T_m, T$ las variables nuevas de tipo $cTree$ introducidas durante la transformación (2). La definición concreta de cada $\nu(T_i)$ para $i = 1 \ldots m$ y de $\nu(T)$ se verá durante la demostración.

Obsérvese que la definición de $(\mu \biguplus \nu)$ asegura que la instancia utilizada en (3) es una instancia admisible tal y como requiere la definición del cálculo $dValSC$ (def. 4.4.2, pág. 91). Además ct representará el APA de (1) salvo por la ausencia de la raíz como indica el enunciado del lema.

(ii) A la inversa, si suponemos que, tal y como indica el apartado (ii), $P^{\mathcal{T}} \vdash_{dValSC} f^{\mathcal{T}}\ \overline{t_n^{\mathcal{T}}} \rightarrow (t^{\mathcal{T}}, ct)$ tenemos que el último paso de la demostración debe corresponder a la aplicación de la regla $AR+FA$ y ser de la forma indicada en (3), donde se ha utilizado una instancia admisible $(\mu \biguplus \nu)$ de la regla $(f^{\mathcal{T}}\ \overline{s}_n \rightarrow (s, T) \Leftarrow C)$ del programa transformado $P^{\mathcal{T}}$, correspondiente a una regla de $P_{\mathcal{A}}$ tal y como indica (2). Por la definición de instancia admisible ν y μ han de ser como se ha dicho al ver la implicación en el otro sentido. Entonces vamos a probar que $P_{\mathcal{A}} \vdash_{SC} f\ \overline{t}_n \rightarrow t$ con una demostración cuyo último paso es el indicado en (1), donde se ha utilizado la regla cuya transformada se utiliza en (3) con una instancia θ tal que

- $dom(\theta) = dom(\mu)$
- $\theta(X) = \mu(X)^{\mathcal{T}^{-1}}$

y tal que ct representa un APA para esa demostración salvo por la ausencia de la raíz.

Estos dos puntos nos muestran que basta con probar que (1) se puede probar sii (3), con ct de la forma adecuada, lo que probará ambos puntos (i) e (ii) simultáneamente.

Para ello vamos a probar que cada una de las premisas de (3) se puede probar sii se puede probar la correspondiente premisa de (1).

1. Premisas $t_i \rightarrow t_i'\theta$ de (1) y sus correspondientes premisas de (3).

   Para cada $i = 1 \ldots n$ se cumple

   $$\textbf{(4)} \qquad P^{\mathcal{T}} \vdash_{dValSC} t_i^{\mathcal{T}} \rightarrow s_i(\mu \biguplus \nu)$$

   sii

   $$\textbf{(5)} \qquad P_{\mathcal{A}} \vdash_{SC} t_i \rightarrow t_i'\theta$$

   Como $s_i \in Pat_\perp$ tenemos que durante su transformación (reglas (TR_2)-(TR_6)) no se introducen variables nuevas, por lo que $s_i(\mu \biguplus \nu) = s_i\mu$. Además, por construcción de las reglas transformadas (regla (TR_1)) se tiene que $s_i \equiv (t_i')^{\mathcal{T}}$ por lo que lo que hay que probar es que se cumple $P^{\mathcal{T}} \vdash_{dValSC} t_i^{\mathcal{T}} \rightarrow (t_i')^{\mathcal{T}}\mu$, y por el corolario A.2.17 esto se cumple sii $P_{\mathcal{A}} \vdash_{SC} t_i \rightarrow t_i'\theta$ (obsérvese que la relación entre θ y μ es aquí la indicada en el corolario).

2. Premisas $C'\theta$ de (1) y sus correspondientes premisas de (3).

   $P^{\mathcal{T}} \vdash_{dValSC} C(\theta \biguplus \nu)$. Según la regla ($\mathrm{TR}_1$) se tiene que C es de la forma

   $$\textbf{(6)} \qquad (C')^{\mathcal{T}}, \quad ctNode\ \text{"}f.k\text{"}\ \ [dVal\, s_1, \ldots, dVal\, s_n]\ (dVal\, s)\ (ctClean\ [(dVal\, r_1, T_1), \ldots, (dVal\, r_m, T_m)]) \rightarrow T$$

   es decir que las primeras condiciones atómicas provienen de la transformación de la condición C' de la regla aplanada y al final se añade una nueva aproximación que tiene como lado derecho T. Tenemos que probar entonces:

2.a) $P^{\mathcal{T}} \vdash_{dValSC} c^{\mathcal{T}}(\theta \biguplus \nu)$ para $c^{\mathcal{T}}$ atómica, $c^{\mathcal{T}} \in (C')^{\mathcal{T}}$. En este caso se tiene que debe existir una $c' \in C'$ atómica tal que $c' \Rightarrow^{\mathcal{T}} c^{\mathcal{T}}$, ya que las reglas ($\mathrm{TR}_7$)-($\mathrm{TR}_{11}$) transforman condiciones atómicas en condiciones atómicas. Además para cada $c'\theta$ se tiene

$$\textbf{(7)} \qquad P_{\mathcal{A}} \vdash_{SC} c'\theta$$

ya que $P_{\mathcal{A}} \vdash_{SC} C'\theta$ por (1). Vamos a fijarnos en las posibles formas de c', que al ser una condición plana (definición 4.3.1, pág. 75) admite las siguientes posibilidades:

2.a.1) $c' \equiv \ a == b$ con $a, b \in Pat_{\perp}$. Entonces $(a == b) \Rightarrow^{\mathcal{T}} (a^{\mathcal{T}} == b^{\mathcal{T}};)$ (regla (TR_7)), por lo que $c^{\mathcal{T}} \equiv a^{\mathcal{T}} == b^{\mathcal{T}}$. La fórmula (7) se escribe en este caso como $P_{\mathcal{A}} \vdash_{SC} a\theta == b\theta$, lo que por el lema A.2.15 nos lleva a que se cumple $P^{\mathcal{T}} \vdash_{dValSC} (a\theta)^{\mathcal{T}} == (b\theta)^{\mathcal{T}}$. Aplicando dos veces el lema A.2.16 esto equivale a que se cumpla $P^{\mathcal{T}} \vdash_{dValSC} (a^{\mathcal{T}})\mu == (b^{\mathcal{T}})\mu$, es decir $P^{\mathcal{T}} \vdash_{dValSC} c^{\mathcal{T}}\mu$. Pero al ser a y b patrones en $a^{\mathcal{T}}$ y en $b^{\mathcal{T}}$ no aparecerá ninguna variable nueva T_i (las reglas de transformación de patrones no introducen variables nuevas) por lo que $P^{\mathcal{T}} \vdash_{dValSC} c^{\mathcal{T}}\mu$ equivale a $P^{\mathcal{T}} \vdash_{dValSC} c^{\mathcal{T}}(\mu \biguplus \nu)$, que es lo que queríamos probar.

2.a.2) $c' \equiv \ a \to b$, con $a \in Pat_{\perp}$, $b \in Pat_{\perp}$. Entonces, por la regla (TR_8) $(a \to b) \Rightarrow^{\mathcal{T}} (a^{\mathcal{T}} \to b^{\mathcal{T}};)$, de donde $c^{\mathcal{T}} \equiv a^{\mathcal{T}} \to b^{\mathcal{T}}$, y el argumento es análogo al punto anterior al ser a, b patrones.

2.a.3) $c' \equiv \ X\ a \to b$, con $X \in Var, a \in Pat_{\perp}$, $b \in Pat_{\perp}$.
Por la regla (TR_9) se tiene que

$$X\ a \to b \Rightarrow^{\mathcal{T}} (X\ a^{\mathcal{T}} \to (b^{\mathcal{T}}, T_X)\,;\,(b^{\mathcal{T}}, T_X))$$

es decir

$$\textbf{(8)} \qquad c^{\mathcal{T}} \equiv X\ a^{\mathcal{T}} \to (b^{\mathcal{T}}, T_X)$$

De (7) tenemos que se cumple $P_{\mathcal{A}} \vdash_{SC} (X\ a \to b)\theta$ y que la prueba es parte de la demostración (1). Para que esta prueba exista $\theta(X) \in Pat_{\perp}$ puede ser de una de las siguientes formas:

- $\theta(X) = g\ \overline{a}_m$ con $g \in DC^{m+1}$, $a_i \in Pat_{\perp}$ para $i = 1 \ldots m$. Entonces tenemos en (1) la prueba de $c'\theta$, es decir de

$$\textbf{(9)} \qquad P_{\mathcal{A}} \vdash_{SC} g\ \overline{a}_m\ a\theta \to b\theta$$

Se tiene que $\mu(X) = \theta(X)^{\mathcal{T}} = \ (\mathrm{TR}_5) \ = g_m^{\mathcal{T}}\ \overline{a}_m^{\mathcal{T}}$ En este caso debe ser $\mu(T_X) = ctVoid$ (el árbol devuelto por las funciones auxiliares). Entonces

$$c^{\mathcal{T}}(\mu \biguplus \nu) = (X\ a^{\mathcal{T}} \to (b^{\mathcal{T}}, T_X))(\mu \biguplus \nu) =$$
$$\mu(X)\ a^{\mathcal{T}}\mu \to (b^{\mathcal{T}}\mu, \nu(T_X)) == g_m^{\mathcal{T}}\ \overline{a}_m^{\mathcal{T}}\ (a^{\mathcal{T}}\mu) \to (b^{\mathcal{T}}\mu, cTVoid)$$

y debemos comprobar que la prueba (9) existe en (1) sii existe en (3) la prueba de

$$\textbf{(10)} \qquad P^{\mathcal{T}} \vdash_{dValSC} g_m^{\mathcal{T}}\ \overline{a}_m^{\mathcal{T}}\ (a^{\mathcal{T}}\mu) \rightarrow (b^{\mathcal{T}}\mu, cTVoid)$$

Esta relación entre ambas pruebas no se deduce por hipótesis de inducción ya que g_m no es una función de FS, ni del lema A.2.16 al no ser $g_m^{\mathcal{T}}\ \overline{a}_m^{\mathcal{T}}\ a^{\mathcal{T}}\mu$ un patrón sino una expresión evaluable, ya que por el apartado 4.4.3 (pág. 83) $g_m^{\mathcal{T}}$ es una función de aridad $m+1$ definida por una única regla:

$$\textbf{(11)} \qquad g_m^{\mathcal{T}}\ \overline{X}_{m+1} \rightarrow\ (g\ \overline{X}_{m+1}, \text{ctVoid})$$

En el caso en que $b\theta \equiv \bot$, la prueba (9) se tiene trivialmente (regla BT) y (10) también se cumple finalizando la prueba con un paso $AR + FA$ de la forma:

$$\frac{a_1^{\mathcal{T}} \rightarrow \bot\ \ldots\ a_m \rightarrow \bot \quad a^{\mathcal{T}}\mu \rightarrow \bot \quad \boxed{g_m^{\mathcal{T}}\ \overline{\bot}_{m+1} \rightarrow u} \quad u \rightarrow u}{g_m^{\mathcal{T}}\ \overline{a}_m^{\mathcal{T}}\ a^{\mathcal{T}}\mu \rightarrow u} \qquad \frac{(g\ \overline{\bot}_{m+1}, \text{ctVoid}) \rightarrow u}{\boxed{g_m^{\mathcal{T}}\ \overline{\bot}_{m+1} \rightarrow u}}$$

en el que hemos llamado u a $(\bot, \text{ctVoid})$ y hemos representado $\underbrace{\bot \cdots \bot}_{m+1}$ por $\overline{\bot}_{m+1}$ para abreviar. En este paso se ha usado una instancia σ de (11) tal que $\sigma(X_i) = \bot$ para $i = 1 \ldots m+1$. Obsérvese que la premisa $u \rightarrow u$, es decir $(\bot, \text{ctVoid}) \rightarrow (\bot, \text{ctVoid})$ se puede probar fácilmente aplicando DC y después BT para $\bot \rightarrow \bot$ y DC de nuevo para ctVoid $\rightarrow$ ctVoid. Análogamente se prueba la premisa $(g\ \overline{\bot}_{m+1}, \text{ctVoid}) \rightarrow u$, mientras que el resto de la premisas sólo requieren la regla BT para ser demostradas.

Supongamos ahora que $b\theta \neq \bot$, lo que implica $b^{\mathcal{T}}\mu \neq \bot$.
Entonces la prueba de (9) sólo resulta posible si ha terminado con un paso DC, correspondiente a la única regla aplicable al ser el lado izquierdo $g\ \overline{a}_m\ a\theta$ con $g \in DC^{m+1}$ y el lado derecho distinto de $b\theta \neq \bot$. Para que DC sea aplicable debe suceder que $b \equiv g\ \overline{b}_{m+1}$ y entonces este último paso será de la forma:

$$\textbf{(12)} \qquad \frac{a_1 \rightarrow b_1\theta\ \ldots\ a_m \rightarrow b_m\theta \quad a\theta \rightarrow b_{m+1}\theta}{g\ \overline{a}_m\ a\theta \rightarrow b\theta}$$

Nos fijamos a continuación en la prueba de (10). Esta debería acabar con un paso $AR + FA$ en el que se utilizara una instancia

$(g^{\mathcal{T}}_m\ \overline{X}_{m+1} \rightarrow\ (g\ \overline{X}_{m+1}, \mathrm{ctVoid}))\sigma$ de la regla (11), que es la única en la definición de $g^{\mathcal{T}}_m$. Vamos a llamar d_i a $a^{\mathcal{T}}_i$ para $i = 1 \ldots m$, d_{m+1} a $(a\theta)^{\mathcal{T}}$ y a probar que si $\sigma(X_i) = d_i$ para $i = 1 \ldots m+1$ esta demostración puede hacerse con un último paso de la forma::

(13)

$$\frac{a^{\mathcal{T}}_1 \rightarrow d_1\ \ldots\ a^{\mathcal{T}}_m \rightarrow d_m\ \ a^{\mathcal{T}}\mu \rightarrow d_{m+1}\ \ \boxed{g^{\mathcal{T}}_m\ \overline{d}_{m+1} \rightarrow u}\ \ u\ \rightarrow (b^{\mathcal{T}}\mu, \mathrm{ctVoid})}{g^{\mathcal{T}}_m\ \overline{a}^{\mathcal{T}}_m\ a^{\mathcal{T}}\mu \rightarrow (b^{\mathcal{T}}\mu, \mathrm{ctVoid})}\ \frac{(g\ \overline{d}_{m+1}, \mathrm{ctVoid}) \rightarrow u}{g^{\mathcal{T}}_m\ \overline{d}_{m+1} \rightarrow u}$$

Debemos comprobar entonces que (12) se cumple en SC sii (13) se cumple en $dValSC$. Para (13) se cumpla deben cumplirse sus premisas, y para esto en particular debe ser $u \equiv (u_1, \mathrm{ctVoid})$ ya que en otro caso no se podría tener la prueba para $u\ \rightarrow (b^{\mathcal{T}}\mu, \mathrm{ctVoid})$. Entonces las premisas de (13)

(14) $\quad P^{\mathcal{T}} \vdash_{dValSC}\ (g\ \overline{d}_{m+1}, \mathrm{ctVoid}) \rightarrow (u_1, \mathrm{ctVoid})$

(15) $\quad P^{\mathcal{T}} \vdash_{dValSC}\ (u_1, \mathrm{ctVoid}) \rightarrow (b^{\mathcal{T}}\mu, \mathrm{ctVoid})$

se podrán probar mediante un paso DC sii el valor $u_1 \in Pat_{\perp}$ verifica:

(16) $\quad P^{\mathcal{T}} \vdash_{dValSC}\ g\ \overline{d}_{m+1} \rightarrow u_1$

(17) $\quad P^{\mathcal{T}} \vdash_{dValSC}\ u_1 \rightarrow b^{\mathcal{T}}\mu$

(17) se cumplirá, al ser $b^{\mathcal{T}}\mu \in Pat_{\perp}$ y por el apartado 1 de la proposición 3.2.1[2] (pág. 57) sii $u_1 \sqsupseteq b^{\mathcal{T}}\mu$, y por (16) y por el apartado 1 de la proposición 3.3.2 (pág. 63) esto último se cumplirá sii

$$P^{\mathcal{T}} \vdash_{dValSC}\ g\ \overline{d}_{m+1} \rightarrow b^{\mathcal{T}}\mu$$

es decir sii, recordando la definición de d_i

$$P^{\mathcal{T}} \vdash_{dValSC}\ g\ \overline{a}^{\mathcal{T}}_m\ a\theta^{\mathcal{T}} \rightarrow b^{\mathcal{T}}\mu$$

o lo que es lo mismo, ya que por el lema A.2.16 $(b\theta)^{\mathcal{T}} = b^{\mathcal{T}}\mu$:

(18) $\quad P^{\mathcal{T}} \vdash_{dValSC}\ g\ \overline{a}^{\mathcal{T}}_m\ a\theta^{\mathcal{T}} \rightarrow (b\theta)^{\mathcal{T}}$

Como $g \in DC^{m+1}$, por la regla de transformación (TR_4):

(19) $\quad (g\ \overline{a}_m\ a\theta)^{\mathcal{T}} \rightarrow (b\theta)^{\mathcal{T}}\ =\ g\ \overline{a}^{\mathcal{T}}_m\ a\theta^{\mathcal{T}} \rightarrow (b\theta)^{\mathcal{T}}$

[2]Hay que notar que aunque la proposición 3.2.1 está definida para el cálculo SC y no para $dValSC$ resulta aplicable en este caso, al no ser la regla DV necesaria en ninguna de estas pruebas (estamos tratando con patrones y estamos $dVal$ sólo aparece aplicada totalmente en las pruebas de $dValSC$).

y del lema A.2.15 se tiene que (18) sii (12). Es decir, si se cumple (13) se cumplen sus premisas (14) y (15), y de ambas llegamos a que se cumple (12). A la inversa, si se cumple (12) ya hemos probado que se cumplen las premisas (14) y (15). Falta por ver que en este caso también se cumplen el resto de las premisas de (13), es decir

- $P^{\mathcal{T}} \vdash_{dValSC} a_i^{\mathcal{T}} \rightarrow d_i$ para $i = 1 \ldots m$. Por definición d_i es $a_i^{\mathcal{T}}$ para $i = 1 \ldots m$, por lo que el resultado se tiene del apartado 2 de la proposición 3.2.1 (pág. 57).
- $P^{\mathcal{T}} \vdash_{dValSC} a^{\mathcal{T}}\mu \rightarrow d_{m+1}$. Por definición d_{m+1} es $(a\theta)^{\mathcal{T}}$ y por el lema A.2.16 $(a\theta)^{\mathcal{T}} = a^{\mathcal{T}}\mu$.

- $\theta(X) = g\ \overline{a}_m$ con $g \in DC^n$, $n > m+1$, $a_i \in Pat_\perp$ para $i = 1 \ldots m$. Similar al caso anterior, por lo que nos apoyamos en éste y sólo señalamos las diferencias. Como en el caso anterior debemos comprobar que la prueba (9) existe en (1) sii existe en (3) la prueba de (10). Sin embargo por el apartado 4.4.3 (pág. 83), al ser $n > m+1$, la regla (11) no es válida, ya que en este caso la regla para $g_m^{\mathcal{T}}$ es :

$$\textbf{(20)} \qquad g_m^{\mathcal{T}}\ \overline{X}_{m+1} \rightarrow\ (g_{m+1}^{\mathcal{T}}\ \overline{X}_{m+1}, \mathrm{ctVoid})$$

Esto hace que en el resto de las fórmulas haya que cambiar cada aparición de g en el programa transformado por por $g_{m+1}^{\mathcal{T}}$, pero los razonamientos aplicados se mantienen. Sólo hay que mencionar que en la transformación (19) en esta ocasión se aplica la regla de transformación (TR_5) en lugar de (TR_4), lo que conlleva el mencionado cambio de g por $g_m^{\mathcal{T}}$ para la el patrón transformado:

$$(g\ \overline{a}_m\ a\theta)^{\mathcal{T}} \rightarrow (b\theta)^{\mathcal{T}}\ =\ g_m^{\mathcal{T}}\ \overline{a}_m^{\mathcal{T}}\ a\theta^{\mathcal{T}} \rightarrow (b\theta)^{\mathcal{T}}$$

- $\theta(X) = g\ \overline{a}_m$ con $g \in FS^n$, $a_i \in Pat_\perp$ para $i = 1 \ldots m$. Si $n > m+2$ el razonamiento es idéntico al del punto anterior. Si es $n = m+2$ el razonamiento es idéntico al de $\theta(X) = g\ \overline{a}_m$ con $g \in DC^{m+1}$ ya estudiado. Nos falta por ver cuando $n = m+1$ (no puede dar $n = m$ porque entonces $\theta(X)$ no sería un patrón).

Como en los casos anteriores, en (1) se debe tener la prueba de

$$\textbf{(9)} \qquad P_{\mathcal{A}} \vdash_{SC}\ g\ \overline{a}_m\ a\theta \rightarrow b\theta$$

Si suponemos que esta prueba existe, por hipótesis de inducción tenemos que

$$\textbf{(21)} \qquad P^{\mathcal{T}} \vdash_{dValSC}\ g^{\mathcal{T}}\ \overline{a}_m^{\mathcal{T}} a\theta^{\mathcal{T}} \rightarrow (b\theta^{\mathcal{T}}, ct_g)$$

para ct_g un patrón total representando el APA de (9) salvo por la raíz, como indica el enunciado del lema. Ahora bien, de (8):

$$\begin{aligned} c^{\mathcal{T}}(\mu \uplus \nu) &= \mu(X)\ a^{\mathcal{T}}\mu \to (b^{\mathcal{T}}\mu, \nu(T_X)) = \\ &= (g\ \overline{a}_m)^{\mathcal{T}}\ a^{\mathcal{T}}\mu \to (b^{\mathcal{T}}\mu, \nu(T_X)) = \mathrm{TR}_6 = \\ &= g^{\mathcal{T}}\ \overline{a}_m^{\mathcal{T}}\ a^{\mathcal{T}}\mu \to (b^{\mathcal{T}}\mu, \nu(T_X)) = \text{ lema A.2.16 } = \\ &= g^{\mathcal{T}}\ \overline{a}_m^{\mathcal{T}}\ a\theta^{\mathcal{T}} \to (b\theta^{\mathcal{T}}, \nu(T_X)) \end{aligned}$$

por lo que basta con tomar $\nu(T_x) = ct_g$ para que por (5) se tenga $P^{\mathcal{T}} \vdash_{dValSC} c^{\mathcal{T}}(\mu \uplus \nu)$. A la inversa, si $P^{\mathcal{T}} \vdash_{dValSC}\ c^{\mathcal{T}}(\mu \uplus \nu)$, es decir

$$P^{\mathcal{T}} \vdash_{dValSC}\ g^{\mathcal{T}}\overline{a}_m^{\mathcal{T}}\ a\theta^{\mathcal{T}} \to (b\theta^{\mathcal{T}}, \nu(T_X))$$

con $\nu(T_X)$ un patrón total ct_g, entonces por hipótesis de inducción se puede probar (9) con un APA de la forma $\overset{g\ \overline{a}_m\ a\theta \to b\theta}{\underset{ct_g}{|}}$

Obsérvese que otros valores de $\theta(X)$ como $\theta(X) = g\ \overline{a}_m$ con $g \in DC^m$, $a_i \in Pat_\perp$ para $i = 1 \ldots m$ no son posibles, ya que entonces tendríamos en (1) la prueba de $(g\ \overline{a}_m\ a \to b)\theta$ y esto no es posible y que se tendría una constructora de aridad m aplicada a $m+1$ argumentos.

2.a.4) $c' \equiv\ g\ \overline{a}_m \to b$ con $g \in FS^m$ y $a_i \in Pat_\perp$ para $i = 1 \ldots m$, $b \in Pat_\perp$. En este caso, por la regla (TR_{11}), se tiene que

$$g\ \overline{a}_m \to b \Rightarrow^{\mathcal{T}} (g^{\mathcal{T}}\ \overline{a}_m^{\mathcal{T}} \to (b^{\mathcal{T}}, T_g)\,;\, (b^{\mathcal{T}}, T_g))$$

es decir,

$$\textbf{(22)} \qquad c^{\mathcal{T}} \equiv g^{\mathcal{T}}\ \overline{a}_m^{\mathcal{T}} \to (b^{\mathcal{T}}, T_g)$$

Como $(g\ \overline{a}_m \to b)\theta = g\ \overline{a}_m\theta \to b\theta$, de la fórmula (7) se tiene que en (1) se encuentra entonces la prueba de

$$\textbf{(23)} \qquad P_{\mathcal{A}} \vdash_{SC}\ g\ \overline{a}_m\theta \to b\theta$$

lo que, por hipótesis de inducción, lleva a que se cumple

$$\textbf{(24)} \qquad P^{\mathcal{T}} \vdash_{dValSC}\ g^{\mathcal{T}}\ (\overline{a}_m\theta)^{\mathcal{T}} \to ((b\theta)^{\mathcal{T}}, ct_g)$$

donde ct_g es un patrón total representando tal que $\overset{g\ \overline{a}_m\theta \to b\theta}{\underset{ct_g}{|}}$ es un *apa* para la prueba de (23) en (1). De (24), y por el lema A.2.16 se tiene:

$$\textbf{(25)} \qquad P^{\mathcal{T}} \vdash_{dValSC}\ g^{\mathcal{T}}\ \overline{a}_m^{\mathcal{T}}\mu \to (b^{\mathcal{T}}\mu, ct_g)$$

Si ahora definimos $\nu(T_g) = ct_g$, podemos escribir (25) como:

$$P^{\mathcal{T}} \vdash_{dValSC} \ (g^{\mathcal{T}}\ \overline{a}^{\mathcal{T}}_m \to (b^{\mathcal{T}}, T_g))\ (\mu \biguplus \nu))$$

que por (22) es $P^{\mathcal{T}} \vdash_{dValSC} \ c^{\mathcal{T}}(\mu \biguplus \nu)$ como queríamos probar. A la inversa, si partimos de (25) y $\nu(T_g)$ es un cierto patrón total, aplicando el lema A.2.16 llegamos a (24), y de allí por hipótesis de inducción a (23) con el APA indicado.

2.b) La condición que falta por comprobar es:

$$\begin{aligned} P^{\mathcal{T}} \vdash_{dValSC} \quad & (ctNode\ "f.k"\ \ [dVal\, s_1, \ldots, dVal\, s_n]\ (dVal\, s) \\ & (ctClean\ [(dVal\, r_1, T_1), \ldots, (dVal\, r_m, T_m)]) \to T)(\mu \biguplus \nu) \end{aligned}$$

En este caso definimos $\nu(T) = ct$ y la prueba anterior se puede escribir como

$$\begin{aligned} P^{\mathcal{T}} \vdash_{dValSC} \quad & ctNode\ "f.k"\ \ [dVal\, s_1\mu, \ldots, dVal\, s_n\mu]\ (dVal\, s\mu) \\ & (ctClean\ [(dVal\, r_1\mu, \nu(T_1)), \ldots, (dVal\, r_m\mu, \nu(T_m))]) \to ct \end{aligned}$$

Por el lema A.2.16 se verifica además que $s_i\mu = t'_i\theta$ con t'_i los parámetros en la regla (2) para $i = 1 \ldots n$, $s\mu = r'\theta$ con r' el lado derecho de esta misma regla y $r_i\mu = v_i\theta$ con v_i el lado derecho de las aproximaciones de la misma regla cuyo lado izquierdo es de la forma $f\ \overline{t}_n$ o $X\ u$ para $i = 1 \ldots m$. Por tanto lo que lo que debemos probar es:

$$\begin{aligned} P^{\mathcal{T}} \vdash_{dValSC} \quad & ctNode\ "f.k"\ \ [dVal\, t'_1\theta, \ldots, dVal\, t'_n\theta]\ (dVal\, r'\theta) \\ & (ctClean\ [(dVal\, r_1\theta, \nu(T_1)), \ldots, (dVal\, r_m\theta, \nu(T_m))]) \to ct \end{aligned}$$

Tenemos que probar que esta prueba existe sii ct es un APA para (1), salvo por la falta de la raíz. Para ello observamos en primer lugar que al ser ct total por hipótesis no puede ser $\perp$, por lo que la prueba anterior, si existe, debe concluir finalizar con un paso DC. Para que esta demostración puede llevarse a cabo ct debe ser de la forma

$$ct \equiv ctNode\ "f.k"\ \ [d_1, \ldots, d_n]\ r\ l$$

donde se debe cumplir

$$\begin{aligned} & P^{\mathcal{T}} \vdash_{dValSC} \quad dVal\ t'_i\theta \to d_i \text{ para } i = 1 \ldots n \\ & P^{\mathcal{T}} \vdash_{dValSC} \quad dVal\ r'\theta \to r \\ & P^{\mathcal{T}} \vdash_{dValSC} \quad ctClean\ [(dVal\, r_1\theta, \nu(T_1)), \ldots, (dVal\, r_m\theta, \nu(T_m))] \to l \end{aligned}$$

para completar la demostración. Las pruebas de llamadas a $dVal$ se pueden resolver mediante la regla DV, por lo que debe ser $d_i \equiv t'_i\theta$ para $i = 1 \ldots n$ y $r \equiv \lceil (r'\theta) \rceil$, por lo que

$$ct \equiv ctNode\ "f.k"\ [\lceil (t'_1\theta) \rceil, \ldots, \lceil (t'_n\theta) \rceil]\ \lceil (r'\theta) \rceil\ l$$

Nos fijamos ahora en el APA de la prueba (2). Tras la raíz (que no consideramos), el siguiente nodo, enmarcado en la figura en un rectángulo, sería $f\ \overline{t'}_n\theta \to u$. Y podemos observar que la raíz de ct representa precisamente esta información:

- El símbolo f viene representado por la cadena "f.k"que incluye además la información adicional de la posición que ocupa la regla utilizada (k).
- La lista $[\lceil (t'_1\theta) \rceil, \ldots, \lceil (t'_n\theta^{\mathcal{T}^{-1}}) \rceil]$ corresponde a la representación como valores de tipo $pVal$ de $t'_1\theta \ldots t'_n\theta$.
- Como se observa en (1) el lado derecho u corresponde con el valor que aproxima el valor derecho $r'\theta$, representado aquí por $\lceil (r'\theta) \rceil$.

En cuanto a los hijos de este nodo, se corresponderán con los hechos básicos consecuencias de reglas $AR + FA$ que pueda haber en las demostraciones de las componentes de $C'\theta$ en (1), ya que $r'\theta \to u$ se trata de una aproximación entre patrones y no puede incluir ninguna regla $AR + FA$.

Y las componentes de $C'\theta$ cuya demostración puede requerir el uso de $AR + FA$ son necesariamente condiciones atómicas de la forma $c'_i\theta = g\ \overline{a}_k \to v$ y provienen o bien de condiciones de la forma $c'_i \equiv g\ \overline{a}_k \to v$ o $c'_i \equiv X\ a \to v$. En cualquiera de los dos casos tenemos en (2) la transformación atómica correspondiente de su transformación afectada por la sustitución $\mu \biguplus \nu$ de la forma: $c_i^{\mathcal{T}}(\mu \biguplus \nu) \equiv g\ \overline{a}_k^{\mathcal{T}} \to (v^{\mathcal{T}}, \nu(T_i))$ y aplicando la hipótesis de inducción los valores $\nu(T_i)$ representan los APAs salvo por la raíz, que no es conclusión un paso $AR + FA$ y por tanto no debe ser incluida.

Obsérvese que hay dos casos en los que las condiciones instanciadas de la forma $g\ \overline{a}_k \to v$ no contribuyen sin embargo al árbol ct:

- Cuando g es una función auxiliar introducida durante la transformación.
- Cuando v es $\bot$.

La función $ctClean$ se encarga de eliminar estos valores de la lista de árboles hijos (ver su definición (ver la definición de esta función en la página 86). El primer caso se reconoce porque las funciones auxiliares devuelven el valor $ctVoid$ como resultado, y el segundo porque el valor v es $\bot$. Es para este último caso por el que la regla (TR_1) incluye no sólo los árboles $T_1, \ldots T_m$ sino también las aplicaciones de $dVal$ a los lados derechos de las aproximaciones correspondientes. Omitimos la prueba de que $ctClean$ efectivamente se comporta tal y como hemos descrito, al ser fácil de comprobar a partir de su definición.

3. $P^{\mathcal{T}} \vdash_{dValSC} (s, T)(\mu \biguplus \nu) \to (u^{\mathcal{T}}, ct)$ en (3) con $P \vdash_{SC} r'\theta \to t$ en (1).

Por la regla (TR_1) se tiene que $r' \Rightarrow^{\mathcal{T}} (s\,;\,)$, es decir $s \equiv (r')^{\mathcal{T}}$, con r' el lado derecho de la regla aplanada (2). Entonces s no puede contener variables nuevas y se tiene que lo hay que probar es $P^{\mathcal{T}} \vdash_{dValSC} (((r')^{\mathcal{T}})\mu, \nu(T)) \to (u^{\mathcal{T}}, ct)$, o lo que es lo mismo al ser $\nu(T) = ct$, $P^{\mathcal{T}} \vdash_{dValSC} (((r')^{\mathcal{T}})\mu, ct) \to (u^{\mathcal{T}}, ct)$. Esta última demostración, si existe, finalizará con un paso DC cuyas premisas son $((r')^{\mathcal{T}})\mu \to u^{\mathcal{T}}$ y $ct \to ct$. La prueba de $ct \to ct$ se tiene por el apartado 2 de la proposición 3.2.1 (pág. 57), por lo que sólo falta por probar $P^{\mathcal{T}} \vdash_{dValSC} ((r')^{\mathcal{T}})\mu \to u^{\mathcal{T}}$. Por el corolario A.2.17 esto si tiene sii $P^{\mathcal{T}} \vdash_{dValSC} r'\theta \to u$ lo que se tiene como premisa de (1).

4. $P^{\mathcal{T}} \vdash_{dValSC} (u^{\mathcal{T}}, ct) \to (t^{\mathcal{T}}, ct)$ en (3) con con $P \vdash_{SC} u \to t$ en (1).

   Esta prueba, si existe, finalizará con un paso DC cuyas premisas serán $u^{\mathcal{T}} \to t^{\mathcal{T}}$ y $ct \to ct$. La prueba de $ct \to ct$ se tiene de nuevo por el apartado 2 de la proposición 3.2.1, mientras que la de $u^{\mathcal{T}} \to t^{\mathcal{T}}$ se tiene sii se tiene la de $u \to t$, por el lema A.2.15, y esta última aproximación aparece como premisa de (1).

Con respecto a la definición de ν y de la observación de las reglas (TR_1)-(TR_{11}) se tiene que cada $T_1, \ldots, T_m$ aparece en el lado derecho de una aproximación y sólo una aproximación en la condición de la regla transformada, por lo que en los casos anteriores no queda ningún valor $\nu(T_i)$ sin definir, y tampoco se dan dos definiciones diferentes para $\nu(T_i)$ para ningún $i = 1 \ldots m$.

□

La demostración del teorema resulta ahora sencilla:

(i) Supongamos que se verifica $P \vdash_{SC} f\ \bar{t}_n \to t$ y que *apa* es un APA para esta demostración de la forma $\overset{f\ \bar{t}_n \to t}{\underset{\mathrm{T}}{|}}$ para cierto árbol T.

Como $f\ \bar{t}_n$ es una expresión plana, en el sentido de la definición 4.3.1 (pág. 75), aplicando la regla (PL_{10}) se tiene que

$$(f\ \bar{t}_n \to t)\ \Rightarrow_{\mathcal{A}}\ (f\ \bar{t}_n \to t)$$

Por el apartado (a) del teorema 4.3.4 (pág. 80), y tomando $\theta = id$, se tiene que $P_{\mathcal{A}} \vdash_{SC} (f\ \bar{t}_n)\nu \to t$ para alguna sustitución ν cuyo dominio son las variables nuevas introducidas durante el aplanamiento de $f\ \bar{t}_n$. Al ser este conjunto de variables nuevas vacío en nuestro caso, se tiene $\nu \equiv id$ y

$$P_{\mathcal{A}} \vdash_{SC} f\ \bar{t}_n \to t$$

con un árbol de prueba abreviado idéntico a *apa*. Entonces el apartado (i) del lema A.2.18 asegura que se cumple $P^{\mathcal{T}} \vdash_{dValSC} f^{\mathcal{T}}\ \overline{t_n^{\mathcal{T}}} \to (t^{\mathcal{T}}, ct)$, donde $ct\ ::\ cTree$ es un patrón total representando T, tal y como queríamos demostrar.

(ii) Supongamos que $P^{\mathcal{T}} \vdash_{dValSC} f^{\mathcal{T}}\ \overline{t_n^{\mathcal{T}}} \to (t^{\mathcal{T}}, ct)$, con ct un patrón total. Por el apartado (ii) del lema A.2.18 se tiene que entonces $P_{\mathcal{A}} \vdash_{SC} f\ \bar{t}_n \to t$ y que ct representa un árbol T tal que $\overset{f\ \bar{t}_n \to t}{\underset{\mathrm{T}}{|}}$ es un APA para $P_{\mathcal{A}} \vdash_{SC} f\ \bar{t}_n \to t$.

Ahora bien, se cumple

$$(f\ \bar{t}_n \to t)\ \Rightarrow_{\mathcal{A}}\ (f\ \bar{t}_n \to t)$$

definiendo entonces $\nu \equiv \theta \equiv id$ se tiene que $P_{\mathcal{A}} \vdash_{SC} (f\ \bar{t}_n \to t)(\theta \biguplus \nu)$, con $dom(\nu) = \emptyset$ al no haber ninguna variable nueva introducida durante el aplanamiento, lo que lleva por el apartado b) del teorema 4.3.4 a que se tenga $P \vdash_{SC} f\ \bar{t}_n \to t$ con $\overset{f\ \bar{t}_n \to t}{\underset{\mathrm{T}}{|}}$ como *apa* tal y como indica el teorema.

■

A.3. Resultados Presentados en el Capítulo 6

A.3.1. Demostración del Teorema 6.4.1

Para demostrar este teorema, enunciado en la pág. 158, necesitamos establecer en primer lugar dos lemas auxiliares.

Lema A.3.1. *Sea $S_i \square W_i$ una configuración obtenida tras i pasos del algoritmo descrito en la sección 6.4.1, y sea $(s \to t) \in S_i$. Entonces se verifica la siguiente fórmula lógica:*

$$var(s) \cap W_i = \emptyset \quad \vee \quad var(t) \cap W_i = \emptyset$$

Demostración.
Razonamos por inducción sobre el valor i.

Caso Base. El resultado se verifica para la configuración inicial debido a la construcción de S_0 y W_0, ya que estamos suponiendo que los dos hechos básicos iniciales no tienen variables en común.

Caso inductivo.
Consideremos el paso i-ésimo del algoritmo ($i \geq 1$): $S_{i-1} \square W_{i-1} \vdash_{\theta_i} S_i \square W_i$. Sea $s \to t$ una aproximación cualquiera de S_i. Podemos distinguir dos casos:

1. $s \to t = (a \to b)\theta_i$ para algún $a \to b$ en S_{i-1}. Por la hipótesis de inducción se tiene que o bien a o b (o ambas) no tienen variables en común con W_{i-1}. Supongamos que $var(a) \cap W_{i-1} = \emptyset$ (análogo si $var(b) \cap W_{i-1} = \emptyset$). Entonces, como $dom(\theta_i) \subseteq W_{i-1}$, $s = a\theta_i = a$ y como W_i coincide con W_{i-1} excepto quizás por la inclusión de algunas variables nuevas, $var(s) \cap W_i = \emptyset$.

2. $s \to t$ es una nueva aproximación introducida bien por la regla $R3$ o por la regla $R6$. En el caso de $R3$, aplicando la hipótesis de inducción a $h\ \overline{a}_m \to h\ \overline{b}_m$ se tiene que cada $a_i \to b_i$ debe cumplir el resultado. En el caso de que la regla que ha introducido la aproximación se trate de $R6$ se tiene por hipótesis de inducción que $var(h\ \overline{a}_m) \cap W_{i-1} = \emptyset$ y por tanto también se cumple $var(a_i) \cap W_i = \emptyset$ para todo $1 \geq i \geq m$.

□

Lema A.3.2. *Sea* $S_i \Box W_i \vdash_{\theta_{i+1}} S_{i+1} \Box W_{i+1}$ *un paso del algoritmo que se describe en la sección 6.4.1. Entonces* $Sol(S_i \Box W_i) = (\theta_{i+1} Sol(S_{i+1} \Box W_{i+1}))\restriction_{W_i}$.

Demostración.

Vamos a probar este resultado examinando la regla aplicada en el paso del algoritmo considerado.

En el caso de las reglas $R1$, $R2$ y $R3$, se tiene que $\theta_{i+1} = id$ y $W_{i+1} = W_i$ y el resultado se sigue a partir de la definición del orden de aproximación $\sqsubseteq$, (ver definición 3.1.1, pág. 48). Consideramos a continuación por separado las reglas $R4$, $R5$ y $R6$.

R4 $Sol(s \to X, S \Box W) = \{X \mapsto s\} Sol(S\{X \mapsto s\} \Box W)$

a) $Sol(s \to X, S \Box W) \subseteq \{X \mapsto s\} Sol(S\{X \mapsto s\} \Box W)$
Sea $\theta \in Sol(s \to X, S \Box W)$. Entonces:
- $X\theta = s\theta$, ya que $X\theta \sqsubseteq s\theta$ y θ es una substitución total.
- $S\theta$ se satisface.
- $\theta = \{X \mapsto s\}\theta$, porque para cada $Y \in Var$:
  - Si $Y \neq X$ entonces $Y\{X \mapsto s\}\theta = Y\theta$.
  - Si $Y = X$ entonces $Y\{X \mapsto s\}\theta = s\theta = X\theta = Y\theta$

Por tanto $S\{X \mapsto s\}\theta = S\theta$ se satisface, y de aquí

$$\theta \in Sol(S\{X \mapsto s\} \Box W)$$

Finalmente, si consideramos de nuevo que $\theta = \{X \mapsto s\}\theta$, llegamos al resultado esperado $\theta \in \{X \mapsto s\} Sol(S\{X \mapsto s\} \Box W)$.

b) $\{X \mapsto s\} Sol(S\{X \mapsto s\} \Box W) \subseteq Sol(s \to X, S \Box W)$
Todo elemento de $\{X \mapsto s\} Sol(S\{X \mapsto s\} \Box W)$ debe ser de la forma $\{X \mapsto s\}\theta$, con $\theta \in Sol(S\{X \mapsto s\} \Box W)$. Entonces:
- $S\{X \mapsto s\}\theta$ se satisface.
- $s\{X \mapsto s\}\theta = X\{X \mapsto s\}\theta$. Para probar esto, nótese que $X \notin var(s)$, porque $X \in W$ implica $var(s) \cap W = \emptyset$, por el lema A.3.1. Entonces $s\{X \mapsto s\}\theta = s\theta = X\{X \mapsto s\}\theta$.

Y de aquí $\{X \mapsto s\}\theta \in Sol(s \to X, S \Box W)$.

R5 $Sol(X \to Y, S\Box W) = \{X \mapsto Y\}Sol(S\{X \mapsto Y\}\Box W)$
Razonamiento análogo al caso anterior.

R6 $Sol(X \to h\,\overline{a}_m, S\Box W) =$
$(\{X \mapsto h\,\overline{X}_m\}Sol(\ldots, X_k \to a_k, \ldots, S\{X \mapsto h\,\overline{X}_m\}\Box W, \overline{X}_m)\restriction_W$

a) $Sol(X \to h\,\overline{a}_m, S\Box W) \subseteq$
$(\{X \mapsto h\,\overline{X}_m\}Sol(\ldots, X_k \to a_k, \ldots, S\{X \mapsto h\,\overline{X}_m\}\Box W, \overline{X}_m)\restriction_W$
Sea $\theta \in Sol(X \to h\,\overline{a}_m, S\Box W)$. Entonces:
- $X\theta = h\ \overline{t}_m$ con $a_k\theta \sqsubseteq t_k$.
- $S\theta$ se satisface.
Consideremos la sustitución total

$$\rho =_{def} \theta \biguplus \{X_1 \mapsto t_1, \ldots, X_m \mapsto t_m\}$$

Entonces :

- $X_k\rho = t_k$, $a_k\rho = a_k\theta$ y por tanto $a_k\rho \ \sqsubseteq X_k\rho$.
- $S\{X \mapsto h\,\overline{X}_m\}\rho = S\theta$ se satisface.

Por tanto $\rho \in Sol(\ldots, X_k \to a_k, \ldots, S\{X \mapsto h\,\overline{X}_m\}\Box W, \overline{X}_m)$.
Además $\{X \mapsto h\,\overline{X}_m\}\rho\restriction_W = \theta$. De aquí que

$$\theta \in (\{X \mapsto h\,\overline{X}_m\}Sol(\ldots, X_k \to a_k, \ldots, S\{X \mapsto h\,\overline{X}_m\}\Box W, \overline{X}_m)\restriction_W$$

b) $(\{X \mapsto h\,\overline{X}_m\}Sol(\ldots, X_k \to a_k, \ldots, S\{X \mapsto h\,\overline{X}_m\}\Box W, \overline{X}_m)\restriction_W \subseteq$
$Sol(X \to h\,\overline{a}_m, S\Box W)$

Todo elemento de

$$(\{X \mapsto h\,\overline{X}_m\}Sol(\ldots, X_k \to a_k, \ldots, S\{X \mapsto h\,\overline{X}_m\}\Box W, \overline{X}_m)\restriction_W$$

debe ser de la forma $\theta = (\{X \mapsto h\,\overline{X}_m\}\rho)\restriction_W$ con
$\rho \in Sol(\ldots, X_k \to a_k, \ldots, S\{X \mapsto h\,\overline{X}_m\}\Box W, \overline{X}_m)$ tal que:

(1) Para cada k, $a_k\rho \sqsubseteq X_k\rho$ se satisface.
(2) $S\{X \mapsto h\,\overline{X}_m\}\rho =$ se satisface.

Por el lema A.3.1, y al ser $X \in W$, se tiene $var(a_k) \cap W = \emptyset$.
También se cumple $var(a_k) \cap (W, \overline{X}_m) = \emptyset$, al ser $\overline{X}_m$ variables nuevas.
Por tanto $a_k\rho \sqsubseteq X_k\rho$ sii $a_k \sqsubseteq X_k\rho$. Si llamamos b_k a cada $X_k\rho$ se cumple

(3) $a_k\rho \sqsubseteq X_k\rho$ sii $a_k \sqsubseteq b_k$.

Esto significa que θ debe ser de la forma $\theta = \{X \mapsto h\,\overline{b}_m\} \uplus \rho\restriction_{(W-\{X\})}$.
Entonces:

- $(X \to h\,\overline{a}_m)\theta = h\,\overline{b}_m \to h\,\overline{a}_m$ lo que se cumple sii para cada k $a_k \sqsubseteq b_k$, y esto es cierto por (3) y (1).
- $S\theta = S(\{X \mapsto h\,\overline{b}_m\} \uplus \rho\restriction_{(W-\{X\})}) = S\{X \mapsto h\,\overline{X}_m\}\rho$, lo que se tiene por (2).

□

Comenzamos la demostración del teorema definiendo un orden lexicográfico bien fundamentado entre configuraciones:

Decimos que $K_i < K_j$ (con $K_i = S_i \Box W_i$, $K_j = S_j \Box W_j$) si se verifican:

a) $\| K_i \|_1 < \| K_j \|_1$, o

b) $\| K_i \|_1 = \| K_j \|_1$ y $\| K_i \|_2 < \| K_j \|_2$, o

c) $\| K_i \|_1 = \| K_j \|_1$ y $\| K_i \|_2 = \| K_j \|_2$, y $\| K_i \|_3 < \| K_j \|_3$.

donde:

- $\| S \Box W \|_1$ = número de apariciones de patrones rígidos $h\,\overline{a}_m$, $m \geq 0$ en alguna $(s \to t) \in S$ tales que:
  a) Si $h\,\overline{a}_m$ es parte de s entonces $var(t) \cap W \neq \emptyset$.
  b) Si $h\,\overline{a}_m$ es parte de t entonces $var(s) \cap W \neq \emptyset$.
- $\| S \Box W \|_2$ = número de apariciones de símbolos $h \in DC \cup FS$ en S.
- $\| S \Box W \|_3$ = tamaño de S (como multiconjunto, teniendo en cuenta las repeticiones).

Ahora hay que verificar que en cada paso i del algoritmo se verifica $K_{i+1} < K_i$. Esto puede hacerse examinando examinando la regla de transformación aplicada en cada paso junto la aproximación seleccionada $(s \to t) \in S_i$:

$R1$ Entonces $\| K_{i+1} \|_1 = \| K_i \|_1$, $\| K_{i+1} \|_2 = \| K_i \|_2$, y $\| K_{i+1} \|_3 < \| K_i \|_3$.

$R2$ Entonces $\| K_{i+1} \|_1 = \| K_i \|_1$, y o bien
- $\| K_{i+1} \|_2 = \| K_i \|_2$, $\| K_{i+1} \|_3 < \| K_i \|_3$ (si $t \in Var$)
o
- $\| K_{i+1} \|_2 < \| K_i \|_2$ (si t no es una variable).

$R3$ O bien $\| K_{i+1} \|_1 < \| K_i \|_1$, o $\| K_{i+1} \|_1 = \| K_i \|_1$ y $\| K_{i+1} \|_2 < \| K_i \|_2$ (ya que el símbolo h ha sido eliminado de S_{i+1}).

R4 El paso del algoritmo aplicado es, en este caso, de la forma:

$$\underbrace{s \to X,\ S\Box W}_{K_i} \vdash_{\{X \mapsto s\}} \underbrace{S\{X \mapsto s\}\Box W}_{K_{i+1}},\ X \neq s, X \in W$$

Si s es una variable entonces $\| K_{i+1} \|_1 = \| K_i \|_1$, $\| K_{i+1} \|_2 = \| K_i \|_2$, y $\| K_{i+1} \|_3 < \| K_i \|_3$. Si s no es una variable debe ser un patrón rígido. Entonces, ya que $X \in W$, el lema A.3.1 asegura que

$$\| S\{X \mapsto s\}\Box W \|_1 = \| S\Box W \|_1 < \| s \to X,\ S\Box W \|_1$$

es decir $\| K_{i+1} \|_1 < \| K_i \|_1$.

R5 Entonces $\| K_{i+1} \|_1 = \| K_i \|_1$, $\| K_{i+1} \|_2 = \| K_i \|_2$, y $\| K_{i+1} \|_3 < \| K_i \|_3$.

R6 Análogamente a *R4* cuando s no es una variable: $\| K_{i+1} \|_1 < \| K_i \|_1$.

Ahora vamos a probar que si $S_j \neq \emptyset$ entonces $Sol(S_0\Box W_0) = \emptyset$ y por tanto no existe ninguna sustitución θ capaz de resolver el sistema y la relación de consecuencia no se cumple. Por el lema A.3.2 basta con comprobar que $Sol(S_j\Box W_j) = \emptyset$. Ya que no puede aplicarse ninguna regla de transformación a esta configuración, debe darse al menos uno de los casos que enunciamos a continuación. Obsérvese que el sistema no puede resolverse en ninguno de los casos.

a) $\bot \to s \in S_j$, $s \neq \bot$. Entonces no existe ninguna sustitución total θ tal que $s\theta \sqsubseteq \bot$.

b) $h\,\overline{a}_m \to g\,\overline{b}_l$ con o bien $h \neq g$ o $m \neq l$. Obvio.

c) $h\,\overline{a}_m \to X$, $h\,\overline{a}_m$ no total, $X \in W_j$. Entonces no existe ninguna sustitución total θ tal que $X\theta \subseteq (h\ \overline{a}_m)\theta$.

d) $X \to Y$, $X \neq Y$, $X \notin W_j$, $Y \notin W_j$. Directamente de la condición de que $dom(\theta) \subseteq W_j$ en toda solución.

e) $X \to h\,\overline{a}_m$, $X \notin W_j$. Como el caso anterior.

Finalmente, si $S_j = \emptyset$ entonces $Sol(\emptyset\Box W_j) = Subst_{W_j}$, donde se tiene que $Subst_{W_j} = \{\theta \in Subst \mid dom(\theta) \subseteq W_j\}$. Consideramos la sustitución $\theta = \theta_1\theta_2 \ldots \theta_j$. Por el lema A.3.2, $Sol(S_0\Box W_0) = (\theta Subst_{W_j}) \restriction_{W_0}$. Ya que $id \in Subst_{W_j}$, $\theta id \restriction_{W_0} = \theta \restriction_{W_0} \in Sol(S_0\Box W_0)$, y por tanto $\theta \restriction_{W_0}$ puede utilizarse para probar que se cumple la relación de consecuencia entre ambos hechos básicos. ■

Apéndice B

Recopilación de Ejemplos

En este apéndice mostramos algunos ejemplos de programas erróneos junto con sus sesiones de depuración. Hemos dividido este muestrario de ejemplos en 4 partes, Las tres primeras corresponden a programas $\mathcal{TOY}$ representando respectivamente programas de programación lógica, de programación funcional, y de programación lógico-funcional. El cuarto apartado muestra algunas sesiones de depuración realizadas con el depurador de Curry. Algunos de los ejemplos están tomados de la literatura sobre depuración declarativa, adaptados a la sintaxis de $\mathcal{TOY}$ o Curry, mientras que otros corresponden a errores producidos al programar en $\mathcal{TOY}$ o han sido creados a propósito para mostrar algún aspecto particular del depurador. En todos los ejemplos hemos utilizado el navegador textual descendente en lugar del navegador $\mathcal{DDT}$ presentado en el capítulo 6. La razón es simplemente que resulta más sencillo reproducir las sesiones de depuración en el formato del depurador en modo texto que en el caso del depurador gráfico, pero todos los programas pueden ser depurados igualmente (en la mayoría de los casos requiriendo un número menor de preguntas al usuario si se utiliza la estrategia *divide y pregunta*) mediante $\mathcal{DDT}$.

B.1. Programación Lógica

Ejemplo B.1.1. De [32], página 178.

En este trabajo se propone el siguiente programa para ilustrar la existencia de respuestas incorrectas, con el síntoma inicial $rev([U,V],[U])$:

```
rev([],[])   :- true
rev([X|Y],Z) :- rev(Y,T), app(T,[X],Z)

app([],X,X) :- true
app([X|Y],Z,T) :- app(Y,Z,T)
```

La sesión de depuración en $\mathcal{TOY}$:

```
TOY> rev([U,V],[U])
```

```
      yes

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

Compiling ...........

Consider the following facts:
1: rev ( [U, V ],  [U ])  -> true

Are all of them valid? ([y]es / [n]ot) / [a]bort) n

Consider the following facts:
1: rev ( [V ],  [V ])  -> true
2: app ( [V ],  [U ],  [U ])  -> true

Are all of them valid? ([y]es / [n]ot) / [a]bort) n
Enter the number of a non-valid fact followed by a fullstop: 2.

Consider the following facts:
1: app ( [],  [U ],  [U ])  -> true

Are all of them valid? ([y]es / [n]ot) / [a]bort) y

Rule number 2 of the function app is wrong.
Buggy node: app ( [V],  [U],  [U])  ->  true
```

Ejemplo B.1.2. De [88], página 171.

En el artículo se presenta este ejemplo (una versión incorrecta del algoritmo de ordenación QuickSort) para mostrar el uso del sistema *TkCalypso* como ejemplo de programa con restricciones:

```
qs([],[]) :- true
qs([Pivot | List],SortList) :-
        partition(Pivot,List,MinList, MaxList),
        qs(MinList,SortMinList),
        qs(MaxList,SortMaxList),
        append([Pivot|SortMinList], SortMaxList, SortList)

partition(_,[],[],[]) :- true
partition(Pivot, [X|List], [X|MinList], MaxList) :-
        X < Pivot,
        partition(Pivot,List,MinList,MaxList)
```

```
partition(Pivot, [X|List], MinList, [X|MaxList]) :-
        X < Pivot,
        partition(Pivot,List,MinList,MaxList)

append([],L,L) :- true
append([X|Xs],L,[X|Ys]) :- append(Xs,L,Ys)
```

La semántica pretendida es:

$qs(X, Y)$ es cierto si Y es una permutación de X y está ordenado de menor a mayor.

$partition(A, B, C, D)$ es cierto si A es un valor, B una lista, C una lista con todos los valores de B menores que A y D una lista con todos los valores de D mayores que A.

El objetivo propuesto es $qs([12, X, Y, 6, Z], L)$ y las respuestas obtenidas representan restricciones de la forma: $X = \_\#2(8.,11)$, indicando por ejemplo que X debe tomar valores entre 8 y 11 ($\_\#2$ es el nombre interno de la variable).

Como nuestro sistema no depura programas con restricciones aritméticas, debemos probar con objetivos sin variables. El programa presenta tanto respuestas perdidas (por ejemplo el objetivo $qs([2, 4, 3], L)$ falla), como respuestas incorrectas. Este es el resultado de la depuración de una respuesta incorrecta:

```
TOY> qs([5,3,1],L)

      yes
      L ==  [ 5, 3, 1 ]

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

Compiling .....

Consider the following facts:
1: qs ( [5.0, 3.0, 1.0 ],  [5.0, 3.0, 1.0 ])  -> true

Are all of them valid? ([y]es / [n]ot) / [a]bort) n

Consider the following facts:
1: partition (5.0,  [3.0, 1.0 ],  [3.0, 1.0 ],  [])  -> true
2: qs ( [3.0, 1.0 ],  [3.0, 1.0 ])  -> true
3: qs ( [],  [])  -> true
4: append ( [5.0, 3.0, 1.0 ],  [],  [5.0, 3.0, 1.0 ])  -> true

Are all of them valid? ([y]es / [n]ot) / [a]bort) n
Enter the number of a non-valid fact followed by a fullstop: 2
```

```
Consider the following facts:
1: partition (3.0,  [1.0 ],  [1.0 ],  [])  -> true
2: qs ( [1.0 ],  [1.0 ])  -> true
3: qs ( [],  [])  -> true
4: append ( [3.0, 1.0 ],  [],  [3.0, 1.0 ])  -> true

Are all of them valid? ([y]es / [n]ot) / [a]bort) y

Rule number 2 of the function qs is wrong.
Buggy node: qs ( [3.0, 1.0],  [3.0, 1.0])  ->  true
```

Tras examinar la segunda regla de `qs` encontramos que el problema se encuentra en la concatenación de las dos listas, porque el pivote (en este caso 3) debe ir en medio y no al comienzo. La regla correcta sería:

```
qs([Pivot | List],SortList) :-
        partition(Pivot,List,MinList, MaxList),
        qs(MinList,SortMinList),
        qs(MaxList,SortMaxList),
        append(SortMinList, [Pivot|SortMaxList], SortList)
```

Ejemplo B.1.3. De [24], página 150.

Aquí se propone un método de ordenación de la burbuja que funciona correctamente cuando el primer argumento es una lista a ordenar, el segundo una variable y sólo examinamos la primera respuesta. Sin embargo con otros modos de uso el programa resulta erróneo; por ejemplo el objetivo

```
ordenarbur([3,2,1], [2,1,3])
```

tiene éxito, cuando debería fallar. Este es el programa:

```
ordenarbur(L,S) :-
    append(X,[A,B|Y],L),
    B < A,
    append(X,[B,A|Y],M),
    ordenarbur(M,S)
ordenarbur(L,L) :- true

append([],L,L) :- true
append([H|T], L, [H|V]) :- append(T,L,V)
```

Sesión de depuración en $\mathcal{TOY}$:

```
TOY> ordenarbur([3,2,1], [2,1,3])

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

Compiling.....

Consider the following facts:
1: ordenarbur ( [3.0, 2.0, 1.0 ],  [2.0, 1.0, 3.0 ])  -> true

Are all of them valid? ([y]es / [n]ot) / [a]bort) n

Consider the following facts:
1: append ( [],  [3.0, 2.0, 1.0 ],  [3.0, 2.0, 1.0 ])  -> true
2: append ( [],  [2.0, 3.0, 1.0 ],  [2.0, 3.0, 1.0 ])  -> true
3: ordenarbur ( [2.0, 3.0, 1.0 ],  [2.0, 1.0, 3.0 ])  -> true

Are all of them valid? ([y]es / [n]ot) / [a]bort) n
Enter the number of a non-valid fact followed by a fullstop: 3.

Consider the following facts:
1: append ( [2.0 ],  [3.0, 1.0 ],  [2.0, 3.0, 1.0 ])  -> true
2: append ( [2.0 ],  [1.0, 3.0 ],  [2.0, 1.0, 3.0 ])  -> true
3: ordenarbur ( [2.0, 1.0, 3.0 ],  [2.0, 1.0, 3.0 ])  -> true

Are all of them valid? ([y]es / [n]ot) / [a]bort) n
Enter the number of a non-valid fact followed by a fullstop: 3.

Rule number 2 of the function ordenarbur is wrong.
Buggy node: ordenarbur ( [2.0, 1.0, 3.0],  [2.0, 1.0, 3.0])  ->  true
```

Ejemplo B.1.4. De [43], página 17.

El siguiente es (otro) programa erróneo para el método de ordenación *quicksort* con objeto de ilustrar las características del sistema *CIAO*:

```
qsort([X|L],R) :-
        partition(L,X,L1,L2),
        qsort(L2,R2), qsort(L1,R1),
        append(R2,[X|R1],R)

qsort([],[]) :- true
```

```
partition([],_B,[],[]) :- true
partition([E|R],C,[E|Left1],Right) :-
        E<C, partition(R,C,Left1,Right)

partition([E|R],C,Left,[E|Right1]) :-
        E>=C, partition(R,C,Left,Right1)

append([],X,X) :- true
append([H|X], Y, [H|Z]) :- append(X,Y,Z)
```

La semántica pretendida es la misma que el el caso de B.1.2. Realmente el programa original propuesto incluye un error adicional en la declaración de *qsort*, donde el orden de X y $L1$ está cambiado en la llamada a *partition*:

```
qsort([X|L],R) :-
        partition(L,L1,X,L2),
        qsort(L2,R2), qsort(L1,R1),
        append(R2,[X|R1],R)

qsort([],[]) :- true
```

Esta incorrección es detectada en el sistema $CIAO$ en tiempo de compilación gracias a que el programa original también incluye una aserción del usuario de la forma:

```
:- calls partition(A,B,C,D) : (ground(A), (ground(B))
```

Según indican los autores una vez el sistema detecta que esta aserción no se verifica en la llamada a *partition* de *qsort* se puede detectar la fuente del error mediante el uso de diagnosis abstracta (método que comentamos brevemente en la Subsección 2.1.9).

En $\mathcal{TOY}$ este error es localizado por el inferidor de tipos. Tras la corrección comprobamos que el programa compila pero el objetivo $qsort([1,2],L)$ tiene éxito para $L = [2,1]$. En el ejemplo de [43] esta respuesta incorrecta es detectada automáticamente por el sistema gracias a otra aserción del usuario y a continuación se puede iniciar un proceso de depuración declarativa (que no se incluye en el citado artículo). En nuestro caso es el usuario el que debe detectar que esta respuesta es incorrecta y comenzar una sesión de depuración como la siguiente:

```
TOY> qsort([1,2],L)

      yes
      L ==  [ 2, 1 ]

Compiling .......
```

```
Consider the following facts:
1: qsort ( [1.0, 2.0 ],  [2.0, 1.0 ])  -> true

Are all of them valid? ([y]es / [n]ot) / [a]bort) n

Consider the following facts:
1: partition ( [2.0 ], 1.0,  [],  [2.0 ])  -> true
2: qsort ( [2.0 ],  [2.0 ])  -> true
3: qsort ( [],  [])  -> true
4: append ( [2.0 ],  [1.0 ],  [2.0, 1.0 ])  -> true

Are all of them valid? ([y]es / [n]ot) / [a]bort) y

Rule number 1 of the function qsort is wrong.
Buggy node: qsort ( [1.0, 2.0],  [2.0, 1.0])  ->  true
```

Como en el ejemplo B.1.2 se observa que el error está en la primera cláusula, al hacer la concatenación de los resultados intermedios, pero en este caso es debido a que están invertidos los dos primeros argumentos en la llamada a *append*. La versión correcta sería:

```
qsort([X|L],R) :-
        partition(L,X,L1,L2),
        qsort(L2,R2), qsort(L1,R1),
        append([X|R1],R2, R)

qsort([],[]) :- true
```

Ejemplo B.1.5. De [63], página 3.

En este trabajo se propone el siguiente programa lógico:

```
isort([],[]) :- true
isort([N|Ns],Ss) :-
        isort(Ns,Ss1),
        insert(N,Ss1,Ss)

insert(N,[],[N]) :- true
insert(N,[S|Ss], [N|Ss]) :-
        N <= S
insert(N,[S|Ss0],[S|Ss]) :-
        N > S,
        insert(N,Ss0,Ss)
```

con el síntoma inicial positivo detectado al obtener la respuesta $X = [1, 3]$ para el objetivo $isort([3, 1, 2], X)$. En el artículo se muestra la sesión de depuración en NU-Prolog, en la que se hacen las siguientes preguntas al usuario:

```
?- isort([3,1,2],X).
X = [1,3] w
isort([3,1,2],[1,3]) Valid? n

isort([1,2],[1]) Valid? n

isort([2],[2]) Valid? y

insert(1,[2],[1]) Valid? n

Incorrect Clause Instance:
insert(1,[2],[1]) :-
  1<=2.

Matching Clauses:
insert(N,[S|Ss],[N|Ss]) :-
  N=<S.
```

La cláusula correcta sería:

```
insert(N,[S|Ss], [N|Ss]) :-
        N <= S
```

También se muestra tras cada pregunta (aunque no lo hayamos incluido aquí) la instancia de cláusula utilizada en cada caso. En este caso el sistema encuentra el error tras realizar 4 preguntas al usuario.

En el caso del depurador de $\mathcal{TOY}$:

```
TOY> isort([3,1,2],X)

      yes
      X ==  [ 1, 3 ]

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

Compiling ....

Consider the following facts:
1: isort ( [3.0, 1.0, 2.0 ],  [1.0, 3.0 ])  -> true

Are all of them valid? ([y]es / [n]ot) / [a]bort) n
```

```
Consider the following facts:
1: isort ( [1.0, 2.0 ],  [1.0 ])  -> true
2: insert (3.0,  [1.0 ],  [1.0, 3.0 ])  -> true

Are all of them valid? ([y]es / [n]ot) / [a]bort) n
Enter the number of a non-valid fact followed by a fullstop: 1.

Consider the following facts:
1: isort ( [2.0 ],  [2.0 ])  -> true
2: insert (1.0,  [2.0 ],  [1.0 ])  -> true

Are all of them valid? ([y]es / [n]ot) / [a]bort) n
Enter the number of a non-valid fact followed by a fullstop: 2.

Rule number 2 of the function insert is wrong.
Buggy node: insert (1.0,  [2.0],  [1.0])  ->  true
```

El sistema encuentra la misma regla, tras realizar 5 preguntas al usuario (3 de tipo sí o no y 2 señalando un hecho no válido). Las diferencias entre ambas sesiones se deben simplemente a la navegación; el árbol de prueba obtenido en $\mathcal{TOY}$ es el mismo que el árbol de prueba que aparece para el ejemplo en [63].

B.2. Programación Funcional

Ejemplo B.2.1. De [78, 68] (páginas 8, 38 respectivamente).

```
insert X []     = [X]
insert X (Y:Ys) = (Y:insert X Ys) <== Y > X
insert X (Y:Ys) = (X:Y:Ys) <== X < Y
insert X (Y:Ys) = (Y:Ys)  <== X==Y

sort []     = []
sort (X:Xs) = insert X (sort Xs)
```

El error en este caso está en la condición de la segunda regla que debería ser $X \leq Y$ en lugar de $Y > X$. Y la sesión de depuración:

```
sort [2,1,3] == R

      yes
```

```
      R ==  [ 2, 3, 1 ]

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

compiling ...

Consider the following facts:
1: sort  [2.0, 1.0, 3.0 ]  ->  [2.0, 3.0, 1.0 ]

Are all of them valid? ([y]es / [n]ot) / [a]bort) n

Consider the following facts:
1: sort  [1.0, 3.0 ]  ->  [3.0, 1.0 ]
2: insert 2.0  [3.0, 1.0 ]  ->  [2.0, 3.0, 1.0 ]

Are all of them valid? ([y]es / [n]ot) / [a]bort) n
Enter the number of a non-valid fact followed by a fullstop: 1.

Consider the following facts:
1: sort  [3.0 ]  ->  [3.0 ]
2: insert 1.0  [3.0 ]  ->  [3.0, 1.0 ]

Are all of them valid? ([y]es / [n]ot) / [a]bort) n
Enter the number of a non-valid fact followed by a fullstop: 2.

Consider the following facts:
1: insert 1.0  []  ->  [1.0 ]

Are all of them valid? ([y]es / [n]ot) / [a]bort) y

Rule number 2 of the function insert is wrong.
Buggy node: insert 1.0  [3.0]  ->   [3.0, 1.0]
```

Esta sesión es similar a la sesión en *Freja* para el programa análogo escrito en Haskell que presentamos en el ejemplo 2.3.2, pág. 34, y su correspondiente sesión de depuración en *Freja*, aunque en el caso de $\mathcal{TOY}$ el depurador no muestra un hijo cada vez sino todos los hijos de un nodo erróneo al mismo tiempo. En ocasiones, como en este ejemplo, este método seguido en $\mathcal{TOY}$ supone para el usuario tener que examinar más nodos que utilizando el método de navegación de *Freja* (mostrar los hijos uno a uno descendiendo por el primer hijo erróneo).

Sin embargo pensamos que nuestro método de navegación simplifica la tarea al usuario programas más complejos (por ejemplo cuando las funciones tienen numerosas definiciones locales) en los que algunos de los nodos tienen un considerable número de nodos hijos. En

este caso siguiendo el método de *Freja* el usuario tiene que contestar afirmativamente a preguntas sobre la validez de los hijos hasta encontrar el nodo erróneo, mientras que en el caso de $\mathcal{TOY}$ puede seleccionar de la lista presentada un nodo erróneo sin tener que contestar a preguntas sobre el resto.

Ejemplo B.2.2. De [78], página 20.

```
data nat = z | s nat

isEven z      = true
isEven  (s N) = if (isEven N) then false
                              else true

isOdd z      = false
isOdd  (s N) = if (isOdd N) then false
                                 else true

foo N = []  <== isOdd N

foo N = [N] <== isEven N
```

La semántica pretendida en este caso es:

- $isEvenN \rightarrow true$ si N par.
- $isEvenN \rightarrow false$ si N impar.
- $isOddN \rightarrow true$ si N impar.
- $isOddN \rightarrow false$ si N par.
- $fooN \rightarrow []$ si $N > 0$, N impar.
- $fooN \rightarrow [N]$ si $N > 0$, N par.

Es decir, foo sólo debe tener éxito para números mayores que cero. Por tanto al lanzar el objetivo $foo\ z == R$ se espera que no se obtenga ninguna respuesta, pero en lugar de eso se obtiene la respuesta incorrecta $R == [z]$. En este caso se tiene que el error es debido a que la función está definida para argumentos que no pertenecen a su dominio en la interpretación pretendida.

Sesión de depuración:

```
TOY> foo z == R

     yes
     R ==  [ z ]
```

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

Consider the following facts:
1: foo z  ->  [z ]

Are all of them valid? ([y]es / [n]ot) / [a]bort) n

Consider the following facts:
1: isEven z  -> true

Are all of them valid? ([y]es / [n]ot) / [a]bort) y

Rule number 2 of the function foo is wrong.
Buggy node: foo z  ->   [z]
```

Ejemplo B.2.3. De [19].

Programa para calcular números primos basado en la *criba de Eratóstenes*. En el modelo pretendido la función *primes* representa la lista (infinita) de los números primos, mientras $sieve(X : Xs)$ debe eliminar de $X : Xs$ todos los elementos que sean divisibles por algún elemento anterior.

```
primes :: [int]
primes = sieve (from 2)

sieve        :: [int] -> [int]
sieve (X:Xs) = X : filter (notDiv X) (sieve Xs)

notDiv     :: int -> int -> bool
notDiv X Y = mod X Y > 0

% Wrong. The expected semantics is:
% notDiv X Y returns true iff X doesn't divide Y

% The rule should be:
% notDiv X Y = mod Y X > 0

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

filter          :: (A -> bool) -> [A] -> [A]
filter P    []  = []
```

```
filter P (X:Xs) = if P X
                  then X : filter P Xs
                  else filter P Xs

take          :: int -> [A] -> [A]
take 0 L = []
take N (X:Xs) = X: take (N-1) Xs <== N>0
```

Sesión de depuración:

```
 TOY> (take 5 primes) == R

      yes
       R ==  [ 2, 3, 4, 5, 6 ]

      Elapsed time: 0 ms.


more solutions (y/n/d) [y]? d
{compiling /home/trend/rafa/tesis/ejemplos/sieve.tmp.out...}
....
Consider the following facts:
1: primes ->  [2, 3, 4, 5, 6 | _ ]
2: take 5  [2, 3, 4, 5, 6 | _ ] ->  [2, 3, 4, 5, 6]

Are all of them valid? ([y]es / [n]ot) / [a]bort) n
Enter the number of a non-valid fact followed by a fullstop: 1.

Consider the following facts:
1: from 2 ->  [2, 3, 4, 5, 6 | _ ]
2: sieve  [2, 3, 4, 5, 6 | _ ] ->  [2, 3, 4, 5, 6 | _ ]

Are all of them valid? ([y]es / [n]ot) / [a]bort) n
Enter the number of a non-valid fact followed by a fullstop: 2.

Consider the following facts:
1: sieve  [3, 4, 5, 6 | _ ] ->  [3, 4, 5, 6 | _ ]
2: filter (notDiv 2)  [3, 4, 5, 6 | _ ] ->  [3, 4, 5, 6 | _ ]

Are all of them valid? ([y]es / [n]ot) / [a]bort) n
Enter the number of a non-valid fact followed by a fullstop: 1.
```

```
Consider the following facts:
1: sieve  [4, 5, 6 | _ ] ->  [4, 5, 6 | _ ]
2: filter (notDiv 3)  [4, 5, 6 | _ ] ->  [4, 5, 6 | _ ]

Are all of them valid? ([y]es / [n]ot) / [a]bort) 2.

Please, answer y, n or a: n
Enter the number of a non-valid fact followed by a fullstop: 2.

Consider the following facts:
1: notDiv 3 4 -> true
2: filter (notDiv 3)  [5, 6 | _ ] ->  [5, 6 | _ ]

Are all of them valid? ([y]es / [n]ot) / [a]bort) n
Enter the number of a non-valid fact followed by a fullstop: 2.

Consider the following facts:
1: notDiv 3 5 -> true
2: filter (notDiv 3)  [6 | _ ] ->  [6 | _ ]

Are all of them valid? ([y]es / [n]ot) / [a]bort) n
Enter the number of a non-valid fact followed by a fullstop: 2.

Consider the following facts:
1: notDiv 3 6 -> true

Are all of them valid? ([y]es / [n]ot) / [a]bort) n

Rule number 1 of the function notDiv is wrong.
Buggy node: notDiv 3.0 6.0  ->  true
```

Resulta interesante comparar esta sesión con la correspondiente sesión para el mismo programa en Curry que puede verse en el ejemplo B.4.1 (pág. 300).

Ejemplo B.2.4. Este ejemplo presenta algunas operaciones aritméticas básicas definidas sobre números naturales representados mediante la notación de Peano.

```
infixr 90 .
(.) :: (B -> C) -> (A -> B) -> (A -> C)
(F . G) X = F (G X)
```

```
data nat = z | s nat

plus::  nat -> nat -> nat
plus X z = X
plus X (s Y) = s (plus X Y)

product::  nat -> nat -> nat
product X Y = nTimes X (plus Y) Y

square::  nat -> nat
square X = product X X

squareInt::  int -> int
squareInt =  natToInt.square.intToNat

nTimes:: nat -> (A ->A)->A->A
nTimes  z F X =  X
nTimes (s N) F X = F (nTimes N F X)

intToNat:: int -> nat
intToNat N = if (N==0) then z else s (intToNat (N-1))

natToInt:: nat -> int
natToInt z = 0
natToInt (s N) = 1+(natToInt N)
```

El significado de las funciones *plus*, *product* y *square*, así como del operador composición '.' es obvio. La función *squareInt* sirve para calcular el cuadrado de un número entero, mientras que *intToNat* y *natToInt* permiten pasar de enteros mayores o iguales que 0 a naturales y viceversa. Finalmente, el propósito de la función *nTimes* es aplicar una función a un valor la cantidad de veces indicada por su primer argumento, es decir que el hecho básico $nTimes\ N\ F\ X \rightarrow M$ se cumple en la interpretación pretendida si $M = F^N X$. Esta función se utiliza para definir el producto de dos números en función de la operación suma.

Sin embargo el programa no funciona correctamente, como prueba el siguiente objetivo, que incluímos seguido de la correspondiente sesión de depuración.

```
TOY> squareInt 2 == R

     yes
     R == 6
```

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


Consider the following facts:
1: squareInt -> natToInt . square . intToNat
2: (natToInt . square . intToNat) 2.0 -> 6.0

Are all of them valid? ([y]es / [n]ot) / [a]bort) n
Enter the number of a non-valid fact followed by a fullstop: 2.

Consider the following facts:
1: (square . intToNat) 2.0 -> (s (s (s (s (s (s))))))
2: natToInt (s (s (s (s (s (s)))))) -> 6.0

Are all of them valid? ([y]es / [n]ot) / [a]bort) n
Enter the number of a non-valid fact followed by a fullstop: 1.

Consider the following facts:
1: intToNat 2.0 -> (s (s z))
2: square (s (s z)) -> (s (s (s (s (s (s))))))

Are all of them valid? ([y]es / [n]ot) / [a]bort) n
Enter the number of a non-valid fact followed by a fullstop: 2.

Consider the following facts:
1: product (s (s z)) (s (s z)) -> (s (s (s (s (s (s))))))

Are all of them valid? ([y]es / [n]ot) / [a]bort) n

Consider the following facts:
1: nTimes (s (s z)) (plus (s (s z))) (s (s z)) ->
            s (s (s (s (s z)))))

Are all of them valid? ([y]es / [n]ot) / [a]bort) y

Rule number 1 of the function product is wrong.
Buggy node: product (s (s z)) (s (s z))  ->  s (s (s (s (s z))))
```

La regla detectada como errónea es la única regla de *product*. Tras examinar el nodo crítico y la regla de programa vemos cual es el problema: se realiza una suma de más. La solución precisa modificar la regla *product* desdoblando su definición en dos reglas:

```
product:: nat -> (A ->A)->A->A
```

```
product  z F X =  X
product (s N) F X = F (nTimes N F X)
```

Ejemplo B.2.5. Este ejemplo se utiliza en el capítulo 6 (sección 6.2, pág. 136) para mostrar las ventajas del depurador gráfico frente al depurador descendente.

```
infixr 50 ++

(++) :: [A] -> [A] -> [A]
[]     ++ Ys = Ys
[X|Xs] ++ Ys = [X|Xs ++ Ys]

foldr :: (A -> B -> B) -> B -> [A] -> B
foldr C N []     = N
foldr C N [X|Xs] = C X (foldr C N Xs)

data tree = node int tree tree | void

insert A void = node A void void
insert A (node B T1 T2) =   if (A > B) then node B T1' T2
                            else node B T1 T2'
                     where
                         T1' = insert A T1
                         T2' = insert A T2

inorder void = []
inorder (node A T1 T2) = inorder T1 ++ [A] ++ inorder T2

sort L = inorder SearchTree
              where SearchTree = foldr insert void L
```

En el citado capítulo se incluyen los detalles acerca del programa, cuya idea general es la utilización de un árbol de búsqueda para ordenar una lista de enteros. Aquí presentamos simplemente la sesión de depuración:

```
TOY> sort [3,2,1] == R

      yes
      R ==  [ 3, 2, 1 ]

more solutions (y/n/d) [y]? d
compiling...
Consider the following facts:
```

```
1: (sort  [3.0, 2.0, 1.0]) ->  [3.0, 2.0, 1.0]

Are all of them valid? ([y]es / [n]ot) / [a]bort) n

Consider the following facts:
1: foldr insert void  [3.0, 2.0, 1.0] ->
        node 1.0 (node 2.0 (node 3.0 void void) void) void
2: inorder (node 1.0 (node 2.0 (node 3.0 void void) void) void) ->
         [3.0, 2.0, 1.0]

Are all of them valid? ([y]es / [n]ot) / [a]bort) n
Enter the number of a non-valid fact followed by a fullstop: 1.

Consider the following facts:
1: foldr insert void  [2.0, 1.0] -> node 1.0 (node 2.0 void void) void
2: insert 3.0 (node 1.0 (node 2.0 void void) void) ->
         node 1.0 (node 2.0 (node 3.0 void void) void) void

Are all of them valid? ([y]es / [n]ot) / [a]bort) n
Enter the number of a non-valid fact followed by a fullstop: 1.

Consider the following facts:
1: foldr insert void  [1.0] -> node 1.0 void void
2: insert 2.0 (node 1.0 void void) ->
        node 1.0 (node 2.0 void void) void

Are all of them valid? ([y]es / [n]ot) / [a]bort) n
Enter the number of a non-valid fact followed by a fullstop: 2.

Consider the following facts:
1: insert 2.0 void -> node 2.0 void void

Are all of them valid? ([y]es / [n]ot) / [a]bort) y

Rule number 2 of the function insert is wrong.
Buggy node: insert 2.0 (node 1.0 void void)  ->
                    node 1.0 (node 2.0 void void) void
```

Ejemplo B.2.6. El propósito del siguiente programa es simplificar expresiones sencillas, que incluyen las operaciones + y *, llamadas *add* y *mul* respectivamente en el programa, y como posibles operandos constantes representadas como $c\ n$ para algún número real n, o variables de la forma $v\ m$ con m un número entero:

```
% simple expressions: constants, variables, +, and *
```

```
data expr = c real | v int | add expr expr |    mul expr expr

% simplify an expression
simplify:: expr ->expr

simplify (c I) = c I
simplify (v I) = v I

simplify (add E1 E2) = simplifyAdd (simplify E1) (simplify E2)
simplify (mul E1 E2) = simplifyMul  (simplify E1) (simplify E2)

%%%%%%%%%% simplifyAdd %%%%%%%%%%%

% E1 or E2 zero
simplifyAdd (c 0)  E2   = E2
simplifyAdd E1    (c 0) = E1

% both E1 and E2 constants
simplifyAdd (c ValueE1) (c ValueE2) =  c (ValueE1+ValueE2)

% other cases
simplifyAdd E1 E2 = add E1 E2

%%%%%%%%%% simplifyMul %%%%%%%%%%%

%  E1 or E2 zero
simplifyMul (c 0) E2 = E2
simplifyMul E1 (c 0) = E1

% E1 or E2 the constant 1
simplifyMul (c 1) E2 = E2
simplifyMul E1 (c 1) = E1

% both E1 and E2 constants
simplifyMul  (c ValueE1) (c ValueE2) =  c (ValueE1*ValueE2)

% other cases
simplifyMul E1 E2 = mul E1 E2
```

El error se encuentra en las dos primeras reglas de $simplifyMul$, destinadas a la simplificación de productos por cero, que deberían ser:

```
%  E1 or E2 zero
simplifyMul (c 0) E2 = c 0
```

```
simplifyMul E1 (c 0) = c 0
```

El siguiente objetivo trata de simplificar la expresión $0 \times (2 \times ((x + 1) \times 1))$, donde la variable x se representa por v 1:

```
TOY>  simplify (mul (c 0)
           (mul (c 2) (mul (add (v 1) (c 1)) (c 1))))==R

       yes
      R ==  (mul (c 2) (add (v 1) (c 1)))

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

El usuario comienza la sesión de depuración porque la respuesta esperada era 0. La sesión de depuración es:

```
Consider the following facts:
1: simplify (mul (c 0)  (mul (c 2) (mul (add (v 1) (c 1)) (c 1)))) ->
     mul (c 2) (add v 1 c 1)

Are all of them valid? ([y]es / [n]ot) / [a]bort) n

Consider the following facts:
1: simplify (c 0) -> c 0
2: simplify (mul (c 2) (mul (add (v 1) (c 1)) (c 1))) ->
      mul (c 2) (add (v 1) (c 1))
3: simplifyMul (c 0) (mul (c 2) (add (v 1) (c 1))) ->
      mul (c 2) (add (v 1) (c 1))

Are all of them valid? ([y]es / [n]ot) / [a]bort) n
Enter the number of a non-valid fact followed by a fullstop: 3.

Rule number 1 of the function simplifyMul is wrong.
```

Recordemos que el depurador sólo localiza una regla errónea cada vez. En esta ocasión la primera regla de las dos erróneas es detectada.

Ejemplo B.2.7. El siguiente programa se utiliza en el apartado 5.5.2 (pág. 126) para mostrar la limitación del depurador de $\mathcal{TOY}$ a programas sin restricciones aritméticas ni de desigualdad.

```
rectan (X,Y) LX LY (X2,Y2) = true
            <== X2 >= X, X2 <X+LX, Y2<=Y, Y2<Y+LY

intersecc A1 A2 P = true <== A1 P, A2 P
```

```
fig1 = rectan (20,20) 50 20
fig2 = rectan (5,5) 30 40
```

En el citado apartado se puede encontrar una descripción de la interpretación de las funciones, que no repetimos aquí. Un ejemplo de sesión de depuración para un cómputo en el que no se utilizan las restricciones aritméticas, y que por tanto puede llevarse a cabo en $\mathcal{TOY}$ es:

```
TOY> intersecc fig1 fig2 (30,0)
      yes

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

Consider the following facts:
1: fig1 -> rectan (20, 20) 50 20
2: fig2 -> rectan (5, 5) 30 40
3: intersecc (rectan (20, 20) 50 20)
              (rectan (5, 5) 30 40 (30, 0)) -> true

Are all of them valid? ([y]es / [n]ot) / [a]bort) n
Enter the number of a non-valid fact followed by a fullstop: 3.

Consider the following facts:
1: rectan (20, 20) 50 20 (30, 0) -> true
2: rectan (5, 5) 30 40 (30, 0) -> true

Are all of them valid? ([y]es / [n]ot) / [a]bort) n
Enter the number of a non-valid fact followed by a fullstop: 1.

Rule number 1 of the function rectan is wrong.
```

Tal y como muestra el depurador la función rectan es errónea. En particular la tercera condición atómica Y2 <= Y debería ser Y <= Y2.

Ejemplo B.2.8. Finalizamos el apartado dedicado a los programas funcionales con otro un ejemplo cuyo objetivo es calcular la *razón aúrea*, presentado también en el capítulo 6 (sección 6.2, pág. 136), donde se puede consultar el significado pretendido de cada función.

```
fib         =   1:1 : (fibAux 1 1)

fibAux N M  = (N+M) :  (fibAux N (N+M))
```

```
goldenApprox = (tail fib) ./. fib

infixr  20 ./.
(X:Xs) ./. (Y:Ys) = (X/Y) : (Xs ./. Ys)

tail  (X:Xs) = Xs

take 0 L = []
take N (X:Xs) = X: take (N-1) Xs <== N>0
```

Un objetivo con respuesta incorrecta junto con su correspondiente sesión de depuración:

```
take 5 goldenApprox == R

      yes
      R ==  [ 1, 2, 1.5, 1.33, 1.25 ]

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

Consider the following facts:
1: goldenApprox ->  [1, 2, 1.5, 1.33, 1.25 | _ ]
2: take 5  [1, 2, 1.5, 1.33, 1.25 | _ ] ->  [1, 2, 1.5, 1.33, 1.25]

Are all of them valid? ([y]es / [n]ot) / [a]bort) n
Enter the number of a non-valid fact followed by a fullstop: 1.

Consider the following facts:
1: fib ->  [1, 1, 2, 3, 4, 5 | _ ]
2: tail  [1, 1, 2, 3, 4, 5 | _ ] ->  [1, 2, 3, 4, 5 | _ ]
3: fib ->  [1, 1, 2, 3, 4 | _ ]
4:  [1, 2, 3, 4, 5 | _ ] ./.  [1, 1, 2, 3, 4 | _ ] ->
     [1, 2, 1.5, 1.33, 1.25 | _ ]

Are all of them valid? ([y]es / [n]ot) / [a]bort) n
Enter the number of a non-valid fact followed by a fullstop: 1.

Consider the following facts:
1: fibAux 1 1 ->  [2, 3, 4, 5 | _ ]

Are all of them valid? ([y]es / [n]ot) / [a]bort) n

Consider the following facts:
1: fibAux 1 2 ->  [3, 4, 5 | _ ]
```

```
Are all of them valid? ([y]es / [n]ot) / [a]bort) n

Consider the following facts:
1: fibAux 1 3 ->  [4, 5 | _ ]

Are all of them valid? ([y]es / [n]ot) / [a]bort) n

Consider the following facts:
1: fibAux 1 4 ->  [5 | _ ]

Are all of them valid? ([y]es / [n]ot) / [a]bort) y

Rule number 1 of the function fibAux is wrong.
Buggy node: fibAux 1 3  ->   [4, 5 | _ ]
```

B.3. Programación Lógico-Funcional

Ejemplo B.3.1. El siguiente programa sirve para ordenar una lista mediante el método de "generación y prueba" (ver ejemplo 2.4.1, página 39 para una breve descripción del funcionamiento del programa).

```
(//) :: A -> A -> A
X // Y = X
X // Y = Y

permSort :: [int] -> [int]
permSort Xs = if (isSorted Y) then Y
                        where Y = permute Xs

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

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

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

La semántica pretendida en este caso es:

- $permSortL \rightarrow Ls$ si Ls es la lista L ordenada de forma creciente.
- $permuteL \rightarrow Ls$ si Ls es una permutación de L.
- $insertXL \rightarrow Ls$ si Ls es el resultado de insertar X en alguna posición de la lista L.
- $isSortedL \rightarrow true$ si L es una lista ordenada de forma creciente.

Sesión de depuración:

```
TOY>  permSort [3,2,1] == R

      yes
      R ==  [ 3 ]

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

Consider the following facts:
1: permSort  [3.0, 2.0, 1.0]  ->  [3.0]

Are all of them valid? ([y]es / [n]ot) / [a]bort) n

Consider the following facts:
1: permute  [3.0, 2.0, 1.0]  ->  [3.0]
2: isSorted  [3.0]  -> true

Are all of them valid? ([y]es / [n]ot) / [a]bort) n
Enter the number of a non-valid fact followed by a fullstop: 1.

Consider the following facts:
1: permute  [2.0, 1.0]  ->  [2.0, 1.0]
2: insert 3.0  [2.0, 1.0]  ->  [3.0]

Are all of them valid? ([y]es / [n]ot) / [a]bort) n
Enter the number of a non-valid fact followed by a fullstop: 2.

Consider the following facts:
1: insert 3.0  [1.0]  ->  [3.0]
2: (//)  [3.0, 2.0, 1.0]  [3.0]  ->  [3.0]

Are all of them valid? ([y]es / [n]ot) / [a]bort) n
Enter the number of a non-valid fact followed by a fullstop: 1.

Consider the following facts:
```

```
1: insert 3.0  []  ->  [3.0]
2: (//)  [3.0, 1.0]  [3.0]  ->  [3.0]

Are all of them valid? ([y]es / [n]ot) / [a]bort) y

Rule number 2 of the function insert is wrong.
Buggy node: insert 3.0  [1.0]  ->   [3.0]
```

En la figura 2.6 de la página 42 se puede ver el árbol de prueba asociado a este cómputo.

Ejemplo B.3.2. Otro ejemplo del método de "generación y prueba" lo representa el siguiente programa, cuyo propósito es resolver el conocido *problema de las n reinas*: sobre un tablero de $n \times n$ casillas hay que lograr colocar n reinas de ajedrez de forma que no se "amenacen" entre ellas. El programa es el siguiente:

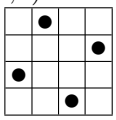
```
solution N = if (noCheck L) then L
             where L  = generateQueens N

%%%%%%%%%%%%%%%% generating %%%%%%%%%%%%%%%%

% generating all the queens
generateQueens N = generateQueensAux N N
generateQueensAux 0 N = []
generateQueensAux M N = [chooseQueen N |
                                  generateQueensAux (M-1) N] <== M>0


% non-deterministic generation of possible coordinates of a queen
chooseQueen N = (choose 1 N, choose 1 N)

% a non-deterministic value between N and M
choose N M = N <== N<=M
choose N M = choose (N+1) M <== N<M


%%%%%%%%%%%%%%%% testing %%%%%%%%%%%%%%%%

% checking  if there is any check in the list of the queens
noCheck [] :- true
noCheck [(X,Y) | L] :-  noCheckQueen (X,Y) L , noCheck L

% no check  between a given queen and a list of queens
noCheckQueen (X,Y) [] :- true
```

```
noCheckQueen (X,Y) [(X2,Y2)|L] :- noCheckTwoQueens (X,Y) (X2,Y2),
                                  noCheckQueen (X2,Y2) L

% no check between two queens
noCheckTwoQueens (X1,Y1) (X2,Y2) :- X1 /=X2, Y1 /= Y2,
                                    abs(X1-X2) /= abs(Y1-Y2)
```

La función principal es *solution* y su interpretación pretendida:

$(solution\ n \to l) \in \mathcal{I}$ sii l es una solución para el problema de las n reinas.

La solución debe ser una lista con las posiciones de las n reinas en el tablero. Cada posición es un par de la forma (x, y) con $1 \leq x, y \leq n$. Por ejemplo se tiene $(solution\ 4 \to [(1,2),(2,4),(3,1),(4,3)]) \in \mathcal{I}$, ya que la lista $[(1,2),(2,4),(3,1),(4,3)]$ representa la disposición (suponiendo que la coordenada (1,1) corresponde a la casilla inferior izquierda):



La función utiliza como funciones auxiliares *generateQueens*, una función capaz de generar todas las listas con n posiciones, y *noCheck* que dada una lista de posiciones compruebas si éstas constituyen una solución. La interpretación de estas funciones y sus funciones auxiliares es (esperamos) clara a partir de sus reglas y de los comentarios incluidos en el código.

Sin embargo el sistema computa una respuesta incorrecta para el siguiente objetivo:

```
TOY> solution 4 == R

     yes
     R ==  [ (1, 1), (2, 3), (1, 1), (2, 3) ]

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

ya que hay dos reinas en la misma posición lo que no constituye una solución válida. La sesión de depuración es entonces:

```
Consider the following facts:
1: solution 4 ->  [(1, 1), (2, 3), (1, 1), (2, 3)]

Are all of them valid? ([y]es / [n]ot) / [a]bort) n

Consider the following facts:
1: generateQueens 4 ->  [(1, 1), (2, 3), (1, 1), (2, 3)]
2: noCheck  [(1, 1), (2, 3), (1, 1), (2, 3)] -> true
```

```
Are all of them valid? ([y]es / [n]ot) / [a]bort) n
Enter the number of a non-valid fact followed by a fullstop: 2.

Consider the following facts:
1: noCheckQueen (1, 1)  [(2, 3), (1, 1), (2, 3)] -> true
2: noCheck  [(2, 3), (1, 1), (2, 3)] -> true

Are all of them valid? ([y]es / [n]ot) / [a]bort) n
Enter the number of a non-valid fact followed by a fullstop: 1.

Consider the following facts:
1: noCheckTwoQueens (1, 1) (2, 3) -> true
2: noCheckQueen (2, 3)  [(1, 1), (2, 3)] -> true

Are all of them valid? ([y]es / [n]ot) / [a]bort) n
Enter the number of a non-valid fact followed by a fullstop: 2.

Consider the following facts:
1: noCheckTwoQueens (2, 3) (1, 1) -> true
2: noCheckQueen (1, 1)  [(2, 3)] -> true

Are all of them valid? ([y]es / [n]ot) / [a]bort) y

Rule number 2 of the function noCheckQueen is wrong.
Buggy node: noCheckQueen (2.0, 3.0)
                         [(1.0, 1.0), (2.0, 3.0)]  ->  true
```

El error está, como señala el depurador, en la segunda regla de *noCheckQueen*, en concreto en la segunda condición $noCheckQueen\ (X2, Y2)\ L$ que debería reemplazarse por $noCheckQueen\,(X, Y)\,L$.

Ejemplo B.3.3. El siguiente ejemplo es un evaluador para expresiones simples que, además de devolver el valor resultado de evaluar la expresión, produce una representación en forma de caracteres de la expresión evaluada.

En programación funcional esto se puede hacer agrupando los dos valores a devolver en una tupla, definiendo por tanto una función de tipo $expr \to (int, [char])$. Para facilitar la estructuración de las reglas de función, así como la legibilidad del código resultante, se pueden utilizar *mónadas* [92]. En programación lógico-funcional además de este recurso también podemos usar *extensiones* [13]. La idea es definir la función evaluadora (de nombre *eval* en el ejemplo) con tipo $expr \to int \to [char]$), utilizando el segundo argumento como argumento de salida para devolver el resultado. Definiendo combinadores de orden superior adecuados el aspecto del programa es muy similar al que se tendría para el caso de las mónadas, como se ve en el ejemplo:

```
%%%%%%%% the output extension %%%%%%%%%%
% the extension combinator
infixr 30 <*>

% the output is a string
type output = [char]

% unit returns the value as well as its string representation
unit:: int -> int  -> output
unit A A =   stringRep A

% the extension combinator is here the list concatenation
(<*>):: output -> output -> output
X <*> Y =  X++Y

[] ++ L = L
[X|Xs] ++ L = [X|Xs ++ L]

%%%%%%%%%% expressions %%%%%%%%%%

infix 20 :/:,:*:,:+:,:-:
data expr = val int  | expr :+: expr | expr :-: expr |
                expr :*: expr| expr :/: expr

% expresion evaluator using the output extension
eval:: expr -> int -> output
eval (val A) R = unit A R
eval (A :+: B)  R = eval A R1  <*> "+" <*> eval B R2
                      <*> " = " <*>  unit (R1+R2) R
eval (A :-: B)  R = eval A R1  <*> "-" <*> eval B R2
                      <*> " = " <*> unit (R1-R2) R
eval (A :*: B)  R = eval A R1 <*> "*" <*> eval B R2
                      <*> " = " <*> unit (R1*R2) R
eval (A :/: B)  R = eval A R1 <*> "/" <*> eval B R2
                      <*> " = " <*> unit (div R1 R2) R

%%%%%%%%% integer to  string %%%%%%%%%%
stringRep N = if N == 0 then "0"
              else if N<10 then LastDigit
                    else  stringRep (div N 10) ++ LastDigit
              where
                 LastDigit = lastDigitRep N
```

```
% string representation of the last digit of the number
lastDigitRep :: int -> [char]
lastDigitRep N = [chr ((div N  10) + (ord '0'))]
```

Las primitivas *chr* y *ord* utilizadas en la última función devuelven, respectivamente, el carácter asociado a un código numérico y viceversa, el valor numérico asociado a un carácter. El programa evalúa correctamente las expresiones pero la representación que computa es incorrecta, debido a un error en la regla para *lastDigitRep*, en la que debería poner *mod* en lugar de *div* para obtener el último dígito de su parámetro.

```
  eval ( (val 4):+:(val 8) ) R == R2

      yes
      R == 12
      R2 ==    "0+0 = 01"

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

Consider the following facts:
1: eval (val 4 :+: val 8) 12 ->  "0+0 = 01"

Are all of them valid? ([y]es / [n]ot) / [a]bort) n

Consider the following facts:
1: eval (val 4) 4 ->  "0"
2: eval (val 8) 8 ->  "0"
3: unit 12 12 ->  "01"
4:  " = " <*>  "01" ->  " = 01"
5:  "0" <*>  " = 01" ->  "0 = 01"
6:  "+" <*>  "0 = 01" ->  "+0 = 01"
7:  "0" <*>  "+0 = 01" ->  "0+0 = 01"

Are all of them valid? ([y]es / [n]ot) / [a]bort) n
Enter the number of a non-valid fact followed by a fullstop: 1.

Consider the following facts:
1: unit 4 4 ->  "0"

Are all of them valid? ([y]es / [n]ot) / [a]bort) n

Consider the following facts:
```

```
1: stringRep 4 ->  "0"

Are all of them valid? ([y]es / [n]ot) / [a]bort) n

Consider the following facts:
1: lastDigitRep 4 ->  "0"

Are all of them valid? ([y]es / [n]ot) / [a]bort) n

Rule number 1 of the function lastDigitRep is wrong.
Buggy node: lastDigitRep 4  ->  "0"
```

En la primera pregunta obtenemos 7 nodos hijos, que es la cantidad de llamadas a función que se encuentran en la segunda regla de *eval*: una llamada por cada una de las 4 apariciones del combinador de orden superior $<*>$, otras 2 por las llamadas recursivas a *eval* y una última por la llamada a *unit*.

B.4. Depuración en Curry

Como hemos indicado en la tesis el depurador de Curry es similar al de $\mathcal{TOY}$. Aquí sólo mostramos algunos ejemplos que permiten apreciar las diferencias fundamentales entre ambos desarrollos.

Ejemplo B.4.1. De [19], adaptado a la sintaxis de Curry.

Este es el mismo programa que mostramos en sintaxis $\mathcal{TOY}$ en el ejemplo B.2.3 (pág. 282). Resulta interesante apreciar las diferencias entre las sesiones de depuración en ambos casos. En Curry las funciones *take* y $filter$ son consideradas fiables, por lo que no forman parte del cómputo y por tanto no aparecen en las preguntas del depurador. El resultado es una sesión de depuración completamente distinta al caso de $\mathcal{TOY}$, más breve y con respuestas más sencillas. En $\mathcal{TOY}$ se puede conseguir un efecto similar usando el navegador visual $\mathcal{DDT}$ y marcando las funciones *take* y $filter$ como *trusted*.

```
primes ::  [Int]
primes = sieve (from 2)

sieve        :: [Int] -> [Int]
sieve (X:Xs) = X : filter (notDiv X) (sieve Xs)

notDiv     :: Int -> Int -> Bool
notDiv X Y = mod X Y > 0

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

```
main  =  take 5 primes
```

Sesión de depuración:

```
Considering the following basic facts:
1. main.main()  -> [2,3,4,5,6]
Are all of them valid? (y/n) n

Considering the following basic facts:
1. main.primes()  -> [2,3,4,5,6|_]
Are all of them valid? (y/n) n

Considering the following basic facts:
1. main.from(2)  -> [2,3,4,5,6|_]
2. main.sieve([2,3,4,5,6|_])  -> [2,3,4,5,6|_]
Are all of them valid? (y/n) n
Write the number of an erroneous basic fact in the list 2

Considering the following basic facts:
1. main.sieve([3,4,5,6|_])  -> [3,4,5,6|_]
2. main.notDiv(2, 3)  -> True
3. main.notDiv(2, 4)  -> True
4. main.notDiv(2, 5)  -> True
5. main.notDiv(2, 6)  -> True
Are all of them valid? (y/n) n
Write the number of an erroneous basic fact in the list 3

 ** Function main.notDiv is incorrect ( main.notDiv(2, 4)  -> True) **

Buggy node found
Debugger exiting
```

En el APA inicial de este cómputo cada nodo correspondiente a la función $filter$ tiene dos hijos: uno para la aplicación de su patrón de orden superior y otro asociado a la llamada recursiva a la propia función. Al ser $filter$ una función fiable, sus nodos asociados se eliminan, pero no así sus hijos, que pasan a depender del padre del nodo eliminado. Por eso se acumulan todas las aplicaciones del parámetro de orden superior (función $notDiv$ en este caso) como hijos de un mismo nodo, lo que se aprecia en la última pregunta del depurador.

También es interesante que los nombres de las funciones vienen precedidos del prefijo *main.*. Esto se refiere al nombre del módulo en el que están definidas.

Ejemplo B.4.2. El siguiente programa muestra una de las diferencias de Curry con respecto a $\mathcal{TOY}$: en Curry se admite el uso de funciones lambda y definiciones locales generales (del mismo tipo que en Haskell [75]), mientras que en $\mathcal{TOY}$ sólo se admiten definiciones locales no recursivas y con patrones en el lado izquierdo. El programa pretende representar el tipo abstracto cola.

```
-- the first element of the queue will occur at the left,
--  i.e. the queue  "Queue 1 (Queue 2 Empty)"
-- is the queue whose first element is 1
data queue A = Empty | Queue  A (queue A)

 -- the first element of the queue
first (Queue N Q) =  N

-- delete the first element of the queue
unqueue (Queue N Q) = Q

-- add new element, that will be the last of the queue
--  (i.e. the rightmost element in the structure)
new N Empty = Queue N Empty
new N (Queue M Q) = Queue N (newQueue M)
                   where newQueue V = new V Q

-- convert the queue in a Toy list
queueToList :: queue A -> [A]
queueToList Q | Q== Empty = []
          | otherwise =   (first Q : queueToList (unqueue Q))


-- insert all the elements of a  list in a queue
-- the head of the list will be the first element of the queue
listToQueue L =foldl (\Q -> (\Element -> new Element Q)) Empty L
```

La interpretación pretendida de cada función se deduce de los comentarios situados. La función *new* utiliza una función definida localmente de nombre *newQueue*, cuyo propósito es añadir el elemento que recibe como último elemento de la cola Q. La función *listToQueue* crea una cola a partir de una lista. Por ejemplo un hecho básico como

$$listToQueue\ [1,2,3]\ \rightarrow\ Queue\ 1\ (Queue\ 2\ (Queue\ 3\ Empty))$$

está en la interpretación pretendida del programa. Esta función incluye dos funciones lambda anidadas: $(\backslash Q- > (\backslash Element- > new\ Element\ Q))$. La primera función recibe un parámetro Q y su propósito es devolver una función que dado un elemento *Element* inserte *Element* en Q. Esto define también el propósito de la segunda función.

Si el programa fuera correcto debería suceder que $queueToList\,.\,lisToQueue = id$, por ser ambas funciones inversas. Por ello el siguiente objetivo muestra que hay algún error:

```
queueToList (listToQueue [1,2,3])
[3,2,1]
```

Podemos utilizar entonces el depurador de Curry, obteniendo la siguiente sesión:

```
 Entering debugger...

Considering the following basic facts:
1. main.listToQueue([1,2,3])  -> Queue 3 (Queue 2 (Queue 1 Empty))
2. main.queueToList(Queue 3 (Queue 2 (Queue 1 Empty)))  -> [3,2,1]
Are all of them valid? (y/n) n
Write the number of an erroneous basic fact in the list 1.

Considering the following basic facts:
1. main.listToQueue._#lambda1(Empty, 1)  -> Queue 1 Empty
2. main.listToQueue._#lambda1(Queue 1 Empty, 2)  ->
     Queue 2 (Queue 1 Empty)
3. main.listToQueue._#lambda1(Queue 2 (Queue 1 Empty), 3)  ->
     Queue 3 (Queue 2 (Queue 1 Empty))
Are all of them valid? (y/n) n
Write the number of an erroneous basic fact in the list 2

Considering the following basic facts:
1. main.new(2, Queue 1 Empty)  -> Queue 2 (Queue 1 Empty)
Are all of them valid? (y/n) n

Considering the following basic facts:
1. main.new.newQueue(Empty, 1)  -> Queue 1 Empty
Are all of them valid? (y/n) y

 ** Function main.new is incorrect
     ( main.new(2, Queue 1 Empty)  -> Queue 2 (Queue 1 Empty)) **

Buggy node found
Debugger exiting
```

En este caso la regla *new* era incorrecta tal y como indica el depurador. La solución es intercambiar los papeles de las variables N y M en esta función, ya que el nuevo elemento N no debe ser el primero de la cola resultante sino el último, conservándose como primero M.

En la sesión de depuración aparecen preguntas sobre las funciones locales y las funciones lambda, ya que éstas también pueden ser erróneas. Una notación como

$$main.listToQueue.\_\#lambda1$$

se refiere a la primera función lambda de la función *listToQueue*.

Es interesante observar que esta función tiene dos parámetros cuando sólo había sido definida con uno. Esto es así porque Curry ha unido las dos funciones en una sola. Una mejora en el depurador sería desactivar estas optimizaciones para evitar que éstas puedan confundir al usuario. Con esta información podemos observar que el hecho básico

$$main.listToQueue.\_\#lambda1(Queue\ 1\ Empty,\ 2)\ \rightarrow\ Queue\ 2\ (Queue\ 1\ Empty)$$

no es válido; el elemento insertado (2 en este caso) debería pasar a ser el último de la lista y no el primero como aquí sucede.

También en la pregunta en la que aparece la función local *newQueue* nos encontramos con que en el hecho básico aparece con dos parámetros cuando estaba definida sólo con uno. El parámetro adicional es el valor Q, que es una variable externa de la que también depende la función. Esta es otra posible mejora para versiones futuras del depurador. Una solución, ya propuesta en [78] es asegurar que los argumentos introducidos aparecen siempre los primeros en la lista de argumentos y apuntar, por ejemplo como parte del prefijo del nombre "extendido" de la función, la cantidad de argumentos de este tipo utilizados y sus nombres. Así el navegador podría mostrar al usuario estos argumentos al presentar el hecho básico. Por ejemplo en la sesión de depuración anterior se podría cambiar el hecho básico referente a *newQueue* por algo como:

$$main.new.newQueue\ 1\ \rightarrow Queue\ 1\ Empty\quad (Q = Empty)$$

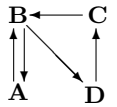
Ejemplo B.4.3. De [18].

El siguiente ejemplo es una programa para buscar caminos en un grafo similar al que presentamos en el ejemplo 2.1.1 (pág. 11) para programación lógica. De hecho el error del programa es uno de los dos errores que aparecían en el citado ejemplo 2.1.1: las variables y, z aparecen intercambiadas en la llamada recursiva contenida en la segunda regla de la función *path* (equivalente al predicado *camino* de2.1.1).

```
data Node = A | B | C | D

graph:: [(Node,Node)]
graph = [(A,B), (B,A), (B,D), (D,C), (C,B)]

path:: [(Node,Node)] -> Node -> Node -> [Node]
path g x  y | x =:= y = [x]
path g x  y | (member (x,z) g) &
```

```
          (path g y z =:= l) = x:l where z,l free

member:: a -> [a] -> Success
member x  (y:ys) = x=:=y
member x  (y:ys) = member x ys

bfs g = step [g]
        where
               step  [] = []
               step  (g:gs) = collect (try g) gs
               collect  []  gs = step gs
               collect  [g]  gs = g : step gs
               collect  (g1:g2:gs)  gs' = step (gs' ++ g1:g2:gs)

-- unpack is defined in the Prelude
-- unpack g | g x = x where x free

main = unpack (head (bfs (\l -> l =:= path graph B C)))
```

Sin embargo el propósito de este programa es más ambicioso, porque no se limita a grafos sin ciclos, sino que utiliza la búsqueda encapsulada de Curry [42] para lograr un recorrido en anchura del grafo. De esto se ocupa la función bfs que ya comentamos en el apartado 5.3 (pág. 113). El grafo, representado como una lista de pares de vértices, es el siguiente:



En este caso el objetivo se ha introducido a través de una función *main*, cuya parte derecha corresponde a la búsqueda del primer camino de B a C que se pueda encontrar en el grafo realizando una búsqueda en anchura. El objetivo devuelve la respuesta incorrecta $[B, C, A, B]$ (que ni siquiera es un camino de B a C), por lo que el usuario inicial el depurador, obteniéndose la siguiente sección:

```
Entering debugger...

Considering the following basic facts:
1. main.main()  -> [B,C,A,B]
Are all of them valid? (y/n) n

Considering the following basic facts:
1. main.bfs(main._debug#main._#lambda1)  ->
     [DebugPrelude.wrap ("solved search goal" <search goal>)|_]
2. main.main._#lambda1([B,C,A,B])  -> Success
```

```
Are all of them valid? (y/n) n
Write the number of an erroneous basic fact in the list 2

Considering the following basic facts:
1. main.graph()  -> [(A,B),(B,A),(B,D),(D,C),(C,B)]
2. main.path([(A,B),(B,A),(B,D),(D,C),(C,B)], B, C)  -> [B,C,A,B]
Are all of them valid? (y/n) n
Write the number of an erroneous basic fact in the list 2

Considering the following basic facts:
1. main.member((B,A), [(A,B),(B,A),(B,D),(D,C),(C,B)])  -> Success
2. main.path([(A,B),(B,A),(B,D),(D,C),(C,B)], C, A)  -> [C,A,B]
Are all of them valid? (y/n) n
Write the number of an erroneous basic fact in the list 2

Considering the following basic facts:
1. main.member((C,B), [(A,B),(B,A),(B,D),(D,C),(C,B)])  -> Success
2. main.path([(A,B),(B,A),(B,D),(D,C),(C,B)], A, B)  -> [A,B]
Are all of them valid? (y/n) y

 ** Function main.path is incorrect
 ( main.path([(A,B),(B,A),(B,D),(D,C),(C,B)],C, A)  -> [C,A,B]) **

Buggy node found
Debugger exiting
```

La segunda pregunta del navegador es sin duda la más compleja. Está formada por dos hechos básicos:

```
1. main.bfs(main._debug#main._#lambda1)  ->
     [DebugPrelude.wrap ("solved search goal" <search goal>)|_]

2. main.main._#lambda1([B,C,A,B])  -> Success
```

De hecho, el nodo número 1 corresponde a un nodo generado internamente, sobre el que el usuario no puede decir si es válido o no. Una mejora del navegador sería eliminar la pregunta acerca de este nodo, a la que el usuario no puede contestar, y que surge habitualmente al hacer depuración incluyendo búsqueda encapsulada. En cambio el usuario sí puede contestar acerca de la validez del segundo hecho básico, ya que significa que el valor $[B, C, A, B]$ hace que la función lambda dentro de *main* tenga éxito, es decir que

$$[B, C, A, B] \ =:= \ path\ graph\ B\ C$$

(donde =:= es la igualdad estricta en Curry) se satisfaga, y esto obviamente no es válido. El resto de las preguntas son más fáciles de contestar, y llevan hasta la función *path* como

causante del error. Aquí el depurador de Curry no puede señalar la segunda regla de *path* sino a la función en general ya que se ha definido *path* mediante una sola regla de función con "guardas".

Apéndice C

Especificación en Haskell de un Depurador Declarativo General

En nuestro artículos [16] presentamos un depurador para síntomas positivos y negativos en programación lógica sin restricciones escrito en Haskell, que presentamos en este apéndice (ligeramente completado y mejorado).

El propósito de este trabajo no era desarrollar un depurador eficiente sino estudiar las características de los depuradores declarativos y en particular de los depuradores declarativos para la programación lógica.

Para ello desarrollamos primero una especificación ejecutable en Haskell de un depurador declarativo general, siguiendo las ideas de [63], para a continuación aplicar el esquema a dos casos particulares: la depuración de respuestas incorrectas y de respuestas perdidas para programas lógicos puros.

Decidimos utilizar el lenguaje funcional Haskell principalmente por dos motivos:

- Los depuradores para lenguajes lógicos presentados hasta entonces estaban escritos en el propio lenguaje, generalmente mediante metaintérpretes. Pensamos que la separación entre el depurador y el lenguaje depurado es clarificadora y mejora la comprensión de los aspectos esenciales del depurador declarativo.

- La expresividad del sistema de tipos de Haskell nos permite representar los diferentes componentes que forman el depurador.

C.1. Estructura del Depurador

Hemos dividido el código en diversos módulos que describimos brevemente a continuación y que presentamos en las secciones siguientes.

- Módulo *LogicProgramming*: En este módulo se representan en Haskell tanto los componentes esenciales de la programación lógica: átomos, términos, cláusulas, programas,

substituciones, etc, como los algoritmos para tratar con estos elementos: generación de variantes de cláusulas, algoritmo de unificación, etc.

- Módulo *Examples*: Este módulo contiene simplemente los programas lógicos a depurar representados mediante los tipos de datos del módulo anterior.
- Módulo *Trees*: Descripción de un tipo árbol y de algunas operaciones sencillas sobre los árboles.
- Módulo *Debugger*: Aquí se presenta un depurador declarativo genérico válido para cualquier paradigma.
- Módulo *PtCat*: Este módulo describe los árboles de prueba positivos y negativos.
- Modulo *Instances*: Define la instancia para síntomas positivos y negativos.
- Módulo *Main*: Finalmente el programa principal se encargará de obtener los árboles de prueba para los cómputos deseados y de mostrar los errores obtenidos.

En las secciones presentan estos módulos más detalladamente. El código correspondiente al depurador completo está al final del capítulo.

C.2. Programación Lógica en Haskell

El módulo *LogicProgramming* incluye la representación de los programas lógicos en Haskell. Dentro de este código se pueden distinguir tres partes diferenciadas.

En la primera se definen los tipos básicos para átomos, cláusulas, etc., así como las funciones para crear y aplicar substituciones.

En la segunda parte se incluyen funciones para obtener variantes de cláusulas y de programas. Estas funciones serán necesarias durante la simulación del cómputo. Para la generación de nuevos nombres de variables se utiliza un número entero que se va incrementando cada vez que se genera un nuevo nombre y que se va pasando de una función a otra mediante una mónada de estado. Dentro de este estado también se incluye un acumulador donde se van almacenando los renombramientos ya realizados de forma que dentro de una cláusula cada variable tenga un único renombramiento.

Es interesante notar que de las dos partes que consituyen el estado (representado por el tipo *Dict*), el contador debe incrementarse siempre, mientras que el almacen de renombramientos debe reinicializarse cada vez que se utiliza en una nueva cláusula. De esto se encarga la llamada a $resetDict$ que se puede ver al comienzo de la función que devuelve una nueva variante de una cláusula, $freshVariant$.

Por último la tercera parte del módulo está constituida por las funciones que se encargan de la unificación de átomos y términos. Para ello se incluye una representación del algoritmo de Martelli y Montanari.

C.3. Programa Ejemplo

El módulo *Examples* incluye un programa lógico de ejemplo representado mediante los tipos descritos en el módulo anterior, en particular el ejemplo 2.1.1 que utilizamos en el Capítulo 2 para ilustrar la depuración declarativa de programas lógicos. Una mejora obvia, necesaria para convertir este depurador 'de juguete' en un depurador real, sería escribir un intérprete capaz de general esta representación a partir de un programa lógico.

C.4. Un Depurador Declarativo Genérico

Para definir un depurador declarativo válido para cualquier paradigma y tipo de error definimos en primer lugar un módulo con algunas funciones básicas sobre árboles, que se puede ver en la definición del módulo *Examples*.

Con este módulo podemos definir el depurador genérico del módulo *Debugger*. El tipo *Debugger* allí definido tiene como parámetros los tres componentes de una instancia particular de depurador declarativo que describimos en 2.2: Una función para determinar la validez de un nodo, una función para extraer la causa del error asociada a un nodo crítico y un árbol de prueba. Con estos parámetros el depurador devolverá una lista de errores. El tipo $IO[Bug]$ denota que durante el proceso se utilizarán operaciones de entrada salida para la comunicación con el usuario (las preguntas para detectar nodos erróneos).

También incluye este módulo la función $allBuggy$, un depurador genérico que devuelve todos los nodos críticos de un árbol. De nuevo el planteamiento es poco realista, ya que para hacer esto hay que visitar (y preguntar al usuario) acerca de todos los nos nodos del árbol, pero sirve para nuestro proposito de ilustrar la estructura de un depurador declarativo.

C.5. Árboles de Prueba

El módulo *PtCat* incluye la definición del árbol de prueba positivo y negativo, llamados respectivamente *Pt* y *Cat* (de *Proof Tree* y *Computed Answers Tree*). Estos árboles son similares a los descritos en la Sección 2.1 pero adaptados al caso de la programación lógica sin restricciones: en lugar de considerar los vínculos de las variables como restricciones de igualdad, éstos se representan mediante sustituciones de variables por términos.

El árbol de prueba positivo corresponde con un árbol de prueba habitual en programación lógica. Cada nodo es una instancia de átomo que representamos como una pareja (átomo, substitución). Preferimos hacerlo así en lugar de tener el átomo con la sustitución ya aplicada para representar mejor la idea de que la sustitución es la respuesta obtenida en el cómputo examinado para el objetivo formado por el átomo. Suponemos que el objetivo inicial es un átomo, lo que se puede hacer sin pérdida de generalidad dado que siempre es posible incorporar un nuevo predicado *start* al programa definido con una única cláusula $init(\overline{X}) : -Goal$ donde *Goal* es el objetivo para el que se ha obtenido una respuesta incorrecta y $\overline{X}$ las variables de dicho objetivo.

Dado que para un mismo objetivo pueden existir diferentes árboles de prueba, uno por

cada respuesta computada, utilizaremos la función *ptList* que devuelve la lista completa de árboles de prueba para el átomo dado.

El árbol de prueba negativo, el CAT, asocia a cada átomo el conjunto formado por todas sus posibles respuestas, representado mediante una lista Haskell. La lista vacía representa que no existe ninguna solución para ese átomo.

C.6. Dos Instancias del Esquema

En el módulo *Instances* podemos ver dos instancias particulares del esquema general, la primera, llamada *wrong* utilizando los árboles positivos Pt y la segunda, llamada *misising* que utiliza los árboles negativos CAT.

El error que se obtendrá para el caso de los síntomas positivos serán cláusulas incorrectas y para señalar estas cláusulas se mostrará una instancia de ellas que, aún siendo válida, no puede ser probada con el programa. En el caso de los síntomas negativos se hallarán predicados con definición incompleta, y para representarlos se mostrarán átomos concretos que son válidos pero no han podido probarse con el programa (los llamados átomos no cubiertos).

La funciones que determinan la validez de los nodos lo hacen mediante preguntas directas al usuario. Nótese que nuestro depurador no tiene en cuenta las respuestas anteriores y no evita las preguntas repetidas.

C.7. Sesiones de Depuración

Utilizando los módulos anteriores podemos escribir una función *main* como la siguiente:

```
module Main where

import LogicProgramming
import PtCat
import Examples
import Debugger
import Instances

main = wrong (head (tail l))
        where
            l = ptList  graph
                          (Pred "camino" [Cons "b" [], Var "Y"])
```

En este módulo se depura la segunda respuesta (que corresponde al segundo árbol de prueba) para el objetivo $camino(b, Y)$. Esta es la sesión de depuración eliminando las preguntas repetidas (cuatro preguntas más):

```
Is Pred "camino" [Cons "b" [],Cons "b" []]  valid? n
Is Pred "arco" [Cons "b" [],Cons "c" []]  valid? y
Is Pred "camino" [Cons "b" [],Cons "c" []]  valid? y
....

The following incorrect clause instances have been found:
[Pred "camino" [Cons "b" [],Cons "b" []] :-
        [Pred "arco" [Cons "b" [],Cons "c" []],
         Pred "camino" [Cons "b" [],Cons "c" []]]]
```

La instancia de cláusula mostrada corresponde con la segunda cláusula de *camino* que es efectivamente la cláusula errónea en el ejemplo.

Dado que para el objetivo $camino(b, Y)$ obtenemos con el programa del ejemplo las respuestas $Y = c$, $Y = b$, pero nos falta la respuesta esperada $Y = d$, podemos escribir otra función *main* para utilizar el depurador de respuestas perdidas:

```
module Main where

import LogicProgramming
import PtCat
import Examples
import Debugger
import Instances

main = missing meow
        where
            meow = cat  graph
                          (Pred "camino" [Cons "b" [], Var "Y"])
```

Esta es la sesión de depuración (de nuevo sin las preguntas repetidas):

```
Is [ Pred "camino" [Cons "b" [],Cons "c" []],
      Pred "camino" [Cons "b" [],Cons "b" []] ]
complete for
Pred "camino" [Cons "b" [],Var "X"] ? n

Is [Pred "arco" [Cons "b" [],Cons "c" []]]
complete for
Pred "arco" [Cons "b" [],Var "Y$2"] ? y

Is [Pred "camino" [Cons "b" [],Cons "c" []]]
 complete for
```

```
Pred "camino" [Var "Y$4",Cons "c" []] ? n

Is [Pred "arco" [Cons "b" [],Cons "c" []]]
 complete for
Pred "arco" [Var "X$16",Cons "c" []] ? y

Is [ Pred "arco" [Cons "a" [],Cons "b" []],
     Pred "arco" [Cons "b" [],Cons "c" []] ]
complete for
Pred "arco" [Var "X$18",Var "Z$20"] ? n

Is []
 complete for
Pred "camino" [Cons "c" [],Cons "b" []] ? y

Is []
 complete for
Pred "camino" [Cons "c" [],Cons "c" []] ? y

Is []
 complete for
Pred "arco" [Cons "c" [],Var "Z$35"] ? n

Is []
 complete for
Pred "arco" [Cons "c" [], Cons "c"] ? y

Is []
 complete for
Pred "arco" [Cons "c" [],Cons "b" []] ? y

The following uncovered atoms have been found:
[Pred "arco" [Var "X$18",Var "Z$20"],
 Pred "arco" [Cons "c" [],Var "Z$35"],
 Pred "arco" [Cons "c" [],Var "Z$50"]]
```

Todos los resultados se refieren a la incompletitud del predicado *arco*. El primer átomo no cubierto no aporta ninguna información adicional, pero el segundo y el tercero sí concretan la información diciendo que falta un átomo de la forma $arco(c, Z)$ para algún valor Z.

Mientras que los árboles obtenidos para el caso de las respuestas incorrectas son análogos a los árboles POS presentados en la sección 2.1.5, pág. 13, los árboles CAT no son equivalentes a los árboles NEG presentados en esa misma sección. La diferencia principal es que en el caso de los árboles NEG cada nodo $C \wedge p(\overline{t_n}) \rightarrow \varphi$ de tipo (A) (aplicación de

las cláusulas del programa) tiene un hijo por cada una de las cláusulas del predicado p lo que permite al depurador mostrar no sólo que un predicado es incompleto, sino cual es la correspondiente fórmula de $IS(P)$ que no es válida en $\mathcal{I}$.

En cambio en los árboles CAT cada nodo tiene como hijo todos los átomos de las cláusulas que han sido utilizados durante el cómputo, pero sin información de a qué cláusula pertenece cada uno, y sin que ni siquiera aparezcan todos los átomos (si en el cuerpo $B_1, \ldots B_n$ algún B_i falla ya no aparecerán en el CAT los átomos B_j con $j > i$). Por ello en el caso del CAT se obtiene una información menos específica.

Módulo *LogicProgramming*

```
module LogicProgramming (
                        AtomInstance,
                        Term(Var,Cons),
                        Atom(Pred),
                        Subst,
                        Clause( (:-)),
                        Body,
                        ClauseInstance,
                        Answer,
                        State,
                        Program,
                        extract, -- initial state for the renaming
                        clauses, -- variants of clauses
                        (.<) -- apply substitution
                       ) where

-- auxiliary datatypes for representing Logic Programs
type Name       = String
type NextName   = Int
data Term       = Var Name | Cons Name [Term] deriving Show
data Atom       = Pred Name [Term] deriving Show
type Head       = Atom
type Body       = [Atom]
type Goal           = [Atom]
data Clause     = Head :- Body deriving Show
type Program        = [Clause]
type Subst      = Term -> Term
type Answer     = Subst
type AtomInstance   = (Atom,Subst)
type BodyInstance   = (Body,Subst)
type GoalInstance   = (Goal,Subst)
type ClauseInstance = (Clause,Subst)


-- Auxiliary functions:

-- substitution composition
(.>) :: Subst -> Subst -> Subst
s1 .> s2 = s2. s1

-- create a substitution of a variable by a term
```

```
createSubst :: Term -> Term ->  Subst
createSubst (Var name) term term2 =
         case term2 of
               Var nameVar      ->
                        if (nameVar== name)
                        then term
                        else Var nameVar
               Cons nameCons l  ->
                       Cons nameCons lSubst
                        where
                             lSubst =
                                  map (createSubst (Var name) term) l



-- Apply substitution to atoms
(.<) :: Atom -> Subst -> Atom
(Pred name terms) .< subst = Pred name (map subst terms)


-----------------------------------------------------------------
-- auxiliary datatype for representing the state
-- used during the renaming of clauses
type OldName = Name
type NewName = Name
type NextValue = Int
type Dict = (NextValue,[(OldName,NewName)])

-- the state monad used for renaming clauses
data State a = State (Dict -> (Dict,a))

instance Monad State  where
         return x = State (\state-> (state,x))
         s >>= f  = bind s f

bind :: (State a) -> (a -> State b) -> (State b)
bind (State st) f =
               State (\dict -> let
                        (newDict,y) = st dict
                        (State trans) = f y
                         in
                          trans newDict)

-- rename the variables of the term
```

```
freshTerm :: Term -> State Term
freshTerm (Var name) = State (rename (Var name))
freshTerm (Cons name l) =
         do
             freshL <- mapM freshTerm l
             return (Cons name freshL)

-- rename the variables of an atom
freshAtom :: Atom -> State Atom
freshAtom (Pred name l) =
         do
             freshL <- mapM freshTerm l
             return (Pred name freshL)


-- resetDict
-- the dictionary must be initalizated
-- for each new clause
resetDict :: State a -> State a
resetDict (State st) =
           State (\dict -> let ( (n,l), y) =
                     st dict
                     in ( (n,[]),y) )


-- rename a term.
--  If some variable is already renamed use this name
--  otherwise produce a new name

rename :: Term -> (Dict -> (Dict, Term))
rename (Var name) dict =
    if (elem name alreadyRenamed)
    then (dict,Var dictName)
    else (newDict, Var newName)
    where
              (n,list) = dict
              alreadyRenamed = map fst list
              newDict = (n+1,(name,newName):list)
              newName = name++"$"++(show n)
              dictName = mylookup name list
```

```
-- look for the value already associated to v in the dict.
mylookup :: OldName -> [(OldName,NewName)] -> NewName
mylookup oldName ((old,new):ls) =
     if (old == oldName) then new
    else mylookup oldName ls

-- renaming of a given clause
freshVariant :: Clause -> State Clause
freshVariant (head :- body)  =
         resetDict (
         do
            freshHead <- freshAtom head
            freshBody <- mapM freshAtom body
            return (freshHead :- freshBody))

-- initial state when renaming:
-- value 0 and empty list of var. names
extract :: State a -> a
extract (State st) = snd (st (1,[]))

-----------------------------------------------------------------

-- the m.g.u.

-- the following datatype represent a equation
--  equations in solved form are distinguished
data Equation = Term :=: Term

mgu:: Atom -> Atom -> Maybe Subst
mgu (Pred name1 l1) (Pred name2 l2) =
   if (name1==name2  &&  length l1==length l2 )
   then martelliMontanari (zipWith (:=:) l1 l2)
   else Nothing
-----------------------------------------------------------------
-- Martelli & Montanari algorithm
martelliMontanari:: [Equation] -> Maybe Subst

martelliMontanari [ ] = Just id

martelliMontanari ( (Var name :=: term ) : ecs) =
        if occursCheck name term
        then Nothing
        else
```

```
           case martelliMontanari (map (applySubst subst) ecs) of
               Nothing -> Nothing
              Just unifier -> Just (subst .> unifier)
            where
               subst = createSubst (Var name) term
               applySubst s (e1 :=: e2) = (s e1) :=: (s e2)

martelliMontanari ( (term :=: Var name  ) : ecs) =
         martelliMontanari ((Var name :=: term) : ecs)

martelliMontanari ( (Cons name1 l1 :=: Cons name2 l2  ) : ecs) =
         if (name1==name2  &&  length l1==length l2 )
         then martelliMontanari ((zipWith (:=:) l1 l2)++ecs)
        else Nothing


-- occurs_check
occursCheck :: Name -> Term -> Bool
occursCheck nameVar (Var name) = name==nameVar

occursCheck nameVar (Cons nameCons l) =
         any (occursCheck nameVar) l


-- return variants of the instances of all the program clauses
-- with heads matching an atom instance
clauses ::
  Program -> AtomInstance -> State [ClauseInstance]
clauses program atomInst=
         do
            freshProgram <- mapM freshVariant program
            clauseInstances <-
                  matchingClauses freshProgram  atomInst
            return clauseInstances


matchingClauses::
    Program -> AtomInstance -> State [ClauseInstance]
matchingClauses [] atomInst = return []
matchingClauses ( (h:-b) :restProg)  (atom, subst) =
         do
             restCl <- matchingClauses restProg (atom,subst)
             case mgu (atom .< subst) h of
```

```
 Nothing -> return restCl
Just unifier ->
        return (( h:-b, subst .> unifier) : restCl)
```

Módulo *Examples*

```
-- logic program examples
module Examples ( graph) where

import LogicProgramming

graph :: Program
graph = [cl1,cl2,cl3,cl4]
         where
              cl1 = Pred "arco" [Cons "a" [], Cons "b" []] :- []
              cl2 = Pred "arco" [Cons "b" [], Cons "c" []] :- []
              cl3 = Pred "camino" [Var "X", Var "Y"] :-
                                      [Pred "arco" [Var "X", Var "Y"]]
              cl4 = Pred "camino" [Var "X", Var "Y"] :-
                                     [Pred "arco" [Var "X", Var "Z"],
                                      Pred "camino" [Var "Y", Var "Z"]]
```

Módulo *Trees*

```
module Trees(Tree, root, forest,mapTree, buildTree) where

-- general computation tree
data Tree node = Root node [Tree node] deriving Show

-- auxiliary functions for manipulating trees
root :: Tree a -> a
root (Root x xs) = x

forest :: Tree a -> [Tree a]
forest (Root x xs) = xs

mapTree:: (a->b) -> Tree a -> Tree b
mapTree f t = Root (f (root t))
                   (map (mapTree f) (forest t))

buildTree :: a -> [Tree a] -> Tree a
buildTree = Root
```

Módulo *Debugger*

```
module Debugger(Debugger, allBuggy ) where

import Trees

-----------------------------------------------------------
-- The parameters neccessary to define any instance

-- function to determine erroneous nodes
type IsErroneous node = node -> IO Bool

-- Extracting the bug from a buggy node
type Extract node bug = (Tree node) -> bug

-- The type of general declarative debuggers

type Debugger node bug =
        (IsErroneous node) -> (Extract node bug) ->
        (Tree node) -> IO [bug]

-----------------------------------------------------------
-- Finding all the buggy nodes
allBuggy :: Debugger node bug
allBuggy erroneous extract tree =
         do
             nonValidRoot <- erroneous treeRoot
             nonValidChildren <- mapM erroneous treeChildren
             childrenBugs <- mapM (allBuggy erroneous extract)
                                              treeForest
             return
                   (if nonValidRoot && (all  not nonValidChildren)
                    then  (extract tree):(concat childrenBugs)
                    else (concat childrenBugs))
             where
                  treeRoot        = root tree
                  treeForest     = forest tree
                  treeChildren  = map root treeForest
```

Módulo *PtCat*

```
-- Positive and Negative proof trees
module PtCat where

import LogicProgramming
import Trees
import Debugger


-- auxiliary datatypes for representing the threes
type Pt = Tree AtomInstance
type Cat    = Tree (AtomInstance, [Answer])


ptRootAtom :: Pt -> Atom
ptRootAtom = fst . root

ptRootSubst :: Pt -> Subst
ptRootSubst = snd . root

ptChildrenAtoms::  Pt -> [Atom]
ptChildrenAtoms = (map (fst.root)).forest

catRootAtom:: Cat -> AtomInstance
catRootAtom = fst . root

catRootAnswers:: Cat -> [Answer]
catRootAnswers = snd . root



-- generating the proof trees
ptList ::  Program -> Atom -> [Pt]
ptList p a =
         listPts
             where
                listAtomTrees = extract (pt p (a,id))
                listPts              = map applySubst listAtomTrees
                applySubst (tree,s) = mapTree (\a -> (a,s)) tree

pt :: Program -> AtomInstance ->
                          State [(Tree Atom,Subst)]
```

```
pt p (a,s) =
         do
             cl <- clauses p (a,s)
             l  <- mapM (ptClause p a) cl
             return (concat l)

-- proof trees obtained using a particular clause
ptClause :: Program -> Atom -> ClauseInstance ->
                                    State [  (Tree Atom, Subst) ]
ptClause p  a (h:-b,sAux) =
         do
             lpts <- ptBody p b sAux
             return (map buildFromChildren lpts)
         where
              buildFromChildren (child,subst) =
                               (buildTree a child, subst)


ptBody :: Program -> Body -> Subst ->
                                  State [ ([Tree Atom], Subst) ]
ptBody p [ ] subst = return [ ([ ],subst)]
ptBody p (x:xs) subst =
         do
             ptFirst <- pt p (x,subst)
             r  <- mapM (restBody p xs) ptFirst
             return (concat r)

restBody :: Program -> Body -> (Tree Atom,Subst) ->
                                    State [ ([Tree Atom], Subst) ]
restBody  p  xs (tree, subst) =
         do
             rest <- ptBody p xs subst
             return (map insertTree rest)
         where
              insertTree (l,s) = (tree:l,s)

-----------------------------------------------------------------------

-- generating the computed answer trees

cat :: Program -> Atom -> Cat
cat p a = extract (instanceCat p (a,id))
```

```
instanceCat :: Program ->AtomInstance -> State Cat
instanceCat p (a,s) =
         do
             cl <- clauses p (a,s)
             l  <- mapM (catClause p (a,s)) cl
             return (buildTree ((a,s), collectAnswers l)
                                        (collectChild l))

collectAnswers :: [Cat]  -> [Answer]
collectAnswers t = concat (map (snd.root) t)

collectChild :: [Cat]    -> [Cat]
collectChild = concat .(map forest)

catClause :: Program ->  AtomInstance -> ClauseInstance ->
                                       State Cat
catClause p atInst (h:-[],s) = return (buildTree (atInst, [s]) [])
catClause p atInst (h:-(x:xs),s) =
         do
             lc <- catBody p (x:xs) s
             return
                   (buildTree (atInst, catRootAnswers (last lc)) lc)

catBody ::  Program ->  Body  -> Subst -> State [Cat]
catBody p [] s      = return [ ]
catBody p (x:xs) s  =
         do
             firstCat   <- instanceCat p (x,s)
             rest  <- mapM (catBody p xs) (catRootAnswers firstCat)
             return (firstCat:(concat rest))
```

Módulo *Instances*

```
module Instances(wrong,missing) where

import LogicProgramming
import Debugger
import PtCat

wrong :: Pt -> IO ()
wrong  tree =
         do
            bugs <- allBuggy erroneous extract tree
            displayIncorrectClauses bugs
         where
               erroneous (atom,subst) =
                         do
                             isValid <- valid (atom .< subst)
                             return (not isValid)
                extract pt = (ptRootAtom pt :- ptChildrenAtoms pt,
                                                    ptRootSubst pt)

displayIncorrectClauses :: [ClauseInstance] -> IO ()
displayIncorrectClauses bugs =
         do
             putStrLn ""
             putStr "The following incorrect clause instances "
             putStrLn "have been found:"
             putStrLn (show (map applyClause bugs))
          where
              applyClause ((h:-b),s) =
                                h .<s :-  map (\a -> (a.<s)) b

valid :: Atom -> IO Bool
valid a =
         do
            putStr("Is ")
            putStr (show a)
            putStr  "  valid? "
            ans <- getLine
            return (ans=="y" || ans=="Y")
```

```
missing :: Cat -> IO ()
missing  tree =
         do
            bugs <- allBuggy erroneous extract tree
            displayUncoveredAtoms bugs
         where
            erroneous ((atom,subst), answers) =
                  do
                     isComplete <- complete (atom .< subst) answers
                     return (not isComplete)
            extract  = catRootAtom

complete :: Atom -> [Answer] -> IO Bool
complete a  ans =
         do
            putStrLn ("Is "++ show answers)
            putStrLn  " complete for "
            putStr (show a)
            putStr  "  ?"
            ans <- getLine
            return (ans=="y" || ans=="Y")
        where
            answers = map (a .<) ans

displayUncoveredAtoms::[AtomInstance] -> IO()
displayUncoveredAtoms bugs =
         do
           putStrLn ""
           putStrLn "The following uncovered atoms have been found: "
           putStrLn (show  (map applyInstance bugs))
         where
           applyInstance (a,s) =  a.<s
```

Bibliografía

[1] M. Abengózar Carneros, P. Arenas Sánchez, R. Caballero, A. Gil Luezas, J.C. González Moreno, J. Leach Albert, F.J. López Fraguas, N. Martí Oliet, J. M. Molina-Bravo, E. Pimentel Sánchez, M. Rodríguez Artalejo, María del Mar Roldán García, J.J. Ruz Ortiz y J. Sánchez Hernández. $\mathcal{TOY}$: *A Multiparadigm Declarative Language. Version 1.0.* Departamento de Sistemas Informáticos y Programación, Universidad Complutense de Madrid, Tech. Report SIP-119/00, February 2002. URL: `http://titan.sip.ucm.es/toy/report.pdf`

[2] E. Albert, M. Hanus y G. Vidal. *A Practical Partial Evaluator for a Multi-Paradigm Declarative Language* En Proc. FLOPS'01, Springer LNCS 2024, 2001.

[3] M. Alpuente, D. Ballis, F. J. Correa y M. Falaschi *Automated Correction of Functional Logic Programs*. En P. Degano, editor, Proc. of European Symposium on Programming (ESOP 2003), Warsaw (Poland), Springer LNCS, Springer-Verlag, Berlin, 2003.

[4] M. Alpuente, F.J. Correa, y M. Falaschi. *A Debugging Scheme for Funcional Logic Programs*. Electronic Notes in Theoretical Computer Science. Vol. 64. Elsevier, 2002.

[5] S. Antoy. *Constructor-based Conditional Narrowing*. En Proc. PPDP'01, ACM Press 2001, 199–206.

[6] S. Antoy, R. Echahed y M. Hanus. *A Needed Narrowing Strategy*. Journal of the ACM Vol. 47, no. 4, 776–822, July 2000.

[7] S. Antoy y M. Hanus. *Functional Logic Design Patterns*. En Proc. FLOPS'02, Springer LNCS 2441:67–87, 2002.

[8] K.R. Apt. *Introduction to Logic Programming*. Handbook of Theoretical Computer Science. Volumen B: Formal Models and Semantics, 495–574. Editor: J. van Leeuwen. Elsevier, Amsterdam & The MIT Press, Cambridge, 1990.

[9] R. Bird, *Introduction to Functional Programming using Haskell*, 2ª edición, Prentice Hall Europe, 1998.

[10] P. Bosco, G. Giovannetti and C. Moiso, *Narrowing vs. SLD-Resolution*, Theoretical Computer Science 59, 3–23, 1988.

[11] Johan Boye, Wlodek Drabent, Jan Maluszyński. *Declarative Diagnosis of Constraint Programs: an assertion-based approach*. Proc. of the 3rd. Int'l Workshop on Automated Debugging-AADEBUG'97, U. of Linkoping Press, 123–141, Linkoping, Sweden, May 1997.

[12] R. Caballero y F.J. López-Fraguas. *A Functional-Logic Perspective of Parsing*. En Proc. FLOPS'99, Springer LNCS 1722:85–99, 1999.

[13] R. Caballero y F.J. López-Fraguas. *Extensions: A Technique for Structuring Functional Logic Programs*. En Proc. PSI'99, Springer LNCS 1755:297–310, 2000.

[14] R. Caballero y F.J. López-Fraguas. *Dynamic Cut with Definitional Trees*. En Proc. FLOPS'02, Springer LNCS 2441:245–258, 2002.

[15] R. Caballero y F.J. López-Fraguas. *Improving Deterministic Computations in Lazy Functional Logic Languages* . The Journal of Functional and Logic Programming, Special Issue on the Sixth International Symposium on Functional and Logic Programming. 23 páginas. EAPLS, 2003.

[16] R. Caballero, F.J. López-Fraguas y M. Rodríguez-Artalejo. *A functional specification of declarative debugging for logic programming* En Proc. of the 8th International Workshop on Functional and Logic Programming. Tecnical Report RR 1021-I. Université Joseph Fourier. Grenoble, 1999.

[17] R. Caballero, F.J. López-Fraguas y M. Rodríguez-Artalejo. *Theoretical Foundations for the Declarative Debugging of Lazy Functional Logic Programs*. En Proc. FLOPS'01, Springer LNCS 2024:170–184, 2001.

[18] R. Caballero y W. Lux. *Declarative Debugging of Encapsulated Search*. Electronic Notes in Theoretical Computer Science. Vol. 76. Elsevier, 2002.

[19] R. Caballero y M. Rodríguez Artalejo. *A Declarative Debugging System for Lazy Functional Logic Programs*. Electronic Notes in Theoretical Computer Science. Vol. 64. Elsevier, 2002.

[20] R. Caballero y M. Rodríguez Artalejo. *DDT: A Declarative Debugging Tool for Functional-Logic Languages*. En Proc. FLOPS'04, Springer LNCS, Vol. 2998, 2004.

[21] M. Cameron, M. García de la Banda, K. Marriott y P. Moulder. *ViMer: A Visual Debugger for Mercury*. In Proc. PPDP03, ACM Press, 56–66, 2003.

[22] O. Chitil, C. Runciman, y M. Wallace. *Freja, Hat and Hood – A comparative Evaluation of Three Systems for Tracing and Debugging Lazy Functional Programs*. En M. Mohnen y P. Koopman, editores, Selected Papers of the 12th International Workshop on Implementation of Functional Languages, Springer LNCS 2011:176–193, 2001.

[23] The CLIP Group. *Program Assertions. The CIAO System Documentation Series*. TR CLIP4/97.1, Facultad de Informática, UPM, August 1997.

[24] W.F. Clocksin y C.S. Mellish. *Programación en Prolog*. Edición española: Editorial Gustavo Gili S.A. Barcelona 1987.

[25] M. Comini, G. Levi, M.C. Meo y G. Vitello. *Abstract Diagnosis*. J. of Logic Programming 39, 43–93, 1999.

[26] P. Cousot y R. Cousot. *Abstract Interpretation: a Unified Lattice Model for Static Analysis of Programs by Construction or Approximation of Fixpoints*. Fourth ACM Symposium on Principles of Programming Languages, 238–252, 1977.

[27] P. Cousot y R. Cousot. *Abstract Interpretation and Applications to Logic Programs*. Journal of Logic Programming 13(2 & 3):103–179, 1992.

[28] L. Damas y R. Milner. *Principal Type Schemes for Functional Programs*. Proc. ACM Symp. on Principles of Programming Languages (POPL'82), ACM Press, 207–212, 1982.

[29] N. Dershowitz and J.P. Jouannaud, *Rewrite Systems*. Handbook of Theoretical Computer Science, volumen B: Formal Models and Semantics, 243–320. Elsevier, Amsterdam, 1991.

[30] W. Drabent, S. Nadjm-Tehrani y J. Maluszynski. *Algorithmic debugging with assertions*. Meta-Programming in Logic Programming. Eds.:H. Abramson, M.H. Rogers. The MIT Press, Cambridge, 383–398, 1989.

[31] M. Falaschi, G. Levi, M. Martelli y C. Palamidessi. *A Model-theoretic Reconstruction of the Operational Semantics of Logic Programs*. Information and Computation 102(1):86–113, 1993.

[32] G. Ferrand. *Error Diagnosis in Logic Programming, an Adaptation of E.Y. Shapiro's Method*. The Journal of Logic Programming 4(3):177–198, 1987.

[33] A. Gill. *Debugging Haskell by observing intermediate data structures*. En Proceedings of the 4th Haskell Workshop, 2000. T.R. University of Nottingham.

[34] E. Giovannetti, G. Levi, C. Moiso y C.Palamidessi. *Kernel-LEAF: A Logic plus Functional Language*. Journal of Computer and System Science 42(2):139–185, 1991.

[35] J.C. González-Moreno, M.T. Hortalá-González, F.J. López-Fraguas y M. Rodríguez-Artalejo. *An Approach to Declarative Programming Based on a Rewriting Logic*. The Journal of Logic Programming 40(1):47–87, 1999.

[36] J.C. González-Moreno, M.T. Hortalá-González y M. Rodríguez-Artalejo. *Polymorphic Types in Functional Logic Programming*. FLOPS'99 special issue of the Journal of Functional and Logic Programming, 2001. URL:
`http://danae.uni-muenster.de/lehre/kuchen/JFLP`

[37] C.A. Gunter y D. Scott. *Semantic Domains*. En J.van Leeuwen (ed.), *Handbook of Theoretical Computer Science*, Elsevier y The MIT Press, Vol. B, Chapter 6, 633–674, 1990.

[38] M. Hanus. *The Integration of Functions into Logic Programming: A Survey*. J. of Logic Programming 19-20. Special issue "*Ten Years of Logic Programming*", 583–628, 1994.

[39] M. Hanus (ed.). *Curry: An Integrated Functional Logic Language*. Version 0.8, 2003. URL: `http://www.informatik.uni-kiel.de/ mh/curry/report.html`.

[40] M. Hanus. *High-Level Server Side Web Scripting in Curry*. Proc. PADL'01, Springer LNCS 1955:76–92, 2001.

[41] M. Hanus y J. Koj. *An Integrated Develepment Environment for Declarative Multi-Paradigm Programming*. A. Kusalik Ed., proceedings of the Eleventh Workshop on Logic Programming Environments, 2001.

[42] M. Hanus y F. Steiner. *Controlling Search in Declarative Programs*. En Proc. PLILP/ALP'98, Springer LNCS 1490:374–390, 1998.

[43] M. Hermenegildo, G. Puebla y F. Bueno. *Using Global Analysis, Partial Specifications, and an Extensible Assertion Language for Program Validation and Debugging*. TR. CLIP8/98.0. Facultad de Informática. Universidad Politécnica de Madrid, 1998.

[44] J. Jaffar y J.L. Lassez. *Constraint Logic Programming*. En Proc. ACM Symp. on Principles of Programming Languages (POPL'87), ACM Press, 111–119, 1987.

y quizá también:

[45] J. Jaffar y M.J. Maher. *Constraint Logic Programming: A Survey*. En Journal of Logic Programming 19&20, 503–581, 1994.

[46] J. Jaffar, M.J. Maher, K.Marriot, y P. J. Stuckey. *Semantics of Constraint Logic Programs* Journal of Logic Programming, 37(1-3):1–46, 1998.

[47] Java. Página WEB `http://java.com`.

[48] T. Johnsson. *Lambda lifting: Transforming programs to recursive equations.* En Procs. Functional Programming Languages and Computer Architecture, Springer LNCS 201:190–203, 1985.

[49] Claude Laï. *Assertions with Constraints for CLP Debugging.* En P. Derasant, M. Hermenegildo and J. Maluszynski (eds.) Analysis and Visualization Tools for Constraint Programming, Capítulo 3, 109–120, Springer LNCS 1870, 2000.

[50] J. Launchbury *Lazy imperative programming.* En Procs. ACM Sigplan Workshop on State in Programming Languages, 1993. YALE/DCS/RR-968, Yale University.

[51] J.W. Lloyd *Foundations of Logic Programming.* Springer Verlag, 1984.

[52] J.W. Lloyd. *Declarative Error Diagnosis.* New Generation Computing 5(2):133–154, 1987.

[53] R. Loogen, F.J. López-Fraguas y M. Rodríguez-Artalejo. *A Demand Driven Computation Strategy for Lazy Narrowing.* Int. Symp. on Programming Language Implementation and Logic Programming (PLILP'93), LNCS 714, Springer Verlag 1993, 184–200.

[54] F.J. López-Fraguas, y J. Sánchez-Hernández. *$\mathcal{TOY}$ a Multiparadigm Declarative System*, En Proc. RTA'99, Springer LNCS 1631:244–247, 1999.

[55] W. Lux and H. Kuchen. *An efficient Abstract Machine for Curry* . En Kurt Beiersdörfer, Gregor Engels and Wilhem Schäfer (ed.) Informatik '99 - GI Jahrestatung, Informatik Aktuell, 390–399, Springer, 1999.

[56] K. Marriot, P. Stuckey. *Programming with Constraints: An Introduction.* Mit Press, Cambridge, MA. 1998.

[57] A. Martelli y M. Montanari. *An efficient unification algorithm.* ACM Transactions on Programming Languages and Systems, 4(2):258–282, 1982.

[58] R. Milner. *A Theory of Type Polymorphism in Programming.* Journal of Computer and Systems Sciences, 17, 348–375, 1978.

[59] B. Möller. *On the Algebraic Specification of Infinite Objects - Ordered and Continuous Models of Algebraic Types*. Acta Informatica 22, 537–578, 1985.

[60] J.J. Moreno-Navarro y M. Rodríguez-Artalejo. *Logic Programming with Functions and Predicates: The Language BABEL* . L. Logic Programming 12: 191–223, 1992.

[61] L. Naish. *Adding equations to NU-Prolog*. Proc. of the Third Interantional Symposium on Programming Language Implementation and Logic Programming, 15–26, August 1991.

[62] L. Naish. *Declarative Debugging of Lazy Functional Programs*. T.R. 92/6. Department of Computer Science, University of Melbourne, 1992.

[63] L. Naish. *A Declarative Debugging Scheme*. Journal of Functional and Logic Programming, 1997-3.

[64] L. Naish y T. Barbour. *A Declarative Debugger for a logical-functional language*. En Graham Forsyth y Moonis Ali (eds.), Eight International Conferencie on Industrial and Engineering Applications of Artificial Intelligence and Expert System - Invited and Additional Papers, Volume 2:91–99, Melbourne, June 1995. DSTO General Document 51.

[65] L. Naish y T. Barbour. *Towards a Portable Lazy Functional Declarative Debugger*. Australian Computer Science Communications, 18(1):401–408, 1996.

[66] L. Naish, P.W. Dart, y J. Zobel. *The NU-Prolog debugging environment*. Proceedings of the Sixth International Conference on Logic Programming. Ed.:Antonio Porto, Cambridge, MA, June 1989. The MIT Press.

[67] H. Nilsson. *Freja: A small non-strict, purely functional language*. MSc dissertation, Department of Computer Science and Applied Mathematics, Aston University, Birmingham, U.K.,1991.

[68] H. Nilsson. *How to look busy while being lazy as ever: The implementation of a lazy functional debugger*. Journal of Functional Programming 11(6):629–671.

[69] H. Nilsson, P. Fritzson. *Algorithmic Debugging for Lazy Funcional Languages*. Proceedings of the Fourth International Symposium on Programming Language Implementation and Logic Programming. 1992.

[70] H. Nilsson, P. Fritzson. *Algorithmic Debugging for Lazy Funcional Languages*. Journal of Functional Programming, 4(3):337–370, 1994.

[71] H. Nilsson y J. Sparud. *The Evaluation Dependence Tree as a basis for Lazy Functional Debugging*. Automated Software Engineering, 4(2):121–150, 1997.

[72] C. Pareja, R. Peña, F. Rubio y C. Segura *Adding Traces to a Lazy Monadic Evaluator*. En Selected Papers of the Eighth International Conference on Computer Aided Systems Theory, EUROCAST'01 Springer LNCS 2178, 2002.

[73] W. Partain. *The nofib Benchmark Suite of Haskell Programs*. En J. Launchbury y PM. Sansom, eds., Workshops in Computing. Springer Verlag, 1993. Se puede acceder a la versión actualizada del conjunto de programas *nofib* desde la URL `http://cvs.haskell.org/cgi-bin/cvsweb.cgi/fptools/nofib`.

[74] L.C. Paulson. *ML for the Working Programmer*. 2ª edición,Cambridge University Press, 1996.

[75] S.L. Peyton Jones (ed.), J. Hughes (ed.), L. Augustsson, D. Barton, B. Boutel, W. Burton, J. Fasel, K. Hammond, R. Hinze, P. Hudak, T. Johnsson, M.P. Jones, J. Launchbury, E. Meijer, J. Peterson, A. Reid, C. Runciman y P. Wadler. *Report on the programming language Haskell 98: a non-strict, purely functional language*. Disponible en `http://www.haskell.org/onlinereport/`, 2002.

[76] B. Pope y L. Naish, *A Program Transformation for Debugging Haskell 98*, En Michael Oudshoorn Ed., Twenty Sixth Australasian Computer Science Conference, Vol. 16:227–236, 2003.

[77] B. Pope y L. Naish, *Practical Aspects of Declarative Debugging in Haskell 98*, In Proc. PPDP03, ACM Press, 230–240, 2003.

[78] B. Pope. *Buddha. A Declarative Debugger for Haskell.* Honours Thesis, Department of Computer Science, University of Melbourne, Australia, Junio 1998.

[79] G.Puebla, F. Bueno and M. Hermenegildo. *An Assertion Language for Constraint Logic Programs* En P. Derasant, M. Hermenegildo and J. Maluszynski (eds.) Analysis and Visualization Tools for Constraint Programming, Capítulo 1, 23–61, Springer LNCS 1870, 2000.

[80] G.Puebla, F. Bueno and M. Hermenegildo. *An Generic Preprocessor for Program Validation and Debugging.* En P. Derasant, M. Hermenegildo and J. Maluszynski (eds.) Analysis and Visualization Tools for Constraint Programming, Capítulo 2, 63–107, Springer LNCS 1870, 2000.

[81] J.A. Robinson y E.E. Sibert. *LOGLISP: Motivation, Design and Implementation.* En K.L. Clark and S.A. Tärnlund (eds.), *Logic Programming*, Academic Press, 299–313, 1982.

[82] E.Y.Shapiro. *Algorithmic Program Debugging.* The MIT Press, Cambridge, Mass., 1982.

[83] SICStus Prolog. Página WEB: `http://www.sics.se/sicstus/`.

[84] J. Sparud. *Tracing and Debuggging Lazy Functional Computations.* PhD Thesis. Department of Computer Science, Chalmers Universitity of Technology. Göteborg, Suecia, 1999.

[85] J. Sparud y H. Nilsson. *The Architecture of a Debugger for Lazy Functional Languages.* En Miraille Ducassé Ed., Proceedings of AADEBUG'95, Saint Malo, Francia, 1995.

[86] J. Sparud y C. Ruciman. *Tracing lazy functional computations using redex trails.* En H. Graser, P. Hartel, y H. Kuchen, editors. Proc. PLILP'97, Springer LNCS 1292:291–308, 1997.

[87] L. Sterling y E.Y. Shapiro. *The Art of Prolog.* The MIT Press, 1986.

[88] A. Tessier y G. Ferrand. *Declarative Diagnosis in the CLP Scheme*. En P. Deransart, M. Hermenegildo y J. Małuszynski (Eds.), Analysis and Visualization Tools for Constraint Programming, Chapter 5, 151–174. Springer LNCS 1870, 2000.

[89] A. Tolmach y A. W. Appel *A Debugger for Standard ML*. Journal of Functional Programming 5(2):155–200, 1995

[90] A.D. Turner. *Miranda: A non-strict functional language with polymorphic types*. En: Proceedings IFIP International Conference on Functional Programming Languages and Computer Architecture, FPCA'85, Springer LNCS 201, 1985.

[91] R. del Vado Vírseda. *A Demand-driven Narrowing Calculus with Overlapping Definitional Trees*. Proc. ACM SIGPLAN Conf. on Principles and Practice on Declarative Programming (PPDP'03), ACM Press, 213–227, 2003.

[92] P. Wadler. *Monads for Functional Programming*. En J. Jeuring y E. Meijer, Eds. Springer LNCS 925. 1995. Notices 33(8), 23–27, 1998.

[93] P. Wadler. *Why no one uses Functional Languages*. SIGPLAN Notices 33(8), 23–27, 1998.