Depuración Declarativa de Programas Lógico Funcionales

Francisco José Correa Zabala
Departamento de Sistemas Informáticos y Computación
Universidad Politécnica de Valencia

Memoria presentada para optar al título de:
Doctor en Informática

Dirigida por:
María Alpuente Frasnedo

Tribunal de lectura:

Presidente:	**Isidro Ramos Salavert**	**U.P. Valencia**
Vocales:	**Moreno Falaschi**	**U. Udine**
	Ginés Moreno Valverde	**U. Castilla la Mancha**
	Ernesto Pimentel Sánchez	**U.C. Malaga**
Secretario :	**Salvador Lucas Alba**	**U.P. Valencia**

Julio de 2002

Resumen

Los lenguajes de programación lógico funcionales integran algunas de las mejores características de los paradigmas declarativos clásicos, en concreto, de la programación lógica y la programación funcional. Cada uno de estos estilos tiene diferentes ventajas con respecto a sus aplicaciones prácticas. Los lenguajes funcionales proporcionan facilidades de abstracción sofisticadas, sistemas de módulos y soluciones "puras" para la integración de facilidades de I/O en la programación declarativa, además de técnicas y estrategias eficientes para la ejecución de los programas. Los lenguajes lógicos permiten la computación con información parcial y están provistos de facilidades de búsqueda de soluciones. Sin embargo, recientemente se ha demostrado que las ventajas de estos estilos pueden combinarse de manera efectiva y útil sobre un lenguaje único. Los lenguajes lógico funcionales modernos ofrecen características de ambos estilos. La semántica operacional de los lenguajes integrados está usualmente basada en *narrowing*, una combinación de la unificación y reducción como mecanismo de evaluación que subsume a la reescritura y a la SLD-resolución.

La depuración de programas lógico funcionales es un importante problema práctico que ha sido escasamente tratado en la literatura. La depuración se puede enfocar desde el punto de vista declarativo y desde el punto de vista procedural. La principal contribución de esta tesis es el desarrollo de métodos de diagnóstico declarativo para la depuración de programas lógico funcionales con respecto al observable de respuestas computadas. Las condiciones impuestas a los programas que consideramos nos permiten definir un marco genérico para la depuración declarativa que es paramétrico con respecto a la estrategia de *narrowing*. En particular nuestro esquema se aplica tanto al *narrowing* impaciente (*llamada por valor*) como al *narrowing* perezoso (*llamada por nombre*). Primero asociamos a un programa lógico funcional, $\mathcal{R}$, un operador (continuo) de consecuencias inmediatas, $T_{\mathcal{R}}^{\varphi}$, el cual es paramétrico con respecto a la estrategia φ de *narrowing* que puede ser tanto perezosa como voraz. Utilizamos el menor punto fijo de este operador para definir la semántica del programa. Demostramos que tal semántica tiene la propiedad de que podemos obtener las respuestas computadas para un objetivo g, con la estrategia φ de *narrowing*, por unificación sintáctica con las ecuaciones de la semántica. Construimos la semántica $\mathcal{O}^{\varphi}(\mathcal{R})$ y mostramos

su correspondencia con la semántica de punto fijo. Entonces mostramos que, dada una especificación deseada $\mathcal{I}$ de un programa $\mathcal{R}$, podemos determinar los errores de corrección y completitud del programa $\mathcal{R}$ por un único paso de este operador. A continuación presentamos una técnica de aproximación de la semántica deseada del conjunto de éxitos. Usamos los conceptos de sobreespecificación $\mathcal{I}^+$ y subespecificación $\mathcal{I}^-$ para aproximar correctamente por exceso (resp. por defecto) la semántica deseada. Al comparar uno de estos conjuntos con el resultado de aplicar el operador de consecuencias inmediatas al otro y, mediante un simple test estático, podemos determinar cuándo alguna de las cláusulas es incorrecta y cuándo existen ecuaciones no cubiertas. Finalmente hemos desarrollado el sistema experimental "BUGGY", que permite realizar la depuración de programas lógico funcionales respecto a la semántica de respuestas computadas por las estrategias de *narrowing* básico, innermost, outermost y necesario. En el sistema presentamos, como complemento natural de la herramienta de diagnóstico, un módulo para la corrección automática de reglas, que utiliza los métodos de inducción de programas lógico funcionales basados en transformaciones de plegado/desplegado.

Abstract

Functional logic programming combines languages which integrate some of the best features of the classical declarative paradigms, namely functional and logic programming. However, each of these programming styles has different advantages w.r.t. practical applications. Functional languages provide sophisticated abstraction facilities, module systems and clean solutions for integrating I/O into declarative programming as well as for techniques and strategies for efficient program execution. Logic languages allow for computing with partial information and provide built-in search facilities. However, recent results show that the advantages of these styles can be efficiently and usefully combined into a single language. Modern functional logic languages offer features from both styles. The operational semantics of integrated languages is usually based on *narrowing*, a combination of unification for parameter passing and reduction as evaluation mechanism which subsumes rewriting and SLD-resolution.

How to debug functional logic programs is an important practical problem which has hardly been addressed in the previous literature. The debugging can be formulated in a declarative or procedural way. The main contribution of this thesis is the development of a declarative diagnosis method w.r.t. computed answers for functional logic programs. The conditions which we impose on the programs which we consider allow us to define a generic framework for declarative debugging which is parametric w.r.t. narrowing strategy. In particular, it both works for eager (*call–by–value*) narrowing as well as for lazy (*call–by–name*) narrowing. We associate a (continuous) immediate consequence operator to our programs which is parametric w.r.t. narrowing strategy φ. We prove that, the answer substitutions computed by narrowing w.r.t. φ can be obtained by standard unification with the equations in the (fixpoint) semantics. We construct the operational semantics, $\mathcal{O}^{\varphi}(\mathcal{R})$, and then we establish the relation between the fixpoint semantics and this operational semantics. Then we show that, given the intended specification of a program $\mathcal{I}$, it is possible to check the correctness and completeness of $\mathcal{R}$ by a single step of the immediate consequence operator. Then we present an efficient methodology which is based on abstract interpretation. We proceed by approximating the intended specification of the success set. We use *over* and *under* specifications $\mathcal{I}^{+}$ and $\mathcal{I}^{-}$ to correctly over- (resp. under-)

approximate the intended semantics. We then use these two sets respectively for the functions in the premises and the consequence of the immediate consequence operator, and by a simple static test we can determine whether some of the clauses are wrong. The basic methodology presented so far has been implemented by a prototype system BUGGY which includes a parser for a conditional functional logic language which can be executed by leftmost innermost narrowing, (innermost) basic narrowing, leftmost outermost narrowing and needed narrowing [Antoy *et al.*, 2000]. Moreover, in the natural extending, then we present an algorithm for automatic program correction which is based on unfolding/folding and is able to automatically infer the correct rules from (positive and negative) evidence.

Resum

Els llenguatges de programació lògic funcionals integren algunes de les característiques dels paradigmes declaratius bàsics, en concret els de la programació lògica i els de la programació funcional. Cadascun d'aquests dos estils tenen diferents advantatges respecte a les seues aplicacions pràctiques. Els llenguatges funcionals proporcionen facilitats d'abstracció sofisticades, sistemes de mòduls i solucions "pures"per a l'integració de facilitats de I/O en la programació declarativa, a més a més de tècniques i d'estratègies eficients per a l'execució dels programes. Els llenguatges lògics permeten la computació amb informació parcial i estan provistos de facilitats de cerca de solucions. En canvi, recentment s'ha demostrat que els advantatges d'aquests estils poden combinar-se d'una forma efectiva i útil sobre un llenguatge únic. Els llenguatges lògic funcionals moderns ofereixen característiques de tots dos estils. La semàntica operacional dels llenguatges integrats està usualment basada en narrowing, una combinació de la unificació i de la reducció com a mecanisme d'aval·luació que compren a la reescriptura i a la SLD-resolució.

La depuració dels programaes lògic funcionals és un important problema pràctic que ha sigut tractat escasament en la literatura. La depuració es pot enfocar des d'un punt de vista declaratiu i des d'un punt de vista procedural. La principal contribució d'aquesta tesis és el desenvolupament de mètodes de diagnòstic declaratiu per a la depuració de programes lògic funcionals respecte a l'observable de respostes computades. Les condicions imposades als programes que considerem ens permeten definir un marc genèric per a la depuració declarativa, el qual és paramètric respecte a l'estratègia de narrowing. En particular, el nostre esquema s'aplica tant al narrowing impacient (cridada per valor) com al narrowing peresós (cridada per nom). Primer associem a un programa lògic funcional, R, un operador (contínuo) de conseqüències immediates, el qual és paramètric respecte a l'estratègia de narrowing que pot ser tant peresosa com voraç. Utilitzem el menor punt fix d'aquest operador per a definir la semàntica del programa. Demostrem que tal semàntica té la propietat de què podem obtindre les respostes computades per a un objectiu G, amb l'estratègia de narrowing, per unificació sintàctica amb les equacions de la semàntica. Construim la semàntica o i mostrem la seua correspondència amb la semàntica de punt fix. Ales-

hores mostrem que, donada una especificació desitjada i d'un programa R, podem determinar els errors de correcció i de competitud del programa R mitjançant un simple pas d'aquest operador. A continuació presentem una tècnica d'aproximació de la semàntica desitjada del conjunt d'èxits. Utilitzem els conceptes de sobreaproximació i de subaproximació per aproximar correctament per excés (respectivament per defecte) la semàntica desitjada. Al comparar un d'aquest conjunt amb el resultat d'aplicar l'operador de conseqüències immediates a l'altre i, mitjançant un simple test estàtic, podem determinar quan algunes de les clàusules és incorrecta i quan existeixen equacions no cobertes. Finalment hem desenvolupat un sistema experimental "BUGGY", que permet realitzar la depuració de programes lògic funcionals respecte a la semàntica de respostes computades per les estratègies de narrowing bàsic, innermost, outermost i necessari. En el sistema presentem, com a complement natural de l'eina de diagnòstic, un mòdul per a la correció automàtica de regles, que utilitza els mètodes d'inducció de programes lògic funcionals basats en transformacions de plegat-desplegat.

Agradecimientos

A mi Marta, mi familia, mis amigos, mis compañeros,...

Al finalizar un proyecto de tal magnitud, son pocas la líneas para describir lo que uno siente y quiere expresar. Esta experiencia sin lugar a dudas es la más grande de mi vida, por su intensidad, por su valor, por lo que aprendí, por todo lo que la vida me ofreció y por lo que siento ahora. Fueron muchas las dificultades que tuvimos que superar, los aprendizajes logrados, las satisfacciones, tristezas y alegrías vividas, y ahora no puedo más que expresar mi satisfacción de haber logrado la meta. Pero este camino no lo crucé sólo, son muchas las personas e instituciones presentes y a la hora de hacer mención explícita se corre el riesgo de olvidarse de alguien. Por eso, antes de hacer alguna mención específica, quiero ofrecer mi gratitud a todos los que me acompañaron con la palabra, con el acto y con el sentimiento. En muchos momentos de este proceso sentí esa pequeña sonrisa, ese toquecito en el hombro, que me empujaba a seguir adelante. Quiero compartir con todos ellos, esta felicidad que me desborda.

- A *María Alpuente*, en quien tuve la fortuna de encontrar una gran asesora, compañera y amiga. Sus exigencias, apoyo y disponibilidad hicieron de este proceso una experiencia de la que estoy seguro, he crecido mucho. Su rigor metodológico, la calidad de su trabajo, su dedicación y amor por la investigación han contribuido a formar el investigador que un día me soñé, con la capacidad de continuar y generar investigación en mi país.

- A *mis profesores*, a los *integrantes tanto docentes e investigadores como administrativos* del Departamento de Sistemas Informativos y Computación, DSIC, por su hospitalidad, amistad, compañía y cariño. Durante mis cuatro estancias en la Universidad Politécnica de Valencia siempre recibí un trato especial y amable, que me hizo sentir como en mi casa. Hasta en mis locuras me acompañaron, no puedo dejar de mencionar la maravillosa experiencia de mi matrimonio en el barco de Isidro. Las navidades compartidas como en familia con los amigos, la semana santa en Extremadura y las reuniones en donde tuve la posibilidad de

compartir agradables momentos con las familias de muchos compañeros y que ahora son mis amigos, que entre muchas otras cosas me dejarán un gran recuerdo de Valencia, de la Universidad Politécnica de Valencia y, lo más importante, de su gente.

- A mis *compañeros del grupo de investigación de Extensiones de la Programación Lógica*, por su apoyo y colaboración, sobre todo las discusiones y dudas planteadas y resueltas, además de los sabios consejos y momentos de apoyo. A Salvador Lucas y Germán Vidal quienes, además de María Alpuente, en los comienzos de este proceso me ayudaron a descubrir el mundo de la integración de paradigmas y me ayudaron a hacer acopio de la extensa literatura de la tuve que proveerme para iniciar mis investigaciones. A Santiago Escobar, además de su amistad, por sus aportes y colaboración en el desarrollo de algunos recursos del sistema Buggy. A María José Ramírez, Ginés Moreno, Pascual Julián, Elvira Albert, César Ferri, y en general, a todos los integrantes del grupo de investigación por su amistad, hospitalidad, disponibilidad y colaboración, con los que siempre pude contar para todo lo que implica el desarrollo de una investigación y con quienes como cómplices del conocimiento nos hemos unido para pensar y decir.

- A *Moreno Falaschi*, por su apoyo y amistad, con el que siempre es un placer trabajar. En las tres estancias de investigación en la universidad de Udine–Italia tuve la oportunidad de compartir el trabajo con personas de su grupo de investigación, de gran calidad científica y humana, con quienes logré algunos de los resultados de esta tesis.

- A *Matilde Celma* coordinadora del Doctorado y en quien siempre encontré apoyo y amistad.

- A mis *compañeros de trabajo de la universidad EAFIT* en especial a Juan Guillermo Lalinde y Alberto Rodríguez por su apoyo decidido, en esta gran empresa. Sin olvidar los demás compañeros con los cuales he compartido experiencias de estudio y trabajo. Además, al iniciar esta experiencia también tuve la fortuna de estar acompañado con los *amigos y colegas de trabajo* del Municipio de Itagüí y del liceo Concejo Municipal de Itagüí.

- A los *amigos de Valencia* con los que he disfrutado de esta experiencia y que me han brindado su compañía a lo largo de este proceso.

- A *mi esposa*, esa gran mujer que me ha acompañado en este proceso, y que ha soportado todas las dificultades y ha compartido todos mis triunfos. Te amo mujer.

- A *mi familia* y *amigos en Colombia*, de los que nos hemos tenido que separar en los kilómetros, pero que siempre han estado ahí, siempre presentes, siempre esperando.

Quiero agradecer también a aquellos *organismos e instituciones* que, de una forma u otra, han colaborado en el desarrollo de los trabajos contenidos en esta tesis: a la Universidad EAFIT en Colombia, donde llevo a cabo mi labor como docente e investigador, de la cual he recibido apoyo tanto a nivel económico como de tiempo y recursos, teniendo de este modo las facilidades para realizar mis actividades, pasantías y compromisos relacionados con el desarrollo de esta tesis. Además de su personal administrativo siempre disponibles para que yo pudiera tener las condiciones adecuadas para realizar esta, mi gran, experiencia. Al Municipio de Itagüí (Liceo Concejo Municipal de Itagüí) de los que recibí apoyo al iniciar este proceso y en la primera estancia en la Universidad Politécnica de Valencia. A las Universidades Politécnica de Valencia y de Udine, donde he realizado diversas estancias, de los cuales he recibido apoyo en compañía de la universidad EAFIT para realizar los viajes y participar en diferentes congresos; y, finalmente, a las diferentes entidades que han cooperado con el Vicerrectorado de la Fundación UPV y su Área de Acción Internacional, en la gestión y financiación conjunta con la universidad EAFIT de las pasantías realizadas en la UPV: La Agencia Española de Cooperación Internacional (AECI), la Dirección de Cooperación de la Generalitat Valenciana y la Corporación COINNOVAR.

Índice General

Capítulo 1

Introducción

En el área de ingeniería de software ocupan un papel preponderante las investigaciones que tienen que ver con el desarrollo de conceptos y herramientas que den soporte a la utilización de técnicas automáticas en el proceso de producción del software. Una propuesta ampliamente aceptada para lograr esta meta es el uso de técnicas automáticas de especificación formal. La lógica matemática subyacente aporta su sintaxis, su semántica y su mecanismo de deducción, el cual debe ser correcto y completo. En consecuencia, el significado de la especificación se puede definir en términos de modelos y su ejecución en términos de deducción.

1.1 Semántica de los lenguajes de programación

Uno de los principales requisitos de un buen lenguaje de programación es el de disponer de una semántica definida de un modo sencillo, flexible y con un buen soporte formal. La tendencia actual es asociar a los lenguajes de especificación una semántica formal definida en un estilo ya estándar, semántica declarativa y semántica operacional, con resultados de equivalencia entre ambas. De esta forma, las especificaciones pueden ser consideradas como teorías que permiten verificar que los modelos que las representan se corresponden con las intenciones del programador, para validar su corrección (y otras propiedades) respecto de la funcionalidad esperada.

1.1.1 Semántica declarativa

El término "declarativa" indica un tipo de semántica que especifica el significado de los objetos sintácticos por medio de su traducción en elementos y estructuras de un dominio matemático conocido. Se han propuesto muchos métodos que se basan en este principio (ver [Palamidessi, 1988] para una clasificación). En relación con nues-

tra investigación, algunos de los más interesantes, que examinaremos en los apartados siguientes, son la semántica por teoría de modelos, la semántica por punto fijo y la semántica denotacional. A menudo se tiende a identificar el método denotacional y el método del punto fijo por la importancia que tiene este último en la aproximación denotacional. Sin embargo, conceptualmente son independientes y conviene identificarlos ya que la semántica por punto fijo se aplica también a otro tipo de aproximaciones.

Semántica por teoría de modelos La semántica por teoría de modelos está basada en las nociones de *interpretación* y de *modelo*. Una interpretación está constituida por un cierto dominio y una función que asocia a los símbolos de base del lenguaje (símbolos de constantes, de función y de predicado) elementos, funciones y relaciones con respecto a dicho dominio, respectivamente. Las conectivas lógicas tienen una interpretación predefinida. A cada fórmula se le asigna un valor de verdad y se definen como modelos de un conjunto de axiomas todas las interpretaciones en las que los axiomas resultan verdaderos. El significado que se da a un programa es el conjunto de sus consecuencias lógicas, i.e. el conjunto de fórmulas que son verdad en todos los modelos del programa. La teoría de modelos ha producido resultados importantes, como el teorema de Löwenheim-Skolem (una teoría consistente tiene al menos un modelo numerable, que puede ser obtenido a partir de un modelo totalmente sintáctico), el teorema de completitud de Gödel (equivalencia entre derivabilidad $\vdash$ e implicación semántica $\models$) y el teorema de Gödel sobre indecibilidad de todas las teorías "interesantes" (es decir, tan potentes, al menos, como la aritmética). En el caso particular de los programas de cláusulas de Horn definidas se cumple también el teorema de existencia del modelo mínimo (de Herbrand) que, en lo que respecta a la verdad de ciertas fórmulas, puede ser considerado como representante de la clase entera de modelos y, por tanto, ser utilizado para caracterizar el significado de este tipo de programas.

Semántica por punto fijo En la aproximación por punto fijo se define el significado de un programa R como un determinado punto fijo $T_{\mathcal{R}}$ de una transformación T, continua, asociada a dicho programa. La continuidad de T implica la existencia de un punto fijo mínimo y garantiza una contrapartida computacional inmediata, que consiste en poder construir tal objeto de un modo iterativo, mediante la aplicación sucesiva de la transformación a partir del elemento más pequeño del dominio. No siempre se requiere la continuidad de tales transformaciones (sobre algunas estructuras, como los retículos completos, la monotonía es suficiente para asegurar la existencia de puntos fijos) y no siempre se elige el menor punto fijo como semántica, como por ejemplo la elección del máximo punto fijo en cierto tipo de especificaciones.

En el caso general, la semántica por punto fijo es la abstracción de un proceso computacional y no proporciona, por sí misma, una semántica declarativa. En el caso de programas en lógica de cláusulas de Horn definidas la situación es muy diferente. Informalmente, la semántica por punto fijo de un programa lógico viene a expresar cómo construir inductivamente un modelo para el mismo en una forma ascendente: dados los hechos y las reglas, se aplican las reglas a los hechos para generar nuevos hechos, se repite la operación sobre el conjunto calculado hasta que no se puedan generar nuevos hechos. El menor punto fijo así obtenido coincide con el modelo mínimo [Kowalski, 1974]. Así pues, está asociado a consecuencias lógicas y tiene un significado declarativo claro. Por otra parte, la caracterización por punto fijo proporciona un enlace entre la semántica por teoría de modelos y la semántica operacional, lo cual permite simplificar las correspondientes demostraciones de equivalencia entre ambas.

Semántica denotacional La aproximación denotacional, que tiene sus orígenes en los trabajos de Scott, Strachey y de Bakker, se articula en las siguientes fases:

i) Subdivisión de los objetos lingüísticos en varios conjuntos (categorías sintácticas o dominios sintácticos) y clasificación de los operadores lingüísticos para obtener otros objetos).

ii) Definición de los dominios semánticos adecuados, algunos de los cuales corresponden a las categorías sintácticas mientras que otros son auxiliares. Esta definición viene dada constructivamente, partiendo de los dominios de base y especificando dominios cada vez más complejos por medio de operadores de dominio. Una técnica alternativa, desarrollada por Scott, consiste en especificar los dominios mediante ecuaciones de dominio. La teoría de Scott se basa, esencialmente, en los conceptos de orden parcial, completitud, algebricidad y continuidad.

iii) Definición de algunas funciones de valuación semántica, que asocian a cada elemento del correspondiente dominio semántico y a cada operador un elemento del espacio funcional sobre los correspondientes dominios semánticos.

1.1.2 Semántica operacional

El significado del programa se define en términos de las acciones que serían ejecutadas por un modelo abstracto de máquina que debe haber sido definido con anterioridad. La definición semántica operacional de un lenguaje de programación equivale a la definición de un intérprete para el mismo independiente de cualquier posible implementación. En el caso de la lógica de predicados, por ejemplo, el procedimiento de demostración se comporta como tal intérprete. Una implemetación concreta de un lenguaje no tiene por qué seguir los mismos mecanismos operacionales de su definición semántica, pero sí ha de obtener los mismos resultados. Una de las aproximaciones más relevantes a la definición de semántica operacional es la definida por Plotkin [Plot-

kin, 1981] como "Structural Operational Semantics" (SOS), basada en un concepto de máquina verdaderamente abstracta y muy simple: el formalismo de los *Sistemas de Transición*. La semántica de las diversas construcciones del lenguaje se especifica, por inducción estructural, mediante reglas de inferencia. Esta presentación es de gran utilidad cuando se intenta demostrar propiedades de los programas puesto que, a menudo, es posible probar dichas propiedades utilizando conjuntamente inducción estructural e inducción sobre la longitud de las secuencias de transición.

1.2 Programación declarativa

La *programación declarativa*, a veces llamada *programación inferencial*, puede entenderse como un estilo de programación en el que el programador especifica *qué* debe computarse en lugar de *cómo* deben realizarse los cómputos. En este paradigma de programación, de acuerdo con Kowalski [Kowalski, 1974, 1979], un

$$programa = lógica + control,$$

y la tarea de programar consiste en centrar la atención en la *lógica* del problema dejando de lado el *control*, que se asume automático, y que es competencia del sistema. Con más precisión, la característica fundamental de la programación declarativa es el uso de la lógica como lenguaje de programación y se puede conceptualizar como sigue:

- Un *programa* es una teoría formal en una cierta lógica, y
- la *computación* se entiende como una forma de inferencia o deducción en dicha lógica.

Como requisito fundamental la lógica empleada debe disponer de un lenguaje, de una semántica operacional y de una semántica declarativa; además de cumplir los correspondientes resultados de corrección y completitud que aseguran que lo que se computa coincide con aquello que es considerado como verdadero, de acuerdo con la noción de verdad que sirve de base a la semántica declarativa.

1.3 Programación lógica

Un lenguaje lógico es aquél en el que los programas son teorías en una cierta lógica y en el que, además, tres conceptos son equivalentes: computación en la máquina, deducción en la lógica y satisfacción en un modelo estándar de la teoría. Adicionalmente, para establecer una distinción entre demostración de teoremas y programación lógica es necesario un requerimiento de eficiencia y la existencia de un mecanismo efectivo de extracción de respuestas.

Para que un lenguaje lógico sea ejecutable es necesario que el correspondiente sistema lógico sea un subconjunto apropiado debido a que no ha sido posible, hasta ahora, abordar computacionalmente un sistema lógico completo por su gran capacidad expresiva. En consecuencia, la *programación lógica* [Apt, 1990; Kowalski, 1974; Lloyd, 1987a] se define tomando como base subconjuntos de la lógica de predicados, siendo el más popular la lógica de *cláusulas de Horn* (HCL, del inglés *Horn clause logic*), el cual puede emplearse como base para un lenguaje de programación al poseer una semántica operacional susceptible de una implementación eficiente: la *resolución SLD*. Como semántica declarativa utiliza una semántica por *teoría de modelos* y una semántica por *punto fijo* que toman como dominio de interpretación un universo puramente sintáctico: el *universo de Herbrand*. La *resolución SLD* es un método de prueba por refutación que emplea el algoritmo de *unificación* como mecanismo de base y permite la extracción de respuestas dadas como enlace de un valor a una *variable lógica*. La resolución SLD es un método de prueba correcto y completo para la lógica HCL.

Una características de los lenguajes declarativos es que permiten la *gestión automática de la memoria*, evitando una de las mayores fuentes de error que ocurren en los programas implementados en otros paradigmas de programación. Otra, es que el programa puede responder a diferentes consultas (*objetivos*) sin necesidad de efectuar cambios en él, gracias al hecho de emplearse el mecanismo de resolución SLD que permite una *búsqueda indeterminista* (*built-in search*) de soluciones. La misma propiedad, permite computar con *datos parcialmente definidos* y hace posible que la *relación de entrada/salida* no esté fijada de antemano.

1.4 Programación funcional

El fundamento esencial de los *lenguajes funcionales* es el concepto matemático de *función*, las cuales se definen mediante ecuaciones, generalmente recursivas, que constituyen el programa. Desde el punto de vista computacional, la programación funcional se centra en la evaluación de expresiones funcionales para obtener un *resultado*.

En programación funcional la descripción de dominios conduce a la idea de *tipo de datos*. Los lenguajes funcionales modernos son lenguajes *fuertemente basados en tipos* (*strongly typed*), realizan una comprobación de las expresiones que aparecen en el programa para asegurarse que no se producirán errores durante la ejecución del mismo por incompatibilidades de tipo. Adicionalmente, lo que caracteriza a una función, además de su perfil, es que cada elemento de su dominio tiene correspondencia con un único elemento del rango; es decir, el resultado (*salida*) de aplicar una función sobre sus argumentos viene determinado exclusivamente por el valor de éstos (su *entrada*). Esta propiedad de las funciones se denomina *transparencia referencial*. La transparencia referencial permite el estilo de la programación funcional basado en

el razonamiento ecuacional, la substitución de iguales por iguales, y los cómputos deterministas. Por último, desde el punto de vista computacional, otra propiedad de las funciones es su capacidad para ser compuestas. La composición de funciones es la técnica por excelencia de la programación funcional, que permite la construcción de programas mediante el empleo de funciones primitivas o previamente definidas por el usuario y como consecuencia refuerza la modularidad de los programas.

Una de las características de los lenguajes funcionales que va más allá de la mera composición de funciones es el empleo de las mismas como "ciudadanos de primera clase" dentro del lenguaje, de forma que estas puedan almacenarse en estructuras de datos, pasarse como argumento a otras funciones y devolverse como resultados. Estas características se referencian bajo la denominación de *orden superior*. A veces también se dice que una función es de orden superior si algunos de sus argumentos son, a su vez, una función.

Un programa funcional incluye una lista de ecuaciones que definen las funciones y una expresión básica (sin variables) que se pretende evaluar como *objetivo* [Plasmeijer y van Eekelen, 1993; Reade, 1993]. La ejecución de un programa consiste en la evaluación del objetivo de acuerdo con las ecuaciones de definición de las funciones y alguna forma de reducción. La *reducción* es una secuencia de pasos que transforma la expresión inicial, compuesta de símbolos de función y de símbolos constructores de datos, en un *valor* o resultado. La secuencia de pasos dada en el proceso de reducción depende de la estrategia de reducción empleada. Podemos distinguir dos *estrategias de reducción* o *modos de evaluación*: la denominada impaciente y la perezosa. Una estrategia *impaciente* evalúa primero los argumentos de una función mientras que una *perezosa* evalúa los argumentos sólo si su valor es necesario para el cómputo de dicha función, intentando evitar cálculos innecesarios. Si bien la estrategia impaciente es más fácil de implementar en los computadores con arquitecturas convencionales, puede conducir a secuencias de reducción que no terminan en situaciones en las que una evaluación perezosa es capaz de computar un valor. Este hecho, junto con algunas facilidades de programación (por ejemplo, libera al programador de la preocupación por el orden de evaluación de las expresiones [Hudak, 1989]) y ventajas expresivas que proporciona (como la habilidad de computar con estructuras de datos infinitas), han hecho que la posibilidad de utilizar un modo evaluación perezosa sea muy apreciada en los lenguajes funcionales modernos.

1.4.1 Teoría lógico ecuacional y sistemas de reescritura

La lógica ecuacional se define como la restricción del cálculo de predicados de primer orden con identidad, obtenida suprimiendo toda conectiva lógica y todo símbolo de predicado distinto de la igualdad. Un subconjunto importante son las teorías ecuacionales canónicas donde la reescritura o reducción desempeña el papel de intérprete

completo y una construcción algebraica apropiada refleja el modelo estándar. La relación de reescritura se entiende como deducción ecuacional usando las ecuaciones sólo de izquierda a derecha. Un paso de computación se ejecuta buscando un subtérmino de la expresión a reducir que sea exactamente igual a una instancia de la parte izquierda de la definición de una función ("pattern matching") y, si tal término existe, reemplazamos el subtérmino por la correspondiente instancia de la parte derecha.

La *lógica ecuacional* proporciona un soporte sintáctico para la definición de funciones (mediante el uso de ecuaciones[1]) y el razonamiento ecuacional (mediante un sistema de deducción ecuacional que incluye como reglas de inferencia las conocidas propiedades reflexiva, simétrica y transitiva de la identidad, así como las propiedades de cierre por substitución de idénticos por idénticos y de cierre por instanciación).

1.5 Programación multiparadigma

Uno de los desafíos todavía pendientes en el área de investigación sobre lenguajes de programación, consiste en la integración de los distintos paradigmas de programación declarativa, en particular, de los estilos de programación lógica y funcional. Para desarrollar lenguajes integrados prácticos es necesario conseguir que su eficiencia sea comparable a la de los lenguajes imperativos. También es necesario desarrollar técnicas y herramientas que ayuden al programador a manipular y optimizar sus programas.

La integración de la programación lógica y funcional es un área de investigación activa desde principios de los años ochenta. Los lenguajes de programación lógico funcionales permiten integrar algunas de las mejores características de los paradigmas de programación lógica y funcional. De la programación lógica, los lenguajes lógico funcionales obtienen el uso de la unificación, la potencia de las variables lógicas y la inversión de definiciones, un mecanismo de búsqueda automático indeterminista y la posibilidad de trabajar con estructuras de datos parciales. De la programación funcional obtienen, entre otros beneficios, la expresividad de las funciones, el empleo de tipos, el orden superior y un mecanismo de evaluación más eficiente (determinismo y evaluación perezosa). El empleo de la técnica de reducción permite evitar características extralógicas introducidas en la programación con PROLOG para reducir el espacio de búsqueda (*el operador de corte*). En la última década se han realizado importantes avances en el campo de la integración de lenguajes declarativos, una panorámica de los cuales puede encontrarse en [Hanus, 1994b; Hölldobler, 1989; Moreno, 1994].

A pesar de que, potencialmente, los distintos paradigmas de programación decla-

[1] En el caso más general, cláusulas de Horn ecuacionales, en las que el único símbolo de predicado permitido es la igualdad.

rativa ofrecen un soporte más adecuado para el desarrollo rápido y a bajo coste de aplicaciones correctas, así como para la generación de herramientas sofisticadas de ayuda al desarrollo de software, estos lenguajes no están difundidos suficientemente ni han conseguido imponerse a nivel industrial. Esta situación se debe, en gran medida, a la ausencia de un lenguaje lógico funcional integrado *estándar* (aceptado por toda la comunidad), al carácter *experimental* de las implementaciones existentes (inmaduras aún para aplicaciones reales) y a la carencia de herramientas de desarrollo potentes, capaces de optimizar dichas implementaciones y de asistir al programador en la manipulación, transformación y optimización de los programas.

1.5.1 Narrowing

Muchas propuestas para la integración usan sistemas de reescritura de términos condicionales (SRTC) como programas y alguna variante del *narrowing* como mecanismo operacional, soportando así unificación y variables lógicas en un contexto funcional. El procedimiento de *narrowing* puede verse como una extensión de la reescritura que puede ser implementado eficientemente sustituyendo ajuste de patrones (*pattern matching*) por la unificación en el procedimiento de reducción. En un *paso de narrowing condicional* se unifica un subtérmino del objetivo con la parte izquierda de una regla y se reemplaza el subtérmino del objetivo por la parte derecha instanciada de la regla, adicionando al objetivo instanciado las correspondientes instancias de las condiciones de la regla utilizada. El uso de *narrowing* como mecanismo operacional para realizar las computaciones supone una extensión muy potente de los programas lógicos tradicionales y el modelo resultante tiene buenas oportunidades para explotar el paralelismo implícito en el lenguaje. La técnica de narrowing, como medio para la obtención de un conjunto de soluciones a un problema de E-unificación, fue descrita en los trabajos pioneros de [Slagle, 1974] y [Fay, 1979]. La relación definida por Fay en [Fay, 1979], y que denominó *narrowing*, es una variante que posteriormente ha recibido el nombre de *narrowing normalizante* [Réty, 1987; Hanus, 1994b]. *Narrowing normalizante* se caracteriza por la normalización del término a evaluar previa a la aplicación de cada paso de *narrowing*.

En general, el procedimiento de *narrowing* tiene un grado de indeterminismo muy alto, debido a la existencia de dos grados de libertad: elección del subtérmino a reducir y elección de la regla.[2] Esto conduce a un espacio de búsqueda muy amplio. Existen muchas estrategias para reducirlo, eliminando algunas derivaciones inútiles, tales como: *innermost narrowing* [Fribourg, 1985], *narrowing* básico [Hullot, 1980; Middeldorp y Hamoen, 1994], *narrowing* básico innermost [Hölldobler, 1989], outer-

[2] En reescritura existen los mismos grados de libertad, pero en narrowing están acentuados debido a la posibilidad de instanciar variables para producir redexes. Aun más, si el sistema de reescritura es ortogonal, en reescritura el segundo grado de libertad desaparece.

most *narrowing* [Echahed, 1988], *narrowing* selección [Bosco *et al.*, 1988], *narrowing* perezoso [Antoy *et al.*, 1994; Hanus, 1994a; Moreno-Navarro y Rodríguez-Artalejo, 1992; Reddy, 1985], *narrowing* necesario [Antoy *et al.*, 1994], entre otros. Cada una de estas estrategias sigue manteniendo, bajo determinadas condiciones, la completitud del cálculo. En [Aït-Kaci y Nasr, 1989; Bellia y Levi, 1986; Dershowitz y Plaisted, 1988; Moreno-Navarro y Rodríguez-Artalejo, 1992; Moreno, 1994; Reddy, 1985] se puede encontrar una revisión, análisis y clasificación de éstas y otras propuestas para la integración. La colección en [DeGroot y Lindstrom, 1986] constituye una referencia estándar en el campo. La panorámica más completa se presenta en [Hanus, 1994b]. En [Middeldorp y Hamoen, 1994; Hanus, 1994b] pueden encontrarse resúmenes sobre las condiciones que debe cumplir un programa, para que una determinada estrategia sea correcta y completa.

1.6 Depuración de programas

Un problema central en el desarrollo del software es obtener un programa que cumpla con las expectativas del usuario. Una pregunta natural, en este sentido, es si éste cumple, o no, alguna clase de expectativas (requerimientos o propiedades). Para poder formular estas preguntas es necesario especificar, formal o informalmente, tales requerimientos. Se precisa entonces de una *semántica* (formal o informal) del programa , en la cual, lo que el programa computa y lo que se desea computar pueda ser expresado.

Tal como lo hemos planteado en el desarrollo de este capítulo, una ventaja de los lenguajes de programación lógicos y/o funcionales es que facilitan la programación declarativa. Ellos permiten, por lo menos hasta cierto punto, separar la semántica declarativa (QUÉ se computa) de la semántica operacional (CÓMO se computa). En principio, todo razonamiento acerca de la "corrección" de programas puede hacerse a nivel de la semántica declarativa y de este modo nos olvidamos de los detalles relativos al mecanismo de computación.

Esta ventaja se pierde cuando la búsqueda y corrección de los errores se realiza a nivel de la semántica operacional. Sucede en el caso de algunas herramientas comunes que se fundamentan en el análisis de trazas y obligan al programador a que piense en secuencias de pasos de computación. Algunas de estas herramientas presentan una traza que despliega la ejecución real de los programas como ayuda para la depuración. Emparejar la traza de información con el programa para localizar los errores, es una tarea difícil, principalmente debido a los mecanismos de ejecución de lenguajes, como por ejemplo la "vuelta atrás" del Prolog cuyo problema es que realiza una transferencia no local del control y es difícil de seguir. Un intento para superar las deficiencias de las propuestas originales es el modelo de ejecución basado en *cajas de Byrd* [Byrd, 1980].

El modelo de cajas es el fundamento, con algunas variaciones, de una gran cantidad de depuradores en programación lógica [Ducassé, 1999b,a], funcional [Arenas y Gil, 1994] o para lenguajes integrados [Arenas y Gil, 1995; Hanus y Josephs, 1993; Heyer, 1995]. El modelo de cajas está basado en la semántica operacional, pero la información que muestra la traza no corresponde exactamente a la ejecución real. La traza desplegada se aumenta con información que hace más fácil para un usuario realizar un seguimiento de la ejecución real.

En términos generales, una caja representa todas las cláusulas (reglas) que definen un predicado (función), es decir una llamada a subrutina. Una caja tiene cuatro puertos a través de los cuales podemos controlar qué entra y qué sale. Primero, el puerto de *llamada* se activa cuando se ejecuta una llamada o invocación de un predicado/función. Segundo, el puerto de *salida* se activa cuando la llamada al predicado/función tiene éxito. Tercero, el control activa la caja a través del puerto *rehacer* con el mecanismo de vuelta atrás, por ejemplo. Cuarto, se sale de la caja a través del puerto de *fallo* cuando la última alternativa para el predicado/función falla en el intento de resuelta.

El problema de este enfoque es que implica una diferencia entre la metodología utilizada para el diseño y la utilizada para la corrección y búsqueda de errores. Una ventaja de los paradigmas declarativos es que el diseño también puede efectuarse en forma declarativa. Desgraciadamente, los modelos de cajas centran su atención en el comportamiento del programa. Por consiguiente, aunque deseamos diseñar y programar con la semántica declarativa en mente, estos métodos nos obligan a corregir el programa en el ambiente de la semántica operacional. Adicionalmente, los depuradores basados en el modelo de cajas de Byrd solamente muestran la información sobre la llamada actual a ser resuelta. Ésta es una cantidad muy pequeña de información, y para realizar cualquier proceso de corrección de un programa real el usuario tiene que seguir la traza de la ejecución en todo el código.

Es posible tanto *verificar* que el programa satisface el requerimiento en todas las computaciones como mostrar que una determinada computación viola dicho requerimiento. La búsqueda de las discrepancias entre lo que el programa calcula y lo que desea el programador, se puede enfocar desde el punto de vista del *conocimiento procedural*, que tiene en cuenta la secuencia de operaciones aplicadas durante la ejecución, o el *conocimiento declarativo* del programa, el cual determina qué resultados devueltos por los procedimientos son correctos [Naish, 1993]. Por lo planteado anteriormente, es necesario desarrollar métodos para la búsqueda y corrección de programas basados en la semántica declarativa.

Para lograr una mejor claridad de los temas aquí planteados vamos a presentar, de manera muy general e intuitiva, algunos de los términos fundamentales de nuestro marco. Posteriormente daremos una descripción formal de algunos de ellos. Asumi-

remos que el usuario tiene una idea clara sobre los resultados que debe computar el programa.

En un sentido amplio y desde el punto de vista de la depuración declarativa, un *error* ocurre cuando el programa computa algo que el programador no desea, o cuando no computa algo que él quiere. Un *error* en un programa es la causa de que el programa se desvíe de su comportamiento esperado. El propósito de la depuración es eliminar los *errores* de los programas. En general, la *depuración* es un proceso que busca detectar, diagnosticar y corregir los errores en un programa dado. La *detección del error* es la actividad de juzgar si el comportamiento real de un programa es el deseado. Una diferencia entre el comportamiento real y el comportamiento deseado se denomina *síntoma*. Un *síntoma de incorrección* es un elemento calculado por el programa que no está en la semántica deseada. Un *síntoma de incompletitud* es un elemento de la semántica deseada que no es calculado por el programa. El *diagnóstico del error* es la actividad de encontrar el error responsable del síntoma de error. El *diagnóstico* del error comprende la localización del error y su identificación. La *localización del error* identifica una parte del programa como posible causa del síntoma del error, mientras que la *identificación del error* lo clasifica. La *corrección del error* es la actividad de modificar la parte incorrecta del programa en un intento por hacer que el programa se comporte como el programa deseado [Lu, 1994a].

La idea fundamental del diagnóstico declarativo del error es comparar el comportamiento deseado del programa con su comportamiento real. Con ello, el *depurador* podrá encontrar los errores, si los hay. El comportamiento deseado se puede formular de varias formas: preguntándole al usuario, mediante una especificación formal, una especificación aproximada (abstracta) u otra versión (correcta) del programa. Las entidades que proporcionan esta información a la herramienta de diagnóstico se agrupan con un término común llamado el *oráculo*.

Al realizar un estudio del desarrollo histórico en el área de la depuración declarativa es importante mencionar algunas aportaciones. El primer trabajo en el área sobre depuración declarativa se debe a Shapiro [Shaphiro, 1982]. Establece la diferencia entre el diagnóstico por trazas y el diagnóstico del error declarativo en el sentido en que el último se basa en la semántica declarativa del lenguaje. Shapiro denominó a este último *depuración algorítmica*. Con ello genera un nuevo método para la localización de los errores en un programa. En general, asume que el *oráculo* es el usuario. Sus trabajos se basan en la semántica básica del modelo mínimo de Herbrand.

Pereira [Pereira, 1986] plantea métodos para simplificar la interacción con el usuario. En el enfoque de Shapiro, para realizar el diagnóstico de incompletitud el usuario debe proporcionar instancias (en el caso peor, todas) correctas de un átomo. En el enfoque de Pereira, el usuario sólo determina si un conjunto contiene, o no, todas las instancias correctas de un átomo.

Lloyd [Lloyd, 1987a,b] y Ferrand [Ferrand, 1987] probaron resultados de corrección y completitud para el diagnóstico del error declarativo. El enfoque de Lloyd está en el contexto del lenguaje de programación lógica con "corrutinas" y recientemente para lenguajes integrados [Lloyd, 1998]. El depurador es similar al de Shapiro para el diagnóstico de respuestas perdidas. Ferrand, presenta su enfoque basado en un operador de punto fijo. Posteriormente, en compañía de otros autores [Bueno *et al.*, 1997a], presentó ampliaciones de su propuesta para programas lógico con restricciones utilizando argumentos de la interpretación abstracta. Naish mostró, en su amplio estudio sobre el diagnóstico de incompletitud [Naish, 1992] que las "corrutinas" no responden de manera adecuada al enfoque de Pereira. En [Naish, 1992] se presentan diferentes ideas para la implementación de métodos para el diagnóstico de las respuestas perdidas y su aplicabilidad bajo diferentes reglas de cómputo. El mismo autor presenta un esquema para la depuración declarativa [Naish, 1997], el cual ha aplicado a lenguajes funcionales [Naish, 1993]y, a lenguajes logico funcionales [Naish y Barbour, 1994], mediante el uso de los *árboles de demostración* utilizados también por Shapiro y en donde la búsqueda de los errores está guiada por *síntomas*.

Una de las preocupaciones importantes es el problema de la mecanización del oráculo. El depurador de Shapiro permite la memorización de preguntas realizadas al oráculo y de sus respuestas, para asegurar que el usuario no deba responder dos veces a la misma pregunta. Ésta es una mecanización obvia, pero él también propuso otras. Es importante que el usuario sea liberado, tanto como sea posible, de la necesidad de responder las preguntas. Trabajos posteriores (por ejemplo [Drabent *et al.*, 1988] basado en el método de aserciones) investigan la mecanización del oráculo. El usuario proporciona instrucciones al depurador (como por ejemplo propiedades de la semántica deseada del programa) para que el oráculo pueda contestar algunas preguntas por sí mismo.

Recientemente, el marco de las s-semánticas se ha utilizado para el diagnóstico declarativo del error [Comini *et al.*, 1995a,b], con extensiones al uso de técnicas de interpretación abstracta [Comini *et al.*, 1999, 1996, 1994; Lu, 1994a,b] y, más recientemente, usando el método de aserciones [Comini *et al.*, 2001, 2000].

Aunque no hemos hecho mención exhaustiva de las investigaciones llevadas a cabo sobre depuración declarativa realizados en otros paradigmas diferentes de la programación lógica, tales como programación funcional [Naish, 1993; Naish y Barbour, 1995; Nilsson y Fritzson, 1994; Nilsson, 1994; Pope, 1998; Sparud y Nilsson, 1995], programación lógica con restricciones [Bueno *et al.*, 1997a; Drabent *et al.*, 1988; Boye *et al.*, 1997; Bueno *et al.*, 1997b; Lu, 1994a,b] e integración de paradigmas [Naish y Barbour, 1994; Caballero-Roldán *et al.*, 2001; Eklund, 1997], es debido a que sus fundamentos parten de los trabajos realizados en la programación lógica, sin embargo haremos mención detallada de ellos en su momento.

1.6.1 Presentación de la investigación

Nuestra propuesta se centra en el desarrollo de técnicas de depuración declarativa para los programas lógico funcionales entendidos como sistemas de reescritura de términos condicionales y con una semántica basada en (alguna forma de) narrowing condicional. En esta investigación presentamos un esquema genérico para la depuración abstracta de programas lógico funcionales que extiende los métodos propuestos en [Bueno *et al.*, 1997a; Comini *et al.*, 1999] al caso de los lenguajes integrados. Dichas extensiones distan mucho de ser inmediatas y en particular nuestra contribución está en plantear un método de diagnóstico paramétrico con respecto a la estrategia de evaluación del lenguaje: perezosa o impaciente. A nivel general, nuestra propuesta se centra en el desarrollo de técnicas de depuración declarativa para programas lógico funcionales entendidos como sistemas de reescritura de términos condicionales y con una semántica basada en (diferentes formas de) *narrowing condicional*. En este contexto, tomaremos como punto de referencia la semántica dada en [Alpuente *et al.*, 1994], que modela el conjunto de respuestas computadas por narrowing y que viene dada por la siguiente definición:

$$\mathcal{O}^s(\mathcal{R}) \quad = \quad \{(f(\bar{x}) = y)\theta \mid f(\bar{x}) = y \overset{\theta}{\leadsto}{}^* \ \top\}, \tag{1.1}$$

donde R representa el sistema de reescritura considerado. En [Alpuente *et al.*, 2001a] presentamos la formalización de esta semántica, mediante una formulación paramétrica con respecto a las estrategias de *narrowing innermost* y *autermost*. Adicionalmente, mostramos la correspondencia con la semántica de punto fijo, basada en un operador de consecuencias inmediatas paramétrico que hemos definido para este propósito. Probamos que la semántica operacional es completamente abstracta con respecto al observable de respuestas computadas y modela adecuadamente dicho observable. La semántica operacional de punto fijo captura el significado operacional del programa R y garantiza el resultado estándar de la programación lógica de que las sustituciones de respuesta computada para cualquier objetivo (posiblemente conjuntivo) pueden derivarse de ella, por unificación sintáctica de las ecuaciones del objetivo con las ecuaciones de la denotación.

Mediante las semánticas basadas en el $T^s_{\mathcal{R}}$ y $\mathcal{O}^s(\mathcal{R})$ formulamos los conceptos fundamentales de la depuración declarativa, tales como regla incorrecta y ecuación no cubierta como base del desarrollo teórico. Las propiedades que cumplen $T^s_{\mathcal{R}}$, $\mathcal{O}^s(\mathcal{R})$ y el programa $\mathcal{R}$, permiten determinar si un programa es parcialmente correcto, incompleto o totalmente correcto y establecer algoritmos para la depuración con estrategias de narrowing tal como los presentados en [Alpuente *et al.*, 2001a,b,c, 2002d]. Presentamos una semántica de punto fijo abstracta, la cual es paramétrica con respecto a la estrategia de *narrowing*, que caracteriza el conjunto de respuestas computadas abs-

tractas de manera ascendente e independiente del objetivo. Utilizamos una técnica de aproximación de la semántica deseada del conjunto de éxitos, planteamos los conceptos de sobreespecificación $\mathcal{I}^+$ y subespecificación $\mathcal{I}^-$ para aproximar correctamente por exceso y por defecto la semántica deseada.

Usamos estos dos conjuntos para las funciones en las premisas y las consecuencias del *operador de consecuencias inmediatas* y mediante un simple test estático, podemos determinar cuándo alguna de las reglas es incorrecta o una ecuación es no cubierta. De este modo, utilizando el operador de consecuencias inmediatas formulamos un sistema finito de diagnóstico que, a diferencia de otros métodos en la literatura, no requiere determinar previamente ningún tipo de síntomas de incorrección, ni de incompletitud y es totalmente automático (no requiere de la intervención del usuario actuando como un oráculo).

Desarrollamos una implementación [Alpuente *et al.*, 2001b, 2002c] de nuestro sistema de depuración "BUGGY" que demuestra experimentalmente que el método permite encontrar algunos errores comunes sobre una muestra amplia de programas.

Finalmente, hemos aplicado nuestros resultados al desarrollo de técnicas para la inducción de programas lógico funcionales [Alpuente *et al.*, 2002b,a], mediante el uso de operaciones de plegado y desplegado de reglas, cálculo de evidencias positivas y negativas para la inducción de teorías y la utilización de un operador de inversión del operador de consecuencias inmediatas que permite sintetizar reglas que reparan los errores encontrados en el programa.

1.6.2 Estructura de la tesis

Es conveniente realizar algunos comentarios relativos a la terminología utilizada en el desarrollo de esta tesis. Muchos de los conceptos presentados han sido acuñados en inglés y no existen acuerdos claros de la comunidad informática sobre su traducción al Español. En consecuencia, se ha mantenido la terminología inglesa de aquellos conceptos mencionados, como es el caso del principio operacional de "estrechamiento" (*narrowing*) que nosotros optamos por mantener en inglés en el documento. En algunas ocasiones traducimos de forma directa algunos términos ingleses por aquellos vocablos castellanos más comúnmente utilizados, aún cuando existan otras traducciones que se ajustan más fielmente al significado pretendido. Este es el caso del término inglés *instance* que nosotros traducimos por "instancia", en vez de "concrección" o "particularización", que sería más correcto.

Esta tesis se organiza como sigue. El Capítulo 2, presentamos brevemente algunas definiciones y notaciones preliminares. Revisamos los conceptos fundamentales de la programación lógico-funcional, incluyendo una descripción genérica del algoritmo de *narrowing*, el mecanismo de ejecución estándar para los programas considerados. Luego presentamos tres instancias del mismo que pueden verse como refinamientos del

mecanismo original. Estas estrategias se refieren a las optimizaciones conocidas como *narrowing* innermost, outermost y necesario. Con relación a este último, incluimos la transformación de Hanus & Prehofer que utilizaremos en nuestro marco. Mediante esta transformación puede implementarse el mecanismo de *Narrowing necesario* de forma simple y eficiente traduciendo los árboles definicionales en "expresiones case" [Hanus y Prehofer, 1999]. El capítulo se cierra con una presentación de algunos resultados sobre las propiedades de las estrategias que serán necesarios posteriormente.

En el Capítulo 3, definimos un operador de consecuencias inmediatas $T_{\mathcal{R}}^{\varphi}$ para programas lógico funcionales $\mathcal{R}$ que es paramétrico con respecto a la estrategia φ de *narrowing*, que puede ser tanto perezosa como impaciente. En el caso de la estrategia *impaciente*, basta introducir una transformación de aplanamiento que elimina llamadas anidadas y permite *ejecutar objetivos* en la semántica por medio de la unificación estándar. Debido a que en la estrategia perezosa el mecanismo operacional es más complejo, es necesario introducir (adicionalmente al aplanamiento) dos clases de igualdad en la definición de $T_{\mathcal{R}}^{\varphi}$. La igualdad estricta, $\approx$, que modela la igualdad sobre los términos constructores (valores), y la igualdad no estricta, $=$, que es cierta aun si los argumentos son tanto indefinidos como parcialmente definidos, tal como en [Giovannetti *et al.*, 1991]. Construimos la semántica $\mathcal{O}^{\varphi}(\mathcal{R})$ y mostramos su correspondencia con la semántica de punto fijo.

En el Capítulo 4, presentamos un breve recorrido sobre los diferentes enfoques de la depuración declarativa para programas lógicos, programas funcionales y programas lógico funcionales. Pretendemos mostrar los fundamentos teóricos de nuestro marco, al mismo tiempo que damos una comparación con los otros enfoque declarativos. Esto nos permite formular las nociones generales y paramétricas, necesarias para el desarrollo de nuestra propuesta de la depuración declarativa en el marco de los programas lógicos funcionales.

En el Capítulo 5, planteamos una semántica abstracta que aproxima correctamente la semántica de punto fijo de $\mathcal{R}$. Partiendo de la semántica de punto fijo introducida en el Capítulo 3, en este capítulo desarrollamos una semántica abstracta que aproxima el comportamiento observable del SRTC dado y es adecuada para modelar análisis de flujo de datos. Asumimos el marco de interpretación abstracta para el análisis de la insatisfacibilidad ecuacional tal como está definido en [Alpuente *et al.*, 1995]. Del mismo modo que en la semántica concreta, nosotros probamos que los resultados para las técnicas de aproximación son paramétricos con respecto a la estrategia de *narrowing* empleada $\varphi \in \{inn, out\}$. Comenzamos recordando algunas de las definiciones básicas sobre dominios abstractos y los operadores abstractos asociados; ver [Alpuente *et al.*, 1995, 1996, 2001a,c]. Describimos un operador de consecuencias inmediatas nuevo $T_{\mathcal{R}}^{\sharp}$ capaz de aproximar el operador $T_{\mathcal{R}}$, y como consecuencia, la semántica de punto fijo abstracta $\mathcal{F}^{\sharp}(\mathcal{R})$ aproxima correctamente a la semántica concreta. Del mismo

modo que en el caso estándar dicha semántica permite *ejecutar objetivos* por medio de la unificación abstracta, cuyas respuestas computadas aproximan correctamente las respuestas computadas concretas.

En el Capitulo 6, presentamos métodos para el diagnóstico abstracto e ilustramos su uso mediante ejemplos. Utilizamos el sistema sistema experimental "BUGGY" [Alpuente *et al.*, 2001b] como herramienta de apoyo, así como como la versión, "NO-BUG" [Alpuente *et al.*, 2002c], orientada a la corrección automática de programas. Y finalmente, en el Capítulo 7, discutimos algunos trabajos relacionados y planteamos posibilidades de trabajo futuro. En el apéndice mostramos una sesión con nuestro sistema experimental.

Capítulo 2

Preliminares

En este capítulo revisamos los conceptos fundamentales de la programación lógico funcional necesarios para el desarrollo de esta tesis y para que el documento sea autocontenido. Tras introducir una serie de nociones preliminares en la Sección 2.1, y la sección 2.2, se introduce, en la Sección 2.3, el algoritmo de *narrowing*, el mecanismo de ejecución estándar de los programas lógico funcionales. En los siguientes apartados se presentan refinamientos (*narrowing* impaciente, perezoso y necesario) de la estrategia general de *narrowing*. Las cuestiones más específicas referentes a las diferentes reglas y estrategias de transformación de programas serán introducidas, posteriormente, al principio del capítulo correspondiente. Al final del capítulo presentamos algunos resultados generales, necesarios para desarrollos posteriores. Para un mayor detalle sobre los conceptos introducidos en este capítulo, se puede consultar [Dershowitz y Jouannaud, 1990; Hölldobler, 1989; Klop, 1992; Hanus, 1994b].

2.1 Teoría de dominios

2.1.1 Definiciones de carácter general

Emplearemos los símbolos usuales $\cup, \cap, \setminus$ para denotar la unión, intersección y diferencia de conjuntos, respectivamente. Escribiremos $A \uplus B$ para representar el conjunto $A \cup B$ cuando deseemos poner de manifiesto que se obtiene por unión de conjuntos disjuntos A y B (esto es, $A \cap B = \emptyset$). Dado un conjunto A, $\wp(A)$ denota el conjunto de todos los subconjuntos de A.

Funciones

Dados los conjuntos A y B, una *función* $f : A \to B$ es un subconjunto $f \subseteq A \times B$ que verifica las siguientes condiciones[1]: (1) para todo $a \in A$, existe un $b \in B$ tal que $(a, b) \in f$ y (2) si $(a, b), (a, b') \in f$, entonces $b = b'$ Al conjunto A se le denomina el *dominio* de la función y al conjunto B su *codominio*. Cuando $(a, b) \in f$, escribimos $f(a)$ para referirnos al elemento b. Si no se cumple la condición (1), se dice que f es una función *parcial*. Si no se cumple (2), se dice que f es una función *indeterminista*. Por lo general, hablaremos simplemente de 'funciones', dejando que el contexto del discurso aclare, en su caso, los matices específicos que puedan ser relevantes. Dada una función $f : A \to B$ y un subconjunto $C \subseteq A$, escribimos $f_{|C}$ para denotar la *restricción* $f_{|C} : C \to B$ de la función f al subconjunto C: $f_{|C}(x) = f(x)$ para todo $x \in C$. Dadas dos funciones $g : B \to C$ y $f : A \to B$, la *composición* de ambas funciones se denota como $g \circ f : A \to C$.

Una función f es *inyectiva* si para todo $a, a' \in A$, $f(a) = f(a')$ implica que $a = a'$. Una función f es *sobreyectiva* si para todo $b \in B$ existe un $a \in A$ tal que $f(a) = b$. Una función *biyectiva* (o correspondencia biunívoca) es una función inyectiva y sobreyectiva. Si $f : A \to B$ es biyectiva entonces la función $f^{-1} : B \to A$ definida por $f^{-1}(b) = a$ si y sólo si $f(a) = b$ se denomina función *inversa* de f.

Una función $f : A \to A$ es *idempotente* si para todo $a \in A$, $f(f(a)) = f(a)$. Un elemento $a \in A$ se denomina un *punto fijo* de f si $f(a) = a$. La función $id_A : A \to A$ definida por $id_A(a) = a$ para todo $a \in A$ se denomina la función *identidad* para el conjunto A.

Relaciones binarias

Sea A un conjunto y $R \subseteq A \times A$ una relación binaria sobre A. Escribiremos $a\ R\ b$ en lugar de $(a, b) \in R$ y $a \not R\ b$ en lugar de $(a, b) \notin R$. Decimos que R es: *reflexiva*, si para todo $a \in A$, $a\ R\ a$; *irreflexiva*, si para todo $a \in A$, $a \not R\ a$; *transitiva*, si para todo $a, b, c \in A$, $a\ R\ b$ y $b\ R\ c$ implican $a\ R\ c$; *antisimétrica*, cuando $a\ R\ b$ y $b\ R\ a$ implican $a = b$; *simétrica*, cuando $a\ R\ b$ implica $b\ R\ a$.

Denotamos como R^ϵ el *cierre*[2] *reflexivo* de la relación R: $R^\epsilon = R \cup \{(a, a) \mid a \in A\}$. R^+ es el *cierre transitivo* de R y R^* el *cierre reflexivo-transitivo* de R.

Dadas dos relaciones R, S sobre A, denotamos como $R \circ S$ la relación sobre A obtenida por *composición de* de ambas: para todo $a, b \in A$, $a\ R \circ S\ b$ si y sólo si existe un $c \in A$ tal que $a\ R\ c$ y $c\ S\ b$. Dados dos conjuntos A, B y relaciones binarias

[1] El concepto de función que manejamos aquí se denomina a veces *aplicación* o *función total* en la literatura.

[2] Dado un conjunto A incluido en un conjunto E, y una propiedad P sobre elementos de E, el *cierre* de A respecto a P es el menor conjunto incluido en E que contiene A y satisface P.

$R \subseteq A \times A$ y $S \subseteq B \times B$, una función $h : A \to B$ es un *homomorfismo* entre las relaciones R y S si para todo $a, b \in A$, $a\ R\ b$ implica que $h(a)\ S\ h(b)$.

2.1.2 Teoría de retículos y funciones continuas

Una relación binaria $\leq$ sobre $S(\leq: S \times S)$ es un *orden parcial* si, para cada $x, y \in S$,

$$\begin{array}{rcll} \mathtt{x} & \leq & \mathtt{x} & (\texttt{reflexiva}) \\ \mathtt{x} \leq \mathtt{y}, \mathtt{y} \leq \mathtt{x} & \Rightarrow & \mathtt{x} = \mathtt{y} & (\texttt{antisimetrica}) \\ \mathtt{x} \leq \mathtt{y}, \mathtt{y} \leq \mathtt{z} & \Rightarrow & \mathtt{x} \leq \mathtt{z} & (\texttt{transitiva}) \end{array}$$

Un *conjunto parcialmente ordenado* (poset) $(S, \leq)$ es un conjunto S con un orden parcial $\leq$. S está *totalmente ordenado* si, para cada x, y en S, $x \leq y$, ó $y \leq x$. Una *cadena* es un subconjunto (posiblemente vacío) totalmente ordenado de S.

Un *preorden* es una relación binaria reflexiva y transitiva. Un preorden $\leq$ sobre un conjunto S induce en el mismo conjunto S una relación de equivalencia $\cong$ definida como sigue:

$$\mathtt{x} \approx \mathtt{y} \quad \Leftrightarrow \quad \mathtt{x} \leq \mathtt{y}, \mathtt{y} \leq \mathtt{x}$$

Además, $\leq$ induce en $S_{\cong}$ el orden parcial $\leq_{\cong}$ tal que, para cada $[x]_{\cong}$, $[y]_{\cong} \in S_{\cong}$,

$$[\mathtt{x}]_{\cong} \leq_{\cong} [\mathtt{y}]_{\cong} \quad \Leftrightarrow \quad \mathtt{x} \leq \mathtt{y}$$

Una relación binaria "$<$" es *estricta* si y sólo si es antirreflexiva $(\forall x)(x \not< x)$ y transitiva.

Dado un poset $(S, \leq)$ y $X \subseteq S$, $y \in S$ es una *cota superior* de X si y sólo si, para cada $x \in X$, $x \leq y$. Además, $y \in S$ es la *mínima cota superior* de X si y es una cota superior de X y para cada cota superior y' de X se cumple que $y \leq y'$. La menor cota superior de X con respecto a $(S, \leq)$ se denota por $lub_{\mathcal{S}}(X)$ o $\sqcup_{\mathcal{S}} X$, también escribimos $\sqcup_{\mathcal{S}}\{d_1, \ldots, d_n\}$ como $d_1 \sqcup_{\mathcal{S}} \ldots \sqcup_{\mathcal{S}} d_n$. De forma análoga, $y \in S$ es una *cota inferior* de X si y sólo si, para cada $x \in X$, $y \leq x$. Además, $y \in S$ es la *mayor cota inferior* de X ($glb_{\mathcal{S}}(X)$ o $\sqcap_{\mathcal{S}} X$) si y es una cota inferior de X y para toda cota inferior y' de X, $y' \leq y$. También escribimos $\sqcap_{\mathcal{S}}\{d_1, \ldots, d_n\}$ como $d_1 \sqcap_{\mathcal{S}} \ldots \sqcap_{\mathcal{S}} d_n$. Cuando sea claro en el contexto omitiremos el subíndice S. Es fácil verificar que si el *lub* y *glb* existen entonces son únicos.

Ordenes Parciales Completo y Retículos Un *conjunto dirigido* es un poset para el que cualquier subconjunto con mínimo dos elementos tiene una cota superior en el conjunto. Un *orden parcial completo* (*CPO*) S es un poset, tal que toda la cadena D tiene una menor cota superior, es decir existe $lub_S(D)$. Nótese que cualquier conjunto ordenado bajo la relación identidad forma un *CPO*, por supuesto, sin un elemento ínfimo, tales *CPO*s se denominan *discretos*. A cualquier poset $(S, \leq)$ que no tenga elemento ínfimo, se le puede agregar uno que cumpla tal función. El nuevo poset $S_\perp$ se

obtiene añadiendo el elemento $\bot$ a S y extendiendo el orden $\leq$ como $(\forall x \in S)(\bot \leq x)$. Si S es un *CPO* discreto, entonces $S_\bot$ es un *CPO* con elemento ínfimo.

Un *retículo completo* es un poset $(S, \leq)$ tal que, para cada subconjunto X de S, existe $lub(X)$ y $glb(X)$. El símbolo $\top$ representa el *supremo* de S, $lub(S) = glb(\emptyset)$, y $\bot$ el ínfimo $glb(S) = lub(\emptyset)$ de S. Los elementos de un retículo completo son considerados como puntos de información y el orden como una relación de aproximación entre ellos. Así pues, $x \leq y$ quiere decir que x aproxima a y (o que x tiene menos o la misma información que y). Por esto, $\bot$ es el punto con la menor información. Es fácil comprobar que, para cualquier conjunto S, $\wp(S)$ es un retículo completo bajo el orden de inclusión ($\subseteq$), donde *lub* es la unión, *glb* es la intersección, el elemento supremo es S y el ínfimo es $\emptyset$. $(\wp(S))_\bot$ es también un retículo completo.

Dado un retículo completo $(L, \leq)$, el conjunto de todas las funciones parciales $F = [S \rightharpoonup L]$ hereda la estructura del retículo completo L, se define $\mathtt{f} \preccurlyeq \mathtt{g} := \forall \mathtt{x} \in \mathtt{S}.\mathtt{f}(\mathtt{x}) \leq \mathtt{g}(\mathtt{x}), (\mathtt{f} \sqcup \mathtt{g})(\mathtt{x}) := \mathtt{f}(\mathtt{x}) \sqcup \mathtt{g}(\mathtt{x}), (\mathtt{f} \sqcap \mathtt{g})(\mathtt{x}) := \mathtt{f}(\mathtt{x}) \sqcap \mathtt{g}(\mathtt{x}), \bot_\mathtt{F} := \lambda \mathtt{x} \in \mathtt{S}.\bot_\mathtt{L}$ y $\top_F := \lambda x \in S.\ \top_L$.

Sean D, E dos CPOs. El conjunto de todas las funciones continuas de D a E forman un orden parcial completo, con la conocida relación de orden "punto a punto".

Funciones continuas Sean $(L, \leq)$ y $(M, \sqsubseteq)$ retículos (completos), una función $f : L \to M$ es *monótona* si y sólo si

$$(\forall \mathtt{x}, \mathtt{y} \in \mathtt{L})(\mathtt{x} \leq \mathtt{y} \quad \Rightarrow \quad \mathtt{f}(\mathtt{x}) \quad \sqsubseteq \quad f(y)).$$

Además, f es *continua* si y sólo si, para cada cadena no vacía $D \subseteq L$,

$$f(\sqcup_L D) \quad = \quad \sqcup_\mathtt{M} \mathtt{f}(\mathtt{D})$$

Toda función continua es también monótona, ya que $\mathtt{x} \leq \mathtt{y} \Rightarrow \sqcup_\mathtt{M}\{\mathtt{f}(\mathtt{x}), \mathtt{f}(\mathtt{y})\} = \mathtt{f}(\sqcup_\mathtt{L}\{\mathtt{x}, \mathtt{y}\}) = \mathtt{f}(\mathtt{y}) \Rightarrow \mathtt{f}(\mathtt{x}) \sqsubseteq \mathtt{f}(\mathtt{y})$.

Los órdenes parciales completos corresponden a tipos de datos y las funciones computables se modelan como funciones continuas entre ellos.

Se puede demostrar que la composición de funciones monótonas, (continuas) es monótona (continua).

Para expresar de una forma matemática que dos estructuras son "esencialmente la misma" y establecer una relación entre ellas, utilizamos el concepto de isomorfismo. Una función continua $f : D \to E$ entre los *CPO*s D y E, se dice que es un *isomorfismo* si existe una función continua $g : E \to D$ tal que $g \circ f = Id_D$ y $f \circ g = Id_E$. Dos *CPO*s isomorfos son esencialmente el mismo excepto por el renombre de los elementos. Se puede demostrar que una función $f : D \to E$ es un isomorfismo si y sólo si f es biyectiva y, para todo $\mathtt{x}, \mathtt{y} \in \mathtt{D}, \mathtt{x} \leq_\mathtt{D} \mathtt{y} \Leftrightarrow \mathtt{f}(\mathtt{x}) \leq_\mathtt{E} \mathtt{f}(\mathtt{y})$.

2.1.3 Teoría del punto fijo

Dado un poset $(S, \leq)$ y una función $f : S \to S$, un *punto fijo* de f es un elemento $x \in S$ tal que $f(x) = x$. Un *pre-punto fijo* de f es un elemento $x \in S$ tal que $f(x) \leq x$ y, de forma dual, un *post-punto fijo* es un elemento $x \in S$ tal que $x \leq f(x)$. Además, decimos que $x \in S$ es el *menor punto fijo* de f (se denota como $lfp(f)$) si y sólo si x es un punto fijo de f y, para todo punto fijo y de f, se cumple $x \leq y$. De forma dual definimos el *mayor punto fijo* ($gfp(f)$).

El teorema fundamental de Knaster-Tarski establece que el conjunto de todos los puntos fijos de una función monótona f es un retículo completo.

Teorema 2.1.1 (Teorema del Punto Fijo) *Una función monótona f sobre un retículo completo $(L, \leq)$ tiene un menor y un mayor punto fijo. Además,*

$$lfp(f) = \sqcap\{x \mid f(x) \leq x\} = \sqcap\{x \mid x = f(x)\}$$
$$lfp(f) = \sqcup\{x \mid f(x) \leq x\} = \sqcup\{x \mid x = f(x)\}$$

El teorema de Knaster-Tarski es importante para este trabajo por cuanto se aplica a cualquier función monótona sobre un retículo completo. Sin embargo, la mayoría de las veces se hará referencia al menor punto fijo de una función continua que se construirá con las técnicas explicadas anteriormente, como límites superiores de cadenas en *CPO*s. También es útil establecer algunas notaciones y resultados adicionales acerca del punto fijo de funciones continuas definidas sobre retículos (completos).

Primero que todo, un *ordinal* es un conjunto, donde todo elemento es un ordinal también y la clase de ordinales está ordenada por la relación de pertenencia ($\alpha < \beta$ significa que $\alpha \in \beta$). En consecuencia, todo ordinal coincide con el conjunto formado por todos los ordinales más pequeños que él. Los ordinales menores son 0, $1 := \{0\}$, $2 := \{0, \{0\}\}$, etc. Intuitivamente, la clase de ordinales es la secuencia transfinita $0 < 1 < 2 < \ldots < \omega < \omega + 1 < \ldots < \omega + \omega < \ldots < \omega^{\omega}$, etc. Los ordinales se suelen denotar por letras griegas. Un ordinal γ es un *ordinal infinito* si no es 0 o el sucesor de un ordinal; por lo tanto, si $\beta < \gamma$, entonces existe σ tal que $\beta < \sigma < \gamma$. El *primer ordinal infinito*, que es equipotente con el conjunto de los números naturales, se denota como ω. A menudo, en las definiciones de *CPO* y de continuidad, se usan conjuntos dirigidos en lugar de cadenas. Es posible demostrar, que bajo ciertas condiciones, las definiciones resultan equivalentes.

Las *potencias ordinales* de una función monótona $T : S \to S$ sobre un *CPO* S se definen como

$$T{\uparrow}\alpha(x) := \begin{cases} x & \texttt{si } \alpha = 0 \\ T(T \uparrow (\alpha - 1))(x) & \texttt{si } \alpha \text{ es un sucesor ordinal} \\ \sqcup\{T \uparrow \beta(x) \mid \beta < \alpha\} & \texttt{si } \alpha \text{ es un ordinal infinito} \end{cases}$$

Si $x = \emptyset$ simplemente escribimos $T \uparrow n$ para mencionar la n–esima iteración de T que comienza en $\emptyset$. Para el caso especifico del menor ordinal infinito ω escribiremos $T \uparrow \omega(\mathcal{I}) = \cup_{n=0}^{\infty} T \uparrow n(\mathcal{I})$ para facilitar la notación.

El siguiente resultado importante se atribuye a Kleene y nos da una construcción explícita del menor punto fijo de una función continua f en un *CPO* D.

Teorema 2.1.2 *Sea $f : D \to D$ una función continua en un CPO D y sea $d \in D$ un pre-puntofijo de f. Entonces $\sqcup\{f \uparrow n(d) \mid n \leq \omega\}$ es el menor punto fijo de f más grande que d. En particular, $f \uparrow \omega$ es el menor pre-punto fijo y menor punto fijo de f.*

2.2 Conceptos básicos

2.2.1 Términos y ecuaciones

Denotamos por $\mathcal{V}$ un conjunto infinito (numerable) de variables y por Σ un conjunto de símbolos de función f/n o *signatura*, cada uno con una aridad n asociada. $\tau(\Sigma \cup \mathcal{V})$ denota el conjunto de *términos* construidos usando los símbolos de Σ y $\mathcal{V}$. $\tau(\Sigma)$ denota el conjunto de *términos básicos* ("*ground*", sin variables). $\tau(\Sigma)$ es el habitual universo de Herbrand sobre Σ y lo denotaremos por $\mathcal{H}$. $\mathcal{B}$ denota la base de Herbrand y es el conjunto de todas las ecuaciones básicas que pueden ser construidas con elementos de $\mathcal{H}$. Una Σ–ecuación tiene la forma $s = t$ donde $s, t \in \tau(\Sigma \cup \mathcal{V})$. La palabra *expresión* la utilizamos para referirnos a un objeto sintáctico general, hasta el momento una ecuación, un término o una variable. Es decir, e es una expresión si $e \in \mathcal{V}, e \in \tau(\Sigma \cup \mathcal{V})$ ó $e \in \mathcal{B}$.

El conjunto de variables que aparecen en una expresión e se denota por $\mathcal{V}ar(e)$, una expresión t es *básica* si $\mathcal{V}ar(t) = \emptyset$. Adicionalmente, $[s]$ denota el conjunto de instancias básicas del objeto sintáctico e. Una variable es *fresca* si es una variable nueva que no ha sido empleada con anterioridad. Una expresión es *lineal* si no contiene múltiples apariciones de una misma variable. El símbolo " $\overline{\ }$ " denota una secuencia finita símbolos. Similarmente, denotamos por $\overline{t_n}$ a la *lista de objetos sintácticos* $t_1, \ldots, t_n$. La *identidad de objetos sintácticos* se denota mediante el operador relacional "$\equiv$".

Los términos se representan como árboles etiquetados a la manera usual, en donde las etiquetas de los nodos son los símbolos que forman el término. Las *posiciones* (*ocurrencias*) de un término t se representan mediante secuencias de números naturales que se emplean para indentificar un camino de acceso a cada subtérmino de t. Las posiciones están ordenadas por el orden prefijo "$\leq$": de manera que $p \leq q$, si existe un w tal que $p.w = q$, donde $p.w$ denota la concatenación de secuencias p y w. Cuando dos posiciones no esten relacionadas diremos que son *disjuntas* y lo indica-

remos mediante el símbolo "$\not\leq$". Denotamos la secuencia vacía mediante el símbolo "Λ". $O(t)$ denota el conjunto de posiciones del términoconjunto(s)!posiciones de un término t y $\overline{O}(t)$ denota el conjunto de posiciones no variables del término t. $O(t)$ se define recursivamente como sigue:

$$O(t) = \begin{cases} \{\Lambda\} & \text{si } t \in \mathcal{V} \\ \{\Lambda\} \cup \{i.p \mid 1 \leq i \leq n \ \wedge \ p \in O(t_i)\} & \text{si } t = f(t_1, \ldots, t_n) \end{cases}$$

$t|_p$ es el subtérmino de t que ocurre en la posición p. $t[p]$ denota la etiqueta asociada al árbol del término t en la posición $p \in O(t)$, coincide con el símbolo más externo del subtérmino t_p. $t[s]_p$ es el término resultante de reemplazar, en la posición p del término t, el subtérmino $t|_p$ por s. Estas nociones son extendidas de manera natural a secuencias de ecuaciones. Por ejemplo, el conjunto de posiciones no variables de una secuencia de ecuaciones $g \equiv (e_1, \ldots, e_n)$ puede ser definido como: $\overline{O}(g) = \{i.u \mid u \in \overline{O}(e_i), i = 1, \ldots, n\}$. Usaremos $\top$ como notación genérica para una secuencia de la forma $true, \ldots, true$.

Usamos la función estándar *depth* para denotar la máxima profundidad de un término. Así,

$$depth(t) = \begin{cases} 1 & \text{si } t \text{ es una constante o una variable} \\ 1 + max(\{\overline{depth(t_n)}\}) & \text{si } t \text{ es de la forma } f(\overline{t_n}),\ n > 0. \end{cases}$$

2.2.2 Sustituciones y unificadores sintácticos

Una *sustitución* es una función finita $\sigma : \mathcal{V} \to \tau(\Sigma \cup \mathcal{V})$ tal que el conjunto $Dom(\sigma) = \{x \in \mathcal{V} \mid \sigma(x) \neq x\}$ es finito, se denomina *dominio* de σ. La función identidad sobre $\mathcal{V}$ se conoce como *sustitución vacía* y la denotamos por ϵ. Desde el punto de vista de la teoría de conjuntos, denotamos la *sustitución* σ por

$$\{x_1/t_1, \ldots, x_n/t_n\}$$

o en algunos contextos como

$$\{x_1 \mapsto t_1, \ldots, x_n \mapsto t_n\}$$

donde $\sigma(x_i) = t_i$ para $i = 1, \ldots, n$ y $Dom(\sigma) = \{x_1, \ldots, x_n\}$, y $\sigma(x) = x$ para cualquier otra variable x. Además, denotamos por $Ran(\sigma)$ las variables que aparecen en $\{t_1, \ldots, t_n\}$, es decir, si $\sigma = \{x_1/t_1, \ldots, x_n/t_n\}$, entonces $Ran(\sigma) = \mathcal{V}ar(t_1) \cup \ldots \cup \mathcal{V}ar(t_n)$. Para referirnos al conjunto formado por los términos $t_1, \ldots, t_n$ escribiremos $\mathcal{C}od(\sigma)$. El par x_i/t_i se denomina enlace. Una sustitución σ donde todos los t_i son básicos se denomina básica, es decir, si para cada $x \in \mathcal{D}om(\sigma)$ se cumple que $\sigma(x)$ es un término básico. La expresión $E\sigma$, llamada *instancia* de E, es equivalente a $\sigma(E)$ y es la expresión que resulta de reemplazar simultaneamente las variables x de E por los correspondientes términos $\sigma(x)$. Las sustituciones se extienden a morfismos sobre términos como $f(\overline{t_n})\sigma = f(\overline{t_n\sigma})$ para todo término $f(\overline{t_n})$.

La *sustitución compuesta* $\vartheta\sigma$ es la composición de las sustituciones ϑ y σ, está definida como $\vartheta\sigma(x) = (x\vartheta)\sigma$. Un sustitución σ es *idempotente* si $\sigma\sigma = \sigma$ y esto es equivalente a $Dom(\sigma) \cap Ran(\sigma) = \emptyset$ y denotamos por Sub el conjunto de todas las sustituciones idempotentes definidas en $\tau(\Sigma \cup \mathcal{V})$. Consideramos el *preorden* "$\leq$" (*subsumción o generalidad relativa*) habitual entre sustituciones: $\theta \leq \sigma$ sii $\exists\gamma.\ \sigma = \theta\gamma$ y decimos que θ es *más general* que σ. Este preorden induce un (pre-)orden parcial sobre términos dado por $t \leq t'$ si y sólo si $\exists\gamma.\ t' = t\gamma$, es decir si t' es una instancia de t. La correspondiente relación de equivalencia inducida se denomina *varianza*. En adelante $(Sub, \leq)$ denota el retículo de las sustituciones idempotentes, que por abuso, en algunos casos simplemente escribimos Sub.

Un *renombramiento* es una sustitución ρ para la cual existe la inversa ρ^{-1} tal que $\rho\rho^{-1} = \rho^{-1}\rho = \epsilon$. Dos términos t y t' son *variantes* (uno de otro), si existe un renombramiento ρ tal que $t\rho = t'$, es decir, dos términos t y t' son variantes si t es instancia de t' y viceversa. La *restricción*, $\vartheta_{\restriction S}$, de una sustitución ϑ a un conjunto S de variables se define como $\vartheta_{\restriction S}(x) = \vartheta(x)$ si $x \in S$ y $\vartheta_{\restriction S}(x) = x$ si $x \notin S$. Si no hay lugar a ambiguedad, escribimos $\theta_{\restriction E}$ en lugar de $\theta_{\restriction \mathcal{V}ar(E)}$ cuando E una expresión. Escribimos $\theta = \vartheta\ [W]$ si $\theta_{\restriction W} = \vartheta_{\restriction W}$, y $\theta \leq \vartheta\ [W]$ denota la existencia de una sustitución γ tal que $\theta\gamma = \vartheta\ [W]$.

Un *unificador* de dos términos s y t es una sustitución σ tal que $s\sigma = t\sigma$. Si un unificador θ de los términos t y s cumple que $\theta \leq \sigma$ para cualquier otro unificador σ, decimos que θ es el *unificador más general (mgu)* de t y s, escribimos $\theta = mgu(t, s)$. El mgu es único salvo renombramientos. Más aun, si dos términos son unificables entonces existe un unificador más general idempotente entre ellos. Estas nociones pueden ser extendidas a otros objetos sintácticos de la manera obvia. Un conjunto de ecuaciones E es unificable, si existe una sustitución ϑ tal que para toda ecuación $s = t$ en E tenemos que $s\vartheta = t\vartheta$. Decimos que ϑ es un *unificador* de E [Lassez *et al.*, 1988].

Describimos el retículo de ecuaciones sintácticas [Codish *et al.*, 1991]. Eqn denota el conjunto finito de ecuaciones sobre términos posiblemente cuantificadas existencialmente [Maher, 1988]. Además, $E \leq E'$ si E' implica lógicamente a E. El elemento $fail$ denota el conjunto de ecuaciones insatisfacible, el cual implica lógicamente a todos los conjuntos de ecuaciones. Del mismo modo, el conjunto vacío de ecuaciones, denotado por $true$, es implicado por todos los elementos de Eqn. Nótese que $(Eqn, \leq)$ es un retículo ordenado por $\leq$ cuyo elemento mínimo es $true$ y el elemento máximo es $fail$. Los elementos de Eqn son vistos como conjunciones de ecuaciones (cuantificadas) y tratados módulo equivalencia lógica, el preorden $\leq$ se extiende a un orden parcial sobre el retículo.

Un conjunto de ecuaciones está en forma *resuelta* si es $fail$ o tiene la forma $\exists y_1 \ldots \exists y_m.\ \{x_1 = t_1, \ldots, x_n = t_n\}$, donde cada x_i es una variable distinta que

no ocurre en alguno de los términos t_i y cada y_i ocurre en algún t_j. Todo conjunto de ecuaciones E puede ser transformado en una forma resulta equivalente, $solve(E)$. Existe un isomorfismo natural entre el conjunto de sustituciones $\theta = \{x_1/t_1, \ldots, x_n/t_n\}$ y los conjuntos de ecuaciones sin cuantificar $\widehat{\theta} = \{x_1 = t_1, \ldots, x_n = t_n\}$. Una sustitución $\theta = \{x_1/t_1, \ldots, x_n/t_n\}$ es un *unificador* de un conjunto de ecuaciones E sii $\widehat{\theta} \Rightarrow E$. El $mgu(E)$ denota el *unificador más general* de un conjunto de ecuaciones sin cuantificar E. Escribimos $mgu(\{s_1 = t_1, \ldots, s_n = t_n\}, \{s'_1 = t'_1, \ldots, s'_n = t'_n\})$ para denotar el unificador más general del conjunto ecuaciones $\{s_1 = s'_1, t_1 = t'_1, \ldots, s_n = s'_n, t_n = t'_n\}$. Para cualquier conjunto de ecuaciones no cuantificado existe un unificador más general [Lassez *et al.*, 1988], mientras que para el conjunto de ecuaciones cuantificadas esto no es cierto en general [Maher, 1990]. Por abuso en ocasiones trataremos los conjuntos de ecuaciones como secuencias.

2.2.3 Programas

Un *sistema de reescritura de términos condicional* (SRTC) es un par $(\Sigma, \mathcal{R})$, donde $\mathcal{R}$ es un conjunto finito de reglas de reescritura de la forma $(\lambda \rightarrow \rho \Leftarrow C)$, donde $\lambda, \rho \in \tau(\Sigma \cup \mathcal{V})$ y $\lambda \notin \mathcal{V}$. Cuando $\mathcal{V}ar(r) \cup \mathcal{V}ar(C) \subseteq \mathcal{V}ar(l)$ se dice que los programas son de tipo 1 o 1-SRTC. Por su parte, los 2-SRTC satisfacen $\mathcal{V}ar(r) \subseteq \mathcal{V}ar(l)$ y los 3-SRTC cumplen $\mathcal{V}ar(r) \subseteq \mathcal{V}ar(l) \cup \mathcal{V}ar(C)$ [Middeldorp y Hamoen, 1994]. La condición C es una secuencia (posiblemente vacía) de ecuaciones $e_1, \ldots, e_n$, $n \geq 0$. Las variables en ρ o C que no aparecen en λ reciben el nombre de *variables extras*. Cuando una regla de reescritura no posee condición, escribimos simplemente $\lambda \rightarrow \rho$. Cuando las condiciones de todas las reglas de $\mathcal{R}$ son vacías, tenemos que $(\Sigma, \mathcal{R})$ es un sistema de reescritura de términos incondicional (SRT). Escribiremos a menudo simplemente $\mathcal{R}$ en lugar de $(\Sigma, \mathcal{R})$ para denotar un SRTC. Escribimos $r \ll \mathcal{R}$ para denotar que r es una nueva variante de una regla de $\mathcal{R}$ tal que r contiene solamente *variables frescas* , es decir no contiene variables que se encuentren previamente durante la computación (aparte estandarizadas).

Un *objetivo ecuacional* es una regla sin cabeza o, equivalentemente, una secuencia de ecuaciones. Denotamos *Goal* el conjunto de objetivos ecuacionales $\Leftarrow g$ que, en lo que sigue, denotamos simplemente por g.

Se debe notar que, en el marco de los programas incondicionales, todavía es posible tratar con reglas con condiciones y objetivos formados por conjunciones de ecuaciones, pero en este caso las condiciones en las reglas deben tratarse usando las funciones predefinidas $\wedge$ y $\Rightarrow$ (a las que pueden añadirse también las construcciones `if_then_else` y `case_of`) que se reducen usando reglas de definición estándar [Moreno-Navarro y Rodríguez-Artalejo, 1992], i.e.:

$$true \wedge true \rightarrow true$$
$$true \Rightarrow x \rightarrow x$$

Dado un término t, el subtérmino t_p en la posición p de t que coincide con la parte izquierda instanciada $l\sigma$ de una regla de reducción $(l \to r \Leftarrow C)$ se denomina *redex* (*reducible expression*).

Dado un SRTC $(\Sigma, \mathcal{R})$, asumimos que la signatura Σ está particionada en dos conjuntos disjuntos $\Sigma = \mathcal{C} \uplus \mathcal{D}$, donde $\mathcal{D} = \{f \mid (f(t_1, \ldots, t_n) \to r \Leftarrow C) \in \mathcal{R}\}$ y $\mathcal{C} = \Sigma \setminus \mathcal{D}$. Los símbolos de $\mathcal{C}$ se denominan *constructores* y los símbolos de $\mathcal{D}$ se denominan *funciones definidas* llamadas también operaciones. Escribimos $c/n \in \mathcal{C}$ y $f/n \in \mathcal{F}$ para referirnos a constructores y operaciones n-arios, respectivamente. A elementos de $\tau(\mathcal{C} \cup \mathcal{V})$ se les denomina *términos constructores*. Si σ es una sustitución y $\mathcal{C}od(\sigma) \subseteq \tau(\mathcal{C} \cup \mathcal{V})$ entonces σ se dice que es una *sustitución constructora*. Un *patrón* es un término de la forma $f(\bar{d})$ donde $f/n \in \mathcal{D}$ y $\bar{d}$ son términos constructores.

Un término s se *reescribe condicionalmente* a un término t, y lo denotamos por $s \to_{\mathcal{R}} t$, si existe una regla $(\lambda \to \rho \Leftarrow s_1 = t_1, \ldots, s_n = t_n) \in \mathcal{R}$, una posición $p \in O(s)$, una sustitución σ tal que $s|_p \equiv \lambda\sigma$, $t \equiv s[\rho\sigma]_p$ y, para todo $i = 1, \ldots, n$, existe un término w_i tal que $s_i\sigma \to^*_{\mathcal{R}} w_i$ y $t_i\sigma \to^*_{\mathcal{R}} w_i$, donde $\to^*_{\mathcal{R}}$ denota el cierre reflexivo y transitivo de la relación $\to_{\mathcal{R}}$. Cuando no haya lugar a confusión, omitiremos el subíndice $\mathcal{R}$ de la relación de reescritura $\to_{\mathcal{R}}$. $\to^+$ denotan el *cierre transitivo* de la relación $\to$.

Un término s es *irreducible* o está en *forma normal* si no existe ningún término t tal que $s \to t$. Denotamos por $s{\downarrow}$ la forma normal de s. Un término s está en *forma normal en cabeza* si no se reduce a un redex[3]. Una sustitución σ se dice *normalizada* si $x\sigma$ está en forma normal para todo $x \in Dom(\sigma)$. Un SRTC $\mathcal{R}$ se dice *noetheriano* cuando no existe una secuencia infinita de términos de la forma $s_0 \to_{\mathcal{R}} s_1 \to_{\mathcal{R}} \ldots$; $\mathcal{R}$ se dice *confluente* si para todo término s tal que $s \to^*_{\mathcal{R}} t_1$ y $s \to^*_{\mathcal{R}} t_2$, entonces $t_1 \downarrow t_2$, es decir, existe un término t tal que $t_1 \to^*_{\mathcal{R}} t$ y $t_2 \to^*_{\mathcal{R}} t$. Un SRTC $\mathcal{R}$ noetheriano y confluente se denomina *canónico* [Klop, 1992].

Un SRTC $\mathcal{R}$ es *lineal por la izquierda* si el término l es lineal para toda regla $l \to r \Leftarrow C \in \mathcal{R}$. Un SRTC está *basado en constructores (CB)* si para toda regla $l \to r \Leftarrow C \in \mathcal{R}$, l es un patrón. Diremos que $\mathcal{R}$ es *ortogonal* si, además de ser lineal por la izquierda, para toda regla $l \to r \Leftarrow C \in \mathcal{R}$ y para cada subtérmino no variable $l|_p$ de l no existe ninguna regla $l' \to r' \in \mathcal{R}$ tal que $l|_p$ y l' unifiquen (*no solapamiento*)[4].

Una clase importante de sistemas de reescritura condicionales son los llamados *sistemas decrecientes*. Decimos que un SRTC es decreciente si existe una extensión bien fundada $\succ$ de la relación de reescritura $\to_{\mathcal{R}}$ con las siguientes propiedades:

[3]Nótese que todo término encabezado por un símbolo constructor está obviamente en forma normal en cabeza.

[4]Cuando $\mathcal{R}$ es CB, entonces la condición de ortogonalidad se simplifica, ya que al ser patrones las partes izquierdas de todas sus reglas, es suficiente con que no existan pares de reglas $l \to r \Leftarrow C$ y $l' \to r' \Leftarrow C$ en $\mathcal{R}$ tal que l y l' unifiquen.

1. $\succ$ cumple la *propiedad de subtérmino*, es decir, $t \succ t|_p$ para todo $p \in O(t) - \{\Lambda\}$, y

2. si $(l \to r \Leftarrow C) \in \mathcal{R}$ y σ es una sustitución, entonces $l\sigma \succ r\sigma$ y, para todo $s = t \in C$, $l\sigma \succ s\sigma$ y $l\sigma \succ t\sigma$.

Una teoría ecuacional de Horn $\mathcal{E}$ consiste en un conjunto finito cláusulas ecuacionales de Horn de la forma $(l = r) \Leftarrow C$. Un *objetivo ecuacional* es una secuencia de ecuaciones $\Leftarrow C$, es decir, una cláusula ecuacional de Horn sin cabeza. Usualmente omitimos el símbolo $\Leftarrow$. Un objetivo de la forma $\Leftarrow x = y$, con $x, y \in V$ es llamado un *objetivo trivial*. Una teoría ecuacional de Horn $\mathcal{E}$, que cumple que $\lambda \notin V$ y $\mathcal{V}ar(\rho) \subseteq \mathcal{V}ar(\lambda)$ para cada cláusula $(l = r) \Leftarrow C$, puede ser vista como un SRTC $\mathcal{R}$, donde la reglas son las cabezas (implícitamente orientadas de izquierda a derecha) y las condiciones los respectivos cuerpos.

Cada teoría ecuacional de Horn $\mathcal{E}$ genera la relación congruencia más pequeña $=_\mathcal{E}$ que se denomina $\mathcal{E}$-*igualdad* sobre el conjunto de términos $\tau(\Sigma \cup V)$ (la menor teoría ecuacional que contiene todas consecuencias lógicas de $\mathcal{E}$ bajo la *relación de inferencia*[5] $\models$ que obedece los axiomas de la igualdad, Eq.[6]

de la igualdad para $\mathcal{E}$). $\mathcal{E}$ es una presentación o axiomatización de $=_\mathcal{E}$. Por abuso de notación, algunas veces nos referiremos a la teoría ecuacional $\mathcal{E}$ para denotar la teoría axiomatizada por $\mathcal{E}$. $\mathcal{H}/\mathcal{E}$ denota la partición más fina $\tau(\Sigma)/=_\mathcal{E}$ inducida por $=_\mathcal{E}$ sobre el conjunto de términos básicos $\tau(\Sigma)$. $\mathcal{H}/\mathcal{E}$ usualmente denota el *álgebra inicial* de $\mathcal{E}$ [Klop, 1992]. Satisfacibilidad en $\mathcal{H}/\mathcal{E}$ es $\mathcal{E}$-$unificabilidad$, esto es, dado un conjunto de ecuaciones E, E es $\mathcal{E}$-unificable sii existe una sustitución σ tal que $\mathcal{E} \models E\sigma$, es decir, para toda ecuación $(s = t) \in E$, se cumple $s\sigma =_\mathcal{E} t\sigma$ y por tanto $\mathcal{E} \models (s\sigma = t\sigma)$. [Klop, 1992]. La sustitución σ es llamada un $\mathcal{E}$-unificador de E. $=_\mathcal{R}$ es el cierre reflexivo, simétrico, y transitivo de $\to_\mathcal{R}$. Si $\mathcal{E}$ es el conjunto de ecuaciones (condicionales) correspondientes a $\mathcal{R}$, entonces $=_\mathcal{E}$ y $=_\mathcal{R}$ coinciden. Por medio de esta correspondencia, la noción de $\mathcal{R}$-unificación es definida implícitamente. Dado un conjunto de variables $W \subseteq \mathcal{V}$, la $\mathcal{R}$-igualdad se extiende a sustituciones de la forma estándar: $\sigma =_\mathcal{R} \theta\ [W]$ sii $x\sigma =_\mathcal{R} x\theta$, $\forall x \in W$. Decimos que $\theta \leq_\mathcal{R} \sigma\ [W]$ si existe una sustitución γ tal que $\theta\gamma =_\mathcal{R} \sigma\ [W]$, esto es $x\theta\gamma =_\mathcal{R} x\sigma$ para todo $x \in W$. Un conjunto de $\mathcal{E}$-unificadores S de un conjunto de ecuaciones E se dice *completo* sii todo $\mathcal{E}$-unificador σ de E se puede factorizar como $\sigma =_\mathcal{E} \theta\gamma\ [\mathcal{V}ar(E)]$, donde $\theta \in S$.

Asumimos la existencia de, al menos, un tipo primitivo *Bool* que contiene los constructores booleanos constantes (de aridad 0) $true$ y $false$. Asumimos además

[5] Traducción de *Entailment relation*.

[6] El conjunto Eq de axiomas de la igualdad para un programa dado $\mathcal{R}$ son:

$x = x \Leftarrow$	(reflexiva)
$x = y \Leftarrow y = x$	(simetría)
$x = z \Leftarrow x = y, y = z$	(transitividad)
$f(x_1, \ldots, x_n) = f(y_1, \ldots, y_n) \Leftarrow x_1 = y_1, \ldots, x_n = y_n$	(f-sustitutividad), con $f/n \in \Sigma$.

que Σ contiene también el símbolo primitivo de función binario "=" para representar la igualdad, escrito en notación infija, que nos permite interpretar las ecuaciones $s = t$ como términos, con $s, t \in \tau(\Sigma \cup \mathcal{V})$. El término *true* se considera también una ecuación. En algunos contextos, debido a la presencia de funciones no terminantes, se asume un nuevo tipo de símbolo primitivo de función binario "$\approx$" para representar la *igualdad estricta* entre dos términos, que también se escribe en notación infija. La igualdad estricta $\approx$ considera dos términos iguales sólo si éstos se reducen a un mismo término básico y formado únicamente por símbolos constructores.

Un redex $t|_p$ de un término t es un *redex más externo* si no existe otro redex $t|_q$ de t tal que $q < p$. Un paso de reescritura dado sobre un redex más externo se denomina *paso de reescritura más externo (outermost)*. La reescritura más externa es la base operacional de los lenguajes funcionales perezosos.

2.3 El procedimiento de *narrowing*

En este apartado presentamos el mecanismo de computación asociado al lenguaje que nos permite obtener las soluciones para un objetivo en un programa dado. A continuación presentamos la extensión del cálculo de reescritura sustituyendo el emparejamiento por la unificación de términos. Consideremos un programa $\mathcal{R}$. Si queremos evaluar una función cuyos argumentos son básicos (*ground*), basta emparejar la llamada a función $f(t_1, \ldots, t_n)$ con la parte izquierda l de alguna regla $(l \rightarrow r \Leftarrow C)$ del programa cuya condición se verifica en el sentido mencionado en el apartado anterior, realizando una secuencia de pasos de reescritura aplicados a los términos resultantes. Por otro lado, si tenemos una llamada a función cuyos argumentos contienen posiblemente variables, entonces generalmente es necesario instanciar estas variables a los términos apropiados para poder aplicar un paso de reescritura. Esto se puede llevar a cabo usando unificación en lugar de emparejamiento en el paso de reescritura, y recibe el nombre de *narrowing* [Fay, 1979; Slagle, 1974]. Informalmente, reducir por *narrowing* una expresión consiste en aplicarle una sustitución tal que la expresión resultante se pueda reducir, y entonces reducirla [Hullot, 1980]. El procedimiento de *narrowing* no sólo subsume la *reescritura*, sino también la *resolución SLD* de los programas lógicos. Formalmente, sea $\mathcal{R}$ un programa y g un objetivo ecuacional, decimos que g se reduce por *narrowing condicional* a g' si:

1. existe una posición no variable u de g (es decir, $u \in \overline{O}(g)$);
2. una regla $r \equiv (\lambda \rightarrow \rho \Longleftarrow C) \ll \mathcal{R}$
3. una sustitución σ tal que $g_{|_u}\sigma = \lambda\sigma$ y
4. $g' \equiv (C, g[\rho]_u)\sigma$

En este caso, escribimos $g \overset{u,r,\sigma}{\leadsto} g'$, $g \overset{u,\sigma}{\leadsto} g'$ o simplemente $g \overset{\sigma}{\leadsto} g'$, si queda claro por el contexto. Generalmente, un *paso de narrowing* se consigue unificando un subtérmino no variable del objetivo con la parte izquierda de la cabeza de una regla de reescritura, reemplazando entonces el subtérmino instanciado por la correspondiente parte derecha de la regla instanciada, y para formar el objetivo resultante agregamos la condición instanciada de la respectiva regla. Veamos un ejemplo.

Ejemplo 1 Dado el programa $\mathcal{R}$:

$$\begin{array}{rcl} \texttt{add}(\texttt{0},\texttt{x}) & \to & \texttt{x}. \\ \texttt{add}(\texttt{s}(\texttt{x}),\texttt{y}) & \to & \texttt{s}(\texttt{add}(\texttt{x},\texttt{y})). \\ \texttt{double}(\texttt{0}) & \to & \texttt{0}. \\ \texttt{double}(\texttt{s}(\texttt{x})) & \to & \texttt{s}(\texttt{s}(\texttt{double}(\texttt{x}))). \end{array}$$

se puede construir la siguiente secuencia de pasos de narrowing para el término $\texttt{double}(\texttt{add}(\texttt{x},\texttt{y}))$[7]:

$$\texttt{double}(\underline{\texttt{add}(\texttt{x},\texttt{y})}) \overset{\{\texttt{x}\mapsto\texttt{0}\}}{\underset{\{\texttt{y}\mapsto\texttt{0}\}}{\leadsto}} \underline{\texttt{double}(\texttt{y})} \\ \texttt{0}$$

con sustitución computada $\{\texttt{x} \mapsto \texttt{0}, \texttt{y} \mapsto \texttt{0}\}$ (restringida a las variables del objetivo inicial). Nótese que esta secuencia de reducciones no puede obtenerse usando reescritura, ya que la expresión $\texttt{add}(\texttt{x},\texttt{y})$ no *empareja* con (no es instancia de) la parte izquierda de ninguna regla.

A menudo se impone que la sustitución σ sea el *mgu* de $t|_p$ y l en la definición de *narrowing*, aunque dicha condición puede relajarse en ocasiones exigiendo únicamente que σ sea un unificador arbitrario (no necesariamente el más general) de $t|_p$ y l. Concretamente, [Padawitz, 1988] distingue entre *narrowing* y *narrowing más general* dependiendo de si σ es un *unificador arbitrario* o necesariamente el más general. La restricción de que σ sea un *mgu* es la base de la mayoría de las variantes de *narrowing* conocidas (como las que veremos en los apartados siguientes) debido fundamentalmente a la ventaja de que los unificadores más generales son computables de forma determinista, mientras que en general pueden existir muchos unificadores independientes. Sin embargo, eliminar la restricción de que los unificadores utilizados en los pasos de *narrowing* sean los más generales es crucial en la definición de la estrategia (más eficiente y refinada) de *narrowing necesario*.

Como hemos visto, un paso de *narrowing* sobre un objetivo concreto siempre considera tres elementos: posición, regla y sustitución. Básicamente, se trata de dar un paso de reescritura sobre una posición concreta del objetivo una vez que previamente se ha aplicado una determinada sustitución sobre el mismo. Formalizamos el procedimiento de *narrowing* usando un sistema de transición etiquetado cuya relación de

[7] En este ejemplo, y en el resto del trabajo, se utilizará la convención de subrayar los subtérminos considerados para realizar el siguiente paso de *narrowing*.

transición $\leadsto$ formaliza los pasos de computación.

Definición 2.3.1 (narrowing condicional $\leadsto$) *Sea $\mathcal{R}$ un SRTC y g un objetivo ecuacional. Formulamos el método de* narrowing condicional *como la relación más pequeña $\leadsto$ que satisface*

$$\frac{u \in \overline{O}(g) \ \wedge \ r \equiv (\lambda \rightarrow \rho \Leftarrow C) \ll \mathcal{R} \ \wedge \ \theta = mgu(\{g_{|u} = \lambda\})}{g \overset{u,r,\theta}{\leadsto} (C, g[\rho]_u)\theta}$$

Una *derivación de narrowing* se define como: $g \overset{\theta}{\leadsto}{}^* g'$ sii $\exists \theta_1, \ldots, \exists \theta_n.\ g \overset{\theta_1}{\leadsto} \ldots \overset{\theta_n}{\leadsto} g'$ y $\theta = \theta_1 \ldots \theta_n$. Decimos que dicha derivación tiene longitud n. Si $n = 0$, entonces $\theta = \epsilon$. Para tratar la unificación sintáctica como un paso de *narrowing*, añadimos al SRTC la regla $(x = x \rightarrow true)$, $x \in \mathcal{V}$. Así, $(s = t) \overset{\sigma}{\leadsto} true$ sii $\sigma = mgu(\{s = t\})$. Denotamos por $\mathcal{R}_+$ la extensión de un SRTC $\mathcal{R}$ con la regla $(x = x \rightarrow true)$. Usaremos el símbolo $\top$ como notación genérica para una secuencia de la forma de la forma $true, \ldots, true$. Una *derivación de éxito* para g en $\mathcal{R}$ es una derivación de la forma $g \overset{\theta}{\leadsto}{}^* \top$ que usa las reglas de $\mathcal{R}_+$. $\theta_{|\mathcal{V}ar(g)}$ se denomina *sustitución de respuesta computada* para g en $\mathcal{R}$.

Un algoritmo de *narrowing* se dice *completo* si genera un conjunto completo de $\mathcal{E}$-unificadores para cualquier sistema de ecuaciones de entrada. Formalmente, un procedimiento de *narrowing* es completo para una clase de programas si se cumple la siguiente condición:

si $\mathcal{E} \models g\sigma$ entonces existe una derivación $g \overset{\theta}{\leadsto}{}^* true$ tal que $\theta \leq_{\mathcal{E}} \sigma \ [\mathcal{V}ar(g)]$

El subíndice $\mathcal{E}$ puede eliminarse de la expresión $\theta \leq_{\mathcal{E}} \sigma \ [\mathcal{V}ar(g)]$ cuando sólo consideremos completitud con respecto a sustituciones normalizadas [Middeldorp y Hamoen, 1994]. Por extensión, dicho subíndice también desaparece al considerar sustituciones constructoras, ya que este tipo de sustituciones son un caso particular de las sustituciones normalizadas en sistemas basados en constructores. El procedimiento de *narrowing* es un algoritmo de $\mathcal{E}$-unificación completo para sistemas de reescritura de términos que satisfacen diferentes restricciones [Hanus, 1994b; Hölldobler, 1989; Middeldorp y Hamoen, 1994]. Por ejemplo, lo es para SRT canónicos [Hullot, 1980] y también para SRT confluentes con respecto a sustituciones normalizadas. *Narrowing* condicional es completo para 1-SRTC's canónicos, para 1-SRTC's confluentes respecto a soluciones normalizadas y para 2-SRTC's y 3-SRTC's terminantes y confluentes por niveles (ver [Middeldorp y Hamoen, 1994]).

2.4 Estrategias de narrowing

La implementación eficiente del *algoritmo de narrowing* es una tarea ciertamente difícil pero muy importante, en especial para áreas estrechamente relacionadas como

son la programación algebraica, la programación lógica con restricciones (CLP), o la demostración automática de teoremas. Afortunadamente, algunos avances recientes han demostrado que, con restricciones y refinamientos adecuados, *narrowing* puede ser implementado tan eficientemente como –y, para algunos problemas, incluso más eficientemente que– la resolución SLD de los programas lógicos tradicionales [Cheong y Fribourg, 1992; Hanus, 1992].

Un componente muy importante para conseguir una implementación eficiente de *narrowing* es el utilizar una estrategia de selección de *redexes* adecuada, ya que el *narrowing ordinario* (Definición 2.3.1) tiene un alto grado de indeterminismo *don't know*[8].

Más concretamente, la ineficiencia del *narrowing* es el resultado de la combinación de los dos grados de libertad del cálculo: 1) la elección del *redex*, y 2) la elección de la regla de reescritura. Una *estrategia de narrowing* consiste, entonces, en sustituir algunas elecciones *don't know* por elecciones *don't care* [Cheong y Fribourg, 1992] (concretamente, las que se relacionan con 1).

Presentamos a continuación un cálculo de *narrowing genérico con estrategia*, a partir del cual se pueden definir como instancias distintos refinamientos del *narrowing* y, por supuesto, el propio procedimiento de *narrowing* ordinario.

La definición del *narrowing genérico con estrategia* es paramétrica con respecto a la función $\varphi : \ Goal \rightarrow \wp I\!N^*$ (*estrategia de reducción*), que asigna a cada objetivo a reducir g un subconjunto de $\overline{O}(g)$, La función φ permite la explotación de un subconjunto de los *redexes* del objetivo, en lugar de explotarlos todos. De esta forma, se puede reducir el espacio de búsqueda manteniendo, bajo diferentes condiciones, la completitud del cálculo.

Definición 2.4.1 (narrowing genérico con estrategia $\overset{\sigma}{\leadsto}_{\varphi}$) *Dado un SRTC $\mathcal{R}$, definimos la relación de* narrowing condicional genérico con estrategia $\overset{\sigma}{\leadsto}_{\varphi}$ *como la menor relación que satisface*[9]

$$\frac{u \in \varphi(g) \ \wedge \ (\lambda \rightarrow \rho \Leftarrow C) \ll \mathcal{R}^{\varphi}_{+} \ \wedge \ \sigma = mgu(\{(g_{|u}) = \lambda\})}{g \ \overset{\sigma}{\leadsto}_{\varphi} (C, g[\rho]_u)\sigma}$$

donde φ denota una estrategia de narrowing cualquiera, $\mathcal{R}^{\varphi}_{+} = \mathcal{R} \cup \{Eq^{\varphi}\}$, donde Eq^{φ} son las reglas que modelan las igualdad sobre los términos.

Informalmente, el cálculo de *narrowing condicional*, presentado en la Definición 2.3.1, puede verse como una instancia de esta relación de *narrowing* genérico con estrategia, cuando la estrategia de reducción se define como $\varphi(g) = \overline{O}(g)$

[8] Preservamos la terminología en inglés para denotar los dos tipos estándar de indeterminismo: indeterminismo *don't know*, cuando todas las opciones deben ser consideradas, e indeterminismo *don't care*, cuando basta con seleccionar una de las opciones e ignorar el resto.

[9] Escribiremos $\overset{u,r,\sigma}{\leadsto}_{\varphi}$ cuando sea necesario distinguir la ocurrencia y la regla usadas.

El número de refinamientos del procedimiento de *narrowing* que se han propuesto en la literatura es enorme. En [Hanus, 1994b] se citan hasta 18 tipos distintos de estrategias de *narrowing*, la mayor parte de las cuales se pueden agrupar en tres clases:

- Sin ninguna estrategia: se deben explotar todos los *redexes* del objetivo en cada paso (e.g., *narrowing* ordinario [Slagle, 1974] y *narrowing* condicional ordinario [Hussman, 1985; Kaplan, 1987]).
- Dar prioridad a los *redexes* más internos: por ejemplo, *narrowing* condicional "innermost" o impaciente [Fribourg, 1985], *narrowing* condicional impaciente básico [Hölldobler, 1989]. Este tipo de estrategias se denominan también *impacientes* o *voraces*.
- Dar prioridad a los *redexes* más externos: por ejemplo, *narrowing* más externo [Echahed, 1988], *narrowing* externo "outer" [You, 1989], *narrowing* perezoso [Reddy, 1985], *narrowing* dirigido por la demanda [Moreno-Navarro y Rodríguez-Artalejo, 1992], y *narrowing* necesario [Antoy *et al.*, 1994].

2.4.1 *Narrowing* innermost

Presentamos en primer lugar una estrategia de *narrowing* en la que los pasos de computación se realizan sólo sobre las ocurrencias más internas (*innermost*) del objetivo ecuacional. Esta estrategia se corresponde con la estrategia de evaluación de Prolog y con la evaluación impaciente (*eager*) de los lenguajes funcionales, también conocida como *llamada por valor (call-by-value)*, debido a que todos los parámetros de una llamada a función deben ser evaluados antes de poder evaluar la propia función. La mayor parte de los conceptos presentados en este punto son una adaptación de [Fribourg, 1985].

Por razones que se discutirán más adelante, en esta sección consideraremos *programas basados en constructores (CB)*. Esto implica que no pueden haber anidamientos en las partes izquierdas de las cabezas de las cláusulas ni axiomas entre los constructores. Ésta es una clase razonable de programas desde el punto de vista de la programación funcional (al menos, cuando se trabaja con géneros 'sorts'). Muchas teorías ecuacionales que ocurren en la práctica siguen esta disciplina, por ejemplo, en la especificación de los tipos abstractos de datos. Un símbolo de *función se dice completamente definido* si no ocurre en ningún término en forma normal, es decir, si todos los símbolos de función son reducibles sobre todos los posibles términos constructores (del género apropiado[10]). Un SRTC $\mathcal{R}$ se dice *completamente definido* (CD,

[10]Ya que los géneros no son relevantes para los objetivos de este trabajo, los omitimos por simplicidad y sólo consideramos signaturas con un género. La extensión a signaturas heterogéneas es inmediata [Padawitz, 1988].

"completely defined") si todos los símbolos de función definidos están completamente definidos. En un SRTC completamente definido, el conjunto de términos en forma normal coincide con el conjunto de términos irreducibles (o constructores) $\tau(\mathcal{C} \cup \mathcal{V})$ sobre $\mathcal{C} \cup \mathcal{V}$. En teorías con un sólo género, es extraño encontrar programas completamente definidos; sin embargo, es usual cuando se usan tipos y cada función se encuentra definida sobre todos los constructores pertenecientes al tipo de sus argumentos.

En una estrategia impaciente, la función de selección de redexes φ, asigna a cada objetivo g una única de las posiciones más internas asociadas a un patrón de g, según el orden prefijo $\leq$. Sin pérdida de generalidad, de las posibles posiciones más internas, seleccionamos aquélla que aparezca más a la izquierda (*leftmost innermost*).

Formalmente, definimos la posición más interna a la izquierda $inn(g)$ de un objetivo g como:

$$inn(g) = p \quad \begin{array}{l} \text{tal que } p \in \overline{O}(g), \quad g_{|_p} \text{ es un patrón y} \\ \text{para cada patrón } g_{|_q}, \quad p \lhd q \end{array}$$

Donde $p \lhd q$ define el orden $\lhd$ entre las posiciones de $\overline{O}(g)$ de la forma:

$$p \lhd q \longleftrightarrow \exists p_0, p_1, p_2 \in I\!N^*,\ \exists i, j > 0.\ p = p_0.i.p_1 \ \wedge\ q = p_0.j.p_2 \ \wedge\ i < j$$

Intuitivamente, $p \lhd q$ implica que la posición p señala un término que aparece más a la izquierda del término señalado por q.

Definición 2.4.2 (narrowing innermost $\leadsto_{inn}$) *El método de* narrowing condicional con la estrategia innermost *se formula como la relación más pequeña $\leadsto_{inn}$ que satisface*

$$\frac{u = inn(g) \ \wedge\ r \equiv (\lambda \rightarrow \rho \Leftarrow C) \ll \mathcal{R}_+^{inn} \ \wedge\ \sigma = mgu(\{g_{|u} = \lambda\})}{g \overset{u,r,\sigma}{\leadsto}_{inn} (C, g[\rho]_u)\sigma}$$

$\mathcal{R}_+^{inn} = \mathcal{R} \cup \{Eq^{inn}\}$, donde Eq^{inn} es la regla que modela la igualdad estandar y esta definida por:

$$x = x \quad \rightarrow \quad true \qquad \% \ x \in \mathcal{V}$$

Como la estrategia *innermost* será la única variante impaciente de *narrowing* que trataremos en esta tesis, en lo que sigue (y sin posibilidad de confusión) la denotaremos genéricamente por *narrowing innermost*, *narrowing impaciente* o en algunos contextos simplemente *inn*. Al formular la *completitud del narrowing innermost* [Fribourg, 1985] considera sustituciones constructoras. Como ya hemos comentado, esto no es suficiente para enunciar un resultado de completitud incluso cuando las reglas del programa son confluentes y terminantes, debido a los problemas que plantea la estrategia impaciente ante funciones definidas parcialmente. Fribourg presenta diferentes condiciones adicionales para asegurar la completitud del cálculo. La más importante consiste en considerar únicamente *funciones totalmente definidas*, lo que

implica que cualquier término básico irreducible es constructor. El siguiente ejemplo de [Hanus, 1994b] muestra la incompletitud del *narrowing* impaciente ante la presencia de funciones parciales.

Ejemplo 2 Consideremos las siguientes reglas, donde a y b son constructores:

$$\begin{array}{rcl} \mathtt{f(a,y)} & \to & \mathtt{a} \\ \mathtt{g(b)} & \to & \mathtt{b} \end{array}$$

Si queremos resolver la ecuación $\mathtt{f(x,g(x))} = \mathtt{a}$, entonces existe la siguiente derivación de éxito (donde restringimos las sustituciones a las variables del objetivo) usando *narrowing*:

$$\underline{\mathtt{f(x,g(x))}} = \mathtt{a} \overset{\{\mathtt{x}\mapsto\mathtt{a}\}}{\leadsto}_{\mathtt{inn}} \underline{\mathtt{a} = \mathtt{a}} \leadsto_{\mathtt{inn}} \mathtt{true}$$

aplicando la primera regla al objetivo original, i.e., $\{\mathtt{x} \mapsto \mathtt{a}\}$ es una solución para la ecuación inicial. Sin embargo, esta derivación no es impaciente, mientras que la única derivación impaciente no es de éxito:

$$\mathtt{f(x,}\underline{\mathtt{g(x)}}\mathtt{)} = \mathtt{a} \overset{\{\mathtt{x}\mapsto\mathtt{b}\}}{\leadsto}_{\mathtt{inn}} \mathtt{f(b,b)} = \mathtt{a}$$

Por tanto, *narrowing* impaciente no puede calcular la solución.

La siguiente proposición, original de [Fribourg, 1985], establece la completitud del procedimiento de *narrowing* impaciente para programas canónicos que sigan la disciplina CB-CD.

Proposición 2.4.3 (completitud del *narrowing innermost*)
Sea $\mathcal{R}$ un SRTC canónico CB-CD, g un objetivo ecuacional y σ una solución básica constructora tal que $\mathcal{V}ar(g) \subseteq Dom(\sigma)$. Entonces, existe una sustitución de respuesta computada θ para $\mathcal{R} \cup \{g\}$ usando $\leadsto_{inn}$ tal que $\theta \leq \sigma\ [\mathcal{V}ar(g)]$.

La condición $\mathcal{V}ar(g) \subseteq Dom(\sigma)$ en la premisa de la proposición anterior garantiza que $\mathtt{g}\sigma$ sea básico. En [Alpuente *et al.*, 1998] encontramos el siguiente ejemplo que revela que esta condición no puede ser eliminada, lo cual se ignora erróneamente, por ejemplo, en [Hanus, 1994b; Hölldobler, 1989].

Ejemplo 3 Consideremos el siguiente programa canónico CB-CD $\mathcal{R}$:

$$\begin{array}{rcl} \mathtt{f(0,y)} & \to & \mathtt{y} \\ \mathtt{g(0)} & \to & \mathtt{0} \end{array}$$

La sustitución básica constructora $\sigma = \{\mathtt{x} \mapsto \mathtt{0}\}$ es una solución de la ecuación $\mathtt{f(x,g(y))} = \mathtt{g(y)}$. Sin embargo, narrowing condicional impaciente no es capaz de computar una respuesta más general (de hecho, sólo computa la respuesta $\{\mathtt{X} \mapsto \mathtt{0}, \mathtt{Y} \mapsto \mathtt{0}\}$). En este caso, la sustitución σ no satisface la condición $\mathcal{V}ar(g) \subseteq Dom(\sigma)$.

Si todas las funciones están completamente definidas y consideramos el programa aumentado $\mathcal{R}_+$, entonces la regla $x = x \to true$ solo calcula sustituciones construc-

toras cuando se aplica. En efecto, si aplicamos la regla $x = x \rightarrow true$ al objetivo $y = g(x)$, es porque el término $g(x)$ es una forma normal y como las funciones son completamente definidas, entonces dicho término es un constructor.

Resulta fácil extender esta estrategia a teorías con funciones que no están completamente definidas, añadiendo simplemente la llamada *regla de reflexión*, que permite ignorar una llamada a función más interna cuando ésta no puede ser reducida [Hölldobler, 1989]. Por simplicidad, asumimos que todas las funciones están completamente definidas y, por tanto, *narrowing* impaciente es suficiente para computar todas las soluciones.

A menudo es útil considerar *narrowing condicional innermost con normalización*, donde se calcula la forma normal del objetivo usando reescritura condicional *innermost* –la reescritura se aplica sólo a aquellos términos cuyos subtérminos están en forma normal– antes de cada paso de *narrowing*. Esto simplifica las demostraciones y, además, puede ser implementado eficientemente [Hanus, 1990, 1991]. Para asegurar que la forma normal de un objetivo sea única y pueda ser calculada con cualquier estrategia de reescritura, vamos a exigir que los programas también sean decrecientes [Dershowitz y Jouannaud, 1990]. *Narrowing innermost* con normalización es la base de varios lenguajes de programación lógico funcionales, como SLOG [Fribourg, 1985], LPG [Bert y Echahed, 1986, 1994], *eager-BABEL* [Kuchen *et al.*, 1990a] y (un subconjunto de) ALF [Hanus, 1990]. Se ha demostrado que, ya que las funciones permiten una evaluación mucho más determinista que los predicados, *narrowing innermost* con normalización permite la ejecución de programas lógico funcionales de forma más eficiente que los programas lógicos equivalentes [Fribourg, 1985; Hanus, 1991].

2.4.2 Narrowing outermost

Como ya hemos comentado, algunas de las restricciones sobre los programas (por ejemplo, estar completamente definidos), se pueden eliminar realizando ligeras variaciones en la estrategia de cálculo $\varphi = inn$. Sin embargo, la exigencia de que las reglas del programa sean terminantes, no puede ser eliminada sin pérdida de completitud. Esto puede ser un inconveniente cuando se quieren explotar técnicas típicas de la programación funcional como, por ejemplo, las que manipulan estructuras de datos infinitas o con funciones parcialmente definidas. En el siguiente punto, presentamos las estrategias de *narrowing* outermost y perezoso que preservan la completitud del *narrowing* para programas no terminantes, bajo ciertas condiciones. Una estrategia perezosa correspondiente para narrowing es *outermost narrowing* donde la siguiente posición de narrowing debe ser la más externa.

Formalmente, definimos la posición más externa a la izquierda $out(g)$ de un obje-

tivo g como:

$$out(g) = p \quad \text{tal que } p \in \overline{O}(g),\ \ g_{|_p} \text{ es un redex outermost y para cada redex outermost } g_{|_q},\ \ p \lhd q$$

Definición 2.4.4 (*narrowing outermost* $\leadsto_{out}$) *El método de* narrowing condicional con la estrategia outermost *se formula como la relación más pequeña* $\leadsto_{out}$ *que satisface*

$$\frac{u = out(g) \ \wedge\ r \equiv (\lambda \to \rho \Leftarrow C) \ll \mathcal{R}_{+}^{out} \ \wedge\ \sigma = mgu(\{g_{|u} = \lambda\})}{g \overset{u,r,\sigma}{\leadsto}_{out} (C, g[\rho]_u)\sigma}.$$

$\mathcal{R}_{+}^{out} = \mathcal{R} \cup \{Eq^{out}\}$ *donde* Eq^{out} *son las reglas que modelan las igualdad sobre los términos constructores* ($\approx$).

La *igualdad estricta* $\approx$ considera dos términos iguales sólo si éstos se reducen a un mismo término básico y formado únicamente por símbolos constructores. La semántica de la relación $\approx$ se puede definir mediante el siguiente conjunto confluente Eq^{out} de reglas de programa:

$$\begin{array}{rcll} c \approx c & \to & true & \%\ c/0 \in \mathcal{C} \\ c(x_1, \ldots, x_n) \approx c(y_1, \ldots, y_n) & \to & (x_1 \approx y_1) \wedge \ldots \wedge (x_n \approx y_n) & \%\ c/n \in \mathcal{C} \end{array}$$

Nótese que la igualdad estricta no posee la propiedad reflexiva $t \approx t$ para todos los términos t. La igualdad estricta es la única definición adecuada de la igualdad para el caso de *programas no terminantes*. Cuando trabajamos con la relación de *narrowing outermost*, consideramos que todos los símbolos de igualdad en los objetivos (y en las condiciones de las reglas) son $\approx$.

Desafortunadamente, esta estrategia es incompleta, como muestra el siguiente ejemplo [Hanus, 1994b].

Ejemplo 4 Consideremos las siguientes reglas, que definen la función f :

$$\begin{array}{rcl} \mathtt{f(0,0)} & \to & \mathtt{0} \\ \mathtt{f(s(x),0)} & \to & \mathtt{1} \\ \mathtt{f(x,s(y))} & \to & \mathtt{2} \end{array}$$

Si queremos resolver la ecuación $\mathtt{f(f(x.y),z)} \approx \mathtt{0}$, la derivación innermost correspondiente es la siguiente:

$$\mathtt{f(f(x,y),z)} \approx \mathtt{0} \overset{\{\mathtt{x}\mapsto\mathtt{0},\mathtt{y}\mapsto\mathtt{0}\}}{\leadsto}_{\mathtt{out}} \mathtt{f(0,z)} \approx \mathtt{0} \overset{\{\mathtt{z}\mapsto\mathtt{0}\}}{\leadsto}_{\mathtt{out}} \mathtt{0} \approx \mathtt{0} \leadsto_{\mathtt{out}} \mathtt{true}$$

Así pues, $\{\mathtt{x} \mapsto \mathtt{0}, \mathtt{y} \mapsto \mathtt{0}, \mathtt{z} \mapsto \mathtt{0}\}$ es una solución de la ecuación inicial. Sin embargo, sólo hay una única derivación de narrowing outermost, usando la última regla:

$$\mathtt{f(f(x,y),z)} \approx \mathtt{0} \overset{\{\mathtt{z}\mapsto\mathtt{s(y)}\}}{\leadsto}_{\mathtt{out}} \mathtt{2} \approx \mathtt{0}$$

De esta forma outermost narrowing no puede calcular la solución anterior.

Se han formulado fuertes restricciones para asegurar la completitud de la estrategia *outermost*, en adición a la confluencia y la terminación de las reglas. En [You, 1989], se muestra que *inn* y *out* no son completos generalmente. Por ejemplo, consideremos el siguiente programa

$$\mathcal{R} = \{\mathtt{f(y,a)} \to \mathtt{true},\ \ \mathtt{f(c,b)} \to \mathtt{true},\ \ \mathtt{g(b)} \to \mathtt{c}\}$$

con el siguiente objetivo

$$\mathtt{f(g(x),x)} =_{\varphi} \mathtt{true}$$

entonces *narrowing innermost* solo calcula la respuesta $\{\mathtt{x/b}\}$ para el objetivo $\mathtt{f(g(x),x)} = \mathtt{true}$ mientras que *narrowing outermost* sólo computa $\{\mathtt{x/a}\}$ para el objetivo $\mathtt{f(g(x),x)} \approx \mathtt{true}$. Para establecer la completitud de *narrowing* con las estrategias $\varphi = inn, out$, se requiere la condición *uniformidad* [Echahed, 1988, 1992; Frutos-Escrig y Fernández-Camacho, 1991; Hanus, 1994b; Padawitz, 1988]: un programa confluente es *uniforme* si y solo si la posición seleccionada por la estrategia φ es también una posición de *narrowing* válida para φ para todas las sustituciones en forma normal aplicadas a ella. Note que el programa $\mathcal{R}$ anterior no satisface la propiedad de uniformidad ya que la posición más externa de el termino $\mathtt{f(g(x),x)}$ no es una posición válida de narrowing si le aplicamos la sustitución $\{\mathtt{x/b}\}$. Similarmente, en el Ejemplo 4 la posición más externa del término $\mathtt{f(f(x,y),z)}$, que corresponde al propio término, no es una posición de narrowing válida si le aplicamos la sustitución $\{\mathtt{z} \mapsto \mathtt{0}\}$. Una condición suficiente para la uniformidad en programa basados en constructores CB y canónicos es [Echahed, 1988]: i) las funciones de $\mathcal{D}$ son completamente definidas (es decir, el conjunto de términos básicos normalizados es $\tau(\mathcal{C})$), y ii) las partes izquierdas de las reglas en $\mathcal{R}$ son *no estrictamente sub–unificables* dos a dos, es decir dos subtérminos en la misma posición de los lados izquierdos de dos reglas que definen una misma función no son unificables por un *mgu* no trivial [Hanus, 1994b]. Por ejemplo, $\mathtt{f(y,a)}$ y $\mathtt{f(c,b)}$ son estrictamente subunificables ya que el *mgu* de los primeros argumentos es la sustitución no trivial $\{\mathtt{y/c}\}$. Debido a que el requerimiento de *no estrictamente sub–unificable* no es satisfecho por muchos programas, en [Echahed, 1988] se formula un método para transformar un programa que satisface i) en un programa que satisface i) y ii) (ver [Echahed, 1988] para detalles).

El siguiente teorema muestra la completitud de narrowing outermost bajo las condiciones ya anunciadas [Echahed, 1988, 1992; Maneth y Vogler, 1995].

Proposición 2.4.5 (completitud del *narrowing* outermost) *Sea $\mathcal{R}$ un SRT canónico CB-CD, lineal por la izquierda, no estrictamente subunificable y sea g un objetivo ecuacional y σ una solución básica constructora tal que $\mathcal{V}ar(g) \subseteq Dom(\sigma)$. Entonces, existe una sustitución de respuesta computada θ para $\mathcal{R} \cup \{g\}$ usando $\leadsto_{\varphi}$*

tal que $\theta \leq \sigma\ [\mathcal{V}ar(g)]$.

Es importante notar que tanto para la estrategia *inn* como para *out* narrowing computa sólo sustituciones constructoras en programas que satisfacen las condiciones para la completitud de la estrategia. Adicionalmente, cuando se considera la igualdad estricta los resultados de completitud pueden ser generalizados para sistemas no terminantes. De acuerdo con [Giovannetti *et al.*, 1991; Moreno-Navarro y Rodríguez-Artalejo, 1992], introducimos un símbolo de constante fresco $\perp$ en Σ para representar valores de expresiones que de otra manera son indefinidos. Para el caso de funciones infinitas, al agregar reglas de la forma $f(x) \rightarrow \perp$ damos un significado que aproxima una expresión que es infinita.

2.4.3 Narrowing perezoso

La estrategia de evaluación perezosa (*lazy evaluation*) en los lenguajes funcionales consiste, básicamente, en retrasar la evaluación de los argumentos de una función hasta que su evaluación sea necesaria para computar el resultado. Este tipo de estrategia se relaciona con el concepto de *llamada por nombre (call-by-name)* debido a que (algunos de) los parámetros de una llamada a función se pasan como una referencia a su nombre y no a sus valores (que únicamente son evaluados si son "demandados" para poder evaluar la propia función). Las estrategias perezosas poseen varias propiedades importantes. En primer lugar, permiten trabajar con estructuras de datos infinitas, ya que los términos no necesitan ser siempre evaluados completamente. Por otra parte, dado un término t, si existe alguna estrategia de evaluación que permite computar la forma normal de t de manera finita, la estrategia perezosa también termina computando dicho valor. Dado que el concepto de posición "demandada" tiene varias interpretaciones posibles, la adaptación de esta estrategia a los lenguajes lógico funcionales ha dado lugar a distintas estrategias perezosas de *narrowing* (ver, por ejemplo, [Antoy *et al.*, 1994; Darlington *et al.*, 1991; Echahed, 1988; Kuchen *et al.*, 1990b; Moreno-Navarro y Rodríguez-Artalejo, 1992; Reddy, 1985; You, 1989]). En nuestro caso, vamos a definir una estrategia de *narrowing* perezoso guiado por la demanda en el estilo de [Reddy, 1985] y, por tanto, similar al mecanismo operacional del lenguaje lógico funcional *BABEL* [Moreno-Navarro y Rodríguez-Artalejo, 1992]. Informalmente, esta estrategia sólo permite realizar pasos de *narrowing* sobre *redexes* internos del objetivo cuando estos son *demandados* por la parte izquierda de alguna regla del programa.

Debido a la presencia de funciones no terminantes, los resultados de completitud para *narrowing* perezoso se establecen con respecto a la igualdad estricta "$\approx$", que como ya hemos dicho anteriormente, considera la identidad entre objetos finitos, i.e., dos términos son iguales sólo si éstos se reducen a un mismo término básico y forma-

do únicamente por símbolos constructores. La igualdad estricta es la única definición adecuada de la igualdad para el caso de programas no terminantes. Cuando trabajamos con la relación de *narrowing* perezoso, consideramos que todos los símbolos de la igualdad en los objetivos son $\approx$. Puesto que no se exige que las reglas de los programas sean terminantes, la propiedad de confluencia de los programas se vuelve indecidible. Pese a todo, existen una serie de condiciones suficientes (decidibles) que garantizan la confluencia en el caso de programas CB, incluso en ausencia de la propiedad de terminación: linealidad por la izquierda y no solapamiento de reglas [Moreno-Navarro y Rodríguez-Artalejo, 1992]. A continuación, formulamos un procedimiento de *narrowing* perezoso que es completo para programas que satisfacen esta doble condición, que como ya hemos avanzado anteriormente, es conocida también como *ortogonalidad* [Klop, 1992]. Cuando trabajamos con programas CB lineales por la izquierda, la unificación entre las partes izquierdas de las reglas y los términos del objetivo posee siempre una serie de características comunes que, en general, denominaremos como *problema de unificación lineal*.

Definición 2.4.6 (problema de unificación lineal) Un problema de unificación lineal es un par de términos $\langle f(d_1, \ldots, d_n), f(t_1, \ldots, t_n)\rangle$, donde $f(d_1, \ldots, d_n)$ y $f(t_1, \ldots, t_n)$ no comparten variables y $f(d_1, \ldots, d_n)$ es un patrón lineal.

Los problemas de unificación lineal se pueden resolver mediante cualquier versión del algoritmo de unificación (ver, por ejemplo, [Lassez *et al.*, 1988]). Siguiendo [Moreno-Navarro y Rodríguez-Artalejo, 1992], distinguimos el caso en el que la unificación no tiene éxito debido a un conflicto entre un constructor c y un símbolo de función f, ya que tal situación se puede considerar como una demanda de mayor evaluación de f. El siguiente algoritmo es una reformulación del algoritmo de unificación para el caso de los problemas de unificación lineales de [Moreno-Navarro y Rodríguez-Artalejo, 1992]. Nótese que, debido a la linealidad por la izquierda, la comprobación conocida como "occur-check" no es necesaria.

Definición 2.4.7 (configuración LU) Una configuración LU es un par (U, σ), donde U es un conjunto y σ es una sustitución.

Definición 2.4.8 (relación de unificación $\rightarrow_{\mathsf{LU}}$) Definimos la relación de unificación $\rightarrow_{\mathsf{LU}}$ como la menor relación que satisface:

1. $(\{c(d_1, \ldots, d_n) \downarrow_p c(t_1, \ldots, t_n)\} \cup U, \sigma) \rightarrow_{\mathsf{LU}} (\{d_1 \downarrow_{p.1} t_1, \ldots, d_n \downarrow_{p.n} t_n\} \cup U, \sigma)$, donde $c/n \in \mathcal{C}$, $n \geq 0$.
2. $(\{x \downarrow_p t\} \cup U, \sigma) \rightarrow_{\mathsf{LU}} (U\{x \mapsto t\}, \sigma\{x \mapsto t\})$, donde $t \notin \mathcal{X}$.
3. $(\{d \downarrow_p x\} \cup U, \sigma) \rightarrow_{\mathsf{LU}} (U\{x \mapsto d\}, \sigma\{x \mapsto d\})$.
4. $(\{c(d_1, \ldots, d_n) \downarrow_p c'(t_1, \ldots, t_m)\} \cup U, \sigma) \rightarrow_{\mathsf{LU}} (\{\mathtt{fail}\}, \sigma)$, donde $c/n, c'/m \in \mathcal{C}$, $c \neq c'$, y $n, m \geq 0$.

Es fácil observar que las computaciones $\rightarrow^*_{\mathsf{LU}}$ pueden terminar de tres formas distintas: computando el *mgu* de los términos de entrada, devolviendo `fail` (en el caso de que los términos no unifiquen) o devolviendo un conjunto de pares $c(d_1, \ldots, d_n) \downarrow_p f(t_1, \ldots, t_n)$, en el que cada posición p indica una posición demandada. La siguiente definición formaliza dicho comportamiento.

Definición 2.4.9 (comportamiento de $\rightarrow_{\mathsf{LU}}$)
Sea $\Gamma = \langle f(d_1, \ldots, d_n), f(t_1, \ldots, t_n)\rangle$ un problema de unificación lineal. Dada la computación[11]:

$$(\{d_1 \downarrow_1 t_1, \ldots, d_n \downarrow_n t_n\}, \epsilon) \rightarrow^*_{\mathsf{LU}} (U, \sigma) \not\rightarrow_{\mathsf{LU}}$$

definimos la función $\mathsf{LU}(\Gamma)$ como sigue:

$$\mathsf{LU}(\Gamma) = \begin{cases} (\textsc{Success}, \sigma) & \text{si } U = \emptyset \\ (\textsc{Fail},) & \text{si } U = \{\texttt{fail}\} \\ (\textsc{Demand}, P) & \text{en cualquier otro caso, donde} \\ & P = \{p \mid (c(d'_1, \ldots, d'_m) \downarrow_p g(t'_1, \ldots, t'_k)) \in U\} \\ & \text{es el conjunto de posiciones demandadas.} \end{cases}$$

Informalmente, una estrategia perezosa debe seleccionar, en cada paso de computación, la ocurrencia del término completo, excepto cuando hayan subtérminos demandados de acuerdo con el algoritmo de unificación lineal. De esta forma, una estrategia de *narrowing* perezoso solamente selecciona los redexes demandados e un término.

A continuación formulamos la relación de em narrowing *perezoso* de una forma similar a como lo hemos hecho para el *narrowing* impaciente, por medio de un sistema de transición que es instancia de la relación genérica de *narrowing* con estrategia.

Definición 2.4.10 (*narrowing* perezoso $\rightsquigarrow_{np}$) La relación de narrowing perezoso $\rightsquigarrow_{np}$ se define como una instancia de la relación de narrowing genérico con estrategia donde la estrategia φ^{np} se define de manera inductiva como sigue:

$$\begin{array}{lcl} \varphi^{np}(g) & = & \bigcup_{k=1}^{m} \varphi_{\text{-}}(g, \Lambda, k) \\ \varphi_{\text{-}}(g, p, k) & = & \text{si } l_k[\Lambda] = g[p] \text{ entonces} \\ & & \quad \begin{cases} \{\langle p, R_k, \sigma\rangle\} & \text{si } \mathsf{LU}(\langle l_k, g|_p\rangle) = (\textsc{Success}, \sigma) \\ \bigcup_{p' \in P} \bigcup_{k'=1}^{m} \varphi_{\text{-}}(g, p.p', k') & \text{si } \mathsf{LU}(\langle l_k, g|_p\rangle) = (\textsc{Demand}, P) \\ \emptyset & \text{si } \mathsf{LU}(\langle l_k, g|_p\rangle) = (\textsc{Fail},) \end{cases} \end{array}$$

donde $R_k = (l_k \rightarrow r_k \Leftarrow C_k) \ll \mathcal{R}_+$, $1 \leq k \leq m$. Las computaciones se realizan usando las reglas del programa extendido $\mathcal{R}_+ = \mathcal{R} \cup Eq^{out}$.

Tal y como se expone en [Moreno-Navarro y Rodríguez-Artalejo, 1992], la sustitución computada σ en cada paso de *narrowing* perezoso puede verse como la unión

[11] Por $(U, \sigma) \not\rightarrow_{\mathsf{LU}}$ indicamos que no es posible realizar una transición $\rightarrow_{\mathsf{LU}}$ a partir del estado (U, σ).

de dos sustituciones $\sigma_g \cup \sigma_r$ representando las restricciones de σ a las variables del objetivo y a las variables de la regla empleada, respectivamente. Es interesante destacar que, debido a la linealidad por la izquierda y a la disciplina de constructores de los programas, la sustitución σ_g no contiene nunca símbolos de función definidos [Moreno-Navarro y Rodríguez-Artalejo, 1992], i.e., es una sustitución constructora. La siguiente proposición, original de [Moreno-Navarro y Rodríguez-Artalejo, 1992], establece la completitud del cálculo.

Proposición 2.4.11 (completitud del narrowing perezoso) [Moreno-Navarro y Rodríguez-Artalejo, 1992] *Sea $\mathcal{R}$ un programa CB ortogonal, lineal por la izquierda y no ambigüo, g un objetivo ecuacional y σ una sustitución básica constructora tal que, para toda ecuación $s \approx t$ en g, se cumple $(s\sigma)\!\downarrow = (t\sigma)\!\downarrow$, donde $(t\sigma)\!\downarrow \in \tau(\mathcal{C})$. Entonces, existe una sustitución de respuesta computada θ para $\mathcal{R} \cup \{g\}$ usando $\leadsto_{np}$ tal que $\theta \leq \sigma$ $[\mathcal{V}ar(g)]$.*

A diferencia de lo que ocurre en las derivaciones de reescritura perezosas, la aplicación de diferentes reglas a una posición (demandada) particular puede conducir a diferentes soluciones cuando se consideran derivaciones de *narrowing* perezosas. Además, al intentar aplicar diferentes reglas a un objetivo, puede darse el caso de que los argumentos a evaluar requeridos sean distintos. Como consecuencia, las estrategias simples de *narrowing* perezoso corren el riesgo de realizar pasos de computación innecesarios en derivaciones de *narrowing*. El siguiente ejemplo ilustra este punto [Hanus, 1994b].

Ejemplo 5 Consideremos el siguiente conjunto de reglas que definen la relación $\leq$ y la suma sobre los números naturales:

$$\begin{array}{rcll} \mathtt{0} & \leq & \mathtt{y} & \rightarrow \mathtt{true} \\ \mathtt{s(x)} & \leq & \mathtt{0} & \rightarrow \mathtt{false} \\ \mathtt{s(x)} & \leq & \mathtt{s(y)} & \rightarrow \mathtt{x \leq y} \\ \mathtt{0} & + & \mathtt{y} & \rightarrow \mathtt{y} \\ \mathtt{s(x)} & + & \mathtt{y} & \rightarrow \mathtt{s(x + y)} \end{array}$$

Ahora, pretendemos resolver la ecuación $\mathtt{x \leq x + y \approx z}$ por *narrowing* perezoso. Una primera solución podría computarse aplicando la primera regla para $\leq$ sin evaluar el subtérmino $\mathtt{x + y}$:

$$\underline{\mathtt{x \leq x + x}} \approx \mathtt{z} \overset{\{\mathtt{X} \mapsto \mathtt{0}\}}{\leadsto}_{\mathtt{np}} \underline{\mathtt{true} \approx \mathtt{z}} \overset{\{\mathtt{B} \mapsto \mathtt{true}\}}{\leadsto}_{\mathtt{np}} \mathtt{true}$$

De esta forma tenemos que $\{\mathtt{x} \mapsto \mathtt{0}, \mathtt{z} \mapsto \mathtt{true}\}$ es una solución para la ecuación inicial. A la hora de calcular nuevas soluciones, la alternativa consistiría en aplicar la segunda y tercera reglas para $\leq$, pero en ambos casos es necesario evaluar el subtérmino $\mathtt{x + y}$. Si escogemos la regla $\mathtt{0 + y} \rightarrow \mathtt{y}$ para dar el primer paso de evaluación, obtenemos la

siguiente derivación de *narrowing* perezoso:

$$\mathtt{x} \leq \underline{\mathtt{x} + \mathtt{y}} \approx \mathtt{z} \overset{\{\mathtt{x} \mapsto 0\}}{\leadsto}_{\mathtt{np}} \underline{\mathtt{0} \leq \mathtt{y}} \approx \mathtt{z} \overset{\{\}}{\leadsto}_{\mathtt{np}} \underline{\mathtt{true} \approx \mathtt{z}} \overset{\{\mathtt{z} \mapsto \mathtt{true}\}}{\leadsto}_{\mathtt{np}} \mathtt{true}$$

En el segundo paso sólo la primera regla para $\leq$ es aplicable. La solución computada $\{\mathtt{x} \mapsto \mathtt{0}, \mathtt{z} \mapsto \mathtt{true}\}$ es idéntica a la anterior, pero ahora se han dado pasos superfluos, ya que para alcanzar esta solución no es necesario evaluar el término $\mathtt{x} + \mathtt{y}$.

Para evitar pasos innecesarios en derivaciones de *narrowing* perezosas es posible cambiar el criterio de instanciación de variables y aplicación de reglas. En el ejemplo anterior, la evaluación de un término de la forma $\mathtt{x} \leq \mathtt{t}$ podría comenzar por instanciar la variable $\mathtt{x}$ bien a $\mathtt{0}$ o bien a $\mathtt{s}(\mathtt{x}')$ y evaluar t únicamente en el segundo caso, cuando es estrictamente necesario. Este refinamiento de *narrowing* perezoso se conoce como *narrowing* necesario y constituye la base del lenguaje *Curry* [Antoy *et al.*, 1994; Hanus *et al.*, 1995]. La nueva estrategia disfruta de propiedades importantes, como la optimalidad e independencia de soluciones, y puede verse como un refinamiento del *narrowing* perezoso .

2.5 *Narrowing* necesario

La estrategia de *narrowing* necesario [Antoy *et al.*, 1994] es completa para programas *inductivamente secuenciales*, al tiempo que únicamente computa pasos *inevitables* para resolver objetivos. Dicha estrategia es óptima con respecto a la longitud de las derivaciones (en implementaciones basadas en grafos) y al número de soluciones computadas. Como resumen, diremos que las principales ventajas de esta estrategia frente a otras existentes en la literatura incluyen: la gran clase de sistemas de reescritura sobre los que es aplicable (sistemas inductivamente secuenciales), la optimalidad o minimalidad en la longitud de las derivaciones, la independencia de los unificadores que calcula y la facilidad con que puede ser implementada [Antoy *et al.*, 1994].

La noción de *inevitable* o *necesario* es bien conocida en reescritura . Los sistemas *ortogonales* tienen la propiedad de que, para todo término t que no esté en forma normal, existe un redex, llamado *necesario*, que eventualmente debe ser reducido para calcular la forma normal de t [Klop y Middeldorp, 1989; O'Donnell, 1977]. Más aún: la reducción reiterada de redexes necesarios es suficiente para calcular la forma normal de un término, si ésta existe. Es decir, en sistemas ortogonales, es suficiente considerar redexes necesarios. Sin embargo, este concepto es indecidible en el caso general y se han propuesto familias restringidas de programas sobre las que es posible decidir cuando el conjunto de redexes es necesario.

A continuación, pasamos a extender estos resultados al caso de la relación de *narrowing* sobre programas inductivamente secuenciales (que son una subclase de los

sistemas ortogonales sobre la que el concepto de redex necesario es decidible)[12]. Al intentar extender algunos conceptos de la reescritura necesaria al caso de *narrowing*, nos encontramos con el hecho de que, ahora, debemos considerar la instanciación de un término antes de que éste pueda ser reducido. Afortunadamente, este problema tiene una solución eficiente sobre programas inductivamente secuenciales, que pasa por relajar el requerimiento (exigido hasta ahora en todas las variantes de *narrowing* estudiadas) de que los unificadores usados en los pasos de *narrowing* fuesen los más generales. Los unificadores que se calculan ahora, de alguna forma "anticipan" ciertos enlaces para las variables que, de todos modos, se hubieran computado más adelante en la derivación. Esto complica ligeramente la definición de la nueva estrategia, pero en contrapartida permite obtener el refinamiento de *narrowing* más eficiente conocido. Sea $A = (t \rightarrow_{u,l\rightarrow r} t')$ un paso de reescritura aplicado sobre una posición u de un término arbitrario t usando una regla $l \rightarrow r$ y obteniendo un nuevo término t'. El conjunto de *descendientes* de una posición v tras el paso A, denotado por $v\backslash A$ es:

$$v\backslash A = \begin{cases} \emptyset & \text{si } u = v, \\ \{v\} & \text{si } u \not\leq v, \\ \{u.p'.q \mid r_{|p'} = x\} & \text{si } v = u.p.q \text{ y } l|_p = x, \text{ donde } x \in \mathcal{X}. \end{cases}$$

Todos los descendientes $\{v_1, \ldots, v_n\}$ de una misma posición v se denominan *hermanos* entre sí, mientras que v se denomina *antecedente* de todos ellos. El conjunto de *descendientes* de una posición v tras una secuencia de reducción B se define inductivamente como sigue:

$$v\backslash B = \begin{cases} \{v\} & \text{si } B \text{ es una derivación nula,} \\ \bigcup\limits_{w\in v\backslash B'} w\backslash B'' & \text{si } B = B'B'', \text{ donde } B' \text{ es el paso inicial de } B. \end{cases}$$

Dado un conjunto de posiciones P, definimos $P\backslash B = \bigcup_{p\in P} p\backslash B$. Decimos que una posición p de un término t es *necesaria* si y sólo si en toda derivación de reescritura que reduce t a su forma normal, algún descendiente de $t|_p$ es reducido a la posición raíz (i.e., el paso de reescritura se aplica sobre la posición Λ de $t|_p$). Como una posición identifica unívocamente a un subtérmino dentro un término, la noción de descendiente para (sub)términos se extiende directamente a partir de la correspondiente noción para posiciones.

Una noción equivalente (y más intuitiva) de descendiente de una posición se propone en [Klop y Middeldorp, 1989]. Sea $t \rightarrow^* t'$ una secuencia de reducción y s un subtérmino de t. Los descendientes de s en t' se calculan como sigue: subrayar la raíz de s y realizar la secuencia de reducción $t \rightarrow^* t'$. Entonces, cada subtérmino de t' con la raíz subrayada es un descendiente de s.

[12]Los sistemas inductivamente secuenciales son una clase de programas basados en constructores incluidos en los sistemas *fuertemente secuenciales* que, a su vez, son ampliamente utilizados para implementar eficazmente cómputos necesarios

Ejemplo 6 Consideremos la operación que duplica su argumento mediante una suma (las reglas para la suma aparecen en el Ejemplo 5).

$\mathtt{R} \;=\; \mathtt{double(x)} \rightarrow \mathtt{x} + \mathtt{x}$

En la siguiente reducción de $\mathtt{double}(\mathtt{0}+\mathtt{0})$ mostramos, como términos subrayados, los descendientes de $\mathtt{0}+\mathtt{0}$:

$\mathtt{double}(\underline{\mathtt{0}+\mathtt{0}}) \rightarrow_{1,\mathtt{R}} \; (\underline{\mathtt{0}+\mathtt{0}}) + (\underline{\mathtt{0}+\mathtt{0}})$

El conjunto de descendientes de la posición 1 en la reducción anterior es $\{1, 2\}$.

Para dar una definición precisa de la clase de programas inductivamente secuenciales y de la estrategia de *narrowing* necesario, presentamos a continuación el concepto de *árbol definicional* [Antoy, 1992]. Se dice que $\mathcal{P}$ es un árbol definicional con patrón π si y sólo si la profundidad de $\mathcal{P}$ es finita, π es un patrón y se da alguno de los siguientes casos:

$\mathcal{P} = rule(\pi \rightarrow r)$, donde $\pi \rightarrow r$ es una variante de alguna regla de $\mathcal{R}$.

$\mathcal{P} = branch(\pi, o, \pi_1, \ldots, \pi_n)$, donde o es una posición de una variable en π (la *posición inductiva*), $c_1, \ldots, c_n \in \mathcal{C}$ son diferentes constructores, para algún $n > 0$, y, para todo $\pi_1, \ldots, \pi_n$ tenemos que $\pi_i = \pi[c_i(\overline{x_{n_i}})]_o$, donde n_i es la aridad de c_i y $\overline{x_{n_i}}$ son variables nuevas.

El *árbol definicional* de un conjunto finito de patrones lineales (para una función)[13] S es un conjunto no vacío $\mathcal{P}$ de patrones lineales parcialmente ordenados por el orden de subsumción, que tiene las siguientes propiedades:

Propiedad de la raíz: Existe un elemento mínimo llamado *patrón* del árbol definicional que denotamos por $pattern(\mathcal{P})$.

Propiedad de las hojas: Los elementos maximales, llamados *hojas*, son los elementos de S. El resto de los elementos, los no maximales, se llaman *ramas*.

Propiedad del padre: Si $\pi \in \mathcal{P}$, $\pi \neq pattern(\mathcal{P})$, existe un único $\pi' \in \mathcal{P}$, llamado el *padre* de π (siendo π el *hijo* de π'), tal que $\pi' < \pi$ y no existe otro patrón $\pi'' \in \mathcal{T}(\mathcal{C} \cup \mathcal{F}, \mathcal{X})$ que verifique $\pi' < \pi'' < \pi$.

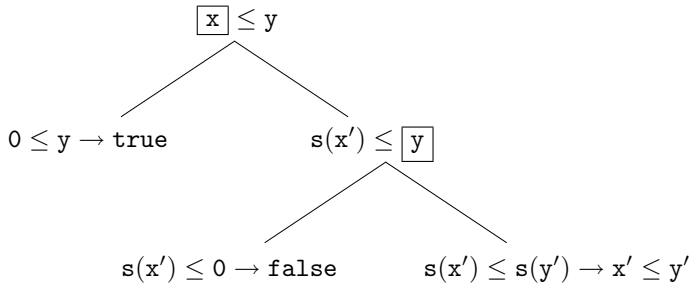
Propiedad de inducción: Dado $\pi \in \mathcal{P} \backslash S$, existe una posición o de $pattern(\mathcal{P})$ con $pattern(\mathcal{P})|_o \in \mathcal{X}$ (llamada *posición inductiva*), y constructores $c_1, \ldots, c_n \in \mathcal{C}$ con $c_i \neq c_j$ para $i \neq j$, tal que, para todo $\pi_1, \ldots, \pi_n$ que tiene como padre a π, $\pi_i = \pi[c_i(\overline{x_{n_i}})]_o$ (donde $\overline{x_{n_i}}$ son variables nuevas distintas) para todo $1 \leq i \leq n$.

[13] En contraste con la definición original de [Antoy, 1992], en lo que sigue utilizaremos la siguiente definición "declarativa" de [Antoy, 1997] ya que resulta más apropiada para demostrar los resultados y propiedades de las transformaciones basadas en *narrowing* necesario que estudiaremos más adelante.

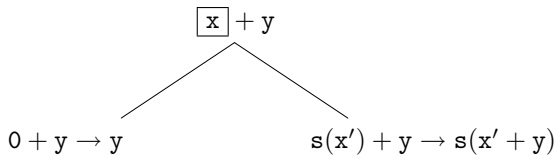
Si $\mathcal{R}$ es un programa o sistema de reescritura de términos ortogonal y f/n es un símbolo de función definido, decimos que $\mathcal{P}$ es un *árbol definicional de* f si $pattern(\mathcal{P}) = f(\overline{x_n})$ para una serie de variables distintas $\overline{x_n}$, siendo las hojas de $\mathcal{P}$ todas y cada una de las variantes de las partes izquierdas de las reglas de $\mathcal{R}$ que definen f. Debido a la ortogonalidad de $\mathcal{R}$, podemos asignar a cada hoja una sola regla que define f. Una función definida se llama *inductivamente secuencial* si tiene un árbol definicional asociado, lo que intuitivamente se corresponde con una definición "por casos" de la misma. Un sistema de reescritura $\mathcal{R}$ es *inductivamente secuencial* si todas sus funciones definidas son inductivamente secuenciales. Un SRT inductivamente secuencial puede verse como un conjunto de árboles definicionales, definiendo cada uno de ellos un símbolo de función distinto. Como en general pueden existir más de un árbol definicional para una misma función inductivamente secuencial, en lo que sigue asumimos que existe un árbol definicional fijo para cada función definida.

Para facilitar la comprensión del concepto de árbol definicional, a menudo es conveniente dar una representación gráfica en la que cada nodo se etiqueta con un patrón, la posición inductiva en las ramas se enmarca dentro de una caja, y las hojas contienen las reglas correspondientes.

Ejemplo 7 El siguiente gráfico muestra un árbol definicional para la función "$\leq$" del Ejemplo 5:



A su vez, un árbol definicional para la función "+" del mismo Ejemplo 5 puede ilustrarse como sigue:



La siguiente proposición muestra que cualquier función definida por una sola regla es siempre inductivamente secuencial.

Proposición 2.5.1 [Alpuente *et al.*, 1999b] *Si $f(\overline{t_n})$ es un patrón lineal, entonces existe un árbol definicional para el conjunto $\{f(\overline{t_n})\}$ con patrón $f(\overline{x_n})$.*

Para definir el cálculo de *narrowing* necesario asumimos que t es un término encabeza-

do por un símbolo definido y que $\mathcal{P}$ es un árbol definicional con patrón $pattern(\mathcal{P}) = \pi$ tal que $\pi \leq t$ [14]. Definimos una función λ de términos y árboles definicionales en conjuntos de tuplas (posición, regla, sustitución) como la menor relación que satisface las siguientes propiedades. Consideramos dos casos para $\mathcal{P}$ [15]:

1. Si π es una hoja, i.e., $\mathcal{P} = \{\pi\}$, y $\pi \to r$ es una variante de una regla del programa, entonces

$$\lambda(t, \mathcal{P}) = \{(\Lambda, \pi \to r, \epsilon)\} \ .$$

2. Si π es una rama, consideramos la posición inductiva o de π y algún hijo $\pi_i = \pi[c_i(\overline{x_n})]_o \in \mathcal{P}$. Sea $\mathcal{P}_i = \{\pi' \in \mathcal{P} \mid \pi_i \leq \pi'\}$ el árbol definicional donde todos los patrones son instancias de π_i. Entonces tenemos los siguientes casos para el subtérmino $t|_o$:

$$\lambda(t, \mathcal{P}) \ni \begin{cases} (p, R, \tau\sigma) & \text{si } t|_o = x \in \mathcal{X},\ \tau = \{x \mapsto c_i(\overline{x_n})\}, \text{ y} \\ & \quad (p, R, \sigma) \in \lambda(t\tau, \mathcal{P}_i); \\ (p, R, \sigma) & \text{si } t|_o = c_i(\overline{t_n}) \text{ y } (p, R, \sigma) \in \lambda(t, \mathcal{P}_i); \\ (o.p, R, \sigma) & \text{si } t|_o = f(\overline{t_n}) \text{ para } f \in \mathcal{F} \text{ y } (p, R, \sigma) \in \lambda(t|_o, \mathcal{P}') \\ & \quad \text{donde } \mathcal{P}' \text{ es un árbol definicional para } f. \end{cases}$$

En términos informales, *narrowing* necesario aplica una regla si es posible (caso 1), o busca el subtérmino correspondiente a la posición inductiva de la rama (caso 2): si es una variable, ésta se instancia al constructor de algún hijo; si ya es un constructor, procedemos con el hijo correspondiente; si es una función, entonces la evaluamos recursivamente aplicando *narrowing* necesario. De esta forma, la nueva estrategia difiere de los lenguajes funcionales perezosos sólo en la (posible) instanciación de las variables libres.

Obsérvese que durante la computación de λ componemos la sustitución calculada en cada paso recursivo (que puede ser la identidad) con la sustitución obtenida en pasos anteriores. Así, cada paso de *narrowing* necesario puede representarse como $(p, R, \varphi_1 \cdots \varphi_k)$, donde cada φ_j es o bien la sustitución identidad o bien el reemplazamiento (computado en cada paso recursivo) de una sola variable. A esta notación se le llama *representación canónica* de un paso de *narrowing* necesario. Al igual que en los procedimientos de demostración de la programación lógica, asumimos que los árboles definicionales siempre contienen variables nuevas cuando se usan en pasos sucesivos de *narrowing*. Esto implica que todas las sustituciones computadas son idempotentes (asumiremos implícitamente esta propiedad en lo que sigue).

[14] Cuando t está encabezado por un símbolo constructor, la estrategia se generaliza de forma sencilla, como se indica en [Antoy *et al.*, 1994].

[15] Esta descripción dada en [Alpuente *et al.*, 1999b] es ligeramente diferente a la mostrada por [Antoy *et al.*, 1994] pero genera el mismo conjunto de pasos de *narrowing* necesario.

Definición 2.5.2 (*narrowing* necesario $\leadsto_{nn}$) La relación de narrowing necesario $\leadsto_{nn}$ se define como una instancia de la relación de narrowing genérico con estrategia donde la estrategia $\varphi^{nn}(t) = \lambda(t, \mathcal{P})$, siendo t un término encabezado por un símbolo de función definido con árbol definicional $\mathcal{P}$. Las computaciones se realizan usando las reglas del programa extendido $\mathcal{R}_+ = (\mathcal{R} \cup Eq^{out})$.

De esta forma tenemos que para todo $(p, R, \sigma) \in \lambda(t, \mathcal{P})$, $t \overset{p,R,\sigma}{\leadsto}_{nn} t'$ es un paso de *narrowing* necesario. Decimos, además, que este paso es *determinista* si $\lambda(t, \mathcal{P})$ contiene exactamente un solo elemento. El siguiente ejemplo ilustra la definición anterior, al tiempo que muestra algunas de las ventajas de las que disfruta la relación de *narrowing* necesario frente al *narrowing* perezoso presentado en la sección anterior.

Ejemplo 8 Consideremos de nuevo los árboles definicionales del Ejemplo 7. La función λ computa el siguiente conjunto para el término inicial $\mathtt{w} \leq \mathtt{w} + \mathtt{w}$:

$$\{(\Lambda, \mathtt{0} \leq \mathtt{y} \to \mathtt{true}, \{\mathtt{w} \mapsto \mathtt{0}\}),\ (\mathtt{2}, \mathtt{s(x)} + \mathtt{y} \to \mathtt{s(x + y)}, \{\mathtt{w} \mapsto \mathtt{s(x)}\})\}$$

Lo que corresponde a los siguientes pasos de *narrowing* necesario:

$$\underline{\mathtt{w} \leq \mathtt{w} + \mathtt{w}} \overset{\{\mathtt{w} \mapsto \mathtt{0}\}}{\leadsto}_{\mathtt{nn}} \mathtt{true} \qquad (*)$$

$$\mathtt{w} \leq \underline{\mathtt{w} + \mathtt{w}} \overset{\{\mathtt{w} \mapsto \mathtt{s(x)}\}}{\leadsto}_{\mathtt{nn}} \mathtt{s(x)} \leq \mathtt{s(x + s(x))}$$

Obsérvese que la estrategia de *narrowing* perezoso generaría un paso más de la forma:

$$\mathtt{w} \leq \underline{\mathtt{w} + \mathtt{w}} \overset{\mathtt{2,0+y} \to \mathtt{y}, \{\mathtt{w} \mapsto \mathtt{b}\}}{\leadsto}_{\mathtt{nn}} \mathtt{0} \leq \mathtt{0}$$

Este paso (perezoso, pero no necesario) es obviamente redundante, ya que conduce a una derivación de éxito (en dos pasos) en la que se repite la solución ya obtenida tras el paso $(*)$.

Los siguientes resultados muestran algunas propiedades importantes del *narrowing* necesario. En primer lugar, observamos que cada sustitución computada en un paso de *narrowing* necesario instancia solamente variables del término inicial.

Proposición 2.5.3 [Alpuente *et al.*, 1999b] *Si $(p, R, \varphi_1 \cdots \varphi_k) \in \lambda(t, \mathcal{P})$ es un paso de narrowing necesario, entonces, para $i = 1, \ldots, k$, $\varphi_i = \epsilon$ o $\varphi_i = \{x \mapsto c(\overline{x_n})\}$ (donde $\overline{x_n}$ son variables diferentes) con $x \in \mathcal{V}ar(t\varphi_1 \cdots \varphi_{i-1})$.*

El siguiente lema muestra que, para pasos de *narrowing* necesario diferentes (que computan sustituciones diferentes), siempre existe una variable que es instanciada a constructores diferentes:

Lema 2.5.4 [Alpuente *et al.*, 1999b] *Sea t un término encabezado por un símbolo de función definido, $\mathcal{P}$ un árbol definicional con $pattern(\mathcal{P}) \leq t$ y $(p, R, \varphi_1 \cdots \varphi_k)$, $(p', R', \varphi'_1 \cdots \varphi'_{k'}) \in \lambda(t, \mathcal{P})$, $k \leq k'$. Entonces, para todo $i \in \{1, \ldots, k\}$,*

- *o bien $\varphi_1 \cdots \varphi_i = \varphi'_1 \cdots \varphi'_i$, o bien*

- *existe algún* $j < i$ *tal que*

  1. $\varphi_1 \cdots \varphi_j = \varphi'_1 \cdots \varphi'_j$, *y*
  2. $\varphi_{j+1} = \{x \mapsto c(\cdots)\}$ *y* $\varphi'_{j+1} = \{x \mapsto c'(\cdots)\}$ *con* $c \neq c'$.

Para programas inductivamente secuenciales, *narrowing* necesario es correcto y completo con respecto a ecuaciones estrictas y sustituciones constructoras como soluciones. Además, *narrowing* necesario nunca calcula soluciones redundantes, es decir, todas las respuestas computadas son *independientes*. Dos sustituciones θ y σ son independientes sobre un conjunto de variables V sii existe alguna variable $x \in V$ tal que $x\theta$ y $x\sigma$ no son unificables.

Teorema 2.5.5 (corrección, completitud y minimalidad [Antoy *et al.*, 1994]**)** *Sea* $\mathcal{R}$ *un programa inductivamente secuencial y* e *una ecuación.*

1. *(Corrección) Si* $e \overset{\sigma}{\leadsto}{}^*{}_{nn} true$ *es una derivación de narrowing necesario, entonces* σ *es una solución para* e.
2. *(Completitud) Para cada sustitución constructora* σ *que sea una solución de* e*, existe una derivación de narrowing necesario* $e \overset{\sigma'}{\leadsto}{}^*{}_{nn}$ *true tal que* $\sigma' \leq \sigma\ [\mathcal{V}ar(e)]$.
3. *(Minimalidad) Si* $e \overset{\sigma}{\leadsto}{}^*{}_{nn}$ *true y* $e \overset{\sigma'}{\leadsto}{}^*{}_{nn}$ *true son dos derivaciones de narrowing necesario distintas, entonces* σ *y* σ' *son independientes sobre* $\mathcal{V}ar(e)$.

Para finalizar este apartado, presentamos a continuación una última e interesante propiedad de la estrategia de *narrowing* necesario. En general, una ventaja importante de los lenguajes lógico funcionales frente a los lenguajes lógicos puros es su mejor comportamiento operacional al evitar pasos de computación no deterministas. Una razón por la que se cumple este hecho está en que las estrategias (necesarias) dirigidas por la demanda pueden evitar la evaluación de expresiones potencialmente indeterministas [Alpuente *et al.*, 1999b]. Por ejemplo, consideremos las reglas del Ejemplo 5 y el término $\mathtt{0} \leq \mathtt{X} + \mathtt{X}$. *Narrowing* necesario evalúa este término a `true` mediante un solo paso determinista. En el programa lógico equivalente, este mismo término anidado debe ser *aplanado* y expresado como una conjunción de dos llamadas a predicado como $+(\mathtt{X}, \mathtt{X}, \mathtt{Z}) \wedge \leq(\mathtt{0}, \mathtt{Z}, \mathtt{B})$, lo que produce una evaluación indeterminista debido a la llamada al predicado $+(\mathtt{X}, \mathtt{X}, \mathtt{Z})$. Otro hecho que muestra el mejor comportamiento operacional de los lenguajes lógico funcionales es la capacidad de ciertas estrategias de evaluación, como es el caso de *narrowing* necesario, para evaluar términos básicos de una forma completamente determinista, lo cual es muy importante para asegurar una implementación eficiente de las evaluaciones funcionales puras. Esta propiedad, que es obvia en la definición del *narrowing* necesario, se formaliza en la siguiente proposición. Con este fin, decimos que un término t es evaluable *deterministamente* (con

respecto a *narrowing* necesario) si cada paso en una derivación de *narrowing* para t es determinista. Además, un término t es normalizable *deterministamente* a un término constructor c (con respecto a *narrowing* necesario) si t es evaluable deterministamente y existe una derivación de *narrowing* necesario $t \overset{\epsilon}{\leadsto}^*_{nn} c$ (i.e., c es la forma normal de t).

Proposición 2.5.6 [Alpuente *et al.*, 1999b] *Sea $\mathcal{R}$ un programa inductivamente secuencial y t un término.*

1. *Si $t \overset{\epsilon}{\leadsto}^*_{nn} c$ es una derivación de narrowing necesario, entonces t es normalizable deterministamente a c.*

2. *Si t es básico, entonces t es evaluable deterministamente.*

El mecanismo de *Narrowing necesario* puede ser implementado de forma simple y eficiente mediante la traducción de los árboles definicionales en "expresiones case" tal como se propone [Hanus y Prehofer, 1999]. Mediante el uso de "expresiones case", cada símbolo de función inductivamente secuencial esta definido por una única regla de programa, donde la parte izquierda es el símbolo de función aplicado a un conjunto de variables disjuntas entre si y su lado derecho es una representación del correspondiente árbol definicional mediante expresiones *case*. Por ejemplo, las reglas para la función $\leq$ definidas en el Ejemplo 7 se representan mediante la siguiente regla de programa:

$$\begin{array}{lllllllll} x \leq y \rightarrow x \ case\ x\ of & 0 & : & true, & & & & & \\ & s(x_1) & : & case\ y\ of & 0 & : & false, & & \\ & & & & s(y_1) & : & x_1 \leq y_1 & & \end{array}$$

Aunque esta regla no es una regla de reescritura en el sentido convencional (debido a la presencia de variables frescas), tiene una lectura operacional única que se logra mediante la especificación de una semántica particular para las expresiones *case*. Una expresión *case* se considera como un símbolo de función cuya semántica está definida por un conjunto de reglas de reescritura. Por ejemplo, en la regla anterior, la expresión *case* es una función de aridad 5:

$$case(x, 0, true, s(x_1), case(y, 0, false, s(y_1), x_1 \leq y_1))$$

junto con las siguientes reglas de reescritura, donde el símbolo "_" denota una variable anónima arbitraria

$$\begin{array}{llllllllll} case & 0 & of & 0 & : & z, & \_ & : & \_ & \rightarrow & z \\ case & s(x) & of & \_ & : & \_, & s(x) & : & z & \rightarrow & z \end{array}$$

Aunque la parte izquierda de la última regla es *no–lineal*, las multiples ocurrencias de la variable x se usan para pasar subtérminos de un lugar a otro y no para comparar los términos.

La función $\mathcal{C}ase(\mathcal{T})$ formaliza la transformación de un árbol definicional $\mathcal{T}$ a un término con expresiones *case* y se define como sigue:

$$\begin{array}{lcl} \mathcal{C}ase(rule(l \to r)) & = & r \\ \mathcal{C}ase(branch(\pi, o, \overline{\mathcal{T}_k})) & = & case\ \pi_{|_o}\ of \quad pat(\mathcal{T}_1)_{|_o} \ : \ \mathcal{C}ase(\mathcal{T}_1), \\ & & \qquad\qquad\qquad\quad \vdots \\ & & \qquad\qquad\qquad\quad pat(\mathcal{T}_k)_{|_o} \ : \ \mathcal{C}ase(\mathcal{T}_k), \end{array}$$

Si $\mathcal{T}$ es el árbol definicional correspondiente al *patrón* $f(\bar{x}_n)$ de la función f con aridad n, entonces

$$f(\bar{x}_n) \to \mathcal{C}ase(\mathcal{T})$$

es la nueva regla de reescritura para f. Además, una expresión "$case\ \ x\ \ of\ \ \overline{p_n\ :\ x_n}$" que se considera como una función con aridad $2n+1$ con las siguientes n reglas.

$$\begin{array}{llllllllll} case & p_1 & of & p_1 & : & z, \ldots, & \_ & : & \_ & \to \ z \\ \vdots & & & & & & & & & \\ case & p_n & of & \_ & : & \_, \ldots, & p_n & : & z & \to \ z \end{array}$$

En resumen, cada función inductivamente secuencial f se transforma en una nueva función (llamada también f) definida por una sola regla de reescritura, cuya parte izquierda es el término $f(\bar{x})$, con $\bar{x}$ una tupla de variables distintas, y donde la correspondiente parte derecha es una "construcción case" que representa el correspondiente árbol definicional. El siguiente ejemplo ilustra la transformación anteriormente explicada sobre expresiones *case*, ver [Hanus y Prehofer, 1999] para una discusión más amplia.

Ejemplo 9 *Consideremos el siguiente programa inductivamente secuencial* `double/1` *para calcular la adición y el doble de los números naturales en notación unaria.*

$$\begin{array}{rcl} \texttt{add(0,x)} & \to & \texttt{x.} \\ \texttt{add(s(x),y)} & \to & \texttt{s(add(x,y)).} \\ \texttt{double(0)} & \to & \texttt{0.} \\ \texttt{double(s(x))} & \to & \texttt{s(s(double(x))).} \end{array}$$

Las reglas de este programa pueden ser representadas por el siguiente árbol definicional.

$$\begin{array}{lll} branch(\texttt{add(x,y)}, \texttt{1}, & rule(\texttt{add(0,x)} & \to \ \texttt{x}), \\ & rule(\texttt{add(s(x),y)} & \to \ \texttt{s(add(x,y))})). \end{array}$$

$$\begin{array}{lll} branch(\texttt{double(x)}, \texttt{1}, & rule(\texttt{double(0)} & \to \ \texttt{0}), \\ & rule(\texttt{double(s(x))} & \to \ \texttt{s(s(double(x)))})). \end{array}$$

Las reglas transformadas a expresiones case *son como sigue*

$$
\begin{aligned}
\mathtt{add(x,y)} &\rightarrow \mathtt{case_1(x,0,y,s(x1),s(add(x1,y)))}.\\
\mathtt{case_1(0,0,z,\_,\_)} &\rightarrow \mathtt{z}.\\
\mathtt{case_1(s(x),\_,\_,s(x),z)} &\rightarrow \mathtt{z}.\\
\mathtt{double(x)} &\rightarrow \mathtt{case_2(x,0,0,s(x1),s(s(double(x1))))}.\\
\mathtt{case_2(0,0,z,\_,\_)} &\rightarrow \mathtt{z}.\\
\mathtt{case_2(s(x),\_,\_,s(x),z)} &\rightarrow \mathtt{z}.
\end{aligned}
$$

Mediante la transformación anterior es posible probar que existe una *equivalencia fuerte* entre las derivaciones de *narrowing necesario* ejecutado con el programa original y las derivaciones por *narrowing leftmost outermost*, en el respectivo programa transformado a expresiones *case*, como lo muestra el siguiente teorema, tomado de [Hanus y Prehofer, 1999] y que transcribimos con una ligera adaptación notacional a nuestro marco. Si bien Hanus & Prehofer no consideran reglas condicionales de manera explicita, esta caracterización no constituye una dificultad sustancial para nuestra propuesta (esto puede ser resuelto mediante el proceso de "decondicionalización", esto es, la codificación de las reglas condicionales se realiza por medio de la función predefinida "if" [Antoy, 2001]).

Para una formulación mas simple del teorema establecemos la siguiente notación: denotamos por $\mathcal{R}$ un sistema de reescritura inductivamente secuencial, por $\mathcal{R}'$ la correspondiente versión trasladada que contiene exactamente una regla de reescritura para cada función definida por $\mathcal{R}$, y por $\mathcal{R}_{case}$ las reglas *case* de reescritura adicionales.

Teorema 2.5.7 *Sea g un objetivo. Para cada derivación de* narrowing necesario $g \overset{\sigma}{\leadsto}{}^*{}_{nn} \top$ *con respecto a $\mathcal{R}$ existe una derivación de* narrowing leftmost outermost $g \overset{\sigma}{\leadsto}{}^*{}_{out} \top$ *con respecto a $\mathcal{R}' \cup \mathcal{R}_{case}$ y viceversa*

Una transformación similar se presenta en [Zartmann, 1997], donde los programas inductivamente secuenciales se traducen a una forma *uniforme*, la cual tiene únicamente reglas planas con partes izquierdas no–subunificables dos a dos, donde la *equivalencia fuerte* entre derivaciones de *narrowing necesario* y derivaciones de *narrowing leftmost outermost* también es cierta.

Es importante tener en cuenta que se necesitan diferentes funciones *case*, diferenciadas por ejemplo con el uso de subíndices, para expresiones *case* asociadas con diferentes patrones. La idea fundamental de la transformación es que, después de que los patrones de emparejamiento se han compilado a expresiones *case*, los árboles definicionales no son necesarios para guiar los pasos de reducción, si no que son guiados por la distinción de los *case* en las partes derechas de cada regla. Entonces, vía esta transformación no perdemos (demasiada) generalidad debido al desarrollo de nuestra metodología para el (mas simple) *narrowig leftmost–outermost*; esto simplifica el razonamiento acerca de computaciones y, por consiguiente, la verificación de propiedades semánticas, por ejemplo, la completitud fuerte. Nosotros preferimos mantener

nuestro marco simple mientras no perdamos, por supuesto, demasiada generalidad.

2.6 Algunos aspectos generales

Para simplificar la notación, en adelante con el símbolo $I\!R_\varphi$ denotamos la clase de programas que satisfacen las condiciones para la completitud para la estrategia φ. Además, para referirnos a una $\mathcal{V}$–interpretación, cuando no haya lugar a confusión, simplemente la enunciaremos como "interpretación", ya que en nuestro marco todas las interpretaciones contienen variables.

2.6.1 Composición paralela ecuacional

Informalmente, la *composición paralela*, denotada por $\Uparrow$, es la operación de unificación generalizada a sustituciones, ver [Hermenegildo y Rossi, 1989; Jacquet, 1989; Palamidessi, 1990]. Concretamente, cuando dos subobjetivos (del mismo objetivo) se explotan en paralelo, las sustituciones computadas (independientemente) deben ser combinadas para obtener el resultado final. Esta "combinación" se puede realizar como sigue. Dadas dos sustituciones idempotentes θ_1 y θ_2, definimos:

$$\theta_1 \Uparrow \theta_2 = mgu(\widehat{\theta_1} \cup \widehat{\theta_2}).$$

La composición paralela de sustituciones es idempotente, conmutativa, asociativa y tiene como elemento neutro ϵ. El operador $\Uparrow$ se extiende a conjuntos de sustituciones de la forma:

$$\Theta_1 \Uparrow \Theta_2 = \begin{cases} \bigcup_{\theta_1 \in \Theta_1, \theta_2 \in \Theta_2} \{\theta_1 \Uparrow \theta_2\} & \text{si es diferente de } \{fail\} \\ \emptyset & \text{en otro caso.} \end{cases}$$

La composición paralela fue propuesta en [Palamidessi, 1990] como base para una caracterización composicional de la semántica de los programas lógicos en cláusulas de Horn. Podemos generalizar la noción de composición paralela para considerar la unificación bajo teorías de Horn ecuacionales. En la siguiente definición formalizamos la noción de *composición paralela ecuacional*, denotada por $\Uparrow_{\mathcal{E}}$ [Alpuente *et al.*, 1996].

Definición 2.6.1 *Sean $\theta_1, \theta_2 \in Sub$. Definimos el operador $\Uparrow_{\mathcal{E}} : Sub \times Sub \to \wp Sub$ como sigue:*

$$\theta_1 \Uparrow_{\mathcal{E}} \theta_2 \ = \mathcal{U}_{\mathcal{E}}(\widehat{\theta_1} \cup \widehat{\theta_2}).$$

En donde $\mathcal{U}_{\mathcal{E}}(E)$ denota el conjunto de todos los $\mathcal{E}$-unificadores de E. Es inmediato demostrar que el operador $\Uparrow_{\mathcal{E}}$ es conmutativo, asociativo y tiene como elemento nulo $fail$. Veamos un ejemplo, de [Alpuente *et al.*, 1996], que ilustra el comportamiento del operador semántico de composición paralela $\Uparrow_{\mathcal{E}}$.

Ejemplo 10 *Sea* $\mathcal{R} = \{f(0) = 0,\ f(g(x)) = g(x),\ g(0) = c(0),\ g(c(x)) = g(x)\}$.

1. *Si* $\theta_1 = \{x/g(z)\}$ *y* $\theta_2 = \{x/c(z)\}$, *entonces* $\{x/c(0), z/0\} \in \theta_1 \Uparrow_{\mathcal{R}} \theta_2$.

2. *Si* $\theta_1 = \{x/f(0)\}$ *y* $\theta_2 = \{x/g(z)\}$, *entonces* $\theta_1 \Uparrow_{\mathcal{R}} \theta_2 = \emptyset$.

El operador $\Uparrow_{\mathcal{E}}$ se puede extender a conjuntos de sustituciones como sigue.

Definición 2.6.2 *Dados* $\Theta_1, \Theta_2 \in \wp Sub$, *definimos:*

$$\Theta_1 \Uparrow_{\mathcal{E}} \Theta_2 = \bigcup_{\theta_1 \in \Theta_1, \theta_2 \in \Theta_2} \theta_1 \Uparrow_{\mathcal{E}} \theta_2.$$

2.6.2 Estrategia independiente del entorno

En este apartado, introducimos las condiciones que garantizan que la relación de *narrowing* genérico con estrategia $\leadsto_\varphi$ es composicional con respecto al operador conjuntivo, i.e. con respecto a la unión o concatenación de secuencias de ecuaciones, usando el operador sintáctico $\Uparrow$ de composición paralela de respuestas. En primer lugar, introducimos una condición sobre las estrategias de *narrowing* que limita las posibles instancias de la relación de *narrowing* genérico.

Definición 2.6.3 (estrategia independiente del entorno) *Una estrategia de narrowing* φ *es* independiente del entorno *si para toda derivación de la forma:*

$$(g_1, g_2) \overset{\sigma}{\leadsto}{}^*{}_{\varphi} (g_1', g_2)\sigma$$

en la que no se ha explotado ninguna ocurrencia correspondiente a g_2, *se cumple que:*

$$\varphi(g_2\sigma) \subseteq \varphi(g_2).$$

Por abuso de notación, en ocasiones diremos que la propia relación $\leadsto_\varphi$ *es independiente del entorno.*

Informalmente, una estrategia de *narrowing* se dice independiente del entorno si, dado un objetivo compuesto (g_1, g_2) y un SRTC $\mathcal{R}$ las sustituciones computadas por pasos de *narrowing* sobre ecuaciones pertenecientes a g_1 pueden restringir las ocurrencias explotables de g_2, pero nunca pueden dar lugar a nuevos *redexes* en g_2.

Lema 2.6.4 *Sea* $\mathcal{R}$ *en* $\mathbb{I}\!\mathbb{R}_\varphi$, *la estrategia* $\varphi \in \{inn, out\}$ *de narrowing condicional con respecto a* R *es independiente del entorno.*

Demostración. El resultado se sigue trivialmente del hecho de que si $\mathcal{R}$ está en $\mathbb{I}\!\mathbb{R}_\varphi$, entonces la sustitución computada es constructora. Entonces, dado un objetivo de la forma (g_1, g_2) si existe una derivación de la forma:

$$(g_1, g_2) \overset{\sigma}{\leadsto}{}^*{}_{\varphi} (g_1', g_2)\sigma$$

en la que no se ha explotado ninguna ocurrencia correspondiente a g_2 como σ es una sustitución constructora y por lo tanto no introduce redexes en g_2, entonces se cumple

que:

$$\varphi(g_2\sigma) \subseteq \varphi(g_2).$$

□

Lema 2.6.5 *Sea $\mathcal{R}$ un programa en $I\!R_\varphi$ y g_1, g_2 un objetivo. $(g_1, g_2) \overset{\theta}{\leadsto}{}^*{}_\varphi \top$ sii $g_1 \overset{\sigma_1}{\leadsto}{}^*{}_\varphi \top, g_2 \overset{\sigma_2}{\leadsto}{}^*{}_\varphi \top$ tal que $\theta = \sigma_1 \Uparrow \sigma_2$, para $\varphi \in \{inn, out\}$.*

Demostración. En [Vidal, 1996], la demostración está dada para cualquier estrategia que sea "independiente del entorno" y por el Lema 2.6.4 la estrategias $\varphi \in \{inn, out\}$ son independientes del entorno. □

Capítulo 3

Denotación de un programa lógico funcional

Cuando definimos una semántica para un lenguaje de programación resulta esencial especificar qué propiedades computacionales nos interesa "observar". En el caso de los programas lógicos funcionales, uno puede estar interesado en observar los valores calculados por las derivaciones de éxito (como son las sustituciones de respuesta computadas), por derivaciones parciales o por derivaciones falladas finitamente. Para cada posible elección, una semántica que modele correctamente el *observable* deberá mostrar una noción diferente de equivalencia de programas. Como ya hemos dicho, a lo largo de este trabajo consideraremos como *observable* el conjunto de las sustituciones de respuesta computada.

3.1 $\mathcal{V}$-Interpretaciones

La semántica operacional es una función del conjunto de programas a un conjunto de "denotaciones" de programas tales que, dado un programa $\mathcal{R}$, devuelve un conjunto de "resultados" de las computaciones de $\mathcal{R}$. Para programas funcionales, la semántica operacional se puede definir en términos del conjunto de todos los valores que un programa puede calcular. Para el caso de lenguajes lógico funcionales, dado que las expresiones (en nuestro caso ecuaciones) pueden ser resueltas por *narrowing*, debemos considerar una semántica no básica, la cual se puede plantear en términos del conjunto de ecuaciones que al ser evaluadas por *narrowing* con estrategia φ y el programa $\mathcal{R}$ su resultado es $\top$. Con el objetivo de formular una semántica que modele el conjunto de respuestas computadas para un programa $\mathcal{R}$, la base de Herbrand usual debe ser extendida al conjunto de todas las ecuaciones (posiblemente) no básicas, módulo

renombramiento de variables.

Definición 3.1.1 *$\mathcal{H}_\mathcal{V}$ denota al $\mathcal{V}$-universo de Herbrand que permite variables en sus elementos, y se define como $\tau(\Sigma \cup \mathcal{V})/_\cong$*

La relación $\cong$ es la relación de equivalencia inducida por el preorden $\leq$ de "generalidad relativa" entre términos. Por simplicidad, los elementos de $\mathcal{H}_\mathcal{V}$ tienen la misma representación que los elementos $\tau(\Sigma \cup \mathcal{V})/_\cong$ y también se llaman términos.

Definición 3.1.2 *La $\mathcal{V}$-base de Herbrand es el conjunto de todas las ecuaciones $s = t$ módulo renombramiento de variables, donde $s, t \in \mathcal{H}_\mathcal{V}$. Se denota por $\mathcal{B}_\mathcal{V}$.*

Definición 3.1.3 *Una $\mathcal{V}$-Interpretación es un subconjunto de la $\mathcal{V}$-base de Herbrand.*

Dado que $[s]$ es el conjunto de *instancias básicas* (*ground*) de la expresión s, entonces la base de Herbrand estándar B es igual a $[\mathcal{B}_\mathcal{V}]$ (más estrictamente hablando $\mathcal{B} = \bigcup_{e \in \mathcal{B}_\mathcal{V}} [e]$). El preorden $\leq$ sobre $\tau(\Sigma \cup \mathcal{V})$ induce una relación de orden sobre $\tau(\Sigma \cup \mathcal{V})/\cong$ (y, por tanto, también sobre $\mathcal{H}_\mathcal{V}$). El orden sobre $\mathcal{H}_\mathcal{V}$ induce un orden sobre $\mathcal{B}_\mathcal{V}$, donde $s' = t' \leq s = t$ si $s' \leq s$ y $t' \leq t$. El conjunto potencia de $\mathcal{B}_\mathcal{V}$ es un retículo completo bajo la inclusión de conjuntos.

Siguiendo a [Alpuente *et al.*, 2001a,c] a continuación presentamos una semántica $\mathcal{F}^{ca}_\varphi(\mathcal{R})$ para el programa $\mathcal{R}$ tal que las sustituciones de respuestas computadas por la estrategia de *narrowing* φ para cualquier objetivo (posiblemente conjuntivo) g en $\mathcal{R}$ se pueden obtener a partir de $\mathcal{F}^{ca}_\varphi(\mathcal{R})$ por simple unificación de las ecuaciones del objetivo con las ecuaciones de la denotación. Las ecuaciones en el objetivo deben ser aplanadas primero, es decir, los subtérminos se deben desanidar de tal manera que la estructura del término sea directamente accesible a la unificación.

Definición 3.1.4 (Objetivo plano con respecto a φ)
Una ecuación plana *es una ecuación de la forma $f(d_1, \ldots, d_n) = d$ o $d_1 =_\varphi d_2$, donde $d, d_1, \ldots, d_n \in \tau(\mathcal{C} \cup V)$ son términos constructores. Un* objetivo plano *es un conjunto de ecuaciones planas.*

Se debe notar que en el caso en que la estrategia sea *outermost*, $\varphi = out$, un objetivo plano puede contener las dos clases de igualdad. La igualdad estricta, $\approx$, da a la igualdad un significado más débil, de identidad entre objetos finitos, debido a que está definida solo para estructuras de datos finitas y completamente determinadas. La igualdad estándar (no–estricta) $=$, está definida sobre estructuras de datos parcialmente determinadas o infinitas [Giovannetti *et al.*, 1991; Moreno-Navarro y Rodríguez-Artalejo, 1992]. Entonces, después de realizar el aplanamiento de acuerdo con la estrategia φ, esperamos resultados diferentes para un mismo objetivo, como lo muestra el siguiente ejemplo.

Ejemplo 11 *Consideremos de nuevo el programa $\mathcal{R}$ dado en el Ejemplo 1:*

$$\begin{array}{rcl} \mathtt{add(0,x)} & \rightarrow & \mathtt{x}. \\ \mathtt{add(s(x),y)} & \rightarrow & \mathtt{s(add(x,y))}. \\ \mathtt{double(0)} & \rightarrow & \mathtt{0}. \\ \mathtt{double(s(x))} & \rightarrow & \mathtt{s(s(double(x)))}. \end{array}$$

Dado el objetivo[1]*:* $\mathtt{g} \equiv \mathtt{add(double(s(0)),s(0))} = \mathtt{s^3(0)}$.
El resultado de aplanar el objetivo g es $\mathtt{add(w,s(0))} = \mathtt{z},\ \mathtt{double(s(0))} = \mathtt{w},\ \mathtt{s^3(0)} = \mathtt{z}$ *utilizando la estrategia* $\varphi = inn$
En el caso en que $\varphi = out$ *y dado el objetivo:* $\mathtt{g} \equiv \mathtt{add(double(s(0)),s(0))} \approx \mathtt{s^3(0)}$.
El resultado de aplanar el objetivo g es $\mathtt{add(w,s(0))} = \mathtt{v},\ \mathtt{double(s(0))} = \mathtt{w},\ \mathtt{s^3(0)} \approx \mathtt{z}, \mathtt{v} \approx \mathtt{z}$

Como se puede observar en el Ejemplo 11 para el caso $\varphi = out$ aparecen ecuaciones de la forma $s \approx t$ con $s, t \in \mathcal{V}$, que serán resueltas por las reglas Eq^{out} que se añaden al programa. Mientras que si $\varphi = inn$ no consideramos como objetivo inicial ecuaciones $x = y$ con $x, y \in \mathcal{V}$, puesto que al unificar con las ecuaciones de Eq^{inn} obtendríamos soluciones espúreas con el mecanismo que discutiremos más adelante. Además, en un objetivo plano con respecto a $\varphi = out$ las únicas ecuaciones no estrictas son de la forma $f(d_1, \ldots, d_n) = x$. Esto permite, por ejemplo, la eliminación de la ecuación $f(a) = x$, cuando $f(a)$ no debe ser seleccionado por *narrowing*,(es decir, cuando su valor no se requiere para reducir $g(f(a)))$, ya que la igualdad estándar $=$ es la única que obedece el axioma de reflexividad: para todo x, $x = x$.

El procedimiento de *aplanamiento* para conjuntos de ecuaciones que produce objetivos planos con respecto a *inn* y *out*, puede ser encontrado en [Bosco *et al.*, 1988; Giovannetti *et al.*, 1991]. En lo que sigue, presentamos el procedimiento de aplanamiento de forma genérica $flat_\varphi(E)$. Para simplificar el desarrollo de nuestro marco teórico planteamos una definición paramétrica de la transformación de aplanamiento. Basado en el concepto de "contexto de datos", presentamos la siguiente definición como auxiliar y previa a nuestra definición de aplanamiento paramétrico.

De acuerdo con [Giovannetti *et al.*, 1991], presentamos los términos como aplicaciones de un "contexto de datos"(es decir, un término constructor con algunos *huecos*) a términos con un operador en cabeza (términos con un símbolo de función definido en el nivel más externo). Entonces, t es representado como $e[t_1, \ldots, t_n]$, donde e es la parte externa de t y contiene únicamente símbolos constructores (si los hay) y $t_1, \ldots, t_n$ son los subtérminos de t cuyo símbolo más externo tiene un operador en cabeza. En particular, si t es un término con una operación en cabeza entonces puede ser obtenido como una aplicación de [ ] (el contexto vacío) al mismo t.

[1] En los ejemplos utilizamos la expresión $\mathtt{s^n(x)}$ para abreviar $\mathtt{s(s(}\ldots\text{n veces}\ldots\mathtt{(x)))}$.

Definición 3.1.5 (pre-aplanamiento) *La función $flat^{\varphi}_{-}(s)$ aplicada a s está definida inductivamente como sigue.*

$flat^{\varphi}_{-}(s) =$

$$\begin{cases} flat^{\varphi}_{-}(e_1), \ldots, flat^{\varphi}_{-}(e_n) & \text{si } s \equiv e_1, \ldots, e_n \\ \\ flat^{\varphi}_{-}(t_1 = z_1), \ldots, flat^{\varphi}_{-}(t_n = z_n), & \text{si } s \equiv e[t_1, \ldots, t_n] =_{\varphi} \\ flat^{\varphi}_{-}(t'_1 = y_1), \ldots, flat^{\varphi}_{-}(t'_m = y_m), & \quad e'[t'_1, \ldots, t'_m] \\ e[z_1, \ldots, z_n] =_{\varphi} e'[y_1, \ldots, y_m] & \quad z_1, \ldots, z_k, y_1, \ldots, y_k, \\ & \quad \text{son variables frescas} \\ \\ flat^{\varphi}_{-}(f(t_1, \ldots, t_n) = x) & \text{si } s \equiv x =_{\varphi} f(t_1, \ldots, t_n), \\ & \quad \text{con } x \in \mathcal{V} \\ \\ flat^{\varphi}_{-}(t_{1,1} = z_{1,1}), \ldots, flat^{\varphi}_{-}(t_{1,m_1} = z_{1,m_1}), & \text{si } s \equiv f(e_1[t_{1,1}, \ldots, t_{1,m_1}], \ldots, \\ \ldots, flat^{\varphi}_{-}(t_{n,1} = z_{n,1}), \ldots, flat^{\varphi}_{-}(t_{n,m_n} = z_{n,m_n}), & \quad e_n[t_{n,1}, \ldots, t_{n,m_n}]) =_{\varphi} x \\ f(e_1[z_{1,1}, \ldots, z_{1,m_1}], \ldots, e_n[z_{n,1}, \ldots, z_{n,m_n}]) = x & \quad z_{1,1}, \ldots, z_{1,m_1}, \ldots, z_{n,1}, \ldots, \\ & \quad z_{n,m_1} \text{ son variables frescas} \\ \\ s & \text{de otro modo} \end{cases}$$

Para el calculo de la expresión $flat^{\varphi}_{-}(s)$, se debe tener en cuenta que el conjunto de ecuaciones s, es tal que toda ecuación $e \in s$ es de la forma $e \equiv (l =_{\varphi} r)$, tanto para ecuaciones que pertenezcan a un objetivo como para las que pertenecen a una condición de una regla. El resultado de la operación es un conjunto de ecuaciones que puede tener la forma $f(d_1, \ldots, d_n) = d$ con $d_1, \ldots, d_n, d \in \tau(\mathcal{C} \cup \mathcal{V})$ tanto $\varphi = inn$ como para $\varphi = out$, o la ecuación es de la forma $d_1 =_{\varphi} d_2$ en donde claramente serán tratadas de diferente forma de acuerdo con la estrategia φ.

Por medio del aplanamiento, la unificación de una expresión compleja se descompone en varias más simples. Sin embargo, algunas ecuaciones que resultan de la transformación anterior son triviales y no contribuyen a la semántica que pretendemos establecer en nuestro marco. Por tanto, podemos eliminarlos después de aplicar el unificador más general al resto de ecuaciones. Formalizamos esta idea como sigue.

Una ecuación de la forma $x = y$, con $x, y \in V$ se denomina *ecuación trivial*. Note que, los objetivos de la forma $x \approx y$ no se consideran triviales. Dado un conjunto de ecuaciones g, definimos $split(g) = (g_1, g_2)$ como una función que divide a g en dos conjuntos disjuntos $g = g_1 \biguplus g_2$ tales que todas las ecuaciones en g_2 son triviales y ninguna ecuación en g_1 es trivial. Ahora estamos listos para completar la definición de la transformación de aplanamiento.

Definición 3.1.6 (Aplanamiento con estrategia φ) *La función $flat_{\varphi}(s)$ para una expresión s está definida como sigue. Sea $g = flat^{\varphi}_{-}(s)$ el pre–aplanamiento de s, y $split(g) = (g_1, g_2)$. Entonces, definimos $flat_{\varphi}(s) = g_1 mgu(g_2)$ como el aplanamiento*

de g.

El siguiente ejemplo ilustra las diferentes salidas de la transformación de aplanamiento de acuerdo con el valor de φ

Ejemplo 12 *Consideremos el conocido programa no terminante*[2] `from/1`*:*

$$\begin{array}{rcl} \texttt{from(x)} & \rightarrow & \texttt{[x|from(s(x))]}. \\ \texttt{first([x|y])} & \rightarrow & \texttt{x}. \end{array}$$

Sea $g \equiv \texttt{first([first([0])])} = \texttt{first([0,1])}$ *y* $\varphi = inn$*. Entonces, el pre-aplanamiento de* g *es* $g_0 = flat^{inn}_{-}(g) = (\texttt{first([0])} = \texttt{y},\ \texttt{first(y)} = \texttt{w},\ \texttt{first([0,1])} = \texttt{z},\ \texttt{w} = \texttt{z})$*, y* $split(g_0) = (\{\texttt{first([0])} = \texttt{y},\ \texttt{first(y)} = \texttt{w},\ \texttt{first([0,1])} = \texttt{z}\}, \{\ \texttt{w} = \texttt{z}\})$*. Por consiguiente* $flat_{inn}(g) = (\texttt{first([0])} = \texttt{y},\ \texttt{first(y)} = \texttt{w},\ \texttt{first([0,1])} = \texttt{w})$*.*

Ahora bien, consideremos un objetivo diferente $g' \equiv (\texttt{first(from(s(x)))} \approx \texttt{z})$ *con* $\varphi = out$*. Entonces, el pre–aplanamiento de* g' *es* $g'_0 = flat^{out}_{-}(g') = (\texttt{from(s(x))} = \texttt{y},\ \texttt{first(y)} = \texttt{w},\ \texttt{w} \approx \texttt{z})$*, y* $split(g'_0) = (\{\texttt{from(s(x))} = \texttt{y},\ \texttt{first(y)} = \texttt{w},\ \texttt{w} \approx \texttt{z}\}, \{\ \})$*. Por lo tanto* $flat_{out}(g') = g'_0$*.*

El conjunto de ecuaciones planas que se obtiene por medio de los resultados del aplanamiento no puede ser transformado de nuevo por aplanamiento. En la conversión a forma plana están implícitos los axiomas de transitividad y f-sustitutividad [Knill *et al.*, 1993].

Cualquier secuencia de ecuaciones E puede ser transformada en una secuencia plana de ecuaciones equivalente, $flat_{\varphi}(E)$. Dicha transformación conserva las respuestas computadas [Bosco *et al.*, 1988; Giovannetti *et al.*, 1991]. Para una discusión completa, ver Lema 6.3.7 en [Vidal, 1996].

Lema 3.1.7 *Sea g un conjunto de ecuaciones. $g \leadsto^*_{\theta} \top$ si y solo si $flat_{\varphi}(g) \leadsto^*_{\theta'} \top$ y $\theta_{\restriction g} = \theta'_{\restriction g}$, para $\varphi = \{inn, out\}$.*

Demostración. Dado que la transformación es correcta, dado que los únicos axiomas que se aplican en el proceso de transformación son sustitutividad, transitividad y reflexividad. Es trivialmente completa, ya que es suficiente "plegar" el conjunto de ecuaciones resultado del aplanamiento, a su posición original haciendo uso de la regla de unificación sintáctica y, a continuación, imitar en su mismo orden los pasos de la derivación para el objetivo original.

[2] Si consideramos la igualdad estricta $\approx$ en lugar de la igualdad no estricta $=$, los resultados de completitud para la estrategia *narrowing outermost* los encontramos en [Echahed, 1988, 1992]. Si estamos interesados únicamente en computar valores finitos y totales de expresiones con reglas no terminantes con un pequeño esfuerzo, ver [Frutos-Escrig y Fernández-Camacho, 1991].

En la definición de narrowing consideramos que la sustitución se aplica al objetivo resultante, entonces todos los enlaces de las variables se propagan al objetivo completo y por lo tanto no se presenta el problema de variables compartidas (*sharing*) anunciado en [Vidal, 1996].

Para completar la demostración, estamos interesados en las respuestas computadas para un objetivo y su correspondiente versión plana. Luego no es importante el orden de aplicación de las reglas en las respectivas derivaciones.

Por último, las diferencias establecidas al definir el proceso de aplanamiento tanto para $\varphi = inn$ como para $\varphi = out$ y ya comentadas, permiten mantener la corrección y completitud del calculo. Recordemos, que para el caso *out* las ecuaciones de la forma $t_1 \approx t_2$ son ejecutadas por narrowing con las reglas que definen a $\approx$. □

La semántica por punto fijo nos permite reconstruir la semántica operacional descendente (*top-down*) así como el computo ascendente (*bottom-up*) de un modelo que es completamente independiente del objetivo.

3.2 Semántica de punto fijo

En esta sección se introduce un operador de consecuencias inmediatas $T_{\mathcal{R}}^{\varphi}$ que modela las respuestas computadas con respecto a $\varphi \in \{inn, out\}$. Para cualquier programa $\mathcal{R}$, denotamos por $\Phi_{\mathcal{R}}$ el conjunto de ecuaciones de la forma $f(x_1, \ldots, x_n) = f(x_1, \ldots, x_n)$, para cada símbolo de función $f/n \in \mathcal{D}$. Denotamos por $\Im_{\mathcal{R}}^{\varphi}$ el conjunto de ecuaciones de la forma $c(x_1, \ldots, x_n) =_{\varphi} c(x_1, \ldots, x_n)$ para cada símbolo constructor c/n de $\mathcal{C}$. Como veremos más adelante, estos *axiomas reflexivos funcionales* desempeñan un papel importante en la definición en la semántica de punto fijo de $\mathcal{R}$.

En lenguajes no estrictos, si el significado debe tener carácter composicional aun en la presencia de estructuras de datos infinitas y funciones parciales, entonces los términos no normalizables, que pueden ocurrir como subtérminos en expresiones normalizables, también deben ser considerados en la denotación. Tal denotación esta acotada por el conjunto de todos los resultados parciales de las computaciones infinitas unido al orden de aproximación sobre ellos o, equivalentemente, la estructura de datos infinita definida como la menor cota superior de su correspondiente conjunto [Giovannetti *et al.*, 1991; Moreno-Navarro y Rodríguez-Artalejo, 1992]. De acuerdo con [Giovannetti *et al.*, 1991; Moreno-Navarro y Rodríguez-Artalejo, 1992], introducimos una nueva constante $\perp$ en Σ para representar los valores de aquellas expresiones que de otro modo quedarían (totalmente) indefinidas. Para el caso de funciones infinitas, al agregar reglas de la forma $f(\bar{x}) \to \perp$ damos un significado de "expresión que aproxima" a una infinita.

Definición 3.2.1 *Sea $\mathcal{R}$ un programa de $\mathbb{R}_\varphi$ e $\mathcal{I}$ una V−interpretación de Herbrand. Entonces,*

$$T^\varphi_\mathcal{R}(\mathcal{I}) = \Phi_\mathcal{R} \cup \Im^\varphi_\mathcal{R} \cup \{ e \in \mathcal{B}_V \mid \begin{array}{l} (\lambda \to \rho \Leftarrow C) \ll \mathcal{R}^\varphi, \\ \{l = r\} \cup C' \subseteq \mathcal{I}, \\ mgu(flat_\varphi(C), C') = \sigma, \\ mgu(\{\lambda = r_{|u}\}\sigma) = \theta, \\ u \in \overline{O}^\varphi(r), \\ e = (l = r[\rho]_u)\sigma\theta \ \}. \end{array}$$

donde $\mathcal{R}^\varphi = \mathcal{R}_+$ si $\varphi = inn$, mientras que $\mathcal{R}^\varphi = \mathcal{R}_+ \cup \{f(x_1, \ldots, x_n) \to \bot \mid f/n \in \mathcal{D}$ y $x_1, \ldots, x_n$ son variables distintas $\}$ si $\varphi = out$.

Las siguientes resultados nos permiten definir una semántica por punto fijo para programas lógico ecuacionales.

Proposición 3.2.2 *El operador $T^\varphi_\mathcal{R}$ es continuo sobre el réticulo completo de interpretaciones de Herbrand, $\varphi \in \{inn, out\}$.*

Demostración. Vamos a probar que, para cualquier secuencia infinita $I_1 \subseteq I_2 \subseteq \ldots$

$$T^\varphi_\mathcal{R}(\cup^\infty_{n=1}\mathcal{I}_n) = \cup^\infty_{n=1} T^\varphi_\mathcal{R}(\mathcal{I}_n)$$

Si $e \in T^\varphi_\mathcal{R}(\cup^\infty_{n=1}\mathcal{I}_n)$ por definición de $T^\varphi_\mathcal{R}$ equivale a

$$e \in \Phi_\mathcal{R} \cup \Im^\varphi_\mathcal{R} \cup \{e' \in \mathcal{B}_V \mid \begin{array}{l} (\lambda = \rho \Leftarrow C) \ll \mathcal{R}^\varphi, \ \{l = r\} \cup C' \subseteq \cup^\infty_{n=1}\mathcal{I}_n, \\ mgu(flat_\varphi(C), C') = \sigma, \ mgu(\{\lambda = r_{|u}\}\sigma) = \theta, \\ u \in \overline{O}^\varphi(r), \ e' = (l = r[\rho]_u)\sigma\theta \ \} \end{array}$$

si y solo si existe i, $1 \leq i \leq n$ (definición de $\cup^\infty_{n=1}\mathcal{I}_n$)

$$e \in \Phi_\mathcal{R} \cup \Im^\varphi_\mathcal{R} \cup \{e' \in \mathcal{B}_V \mid \begin{array}{l} (\lambda = \rho \Leftarrow C) \ll \mathcal{R}^\varphi, \ \{l = r\} \cup C' \subseteq \mathcal{I}_i, \\ mgu(flat_\varphi(C), C') = \sigma, \ mgu(\{\lambda = r_{|u}\}\sigma) = \theta, \\ u \in \overline{O}^\varphi(r), \ e' = (l = r[\rho]_u)\sigma\theta \ \} \end{array}$$

si y solo si para algún i, $1 \leq i \leq n$, tenemos que $e \in T^\varphi_\mathcal{R}(\mathcal{I}_i)$

si y solo si $e \in \cup^\infty_{n=1} T^\varphi_\mathcal{R}(\mathcal{I}_n)$

□

Debido a la continuidad del operador de consecuencias inmediatas entonces, por el teorema de Kleene, el menor punto fijo de $T^\varphi_\mathcal{R}$ existe $lfp(T^\varphi_\mathcal{R}) = T^\varphi_\mathcal{R} \uparrow \omega$.

Definición 3.2.3 *La semántica del menor punto fijo de un programa $\mathcal{R}$ en $\mathbb{R}_\varphi$ está definida como $\mathcal{F}_\varphi(\mathcal{R}) = lfp(T_\mathcal{R}^\varphi)$, $\varphi \in \{inn, out\}$.*

La siguiente semántica es equivalente a la semántica de respuestas computadas, tal como probaremos en lo que sigue.

Definición 3.2.4 *La expresión $\mathcal{F}_\varphi^{ca}(\mathcal{R})$ denota el conjunto $\{l = r \in lfp(T_\mathcal{R}^\varphi) \mid r$ no contiene ningún símbolo de función definidos $f/n \in \mathcal{D}\}$, $\varphi \in \{inn, out\}$.*

Ejemplo 13 *Consideremos el siguiente programa $\mathcal{R}$ con respecto a la estrategia de narrowing $\varphi = inn$.*

$$\begin{array}{lcl} \{\mathtt{g(x)} & \to & 0. \\ \mathtt{f(0)} & \to & 0. \\ \mathtt{f(s(x))} & \to & \mathtt{f(x).}\} \end{array}$$

La semántica por punto fijo se obtiene usando el operador $T_\mathcal{R}^\varphi$, partiendo de los axiomas reflexivos $\Phi_\mathcal{R} \cup \Im_\mathcal{R}^\varphi = \{\mathtt{0 = 0},\ \mathtt{s(x) = s(x)},\ \mathtt{g(x) = g(x)},\ \mathtt{f(x) = f(x)}\}$, desarrollamos las potencias del operador de consecuencias inmediatas y empleamos $\mathcal{V}$-interpretaciones del siguiente modo:

$$T_\mathcal{R}^{inn} \uparrow 0 = \Phi_\mathcal{R} \cup \Im_\mathcal{R}^\varphi \cup \{\mathtt{0 = 0},\ \mathtt{s(x) = s(x)},\ \mathtt{g(x) = g(x)},\ \mathtt{f(x) = f(x)}\}$$

$$T_\mathcal{R}^{inn} \uparrow 1 = T_\mathcal{R}^{inn} \uparrow 0 \cup \{\mathtt{g(x) = 0},\ \mathtt{f(0) = 0},\ \mathtt{f(s(x)) = f(x)}\}$$

$$T_\mathcal{R}^{inn} \uparrow 2 = T_\mathcal{R}^{inn} \uparrow 1 \cup \{\mathtt{f(s(0)) = 0},\ \mathtt{f(s^2(x)) = f(x)}\}$$

$$\vdots$$

$$\begin{array}{rcl} T_\mathcal{R}^{inn} \uparrow \omega = lfp(T_\mathcal{R}^{inn}) & = & \{\mathtt{0 = 0},\ \mathtt{s(x) = s(x)},\ \mathtt{g(x) = g(x)}, \\ & & \mathtt{f(x) = f(x)},\ \mathtt{g(x) = 0},\ \mathtt{f(0) = 0}, \\ & & \mathtt{f(s(0)) = 0}, \ldots, \mathtt{f(s^n(0)) = 0}, \ldots, \\ & & \mathtt{f(s(x)) = f(x)}, \ldots,\ \ \mathtt{f(s^n(x)) = f(x)}, \ldots\} \end{array}$$

Por la Definición 3.2.3 la semántica de punto fijo es:

$$\begin{array}{rcl} \mathcal{F}_{inn}(\mathcal{R}) = lfp(T_\mathcal{R}^{inn}) & = & \{\mathtt{0 = 0},\ \mathtt{s(x) = s(x)},\ \mathtt{g(x) = g(x)}, \\ & & \mathtt{f(x) = f(x)},\ \mathtt{g(x) = 0},\ \mathtt{f(0) = 0}, \\ & & \mathtt{f(s(0)) = 0}, \ldots, \mathtt{f(s^n(0)) = 0}, \ldots, \\ & & \mathtt{f(s(x)) = f(x)}, \ldots,\ \ \mathtt{f(s^n(x)) = f(x)}, \ldots\} \end{array}$$

Mientras que la respectiva semántica de respuestas computadas es (Definición 3.2.4):

$$\begin{array}{rcl} \mathcal{F}_{inn}^{ca}(\mathcal{E}) & = & \{\mathtt{0 = 0},\ \mathtt{s(x) = s(x)},\ \mathtt{g(x) = 0},\ \mathtt{f(0) = 0}, \\ & & \mathtt{f(s(0)) = 0}, \ldots, \mathtt{f(s^n(0)) = 0}, \ldots\} \end{array}$$

Ejemplo 14 *Consideremos el ahora el conocido programa* `from/1` *y como estrategia de narrowing* $\varphi = out$.

$$\begin{array}{lcl} \{\mathtt{from(x)} & \rightarrow & \mathtt{[x|from(s(x))]}. \\ \mathtt{first([x|y])} & \rightarrow & \mathtt{x}.\} \end{array}$$

Las potencias de $T_{\mathcal{R}}^{out}$ *son:*

$$\begin{array}{lcll} T_{\mathcal{R}}^{out} \uparrow 0 & = & \Phi_{\mathcal{R}} \cup \Im_{\mathcal{R}}^{\varphi} & = \{\mathtt{[x|y]} \approx \mathtt{[x|y]},\ \mathtt{s(x)} \approx \mathtt{s(x)},\ \mathtt{from(x)} = \mathtt{from(x)}, \\ & & & \quad \mathtt{first(x)} = \mathtt{first(x)}\} \\ T_{\mathcal{R}}^{out} \uparrow 1 & = & T_{\mathcal{R}}^{out} \uparrow 0 \quad \cup & \{\mathtt{first([x|y])} = \mathtt{x},\ \mathtt{first(x)} = \bot,\ \mathtt{from(x)} = \bot, \\ & & & \quad \mathtt{from(x)} = \mathtt{[x|from(s(x))]}\} \\ & \vdots & & \\ T_{\mathcal{R}}^{out} \uparrow \omega & = & lfp(T_{\mathcal{R}}^{out}) & = \{\mathtt{[x|y]} \approx \mathtt{[x|y]},\ \mathtt{s(x)} \approx \mathtt{s(x)},\ \mathtt{from(x)} = \mathtt{from(x)}, \\ & & & \quad \mathtt{first(x)} = \mathtt{first(x)},\ \mathtt{first(x)} = \bot, \\ & & & \quad \mathtt{first([x|y])} = \mathtt{x},\ \mathtt{from(x)} = \bot, \\ & & & \quad \mathtt{from(x)} = \mathtt{[x|}\bot\mathtt{]}, \ldots, \\ & & & \quad \mathtt{from(x)} = \mathtt{[x|[s(x)|} \ldots \mathtt{[s^n(x)|}\bot\mathtt{]]]}, \ldots, \\ & & & \quad \mathtt{from(x)} = \mathtt{[x|from(s(x))]}, \ldots, \\ & & & \quad \mathtt{from(x)} = \mathtt{[x|[s(x)|} \ldots \mathtt{[from(s^n(x))]]]}, \ldots\} \end{array}$$

Por la Definición 3.2.3 la semántica de punto fijo es:

$$\begin{array}{lcll} \mathcal{F}_{out}(\mathcal{R}) & = & lfp(T_{\mathcal{R}}^{out}) & = \{\mathtt{[x|y]} \approx \mathtt{[x|y]},\ \mathtt{s(x)} \approx \mathtt{s(x)},\ \mathtt{from(x)} = \mathtt{from(x)}, \\ & & & \quad \mathtt{first(x)} = \mathtt{first(x)},\ \mathtt{first(x)} = \bot, \\ & & & \quad \mathtt{first([x|y])} = \mathtt{x},\ \mathtt{from(x)} = \bot, \\ & & & \quad \mathtt{from(x)} = \mathtt{[x|}\bot\mathtt{]}, \ldots, \\ & & & \quad \mathtt{from(x)} = \mathtt{[x|[s(x)|} \ldots \mathtt{[s^n(x)|}\bot\mathtt{]]]}, \ldots, \\ & & & \quad \mathtt{from(x)} = \mathtt{[x|from(s(x))]}, \ldots, \\ & & & \quad \mathtt{from(x)} = \mathtt{[x|[s(x)|} \ldots \mathtt{[from(s^n(x))]]]}, \ldots\} \end{array}$$

Mientras que la respectiva semántica de respuestas computadas es (Definición 3.2.4):

$$\begin{array}{lcl} \mathcal{F}_{out}^{ca}(\mathcal{E}) & = & \{\mathtt{[x|y]} \approx \mathtt{[x|y]},\ \mathtt{s(x)} \approx \mathtt{s(x)},\ \mathtt{first(x)} = \bot, \\ & & \quad \mathtt{first([x|y])} = \mathtt{x},\ \mathtt{from(x)} = \bot, \\ & & \quad \mathtt{from(x)} = \mathtt{[x|}\bot\mathtt{]}, \ldots, \mathtt{from(x)} = \mathtt{[x|[s(x)|} \ldots \mathtt{[s^n(x)|}\bot\mathtt{]]]}, \ldots, \} \end{array}$$

El siguiente lema establece que toda ecuación computada por $T_{\mathcal{R}}^{\varphi}$ también puede ser computada por *narrowing* con respecto a la estrategia φ. En lo que sigue , la notación $g \leadsto_{\varphi}^{n} g'$se utiliza para representar una derivación de *narrowing* de g a g' de longitud n con respecto a φ.

Lema 3.2.5 *Sea $\mathcal{R}$ un programa en $I\!R_\varphi$, $\varphi \in \{inn, out\}$. Entonces*

$$(f(\bar{x})\theta = t) \in T_{\mathcal{R}}^{\varphi} \uparrow k \quad sii \quad f(\bar{x}) = y \overset{\mathcal{R},\theta}{\leadsto}{}^{*}{}_{\varphi} \; t = y$$

en donde $\perp$ no ocurre en t.

Demostración. ($\Longrightarrow$) : Para cada ecuación $(f(\bar{x})\theta = t) \in T_{\mathcal{R}}^{\varphi} \uparrow k$, en donde $\perp$ no ocurre en t, demostremos que existe una derivación de narrowing $f(\bar{x}) = y \overset{\mathcal{R},\theta}{\leadsto}{}^{*}{}_{\varphi} \; t = y$. Razonemos por inducción sobre el número k de iteraciones.

Si $k = 0$ y $(f(\bar{x}) = f(\bar{x})) \in T_{\mathcal{R}}^{\varphi} \uparrow 0$ entonces por el cierre reflexivo de la relación de narrowing tenemos $f(\bar{x}) = y \overset{\mathcal{R},\epsilon}{\leadsto}{}^{*}{}_{\varphi,0} \; f(\bar{x}) = y$

Consideremos el caso inductivo, $k > 0$. Si $(f(\bar{x})\theta = t) \in T_{\mathcal{R}}^{\varphi} \uparrow k$ existe un i, $0 \leq i \leq k$ tal que $(f(\bar{x})\theta = t) \notin T_{\mathcal{R}}^{\varphi} \uparrow (i-1)$ y $(f(\bar{x})\theta = t) \in T_{\mathcal{R}}^{\varphi} \uparrow i$. Por la hipótesis de inducción, existe una derivación de narrowing con estrategia φ tal que $f(\bar{x}) = y \overset{\mathcal{R},\theta}{\leadsto}{}^{*}{}_{\varphi} \; t = y$.

Consideremos las ecuaciones de $T_{\mathcal{R}}^{\varphi} \uparrow k$ que generan nuevas ecuaciones en el $T_{\mathcal{R}}^{\varphi} \uparrow (k+1)$.

Sea $(f(\bar{x})\theta = t_k) \in T_{\mathcal{R}}^{\varphi} \uparrow k$ una ecuación a la cual aplicamos un paso del operador y genera una ecuación tal que $(f(\bar{x})\theta' = t_{(k+1)}) \in T_{\mathcal{R}}^{\varphi} \uparrow (k+1)$ y $(f(\bar{x})\theta' = t_{(k+1)}) \notin T_{\mathcal{R}}^{\varphi} \uparrow k$. Entonces por la definición del operador $T_{\mathcal{R}}^{\varphi} \uparrow (k+1)$, existe un conjunto de ecuaciones $\{f(\bar{x})\theta = t_k\} \cup C' \subseteq T_{\mathcal{R}}^{\varphi} \uparrow k$ y una regla $(\lambda \to \rho \Leftarrow C) \ll \mathcal{R}_\varphi$ tal que existen $\sigma = mgu(flat_\varphi(C), C')$, $\beta = mgu(\{\lambda = t_k|_u\}\sigma)$ y $u \in \overline{O}^{\varphi}(t_k)$ que cumplen que $(f(\bar{x})\theta' = t_{k+1}) \equiv ((f(\bar{x})\theta = t_k[\rho]_u)\sigma\beta)$ donde $\theta' = \theta\sigma\beta$.

Como la regla $(\lambda \to \rho \Leftarrow C)$ es un renombramiento "fresco" o variante de una de las reglas de $\mathcal{R}$ (esto es, todas sus variables son completamente nuevas) y σ depende de las variables de $flat_\varphi(C)$ y C', se cumple que $\mathcal{V}ar(\sigma) \cap \mathcal{V}ar(t_k) = \{\}$ de lo cual se deduce que $\beta = mgu(\{\lambda = t_k|_u\}\sigma) = mgu(\lambda\sigma = t_k|_u)$, $\overline{O}^{\varphi}(t_k) = \overline{O}^{\varphi}(t_k\sigma)$ y $t_{k+1} = (t_k[\rho]_u)\sigma\beta = (t_k[\rho\sigma]_u)\beta$.

Como $(f(\bar{x})\theta = t_k) \in T_{\mathcal{R}}^{\varphi} \uparrow k$, por la hipótesis de inducción podemos asumir que existe una derivación de narrowing con la estrategia φ tal que $f(\bar{x}) = y \overset{\mathcal{R},\theta}{\leadsto}{}^{*}{}_{\varphi} \; t_k = y$. Consideremos un paso de narrowing posterior. Dada la ecuación $t_k = y$ existen $u \in \overline{O}^{\varphi}(t_k) = \overline{O}^{\varphi}(t_k\sigma)$, $mgu(\lambda\sigma = t_k|_u\sigma) = \beta$ y una regla $(\lambda\sigma \to \rho\sigma \Leftarrow C\sigma) \ll \mathcal{R}^{\varphi}$ tal que es posible aplicar un paso de narrowing con la estrategia φ y resulta el objetivo $C\sigma\beta, (t_{k+1} = y))$, con $t_{k+1} = (t_k[\rho]_u)\sigma\beta$. Como $C' \subseteq T_{\mathcal{R}}^{\varphi} \uparrow k$ por la hipótesis inductiva para cada ecuación $f_s(\bar{d}) = d_s$ o $d_m =_\varphi d'_m$ de $C'\sigma$ existe una derivación tal que $f_s(\bar{x}) = y \overset{\delta}{\leadsto}{}^{*}{}_{\varphi} \; d_s = y \overset{\{y/d_s\}}{\leadsto}{}_{\varphi} \; \top$; en el caso de ecuaciones entre términos constructores $d_m =_\varphi d'_m \overset{\epsilon}{\leadsto}{}_{\varphi} \; \top$. Si $\varphi = inn$ se aplica la regla $x = x \to true$ mientras que para $\varphi = out$ aplicamos las reglas para $\approx$ que fueron adicionadas con anterioridad al programa. Luego, para cada ecuación $e \in C'\sigma$ se cumple que $e \overset{\epsilon}{\leadsto}{}^{*}{}_{\varphi} \; \top$, esto es, $C'\sigma \overset{\epsilon}{\leadsto}{}^{*}{}_{\varphi} \top$, por consiguiente $flat_\varphi(C)\sigma \overset{\epsilon}{\leadsto}{}^{*}{}_{\varphi} \top$ y como el aplanamiento $flat_\varphi(C)$ preserva las respuestas computadas por la estrategia de narrowing φ, Lema 3.1.7,

entonces $C\sigma \overset{\epsilon}{\leadsto}^*{}_\varphi \top$. Ya que la estrategia de narrowing φ es correcta y completa con respecto a las respuestas computadas y la sustitución β está en forma normal entonces $C\sigma\beta \overset{\epsilon}{\leadsto}^*{}_\varphi \top$. En definitiva, $t_k = y \overset{\sigma\beta}{\leadsto}^*{}_\varphi\ t_{k+1} = y$, por tanto $f(\bar{x}) = y \overset{\theta\sigma\beta}{\leadsto}^*{}_\varphi\ t_k = y$.

En conclusión, existe una derivación de la estrategia de narrowing φ tal que $f(\bar{x}) = y \overset{\theta'}{\leadsto}^*{}_\varphi\ \ t = y$ donde $\theta' = \theta\sigma\beta$.

($\Longleftarrow$): En la dirección opuesta, vamos a probar un resultado más general, para toda derivación $\mathcal{D} : f(\bar{x}) = y \overset{\theta}{\leadsto}^*{}_\varphi C,\ t = y$ tal que $C \overset{\nu}{\leadsto}^*{}_\varphi \top$ existe un $k \geq 0$ para la cual se cumple que $(f(\bar{x})\theta = t)\nu \in T^\varphi_\mathcal{R} \uparrow k$.

La demostración es por inducción sobre la longitud n de la derivación $\mathcal{D}$

Sea $n = 0$, entonces por el cierre reflexivo de la relación de *narrowing* se cumple que $f(\bar{x}) = y \overset{\epsilon}{\leadsto}^*{}_\varphi\ \ f(\bar{x}) = y$ y por la Definición 3.2.1 $(f(\bar{x}) = f(\bar{x})) \in T^\varphi_\mathcal{R} \uparrow 0$.

Consideremos el paso inductivo, sea $n > 0$. Asumamos que para toda derivación de narrowing $f'(\bar{x}) = y \overset{\theta'}{\leadsto}{}^m_\varphi \quad C', t' = y$ con $m < n$ tal que $C' \overset{\nu'}{\leadsto}^*{}_\varphi \top$, existe $j \geq 0$ para el que se cumple que $(f'(\bar{x})\theta' = t')\nu' \in T^\varphi_\mathcal{R} \uparrow j$.

Para $\varphi \in \{inn, out\}$, es inmediato ver que $\mathcal{D}$ también se puede escribir de la forma:

$$f(\bar{x}) = y \overset{\theta'}{\leadsto}{}^{n-1}_\varphi \quad t' = y \overset{\eta}{\leadsto}_\varphi \quad t = y,$$

donde la respectiva condicion de una regla que ha sido introducida eventalmente, es resuelta antes del paso final de narrowing $t' = y \overset{\eta}{\leadsto}_\varphi \quad t = y$, o

$$f(\bar{x}) = y \overset{\theta'}{\leadsto}^*{}_\varphi\ \ C', t' = y \overset{\alpha}{\leadsto}_\varphi \quad C, t = y \overset{\beta}{\leadsto}^*{}_\varphi\ \ (t\beta = y),$$

donde la subderivación final $C, t = y \overset{\beta}{\leadsto}^*{}_\varphi\ \ (t\beta = y)$ reduce a $\top$ la condición C sin aplicar más pasos de narrowing al término t, y el paso $C', t' = y \overset{\alpha}{\leadsto}_\varphi \quad C, t = y$ aplica narrowing a t' y resulta el término t mediante la aplicación de la regla $(\lambda \to \rho \Leftarrow B) \ll \mathcal{R}_\varphi$ en la posición seleccionada $u \in \overline{O}^\varphi(t')$ tal que $\alpha = mgu(\{\lambda = t'|_u\})$ y $t = (t'[\rho]_u)\alpha$, with $C = (C', B)\alpha$.

En el primer caso, por la hipótesis inductiva existe $j \geq 0$ tal que $f(\bar{x})\theta' = t' \in T^\varphi_\mathcal{R} \uparrow j$, y por lo tanto por las definiciones de φ y $T^\varphi_\mathcal{R}$ se cumple que $f(\bar{x})\theta'\eta = t \in T^\varphi_\mathcal{R} \uparrow j+1$, y la conclusión es inmediata.

En el segundo caso, ya que $C, t = y \overset{\beta}{\leadsto}^*{}_\varphi\ \ (t\beta = y)$, tenemos que $C \overset{\beta}{\leadsto}^*{}_\varphi\ \ \top$ y, por Lema 3.1.7 $flat_\varphi(C) \overset{\beta'}{\leadsto}^*{}_\varphi\ \ \top$, con $\beta = \beta'_{\restriction Var(C)}$, luego $flat_\varphi(C)\beta' \overset{\epsilon}{\leadsto}^*{}_\varphi\ \ \top$. La ecuaciones de $flat_\varphi(C)$ son planas y tienen la siguiente forma: $(f'(x_1, \ldots, x_n) = x_{n+1})\delta$ o $d_l =_\varphi d_r$, donde $\delta = \{x_1/d_1, \ldots, x_{n+1}/d_{n+1}\}$ y d_l, d_r, d_i, $i = 1 \ldots n+1$, son términos constructores.

Por la definición de $T^\varphi_\mathcal{R}$, para cada ecuación $(d_l =_\varphi d_r)\beta'$ en $flat_\varphi(C)\beta'$ existe $m \geq 0$ tal que $flat_\varphi(C)\beta' \in T^\varphi_\mathcal{R} \uparrow m$.

Construyamos ahora el conjunto $C'' = flat_\varphi(C)\beta'$. Note que, ya que β' es una sustitución constructora entonces C'' es un conjunto plano de ecuaciones. Por la completitud de φ, para cada ecuación $(f'(\bar{x}) = y)\delta\beta'$ de C'' existe una derivación

$f'(\bar{x}) = y \overset{\delta\beta'}{\leadsto}{}^{*}{}_{\varphi}\ d_{n+1}\beta' = y$. Por la hipótesis inductiva, para cada ecuación de C'', para algún $j' \geq 0$ se cumple que $(f'(\bar{x})\delta = d_{n+1})\beta' \in T_{\mathcal{R}}^{\varphi} \uparrow j'$.

Finalmente, consideremos el prefijo $f(\bar{x}) = y \overset{\theta'}{\leadsto}{}^{*}{}_{\varphi}\ C', t' = y$ de la derivación $\mathcal{D}$, por la hipótesis inductiva, existe $q \geq 0$ tal que $\{f(\bar{x})\theta'\alpha\beta = t'\alpha\beta\} \subseteq T_{\mathcal{R}}^{\varphi} \uparrow q$.

Por consiguiente, hemos probado que existe $p \geq 0$ (a saber, el máximo de j', q y m) tal que $\{f(\bar{x})\theta'\alpha\beta = t'\alpha\beta\} \cup C'' \subseteq T_{\mathcal{R}}^{\varphi} \uparrow p$, y $\beta = mgu(flat_{\varphi}(C), C'')_{\restriction Var(C)}$. Ahora, ya que existe $u \in \overline{O}^{\varphi}(t')$ y una regla $(\lambda \to \rho \Leftarrow B) \ll \mathcal{R}_{\varphi}$ con $mgu(flat_{\varphi}(B), C'') = \beta_{\restriction(Var(B)}$, $\alpha = mgu(\{\lambda = t'|_u\})$ y $t = (t'[\rho]_u)\alpha$, concluimos que $(f(\bar{x})\theta = t\beta) \in T_{\mathcal{R}}^{\varphi} \uparrow k$, con $k = p + 1$ y $\theta = \theta'\alpha\beta$. □

Recordemos que un objetivo plano de la forma $x = y$, con $x, y \in V$ se denomina un *objetivo trivial*. Los objetivos de la forma $x \approx y$ no son triviales.

El siguiente resultado relaciona las sustituciones de respuesta computada por narrowing con respecto a φ con las sustituciones "computadas" donde no aparece el símbolo $\perp$ en la *semántica (de punto fijo) de respuestas computadas* por unificación estándar.

Teorema 3.2.6 (corrección y completitud fuertes) *Sea $\mathcal{R}$ un programa en $\mathbb{R}_{\varphi}$ y g un objetivo (no–trivial) de acuerdo con φ. Entonces θ es una sustitución de respuesta computada para g en $\mathcal{R}$ con respecto a $\leadsto_{\varphi}$ si y solo si existe $g' \equiv e_1, \ldots, e_n << \mathcal{F}_{\varphi}^{ca}(\mathcal{R})$ tal que $\theta = mgu(flat_{\varphi}(g), g')_{\restriction Var(g)}$.*

Demostración. ($\Rightarrow$): Sea g un objetivo tal que $flat_{\varphi}(g) = (e_1, \ldots, e_n)$ y $g \overset{\theta}{\leadsto}{}^{*}{}_{\varphi}\ \top$. Por medio del Lema 3.1.7, se sigue que $(e_1, \ldots, e_n) \overset{\theta'}{\leadsto}{}^{*}{}_{\varphi}\ \top$ y $\theta_{\restriction g} = \theta'_{\restriction g}$. Por Lema 2.6.5, se cumple que $e_1 \overset{\sigma_1}{\leadsto}{}^{*}{}_{\varphi}\ \top, \ldots, e_n \overset{\sigma_n}{\leadsto}{}^{*}{}_{\varphi}\ \top$ con $\theta' = \sigma_1 \Uparrow \ldots \Uparrow \sigma_n$.

Ya que las ecuaciones e_i, $i = 1, \ldots, n$ son planas, entonces tienen la forma $f_s(d_1^i, \ldots, d_n^i) = d_s^i$ o $d_l^i =_{\varphi} d_r^i$. Consideremos los dos casos de forma separada:

Dada la ecuación $f_s(d_1^i, \ldots, d_n^i) = d_s^i$, podemos escribirla como $f_s(\bar{x})\beta_i = d_s^i$ donde $\beta_i = \{x_1/d_1^i, \ldots, x_n/d_n^i\}$. Ahora bien, para cada i se cumple que, como $f_s(\bar{x})\beta_i = d_s^i \overset{\sigma_i}{\leadsto}{}^{*}{}_{\varphi}\ \top$, entonces $f_s(\bar{x})\beta_i\sigma_i = d_s^i\sigma_i \overset{\epsilon}{\leadsto}{}^{*}{}_{\varphi}\ \top$, esto implica que $f_s(\bar{x}) = y \overset{\beta_i\sigma_i}{\leadsto}{}^{*}{}_{\varphi}{}^{*} d_s^i\sigma_i = y$ donde $d_s^i\sigma_i$ es un término constructor que no incluye $\perp$. Por el Lema 3.2.5 y la definición de $\mathcal{F}_{\varphi}^{ca}(\mathcal{R})$, tenemos $f_s(\bar{x})\beta_i\sigma_i = d_s^i\sigma_i \in \mathcal{F}_{\varphi}^{ca}(\mathcal{R})$. Como resultado podemos considerar $e'_i = e_i\sigma_i$; entonces

$$\begin{aligned} mgu((e_1, \ldots, e_n), (e'_1, \ldots, e'_n)) &= mgu((e_1, \ldots, e_n), (e_1\sigma_1, \ldots, e_n\sigma_n)) \\ &= mgu(\widehat{\sigma_1}, \ldots, \widehat{\sigma_n}) \\ &= \theta' \end{aligned}$$

Sea $d_l^i =_{\varphi} d_r^i$. En el caso de que $\varphi = inn$ entonces $d_l^i = d_r^i \overset{\sigma_i}{\leadsto}{}_{\varphi}\ \top$ por medio de la aplicación de la regla $x = x \to true$. De la Definición 3.2.1, todas las ecuaciones $c(\bar{x}) = c(\bar{x})$, con c/n un símbolo constructor, están en $T_{\mathcal{R}} \uparrow k$ para todo k. Por

consiguiente, existe una sustitución α_i tal que $(d_l^i = d_r^i)\sigma_i = (c(\bar{x}) = c(\bar{x}))\alpha_i$. Esto implica que existe una sustitución θ_i' tal que $\theta_i' = mgu((d_l^i = d_r^i), (c(\bar{x}) = c(\bar{x})))$. Note que, $\theta_{i\restriction \mathcal{V}ar(d_l^i = d_r^i)}' = \sigma_i$. Similarmente, si $\varphi = out$ entonces $d_l^i \approx d_r^i \overset{\sigma_i}{\leadsto}{}^*{}_\varphi \top$ aplicando las reglas que definen la igualdad estricta $\approx$ y que son adicionadas al programa previamente.

($\Leftarrow$): En dirección contraria. Supongamos que existen ecuaciones $e_1', \ldots, e_n'$ tal que $(e_1', \ldots, e_n') \ll \mathcal{F}_\varphi^{ca}(\mathcal{R})$ y cumplen que $\theta' = mgu((e_1, \ldots, e_n), (e_1', \ldots, e_n'))$. Por consiguiente, para todo i, $i = 1, \ldots, n$, existe σ_i tal que $e_i\sigma_i = e_i'\sigma_i$. Como $e_i' \in \mathcal{F}_\varphi^{ca}(\mathcal{R})$, entonces tiene la forma $f_s(\bar{x})\beta_i = d_s^i$ o $d_l^i =_\varphi d_r^i$. Debido a que consideramos únicamente que $\varphi \in \{inn, out\}$ entonces θ' es una sustitución constructora, por lo tanto así es σ_i, para cada i. Por el Lema 3.2.5, existe una derivación de narrowing $f_s(\bar{x}) = y \overset{\beta_i}{\leadsto}{}^*{}_\varphi d_s^i = y$, por lo tanto $f_s(\bar{x})\beta_i = d_s^i \overset{\epsilon}{\leadsto}{}^*{}_\varphi \top$, entonces $f_s(\bar{x})\beta_i\sigma_i = d_s^i\sigma_i \overset{\epsilon}{\leadsto}{}^*{}_\varphi \top$. Similarmente $d_l^i\sigma_i =_\varphi d_r^i\sigma_i \overset{\epsilon}{\leadsto}{}^*{}_\varphi \top$. Ya que $e_i'\sigma_i \overset{\epsilon}{\leadsto}{}^*{}_\varphi \top$, entonces $e_i\sigma_i \overset{\epsilon}{\leadsto}{}^*{}_\varphi \top$ por lo tanto $e_i \overset{\sigma_i}{\leadsto}{}^*{}_\varphi \top$. En general, $e_1 \overset{\sigma_1}{\leadsto}{}^*{}_\varphi \top, \ldots, e_n \overset{\sigma_n}{\leadsto}{}^*{}_\varphi \top$ y por el Lema 2.6.5 $(e_1, \ldots, e_n) \overset{\theta'}{\leadsto}{}^*{}_\varphi \top$ donde $\theta' = \sigma_1 \Uparrow \ldots \Uparrow \sigma_n$. En conclusión, $g \overset{\theta}{\leadsto}{}^*{}_\varphi \top$. □

Ejemplo 15 *Consideremos de nuevo el programa del Ejemplo 13.*

$$\begin{array}{lcl} \{\mathtt{g(x)} & \rightarrow & 0. \\ \mathtt{f(0)} & \rightarrow & 0. \\ \mathtt{f(s(x))} & \rightarrow & \mathtt{f(x).}\} \end{array}$$

La semántica de punto fijo de respuestas computadas es (Definición 3.2.4):

$$\begin{array}{rcl} \mathcal{F}_{inn}^{ca}(\mathcal{E}) & = & \{\mathtt{0 = 0},\ \mathtt{s(x) = s(x)},\ \mathtt{g(x) = 0},\ \mathtt{f(0) = 0}, \\ & & \mathtt{f(s(0)) = 0}, \ldots, \mathtt{f(s^n(0)) = 0}, \ldots\} \end{array}$$

Dado el objetivo $g \equiv (\mathtt{y = f(z)})$*, narrowing innermost computa las respuestas*

$$\{\ \{y/\mathtt{0}, \mathtt{z/0}\},\ \{\mathtt{y/0, z/s(0)}\}, \ldots,\ \{\mathtt{y/0, z/s^n(0)}\}\}$$

con el programa $\mathcal{R}$*, Las cuales coinciden exactamente con el conjunto de sustituciones computadas por unificación del objetivo plano* $\mathtt{f(z) = y}$ *con el conjunto de ecuaciones de* $\mathcal{F}_{inn}^{ca}(\mathcal{R})$*.*

Ejemplo 16 *Consideremos de nuevo el programa* `from/1` *del Ejemplo 14*

$$\begin{array}{lcl} \{\mathtt{from(x)} & \rightarrow & \mathtt{[x|from(s(x))].} \\ \mathtt{first([x|y])} & \rightarrow & \mathtt{x.}\} \end{array}$$

El semántica de respuestas computadas[3] *es:*

[3]Por simplicidad, omitimos la denotación de los símbolos definidos de función "incorporados" al programa "$\approx$" y "$\wedge$", por ejemplo las ecuaciones $(\mathtt{[s^n(x1)|y1]} \approx \mathtt{[s^n(x2)|y2]}) = (\mathtt{x1} \approx \mathtt{x2}) \wedge (\mathtt{y1} \approx \mathtt{y2})$, con $\mathtt{n} > \mathtt{0}$.

$$\begin{aligned}\mathcal{F}^{ca}_{out}(\mathcal{R}) \quad &= \{[\mathtt{x}|\mathtt{y}] \approx [\mathtt{x}|\mathtt{y}], \mathtt{s}(\mathtt{x}) \approx \mathtt{s}(\mathtt{x}),\ \mathtt{from}(\mathtt{x}) = \bot,\ \mathtt{from}(\mathtt{x}) = [\mathtt{x}|\bot], \ldots,\\ &\quad\ \mathtt{from}(\mathtt{x}) = [\mathtt{x}|[\mathtt{s}(\mathtt{x})|\ldots[\mathtt{s}^{\mathtt{n}}(\mathtt{x})|\bot]]], \ldots,\ \mathtt{first}(\mathtt{x}) = \bot,\ \mathtt{first}([\mathtt{x}|\mathtt{y}]) = \mathtt{x}\}\end{aligned}$$

con $n \in \omega$.

Dado el objetivo $g \equiv\ (\mathtt{first}(\mathtt{from}(\mathtt{s}(\mathtt{x}))) \approx \mathtt{z})$*, narrowing outermost sólo computa la respuesta* $\{\mathtt{z}/\mathtt{s}(\mathtt{x})\}$ *en* $\mathcal{R}$*, la cual es también la única sustitución que puede ser computada por unificación del objetivo plano (*$\mathtt{from}(\mathtt{s}(\mathtt{x})) = \mathtt{y},\ \mathtt{first}(\mathtt{y}) = \mathtt{w},\ \mathtt{w} \approx \mathtt{z}$*) con* $\mathcal{F}^{ca}_{out}(\mathcal{R})$.

De acuerdo con el Teorema 3.2.6, $\mathcal{F}^{ca}_{\varphi}(\mathcal{R})$ puede ser utilizada para simular la ejecución de cualquier objetivo (no–trivial) g, esto es, $\mathcal{F}^{ca}_{\varphi}(\mathcal{R})$ puede ser visto como un conjunto (posiblemente infinito) de cláusulas "unitarias", y las sustituciones de respuesta computadas para g en $\mathcal{R}$ pueden ser determinadas por la 'ejecución' de $flat(g)$ en el programa $\mathcal{F}^{ca}_{\varphi}(\mathcal{R})$ por unificación estándar, como si el símbolo de igualdad fuera un predicado ordinario. En lo que sigue, mostraremos la relación las semánticas $\mathcal{F}^{ca}_{\varphi}(\mathcal{R})$ y la nueva semántica operacional de "respuestas computadas" $\mathcal{O}^{ca}_{\varphi}(\mathcal{R})$ que modela el comportamiento de ecuaciones simples y presentamos a continuación.

3.3 Semántica del conjunto de éxitos

Si fijamos nuestra atención sobre el observable de sustituciones de respuestas computadas, podemos ver, al igual que sucedía en el caso de la programación lógica, que la semántica operacional estándar (que obtiene el conjunto de 'valores' básicos calculados por derivaciones de éxito) no es bastante rica como para caracterizar un programa respecto a este observable. Esto queda ilustrado en los siguientes ejemplos [Julián y Moreno, 1996].

Ejemplo 17 Consideremos los programas $\mathcal{R}_1$ y $\mathcal{R}_1$:

$$\begin{aligned}\mathcal{R}_1 &= \{f(0) \to 0\}\\ \mathcal{R}_2 &= \{f(0) \to 0,\quad f(x) \to 0\}\end{aligned}$$

Estos programas tienen la misma semántica operacional estándar, el mismo conjunto ecuaciones de éxitos básicas:

$$\{0 = 0,\ 0 = f(0), \ldots,\ 0 = f^m(0), \ldots,\ f^n(0) = 0,\ f^n(0) = f(0), \ldots,\ f^n(0) = f^m(0)\}$$

así pues, estos programas serán indistinguibles según esta caracterización.

Sin embargo los dos programas tienen diferente comportamiento en términos de las sustituciones de respuestas computadas, como puede verse si consideramos el objetivo $f(x) = 0$, que computa las respuestas $\{x/0\}$ y $\{\}$ cuando se ejecuta en $\mathcal{R}_2$, mientras solamente calcula la respuesta $\{x/0\}$ cuando se ejecuta en $\mathcal{R}_1$. Tenemos que $\mathcal{R}_1$ y

$\mathcal{R}_2$ no son equivalentes respecto al observable de respuestas computadas, por lo que deberíamos dotar al lenguaje de una semántica O para la cual $O(\mathcal{R}_1) \neq O(\mathcal{R}_2)$.[4]

Si no queremos comportamientos "anómalos" de los programas respecto al observable de las respuestas computadas será necesario dar una nueva semántica operacional capaz de reflejar diferencias como las constatadas en el ejemplo anterior. La semántica operacional del *conjunto de éxitos* $\mathcal{O}^{ca}_{\varphi}(\mathcal{R})$ de un programa $\mathcal{R}$ con respecto a la semántica de *narrowing* φ se define considerando las respuestas computadas para llamadas más generales"[Alpuente *et al.*, 2001a].

Definición 3.3.1 *Sea $\mathcal{R}$ un programa en $\mathbb{R}_{\varphi}$. Entonces, $\mathcal{O}^{ca}_{\varphi}(\mathcal{R}) = \Im^{\varphi}_{\mathcal{R}} \cup \{(f(x_1,\ldots,x_n) = x_{n+1})\theta \mid (f(x_1,\ldots,x_n) =_{\varphi} x_{n+1}) \overset{\theta}{\leadsto}{}^{*}_{\varphi} \top$ donde $f/n \in \mathcal{D}$, x_{n+1} y x_i son variables distintas, para $i = 1,\ldots,n\}$.*

Ejemplo 18 *Consideremos de nuevo los programas del ejemplo 17. Allí se vio que los programas $\mathcal{R}_1$ y $\mathcal{R}_2$ tenían diferente comportamiento con respecto al observable de las respuestas computadas, pues computaban diferentes soluciones. Sin embargo, $\mathcal{R}_1 \cong \mathcal{R}_2$. Por tanto podemos decir que el conjunto de éxitos básicos no caracteriza convenientemente a los programas desde el punto de vista del observable de las respuestas computadas. Al revisar el concepto de semántica operacional, de acuerdo con la definición 3.3.1 anterior, y permitir que contenga átomos con términos variables, encontramos:*

$$\mathcal{O}^{ca}_{\varphi}(\mathcal{R}_1) = \{0 = 0,\quad f(x) = f(x),\quad f(0) = 0\}$$

mientras que

$$\mathcal{O}^{ca}_{\varphi}(\mathcal{R}_2) = \{0 = 0,\quad f(x) = f(x),\quad f(0) = 0,\quad f(x) = 0\}$$

Vemos así que esta nueva definición de semántica operacional captura lo que estamos buscando: $\mathcal{O}^{ca}_{\varphi}(\mathcal{R}_1) \neq \mathcal{O}^{ca}_{\varphi}(\mathcal{R}_2)$ cuando $\mathcal{R}_1 \ncong \mathcal{R}_2$. Así pues, esta semántica, modela de forma adecuada el comportamiento del programa respecto al observable de las respuestas computadas.

La equivalencia entre las semánticas operacional y la de menor punto fijo es establecida por medio de la aplicación de el siguiente lema.

Lema 3.3.2 *Si $\mathcal{R} \in \mathbb{R}_{inn}$ entonces $\mathcal{O}^{ca}_{inn}(\mathcal{R}) = \mathcal{F}^{ca}_{inn}(\mathcal{R})$.*
Si $\mathcal{R} \in \mathbb{R}_{out}$, entonces $\mathcal{O}^{ca}_{out}(\mathcal{R}) = \{l = r \in \mathcal{F}^{ca}_{out}(\mathcal{R}) \mid \perp$ no ocurre en $r\}$

Demostración. ($\supseteq$): Sea $(f(\bar{x})\theta_i = t_i) \in \mathcal{F}^{ca}_{\varphi}(\mathcal{R})$ donde $t_i \in \tau(C, X)$ y $\perp$ no ocurre en t_i. Por el Lema 3.2.5, existe una derivación de narrowing con la estrategia φ tal

[4]Diremos que dos programas son equivalentes con respecto a la semántica O, $\mathcal{R}_1 \cong \mathcal{R}_2$, si y solo si las respectivas denotaciones son iguales, $O(\mathcal{R}_1) = O(\mathcal{R}_2)$

que $f(\bar{x}) = y \overset{\theta_i}{\leadsto}{}^*{}_\varphi\ t_i = y$. Ahora bien, como t_n es un término constructor y $\perp$ no ocurre en t_i, entonces $t_i = y \overset{\{y/t_i\}}{\leadsto}{}_\varphi\ \top$. Por la Definición de $\mathcal{O}^{ca}_\varphi(\mathcal{R})$, tenemos que

$$(f(\bar{x}) = y)\theta_i\{y/t_i\} \in \mathcal{O}^{ca}_\varphi(\mathcal{R})$$

Ya que $y \notin \mathcal{V}ar(\theta_i)$ entonces

$$(f(\bar{x})\theta_i = t_n) \in \mathcal{O}^{ca}_\varphi(\mathcal{R})$$

Cuando las ecuaciones son constructoras, diferenciamos dos casos: $\varphi = inn$ o $\varphi = out$.

Si $\varphi = inn$ y $d^i_l = d^i_r \in \mathcal{F}^{ca}_{inn}(\mathcal{R})$, entonces $d^i_l = d^i_r \in \Im^{inn}_{\mathcal{R}}$ el cual es un subconjunto de $\mathcal{O}^{ca}_{inn}(\mathcal{R})$. Se concluye, por la Definición 3.3.1, $d^i_l = d^i_r \in \mathcal{O}^{ca}_{inn}(\mathcal{R})$.

El caso en que $\varphi = out$ y $d^i_l \approx d^i_r \in \mathcal{F}^{ca}_{out}(\mathcal{R})$ es análogo.

Nótese que descartamos en $\mathcal{F}^{ca}_{out}(\mathcal{R})$ y en $\mathcal{O}^{ca}_{out}(\mathcal{R})$ todas las ecuaciones que contienen el símbolo $\perp$ en su parte derecha.

($\subseteq$): Sea $(s = t) \in \mathcal{O}^{ca}_\varphi(\mathcal{R})$, por medio de la definición de $\mathcal{O}^{ca}_\varphi(\mathcal{R})$, se deduce que la ecuación $s = t$ tiene la forma $(f(\bar{x}) = y)\theta'$. Como θ' es una sustitución constructora, entonces la ecuación anterior se puede escribir como $(f(\bar{x})\theta = t_n)$, donde $\theta = \theta'_{\restriction f(\bar{x})}$ y $t_n \in \tau(C \cup V)$. Distinguimos dos casos:

Si $f/m \in \mathcal{C}$ entonces trivialmente se cumple que $(s = t) \in \mathcal{F}^{ca}_\varphi(\mathcal{R})$ (ver la primera parte de la demostración).

Si $f/m \in \mathcal{D}$, por la definición de $\mathcal{O}^{ca}_\varphi(\mathcal{R})$, existe una derivación de narrowing con la estrategia φ

$$f(\bar{x}) = y \overset{\theta}{\leadsto}{}^*{}_\varphi\ t_n = y \overset{\{y/t_n\}}{\leadsto}{}_\varphi\ \top$$

Por el Lema 3.2.5, $(f(\bar{x})\theta = t_n) \in \mathcal{F}^{ca}_\varphi(\mathcal{R})$, por lo tanto $(s = t) \in \mathcal{O}^{ca}_\varphi(\mathcal{R})$ □

El siguiente operador $inprogress(S)$.

Definición 3.3.3 *Sea S un conjunto de ecuaciones y Σ la signatura correspondiente. Definimos:*

$$inprogress(S) = \{\lambda = \rho \in S \mid \perp \textit{ ocurre en } \rho,\ o\ \rho \textit{ contiene símbolos de función definido de } \Sigma\}$$

Como consecuencia de la definición, $inprogress(\mathcal{O}^{ca}_\varphi(\mathcal{R})) = \emptyset$. El siguiente resultado resume la relación entre las denotaciones de punto fijo y operacional de un programa.

Teorema 3.3.4 *La siguiente relación es cierta:*

$$\mathcal{O}^{ca}_\varphi(\mathcal{R}) \quad = \quad \mathcal{F}_\varphi(\mathcal{R}) - inprogress(\mathcal{F}_\varphi(\mathcal{R}))$$

Demostración. Se deriva fácilmente del 3.3.2 □

Para dar mayor claridad, resumimos la relación entre las tres diferentes denotaciones de programa que hasta ahora hemos planteado $\mathcal{F}_\varphi(\mathcal{R})$, $\mathcal{F}^{ca}_\varphi(\mathcal{R})$ y $\mathcal{O}^{ca}_\varphi(\mathcal{R})$. La

semántica composicional de punto fijo $\mathcal{F}_\varphi(\mathcal{R})$ que modela computaciones tanto de éxito como parciales (intermedias y no terminantes) se obtiene por medio de la computación del menor punto fijo, lfp, del operador de consecuencias inmediatas $T_{\mathcal{R}}^{\varphi}$. Un subconjunto de la denotación $\mathcal{F}_\varphi(\mathcal{R})$ es la *semántica (de punto fijo) de respuestas computadas* $\mathcal{F}_\varphi^{ca}(\mathcal{R})$, la que se obtiene de $\mathcal{F}_\varphi(\mathcal{R})$ al eliminar las ecuaciones que modelan computaciones intermedias (esto es, todas las ecuaciones de la forma $f(\bar{t}) = s$ donde s "no ha alcanzado su valor") y esta es la única semántica que nos permite ejecutar objetivos g, no triviales, por unificación simple de $flat(g)$ con las ecuaciones en la denotación. Note que las semántica $\mathcal{F}_\varphi^{ca}(\mathcal{R})$ aún modela funciones no terminantes, mediante la utilización de símbolo $\perp$. Finalmente, la *semántica operacional del conjunto de éxitos* $\mathcal{O}_\varphi^{ca}(\mathcal{R})$ detecta únicamente derivaciones de éxito, esto es, esta modela de forma adecuada el observable de las respuestas computadas.

El siguiente teorema relaciona la semántica no–básica $\mathcal{O}_\varphi^{ca}(\mathcal{R})$ con el semántica estándar del E-modelo mínimo de Herbrand $\mathcal{M}(\mathcal{R})$.

Teorema 3.3.5 *Sea $\mathcal{R}$ un programa en $\mathbb{R}_\varphi$ y $r^{\mp}$ el cierre transitivo y simétrico de la relación r bajo reemplazamiento (propiedad de* f-sustitutividad*). Entonces, $\mathcal{M}(\mathcal{R}) = [\mathcal{O}_\varphi^{ca}(\mathcal{R})]^{\mp}$.*

Demostración. Se sigue inmediatamente de la definición. □

De acuerdo con el Teorema 3.3.5, la semántica no básica $\mathcal{O}_\varphi(\mathcal{R})$ puede verse como una representación no estándar de la semántica del $\mathcal{E}$-modelo mínimo de Herbrand $\mathcal{M}(\mathcal{R})$ que contiene la información necesaria para determinar las respuestas computadas, como lo establece el Teorema 3.2.6.

Capítulo 4

Depuración declarativa

4.1 Programas lógicos

4.1.1 Conceptos básicos

Según [Shaphiro, 1982; Ferrand y Tessier, 1996] el *diagnóstico* se refiere a la identificación de un error en un programa que se comporta incorrectamente, es decir, es el proceso de identificación de la parte del programa responsable de la violación del requerimiento mientras la *depuración* es la identificación, ubicación y corrección del error. En la literatura que se relaciona en esta tesis, generalmente, se utilizan estas dos palabras indistintamente [Comini *et al.*, 1996]. El término *depuración* se identifica con el campo de la programación en el que se compara lo que el programa calcula realmente con lo que el programador desea calcular y su objetivo principal es eliminar los errores que se encuentran en este proceso. El *diagnóstico (depuración) declarativo* se define como sigue [Comini *et al.*, 1996], en concordancia con los enfoques dados en este campo por [Shaphiro, 1982; Lloyd, 1987a; Ferrand, 1987].

Definición 4.1.1 *Sea $\mathcal{P}$ un programa, $S(\mathcal{P})$ la semántica declarativa de $\mathcal{P}$ y $\mathcal{M}$ una especificación de la semántica deseada de $\mathcal{P}$. El diagnóstico declarativo es un método para probar la corrección y completitud de $\mathcal{P}$ con respecto a $\mathcal{M}$, ó determinar los errores y los componentes del programa que son fuente de error, en el caso en que $S(\mathcal{P}) \neq \mathcal{M}$.*

La *depuración declarativa* se relaciona con las propiedades de la teoría de modelos de la programación declarativa. La semánticas declarativas exploradas más comúnmente son: el modelo mínimo de Herbrand [Shaphiro, 1982], el conjunto de modelos de la compleción del programa [Lloyd, 1987b] y el conjunto de consecuencias lógicas atómicas del programa [Ferrand, 1987; Ferrand y Tessier, 1996].

Con miras a dar una definición más global, de tal manera que podamos extender posteriormente a otros paradigmas. Es posible enfocar los métodos de diagnóstico de acuerdo con una propiedad que se desee observar en una computación, mediante la especificación de una semántica que la modele adecuadamente. La definición del *observable* depende del paradigma de programación y de la propiedad a observar. Algunos ejemplos de observables de la programación lógica son: patrones de llamada, respuestas computadas, respuestas parciales, resultantes, derivaciones que terminan, éxitos y fallos (definidos en [Comini *et al.*, 1994]). Para ello vamos a precisar los conceptos de corrección y completitud de un programa mediante una formulación paramétrica con respecto al observable α.

Definición 4.1.2 *Sea $\mathcal{P}$ un programa , α una propiedad observable, $\mathcal{M}_\alpha$ la especificación de la semántica deseada de $\mathcal{P}$ que modela el observable α y $\mathcal{O}_\alpha(\mathcal{P})$ la semántica operacional de $\mathcal{P}$ con respecto a α.*

1. *$\mathcal{P}$ es* parcialmente correcto *con respecto a $\mathcal{M}_\alpha$, si $\mathcal{O}_\alpha(\mathcal{P}) \subseteq \mathcal{M}_\alpha$.*
2. *$\mathcal{P}$ es* completo *con respecto a $\mathcal{M}_\alpha$, si $\mathcal{M}_\alpha \subseteq \mathcal{O}_\alpha(\mathcal{P})$.*
3. *$\mathcal{P}$ es* totalmente correcto *con respecto a $\mathcal{M}_\alpha$, si $\mathcal{M}_\alpha = \mathcal{O}_\alpha(\mathcal{P})$.*

La siguiente definición, dada en [Comini *et al.*, 1995a], es una extensión para el diagnóstico con respecto al observable de respuestas computadas *s*-semántica de las definiciones dadas en [Shaphiro, 1982; Lloyd, 1987b; Ferrand, 1987].

Definición 4.1.3 *Sea $\mathcal{P}$ un programa, $O(\mathcal{P})$ la semántica operacional* [1] *de $\mathcal{P}$ y $\mathcal{M}$ una especificación de la semántica deseada de $\mathcal{P}$.*

1. *$\mathcal{P}$ es* parcialmente correcto *con respecto a $\mathcal{M}$, si $O(\mathcal{P}) \subseteq \mathcal{M}$.*
2. *$\mathcal{P}$ es* completo *con respecto a $\mathcal{M}$, si $\mathcal{M} \subseteq O(\mathcal{P})$.*
3. *$\mathcal{P}$ es* totalmente correcto *con respecto a $\mathcal{M}$, si $\mathcal{M} = O(\mathcal{P})$.*

Los algoritmos de depuración declarativa clásicos están basados en una teoría que exige que la semántica deseada M sea declarada extensionalmente. Sin embargo, en general M es infinita. Algunos algoritmos prácticos pueden manejar este tipo de semánticas trabajando con síntomas que se obtienen usando técnicas de generación de tests. Estos algoritmos son los llamados *algoritmos dirigidos por los síntomas*. Otra forma de manejar semánticas infinitas es utilizar *aproximaciones* de la semántica, que generalmente se basan en la teoría de la *interpretación abstracta*.

[1] O(P) modela el comportamiento operacional del programa con respecto a las sustituciones de respuesta computada. La semántica declarativa equivalente es la *s*-semántica, que subsume la semántica estándar [Falaschi *et al.*, 1993].

La interpretación abstracta [Cousot y Cousot, 1977; Cousot y Halbwachs, 1978; Cousot y Cousot, 1979] es una técnica para el análisis estático del comportamiento de la ejecución de un programa y está basado en la idea de construir aproximaciones de la semántica del programa. Las técnicas de diagnóstico abstracto aparecen como una combinación de tres técnicas bien conocidas: la *depuración declarativa* [Shaphiro, 1982; Lloyd, 1987b], la aproximación conocida como s-semántica (o semántica de respuestas computadas) para la definición de programas que modelen varios comportamientos observables y, por último, la teoría de la *interpretación abstracta*. El *diagnóstico abstracto* pretende extender la depuración declarativa al caso donde las especificaciones son finitas y definen una propiedad del programa (observables) o su semántica. De esta forma, se obtiene un marco potente con el cual desarrollar herramientas para el diagnóstico y la verificación de programas.

Tal como lo hemos mencionado anteriormente, el proceso de depuración de un programa involucra tres fases: detección del error, localización del error y corrección del error [Dershowitz y Lee, 1992]. Este último lo realiza a menudo el programador mientras que los dos primeros se resuelven con la ayuda de un *oráculo* que representa la semántica deseada y responde a preguntas acerca de la validez de las ecuaciones. Sin embargo, existen también propuestas que permiten sintetizar programas a partir de una especificación preliminar ejecutable, un proceso automático de depuración y diagnóstico y un espacio de búsqueda inductiva para programas; como es el caso de [Dershowitz y Lee, 1992].

Las técnicas de depuración declarativa estándar de los programas lógicos puros se formulan en el marco de la teoría de modelos lógica. La formulación más clásica se basa en la semántica del menor modelo de Herbrand del programa, la cual es adecuada para modelar las características de los programas lógicos en el caso átomos básicos *ground*. El enfoque de las *s*-semánticas constituyen una caracterización más cercana al comportamiento del programa y, consecuentemente, son una herramienta más potente de ayuda a las técnicas de depuración declarativa.

En lo que sigue, adoptaremos la noción de interpretación *no ground* de [Falaschi *et al.*, 1993, 1989] que, básicamente, considera como interpretación un subconjunto de la base de Herbrand extendida, $\mathcal{B}_{\mathcal{V}}$, formada por todos los átomos, posiblemente no básicos, inducidos por el programa P (modulo la varianza inducida por el cambio de nombre).[2] También, por abuso, no haremos distinción entre objetivos atómicos y átomos.

[2]Por abuso del lenguaje, estamos utilizando la misma notación que usamos para la base de Herbrand en el caso ecuacional.

4.1.2 Semántica de programas lógicos

Aunque la gran mayoría de las propuestas están definidas considerando *semánticas básicas*, nosotros centramos nuestra atención en el caso no básico, ya que, como lo hemos dicho, para efecto de la depuración declarativa sólo nos interesa el observable de las respuestas computadas.

La semántica que modela el observable de las respuestas computadas, es un conjunto de átomos posiblemente no *básicos* (no *ground*) que puede ser caracterizada como el menor punto fijo (*lfp*) del siguiente operador de consecuencias inmediatas[3]: sea $\mathcal{B}_\mathcal{V}$ es la base de Herbrand con variables, $\mathcal{I}$ un subconjunto de $\mathcal{B}_\mathcal{V}$ y sea $A = p(x_1, \ldots, x_n)$, donde las x_i son variables distintas.

$$\begin{aligned} T^s{}_\mathcal{P}(\mathcal{I}) \quad = \quad \{A\theta \in \mathcal{B}_\mathcal{V} \mid \quad & A' \leftarrow B_1, \ldots, B_n \ll \mathcal{P}, \\ & \{B'_1, \ldots, B'_n\} \subseteq \mathcal{I}, \\ & \theta = mgu((B_1, \ldots, B_n), (B'_1, \ldots, B'_n))\} \end{aligned} \tag{4.1}$$

La misma denotación puede ser obtenida de modo *descendente*, por consideración las respuestas computadas de objetivos atómicos más generales. Sea A un objetivo de la forma $A = p(x_1, \ldots, x_n)$. La expresión $A \overset{\theta}{\longmapsto}{}^* \square$ representa el la derivación finita de éxito del objetivo A con el programa $\mathcal{P}$, donde θ es al respuesta computada.

$$\mathcal{O}^s(\mathcal{P}) = \{A\theta \in \mathcal{B}_\mathcal{V} \mid A \overset{\theta}{\longmapsto}{}^* \square\} \tag{4.2}$$

Por último, recordamos la semántica de un objetivo G con respecto a un programa $\mathcal{P}$, en donde $g \equiv p(x_1, \ldots, x_n)$ y las x_i son variables distintas.

$$\mathcal{O}_P(g) = \{\theta \mid g \overset{\theta}{\longmapsto}{}^* \square\} \tag{4.3}$$

En esta aproximación, es posible plantear el concepto de cláusula incorrecta y átomo no cubierto mediante el uso del operador de consecuencias inmediatas dado en [Comini *et al.*, 1995a].

Definición 4.1.4 *Sea $\mathcal{M}$ una interpretación deseada y $T^s_\mathcal{P}$ el operador de consecuencias inmediatas asociado al programa $\mathcal{P}$.*

1. *Si existe un átomo A tal que $A \notin \mathcal{M}$ y $A \in T^s_{\{c\}}(\mathcal{M})$, entonces la cláusula $c \in \mathcal{P}$ es incorrecta sobre A.*

2. *Un átomo A es no cubierto si $A \in \mathcal{M}$ y $A \notin T^s_\mathcal{P}(\mathcal{M})$.*

En la definición 4.1.4, $T^s_{\{c\}}$ es el operador de consecuencias inmediatas asociado al programa cuya única cláusula es c y $T^s_\mathcal{P}$ se refiere al operador de consecuencias

[3]En general, dicho operador se presenta como una función T, a la que aplicamos la siguiente definición. Sea H un conjunto y $T : 2^H \to 2^H$ un operador monótono. El menor punto fijo de T es el menor subconjunto $\mathcal{I}$ de H tal que $T(\mathcal{I}) \subseteq \mathcal{I}$. Así, $T(\mathcal{I}) \subseteq \mathcal{I} \Rightarrow lfp(T) \subseteq \mathcal{I}$. El mayor punto fijo es el mayor subconjunto $\mathcal{I}$ de H tal que $\mathcal{I} \subseteq T(\mathcal{I})$. De este modo, $\mathcal{I} \subseteq T(\mathcal{I}) \Rightarrow \mathcal{I} \subseteq gfp(T)$.

inmediatas no *ground* de un programa $\mathcal{P}$ que modela respuestas computadas. Esta definición permite un nuevo enfoque en el diagnóstico declarativo, en el que no son necesarios síntomas de incorrección o incompletitud para determinar un error en un programa dado.

Árboles de demostración Los árboles SLD son estructuras usadas comúnmente para describir la semántica operacional de lenguajes lógicos y permiten expresar el comportamiento de un programa al realizar su ejecución. El conjunto de árboles SLD para un programa puede verse como una estructura abstracta con una relación de orden, la inclusión de árboles. Es posible probar que el conjunto de árboles para un programa $\mathcal{P}$ forma un retículo completo bajo la relación de inclusión y el conjunto de árboles generados por un objetivo g vía una regla de computación r en un programa $\mathcal{P}$ es isomorfo al conjunto de resultantes bien formados[4] de g en $\mathcal{P}$ vía r, lo que garantiza que cada resultante se puede identificar con un nodo y cada nodo con un resultante. En este marco, un observable mirada al árbol SLD para abstraer una propiedad, como pueden ser las respuestas computadas. La abstracción de las propiedades de un programa da lugar a una gran cantidad de posibilidades para observar el comportamiento de un programa. Un marco formal algebraico para describir distintos observables se encuentra en [Comini y Levi, 1994].

Las respuestas computadas, en programas lógicos puros, también pueden ser descritas usando *árboles de computación* [Shaphiro, 1982] o *demostración* [Naish, 1997; Lloyd, 1987b; Boye *et al.*, 1997] que difieren de los árboles de búsqueda o ejecución convencionales. Cada nodo en el árbol de computación contiene un átomo "probado" en la ejecución del programa, sus hijos son el cuerpo de la instancia de cláusula que se utilizó para derivar el nodo, un hijo por cada átomo del cuerpo; las hojas son instancias de hechos del programa o predicados del sistema. Es importante tener en cuenta que, una vez obtenido el árbol, que se supone finito, la información contenida es independiente de la estrategia de búsqueda y de la regla de computación. El árbol de demostración se puede construir por un meta intérprete [Safra y Shapiro, 1986], por una transformación del programa [Naish y Barbour, 1994] o se puede usar el átomo para representar de manera *implícita* su propio árbol de demostración [Naish, 1997].

Si un objetivo g es *básico* y su ejecución en un programa $\mathcal{P}$ termina con éxito, la semántica $O_{\mathcal{P}}(g)$ está dada por el conjunto cuyo único elemento es la sustitución ε y la ejecución de g se puede representar en un árbol de computación completo. Si g es no *básico*, cada sustitución obtenida en $O_{\mathcal{P}}(g)$ da origen a un árbol de computación completo. Calcular $g\theta$ y luego ejecutarlo da origen a una tercera posibilidad, la cual es útil para la simulación del oráculo, cuya importancia se describe en la siguiente

[4]Un conjunto de resultantes está bien formado si contiene todos los resultantes previos para cada resultante del conjunto.

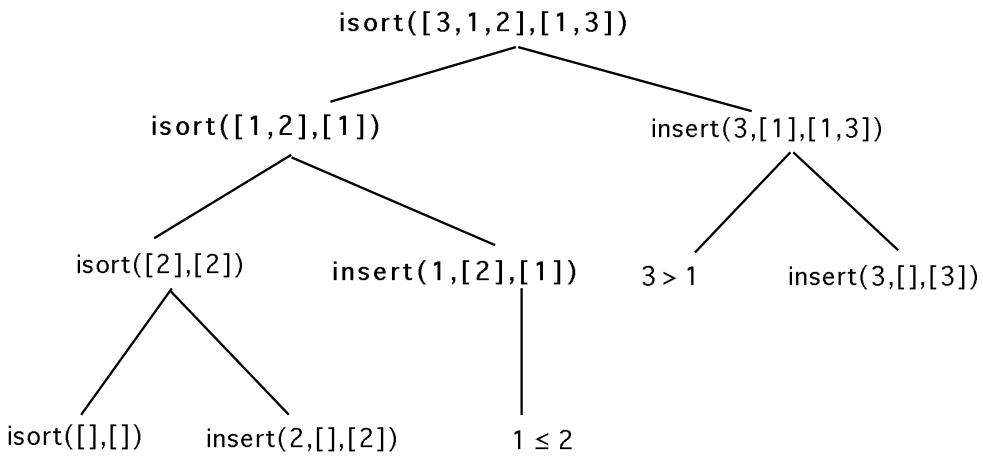


Figura 4.1: Árbol de derivación para el objetivo isort([3, 1, 2], X)

sección.

El siguiente ejemplo, dado en [Naish, 1997], muestra el árbol de demostración (computación) para el programa **isort**; el árbol de derivación de la figura 4.1 corresponde al objetivo isort([3, 1, 2], X) .

```
isort([ ], [ ]).
isort(N.Ns, Ss) :−
          isort(Ns, Ss),
          insert(N, Ss1, Ss).
insert(N, [ ], [N]).
% insert(N, S.Ss, N.S.Ss) :− Ê% correcta
insert(N, S.Ss, N.Ss) :−
          N ≤ S.
insert(N, S.Ss, N.Ss) :−
          N > S,
          insert(N, Ss0, Ss).
```

4.1.3 Semántica declarativa deseada

Dada la formalización de la semántica declarativa de un programa P, es necesario tener alguna información de la semántica declarativa *deseada* del programa. La semántica deseada se define habitualmente mediante el recurso de un *oráculo* que puede ser representado por el programador, quien responde preguntas acerca de la

validez de los átomos [Shaphiro, 1982; Lloyd, 1987b; Naish, 1997], o se puede describir mediante una especificación ejecutable [Dershowitz y Lee, 1987; Edman y Tärnlund, 1983; Drabent *et al.*, 1988; Kanamori *et al.*, 1989], la cual puede resultar particularmente útil cuando las estructuras de datos son complejas.

En la literatura sobre la depuración declarativa se han usado tres tipos de preguntas realizadas al oráculo [Naish, 1992], con los siguientes objetivos:

- Determinar si un átomo, generalmente ground, es satisfacible en la interpretación deseada.
- Determinar las instancias válidas de un objetivo no ground. Requiere que el oráculo provea los enlaces para las variables; si tiene varios, deben ser dados todos en algunos casos. Las instancias deben ser proporcionadas por el usuario o ha de existir una especificación que las determine.
- Determinar si un conjunto de instancias de un átomo no ground (dadas generalmente por el conjunto de las respuestas devueltas por el programa) contiene todas las instancias válidas del átomo.

Los depuradores al estilo de Shapiro [Shaphiro, 1982] utilizan los dos primeros tipos de preguntas y los del estilo Pereira [Pereira, 1986] la primera y tercera. Para minimizar la cantidad de preguntas al oráculo y reducir el espacio de búsqueda del error, se han propuesto las siguientes estrategias. Una discusión detallada de algunos aparece en [Naish, 1992]:

- Utilizar el árbol de computación creado al ejecutar el programa con un objetivo, para realizar la búsqueda del error [Naish, 1997].
- Acumular en una base de datos las respuestas a las preguntas previas, para evitar que el oráculo responda la misma pregunta varias veces [Shaphiro, 1982].
- Suministrar afirmaciones y restricciones sobre el comportamiento de los datos de entrada y salida, para determinar cuando éstas son violadas [Shaphiro, 1982].
- Utilizar versiones previas del programa para las que se conoce su comportamiento [Shaphiro, 1982].
- Utilizar objetivos que se comportan erróneamente para guiar la búsqueda de lo que comúnmente se conoce como *síntomas* [Shaphiro, 1982; Pereira, 1986; Ferrand, 1987; Naish, 1997; Boye *et al.*, 1997].
- Utilizar predicados que permitan generar instancias de átomos, en lugar de que el oráculo las provea [Naish, 1992].

- Preguntar acerca de la satisfacibilidad de un átomo sin necesidad de una instancia del mismo [Naish, 1992].

- Simular el oráculo, mediante una definición de la *s*-semántica para un objetivo y el uso del operador de consecuencias inmediatas [Comini *et al.*, 1995a].

- Definir una especificación parcial positiva de las respuestas computadas por el programa y una especificación parcial negativa de las respuestas no computadas por el programa en el marco de la *s*-semántica [Comini *et al.*, 1995a].

En general, el oráculo es representado por el programador con algunas propiedades que en algunos casos son extremas. Por ejemplo se suele usar un conjunto *finito* para describir la semántica deseada, que expresa exactamente las expectativas del usuario [Shaphiro, 1982]. En este caso, los elementos del conjunto son ground y es posible especificar completamente el oráculo, que puede responder todas las preguntas que se le plantean. Frente a esta situación se propone una alternativa que, agregada a las estrategias mencionadas anteriormente, puede mejorar el proceso de depuración. Dada una pregunta al oráculo en el enfoque conocido como CLP (Constraint Logic Programming), se permite que éste responda con cuatro posibilidades [Boye *et al.*, 1997]: una *afirmación* (assertion) que afecta a la definición del procedimiento mediante una restricción o una regla para establecer condiciones [Bueno *et al.*, 1997b], un *don't know* que elimina el subárbol correspondiente al nodo actualmente visitado y un *si* o un *no* manejados comúnmente.

Una *afirmación* permite expresar propiedades acerca del significado deseado de un programa para algún predicado p. Estas propiedades pueden ser vistas como *especificaciones parciales* de los predicados del programa y son aproximaciones de la semántica deseada del predicado. En [Bueno *et al.*, 1997b] se consideran dos tipos de *afirmaciones*, que corresponden a aproximaciones basadas en *superconjuntos* y *subconjuntos*, las cuales resumimos como sigue. Sea $\mathcal{O}_p$ la semántica del programa con respecto al predicado p e I_p la semántica deseada del programa con respecto al predicado p. Una afirmación del tipo "$:-$`inmodel Pred` $\Rightarrow$ `Cond`" donde *Pred* es un átomo de la forma $p(x_1, \ldots, x_2)$, con $n \geq 0$ y $x_i \neq x_j$ para $i \neq j$, se entiende como una aproximación A_p de tipo "superconjunto" para I_p y se interpreta como: "cualquier átomo de $\mathcal{O}_p$ debe cumplir la propiedad *Cond*". Por ejemplo, la afirmación "$:-$`inmodel qsort(A,B)` $\Rightarrow$ `list(B)`" establece que el resultado de ordenar una lista debe ser una lista. Este tipo de afirmaciones declarativas se utilizan para estudiar el *problema de la corrección*. Si existe un x que pertenece a $\mathcal{O}_p$ y $x \notin A_p$ entonces x es un síntoma de incorrección. Similarmente, una afirmación del tipo "$:-$`inmodel Pred` $\Leftarrow$ `Cond`" representa una aproximación de tipo *subconjunto* A_p de I_p y se interpreta como: "cualquier átomo que satisface *Cond* debe estar en $\mathcal{O}_p$". Por

ejemplo, la afirmación ": $-$`inmodel qsort(A,B)` $\Leftarrow$ (`A` $==$ $[2,1]$, `B` $==$ $[1,2]$)." [5] establece que la lista $[1,2]$ es el orden correcto de la lista $[2,1]$. Este tipo de afirmaciones declarativas se utilizan para establecer cuando un programa es *completo*. Si existe un $x \in A_p$ y x que no pertenece a $\mathcal{O}_p$, entonces x es un síntoma de incompletitud. Es necesario aclarar que este enfoque está dado para programas lógicos con restricciones - CLP y la estructura de las afirmaciones se fija para un contexto más general, propuesto como un lenguaje de afirmaciones para la depuración tanto a nivel declarativo como operacional [Bueno *et al.*, 1997b].

4.1.4 Síntomas y error

Un síntoma es la aparición de una anomalía durante la ejecución de un programa [Ferrand y Tessier, 1996], que resulta de la comparación de la semántica del programa con la semántica deseada por el programador, como lo precisa la siguiente definición [Comini *et al.*, 1995a].

Definición 4.1.5 *Sea $\mathcal{P}$ un programa, $O(\mathcal{P})$ la semántica de $\mathcal{P}$ e $\mathcal{I}$ la semántica deseada de $\mathcal{P}$.*

1. *Un síntoma de incorrección es un átomo A tal que $A \in O(\mathcal{P})$ y $A \notin \mathcal{I}$.*

2. *Un síntoma de incompletitud es un átomo tal que $A \in \mathcal{I}$ y $A \notin O(\mathcal{P})$.*

Desde el punto de vista de la ejecución del programa, un síntoma de incorrección es un resultado que no está en las expectativas del usuario y un síntoma de incompletitud se presenta cuando el programa es incapaz de computar algún resultado esperado por el usuario. El algoritmo de depuración genera todo el *espacio de búsqueda* [6] para un objetivo dado, permite establecer si algunas respuestas están pérdidas en el conjunto de todas las respuestas computadas para el objetivo, con las siguientes posibilidades para ayudar a la exploración del espacio de búsqueda, entre otras: el usuario provee las instancias del objetivo [Shaphiro, 1982], el programa las calcula [Pereira, 1986] o el programa verifica la satisfacibilidad sin que el usuario deba hacer instanciaciones [Naish, 1992].

En [Ferrand y Tessier, 1996] se define el concepto de síntoma mediante el operador abstracto de consecuencias inmediatas. Un síntoma de incorrección[7] es un elemento $h \in lfp(T)$ tal que $h \notin \mathcal{I}$, donde $\mathcal{I}$ es un subconjunto de H que satisface $T(\mathcal{I}) \subseteq \mathcal{I}$. Un síntoma de incompletitud[8] es un elemento $h \in \mathcal{I}$ tal que $h \notin gfp(T)$, donde $\mathcal{I}$ es un subconjunto de H que satisface $\mathcal{I} \subseteq T(\mathcal{I})$. A pesar de la diferencia, ambas definiciones

[5] El símbolo $==$ establece la identidad entre términos.
[6] Conjunto de todas las posibles derivaciones en el árbol de demostración, que se suponen finitas.
[7] Llamado por el autor síntoma (symptom).
[8] Llamado por el autor coo-síntoma (co-symptom)

son equivalentes para programas aceptables[9]. Esto se debe a que el operador de consecuencias inmediatas posee un único punto fijo [Comini *et al.*, 1995a].

En el contexto de las *afirmaciones* (assertions), un síntoma de incorrección de un programa $\mathcal{P}$ es un elemento que pertenece a $\mathcal{O}_p$ y no pertenece al superconjunto A_p de I_p y un síntoma de incompletitud es un elemento que pertenece al subconjunto A_p de I_p y no pertenece a $\mathcal{O}_p$ [Bueno *et al.*, 1997b].

Para la ejecución de los algoritmos de diagnóstico, es necesario dar como entrada un objetivo que sea un síntoma de incorrección o de incompletitud, excepto en el algoritmo propuesto en [Comini *et al.*, 1995a], el cual se ejecuta con objetivos atómicos más generales de la forma $p(X_1, \ldots, X_n)$. La diferencia radica en que la concepción de la depuración se fundamenta en la definición 4.1.4, mientras que las demás se basan en la noción estándar de síntoma.

4.1.5 Respuestas computadas incorrectas

Dado un síntoma de incorrección, decimos que el programa es incorrecto. Esto se debe a la existencia de una cláusula incorrecta, como lo precisa el siguiente teorema dado por [Shaphiro, 1982].

Teorema 4.1.6 *Sea $\mathcal{P}$ un programa y $\mathcal{M}$ una interpretación. Si $\mathcal{P}$ es incorrecto con respecto a $\mathcal{M}$, entonces $\mathcal{P}$ contiene una cláusula incorrecta en $\mathcal{M}$.*

El átomo en un nodo del árbol de computación se considera *erróneo* si no está en la interpretación deseada. Distinguiremos tres clases de nodos de error, de acuerdo con las siguientes definiciones [Naish, 1997].

Nodo erróneo (*erroneous*): Es un nodo que contiene un átomo que no es verdad con respecto a la interpretación deseada. Si el átomo contiene variables, es erróneo si alguna de sus instancias no es verdad en la interpretación deseada. En el ejemplo `isort`, los nodos resaltados son erróneos.

Nodo equívoco (*buggy*): Es un nodo erróneo con hijos válidos. Se origina por una cláusula incorrecta, que es falsa en la interpretación deseada. En el ejemplo, el nodo insert(1, [2], [1]) es equívoco. En [Naish, 1997], se especifica el hecho de que `A` es un nodo equívoco de la siguiente forma.

```
buggy(A) :-
                erroneous(A),
                no existe [A](child(A,B), erroneous(B)).
```

[9]Programas terminantes por la izquierda, esto es, son programas para los cuales las SLD-derivaciones, vía la regla de selección por la izquierda, son finitas. Una definición formal puede ser encontrada en [Comini *et al.*, 1995b].

Nodo equívoco más alto: Es un nodo equívoco cuyos antecesores son erróneos. En el ejemplo, el nodo insert(1, [2], [1]) es el nodo equívoco más alto.

La depuración declarativa para respuestas computadas incorrectas consiste en la búsqueda de un *nodo equívoco*. Para ello, el árbol de computación se recorre preguntando al oráculo por la validez de cada nodo. En [Naish, 1997], se demostración que el proceso de búsqueda para árboles de computación finitos, termina con éxito y, si tiene nodos *equívocos más altos*, los encuentra todos. Este resultado determina la corrección y completitud del siguiente esquema general de algoritmos para el diagnóstico declarativo de respuestas incorrectas, que se aplica al caso de árboles de computación finitos que tengan nodos equívocos más altos y está basado en la siguiente regla.

```
debug(Root, Bug) :−
              erroneus(Root),
              (if existe [Child, Bug1] (
                    child(Root, Child),
                    debug(Child, Bug1))
              then
                    Bug = Bug1
              else
                    Bug = Root).
```

Se invoca al procedimiento `debug` con la raíz del árbol y el resultado es un nodo `buggy`. El algoritmo visita aquellos nodos cuyos padres son erróneos y recorre el árbol de arriba hacia abajo. `Erroneus(A)` verifica la validez del átomo A en la interpretación deseada. Los hijos se representan mediante el predicado `child(Root, Child)`.

Este esquema plantea los aspectos generales para encontrar una cláusula incorrecta y está en concordancia con los desarrollados por otros autores. Las diferencias se deben a la estrategia de búsqueda y a la forma de preguntar al oráculo. En [Shaphiro, 1982], el depurador, para programas definidos, recorre el árbol de abajo hacia arriba partiendo de un síntoma de incorrección y devuelve una cláusula incorrecta, tan pronto encuentra un nodo incorrecto. La siguiente cláusula describe de forma concisa el esqueleto del depurador.

```
debug(A, X) :−
              system(A) → A, X = ok;
              child(A, B),
              debug(B, Xb),
              (Xb ≠ ok → X = Xb;
```

```
erroneus(forall, A, true) → X = ok; X = (A←B)).
```

En la regla anterior, `system` verifica si A es un predicado del sistema, `erroneus` devuelve *true* si toda instancia de A es verdad, utilizando para ello el oráculo. Es importante notar que el programa tiene una cláusula adicional para manejar objetivos conjuntivos y Shapiro propone un segundo algoritmo –divide y consulta– que permite mejorar la estrategia de búsqueda.

En [Lloyd, 1987b], el programa para la depuración es un poco más extenso debido a su diseño para depurar programas extendidos[10], sin embargo las cláusulas para el diagnóstico son esencialmente las mismas.

```
debug(A, X) :−
              child(A, A1 if B),
              succeed(B, B),
              unsatisfiable(B, B),
              debug(B, X).
wrong(A, A1 if B) :−
              unsatisfiable(A, A1),
              child(A, A1 if B),
              erroneus(B, B).
```

En esta regla, `child` determina una instancia ground de una cláusula cuya cabeza empareja con A, `erroneus` provee instancias válidas de B en la interpretación deseada, `unsatisfiable` asegura que las instancias insatisfacibles de A estén dadas y, por último, `succeed` permite verificar si el elemento B devuelto es una sustitución de respuesta computada. La estrategia de búsqueda es de arriba hacia abajo.

En el enfoque de las s-semánticas se establece, para un programa $\mathcal{P}$, la equivalencia entre $O^s_{\mathcal{P}}$ y el menor punto fijo de $T^s_{\mathcal{P}}$ para el observable de respuestas computadas y, en consonancia, permite generar algoritmos cuya estrategia de búsqueda puede ser de arriba hacia abajo (descendente) o de abajo hacia arriba (ascendente). En [Comini *et al.*, 1995a], se propone un algoritmo de diagnóstico cuya estrategia es de arriba hacia abajo y que se diferencia de los presentados anteriormente en que la búsqueda se realiza con objetivos atómicos más generales, no precisa síntomas como entradas y se introduce la posibilidad de simular el oráculo. La diferencia radica en que el proceso de diagnóstico se fundamenta en la definición 4.1.4, se conoce la s-semántica deseada $\mathcal{I}$ y $T_{\mathcal{P}}(\mathcal{I})$ y, al compararlos, se determinan las cláusulas incorrectas del programa cuando el operador $T_{\mathcal{P}}$ tiene un único punto fijo. La simulación del oráculo se basa en este argumento.

[10] En las cláusulas del programa la cabeza es un átomo y el cuerpo es una fórmula de primer orden.

```
incorrect(A :- B) :-
              userdefine(A),
              clause(A, B),
              answer(B),
              freeze(A, A1),
              not(answer(A), A = A1).
```

La simulación del oráculo está dada por la siguiente definición.

Definición 4.1.7 *Sea g un objetivo y $\mathcal{P}$ un programa definido.*

- *El oráculo:*

$$A(g) = \{g\theta \mid g \text{ computa } \theta \text{ en la } s\text{-semántica deseada}\}. \tag{4.4}$$

- *La simulación del oráculo:*

$$\begin{aligned} S(g,\mathcal{P}) \quad = \quad & \{g\theta_1\theta_2 \mid \exists A \leftarrow B_1, B_2, \ldots, B_n \in P, \\ & \exists\theta_1 = mgu(g, A), \\ & \exists\theta_2 : \ (B_1, \ldots, B_n)\theta_1\theta_2 \in A((B_1, \ldots, B_n)\theta_1)\}. \end{aligned} \tag{4.5}$$

El oráculo $A(g)$ se supone dado de manera extensional y representa la s-semántica deseada. En el algoritmo, el predicado de la forma `userdefine`$(p(X1, \ldots, Xn))$ representa los objetivos atómicos más generales, donde p es un predicado que ocurre en el programa y $X1, \ldots, Xn$ son variables distintas. El predicado `clause` determina una cláusula del programa que empareje con el objetivo g al desarrollar el primer paso de la simulación del oráculo $S(g, \mathcal{P})$. El oráculo `answer` instancia sus argumentos de forma no determinista y verifica si una instancia de g pertenece al oráculo $A(g)$ y `freeze` obtiene una copia de la correspondiente instancia del objetivo. En resumen, la simulación del oráculo desarrolla un paso para obtener el objetivo por el uso de las cláusulas del programa y busca en el oráculo las respuestas para el objetivo resultante. Los elementos de $S(g, \mathcal{P})$ y $A(g)$ son clases de equivalencia modulo la varianza, en donde $A(g)$ se calcula por sustituciones aplicadas a g de acuerdo con la s-semántica deseada y $S(g, \mathcal{P})$ de manera similar pero con las cláusulas del programa.

El diagnóstico para programas aceptables tiene éxito y garantiza que encuentra todos las instancias de cláusulas incorrectas para un número finito de objetivos atómicos. El siguiente teorema determina cuándo una cláusula es incorrecta mediante el uso de $S(g, \mathcal{P})$ y $A(g)$ [Comini *et al.*, 1995a].

Teorema 4.1.8 *La cláusula $c \in P$ es incorrecta sobre el átomo $p(X1, \ldots, Xn)\theta$ si y solo si $p(X1, \ldots, Xn)\theta \in S(p(X1, \ldots, Xn),\ \{c\})$ y $p(X1, \ldots, Xn)\theta \notin A(p(X1, \ldots, Xn))$.*

En resumen, para el diagnóstico de respuestas computadas incorrectas se busca una instancia de cláusula cuya cabeza sea incorrecta y el cuerpo válido en la interpretación deseada; variando el tipo de programa sobre el cual se aplica, la semántica utilizada para interpretar el programa, la estrategia de búsqueda, la clase de preguntas, la simulación y el uso de afirmaciones en la concepción del oráculo.

4.1.6 Soluciones perdidas

Un objetivo A *falla inmediatamente* en $\mathcal{P}$ si no existe una cláusula $A' \leftarrow B'$ en $\mathcal{P}$ tal que A' unifique con A. Un objetivo A *falla finitamente* en un programa $\mathcal{P}$ si todas las computaciones de $\mathcal{P}$ sobre A son finitas y cada computación contiene al menos un objetivo que falla inmediatamente [Shaphiro, 1982]. El siguiente teorema sustenta el proceso de diagnóstico para objetivos perdidos:

Teorema 4.1.9 *Sea $\mathcal{P}$ un programa y $\mathcal{M}$ la interpretación deseada. Si $\mathcal{P}$ falla finitamente sobre un objetivo $A \in \mathcal{M}$, entonces existe un objetivo $B \in \mathcal{M}$ en alguna derivación de A tal que ninguna cláusula de $\mathcal{P}$ cubre a B.*

La siguiente cláusula permite definir de manera esquemática el diagnóstico de átomos que no están cubiertos [Naish, 1997].

```
buggy(A) :-
              instance(A, AI),
              valid(AI),
              no existe [B] ( is_clause(AI, B), valid(B)).
```

En general, el algoritmo para encontrar átomos que no están cubiertos parte de un síntoma de incompletitud y busca cada cláusula cuya cabeza unifica con él. Entonces usa el oráculo para comprobar si el cuerpo de la cláusula tiene alguna instancia en la interpretación deseada. Si no existe dicha cláusula, el átomo no está cubierto y termina; la definición del predicado correspondiente al átomo debe ser modificada. Si existe, el algoritmo encuentra un objetivo que falla y es válido según el oráculo en el cuerpo de la cláusula, para ello resuelve cada objetivo componente, utilizando el mismo criterio de manera recursiva, puesto que al menos uno debe fallar [Shaphiro, 1982]. El algoritmo puede encontrar la solución bajo dos formas, un objetivo que sea válido y no empareje con ninguna cláusula o, si empareja con alguna, el cuerpo de la cláusula ha de ser insatisfacible. La existencia de átomos que no están cubiertos puede ser debida a que las cláusulas que definen el átomo tienen alguna instancia que falla. Para corregir este error, se debe adicionar una cláusula o generalizar una existente [Eklund, 1997].

El diagnóstico de átomos que no están cubiertos supone un carga muy grande para el oráculo: determinar si un objetivo tiene solución y encontrarla si existe. Cada vez que se encuentre una cláusula que unifique, el oráculo o el programa deben generar una instancia para que el algoritmo continué. La cantidad de preguntas al oráculo hace necesario diseñar estrategias que permitan disminuir y reducir su complejidad, optimizar la búsqueda y ampliar la posibilidad de respuestas a las preguntas realizadas al oráculo, como lo discutimos en la sección 4.1.3. Los algoritmos clásicos pueden ser encontrados en [Lloyd, 1987b; Ferrand, 1987; Pereira, 1986; Shaphiro, 1982]. Una discusión detallada de ellos se encuentra en [Naish, 1992] donde, adicionalmente, se presentan tres posibles características para enfocar el diagnóstico de respuestas perdidas: un átomo válido que falle, un átomo satisfacible que falle o un átomo que devuelva un conjunto de respuestas incompleto.

En [Comini *et al.*, 1995a] se plantea una versión que difiere en gran medida de las anteriores. No es necesario el uso de síntomas para su ejecución y permite la simulación del oráculo:

```
uncovered(A) :-
              userdefined(A),
              answer(A),
              freeze(A, A1),
              not (clause(A, B), answer(B), A = A1).
```

Los predicados tienen el mismo significado que los de respuestas computadas. La simulación del oráculo está dado por las ecuaciones 4.4 y 4.5 de la definición 4.1.7, se busca ahora elementos del oráculo $A(g)$ que no puedan ser derivados por la simulación del oráculo $A(g, \mathcal{P})$, en lo que se diferencia del algoritmo para determinar cláusulas incorrectas. Por ello, las observaciones mencionadas en la sección 4.1.5 para el caso de respuestas incorrectas son válidas. El diagnóstico para programas aceptables tiene éxito, garantiza que encuentra todos los átomos que no están cubiertos para un número finito de objetivos atómicos y lo sustenta el siguiente teorema [Comini *et al.*, 1995a].

Teorema 4.1.10 *Un átomo* $p(X1, \ldots, Xn)\theta$ *no está cubierto sii* $p(X1, \ldots, Xn)\theta \notin S(p(X1, \ldots, Xn),\ P)$ *y* $p(X1, \ldots, Xn)\theta \in A(p(X1, \ldots, Xn)$.

De manera similar al caso de respuestas computadas incorrectas, en los diferentes algoritmos propuestos para la depuración de soluciones perdidas, la concepción en general es la misma y las diferencias se establecen en cuanto a las estrategias de búsqueda, el uso o no de síntomas y las preguntas realizadas al oráculo. En los enfoques clásicos el manejo del oráculo presenta grandes dificultades debido a la cantidad de información, los supuestos y las restricciones que se deben imponer. Con el uso de las s-semánticas, el horizonte de aplicación es más amplio y es posible probar que la

ausencia de átomos que no están cubiertos[11] implica la completitud para los programas cuyo operador de consecuencias inmediatas asociado tenga un único punto fijo. La implicación anterior no es cierta si la unicidad del operador no se cumple, esto es, exiten programas que no tienen átomos que no están cubiertos y no son completos [Comini *et al.*, 1995a,b].

4.1.7 Programas totalmente correctos

El ideal en la programación es obtener programas totalmente correctos y tener un criterio que permita decidir si el programa es totalmente correcto con respecto a una interpretación deseada. El siguiente teorema establece un resultado en ese sentido [Comini *et al.*, 1995a].

Teorema 4.1.11 *Sea $T^s_{\mathcal{P}}$ el operador de consecuencias inmediatas no ground definido en la ecuación 4.1, $\mathcal{P}$ un programa y $\mathcal{M}$ una interpretación deseada. Si $T^s_{\mathcal{P}}$ tiene un único punto fijo, entonces $\mathcal{P}$ es* totalmente correcto *con respecto a $\mathcal{M}$ si y solo si no existen cláusulas incorrectas y átomos que no están cubiertos.*

La restricción de que el operador de consecuencias inmediatas debe tener un único punto fijo es bastante fuerte y la cumplen los programas aceptables, para los cuales se garantiza que los métodos de diagnóstico pueden estar basados en la definición 4.1.4. Un teorema similar aparece en [Lloyd, 1987b] con la condición de que $\mathcal{P}$ sea un programa extendido para contener fórmulas de primer orden en el cuerpo de las cláusulas y $\mathcal{M}$ la interpretación deseada, la cual es una interpretación de Herbrand normal para la compleción de $\mathcal{P}$.

Los métodos de diagnóstico no pueden ser efectivos bajo las condiciones de finitud que se imponen a la semántica deseada y del programa. Los algoritmos propuestos hasta ahora son inefectivos en el caso en que $\mathcal{M}$ sea infinita o el oráculo puede devolver un número infinito de respuestas para alguna pregunta. El problema puede resolverse con una aproximación finita de la semántica deseada.

En [Comini *et al.*, 1995a] se propone una solución que permite aproximar el comportamiento deseado $\mathcal{M}$ mediante una especificación parcial finita, que se representa por el par $(\mathcal{M}^+, \mathcal{M}^-)$: $\mathcal{M}^+$ es el conjunto que representa la especificación parcial de las respuestas computadas por el programa para objetivos atómicos más generales, llamada *especificación parcial positiva* ($\mathcal{M}^+ \subseteq \mathcal{M}$) y $\mathcal{M}^-$ es el conjunto que representa la especificación parcial de las respuestas no computadas por el programa para objetivos atómicos más generales, llamada *especificación parcial negativa* ($\mathcal{M}^- \subseteq \mathcal{M}$). Ambos conjuntos son finitos. Como la especificación es parcial, los conceptos de corrección y completitud se modifican. Un programa es *p-correcto* para algún objetivo g, si no existe ninguna respuesta computada θ, hasta el momento, tal

[11] En el sentido de la definición 4.1.4

que $g\theta \in \mathcal{M}^-$ y un programa es *p-completo* para algún objetivo g, si toda respuesta computada θ, hasta el momento, satisface $g\theta \in \mathcal{M}^+$. Similarmente, los conceptos de átomo no p-cubierto y cláusula p-incorrecta se modifican según la siguiente definición [Comini *et al.*, 1995a].

Definición 4.1.12 *Sea $\mathcal{M}$ una interpretación deseada y $T^s_{\mathcal{P}}$ el operador de consecuencias inmediatas asociado con el programa $\mathcal{P}$.*

1. *Si existe un átomo A tal que $A \notin (\mathcal{M}^-)'$ [12] y $A \in T_{\{c\}}((\mathcal{M}^-)')$, entonces la cláusula $c \in \mathcal{P}$ es p-incorrecta sobre A.*

2. *Un átomo A es no p-cubierto si $A \in \mathcal{M}^+$ y $A \notin T_{\mathcal{P}}(\mathcal{M}^+)$.*

El problema fundamental es que $(\mathcal{M}^-)'$ puede ser infinito, entonces se supone que $\mathcal{M}^+ = (\mathcal{M}^-)'$ para definir los métodos de diagnóstico con respecto a las especificaciones parciales. Aunque el diagnóstico parcial habitualmente devuelve un subconjunto de cláusulas incorrectas y un superconjunto de los átomos que no están cubiertos, ofrece la posibilidad de presentar mejoras en las especificaciones $\mathcal{M}^+$ y $\mathcal{M}^-$ cada vez que se efectúa una corrección del programa y se puede implementar de manera eficiente con estrategias de arriba hacia abajo o de abajo hacia arriba. A continuación presentamos los algoritmos para detectar cláusulas p-incorrectas y átomos no p-cubiertos, con las correspondientes simulaciones del oráculo dados en [Comini *et al.*, 1995a].

```
pincorrect(A :- B) :-
                npanswer(A),
                freeze(A, A1),
                clause(A, B),
                panswer(B), A = A1.


puncovered(A) :-
                panswer(A),
                freeze(A, A1),
                not (clause(A, B), panswer(B), A = A1).
```

La simulación del oráculo está dada por las siguientes definiciones.

Definición 4.1.13 *Sea g un objetivo y $\mathcal{P}$ un programa definido.*

- *El oráculo positivo:*

$$A^+(g) = \{g\theta \mid g \text{ computa } \theta \text{ en la s-semántica deseada}\}. \tag{4.6}$$

[12] Donde $(\mathcal{M}^-)'$ es el complemento de $\mathcal{M}^-$ con respecto a $\mathcal{M}$.

- *El oráculo negativo:*

$$A^-(g) = \{g\theta \mid g \ \ \textit{no computa } \theta \textit{ en la s-semántica deseada}\}. \tag{4.7}$$

- *La simulación del oráculo:*

$$\begin{aligned} S^+(g,\mathcal{P}) \ = \ & \{g\theta_1\theta_2 \mid \exists A \leftarrow B_1, B_2, \ldots, B_n \in \mathcal{P}, \\ & \exists\theta_1 = mgu(g, A), \\ & \exists\theta_2 : \ (B_1, \ldots, B_n)\theta_1\theta_2 \in A^+((B_1, \ldots, B_n)\theta_1)\}. \end{aligned} \tag{4.8}$$

Teorema 4.1.14 *La cláusula* $c \in \mathcal{P}$ *es p-incorrecta sobre* $p(X1, \ldots, Xn)\theta$ *sii* $p(X1, \ldots, Xn)\theta \in S^+(p(X1, \ldots, Xn),\ \{c\})$ *y* $p(X1, \ldots, Xn)\theta \notin A^-(p(X1, \ldots, Xn))$.

Teorema 4.1.15 *Un átomo* $p(X1, \ldots, Xn)\theta$ *no está cubierto sii* $p(X1, \ldots, Xn)\theta \notin S^+(p(X1, \ldots, Xn),\ \mathcal{P})$ *y* $p(X1, \ldots, Xn)\theta \in A^+(p(X1, \ldots, Xn)$.

La búsqueda se maneja por elementos de la especificación positiva y negativa, obtenidos de los correspondientes oráculos $A^+(g)$ y $A^-(g)$, los cuales están manejados por `panswer` y `nanswer`, respectivamente. Los demás predicados se definen como ya lo hemos considerado en la sección 4.1.5. Una desarrollo completo se encuentra en [Comini *et al.*, 1995a]. En [Comini *et al.*, 1995b] se definen aproximaciones en el marco de la teoría de la interpretación abstracta [Cousot y Cousot, 1977, 1979].

4.2 Programas funcionales

4.2.1 Aspectos generales

El campo de la depuración declarativa de los programas funcionales está mucho menos definido y son pocas las investigaciones en él.

Desde el punto de vista de la depuración declarativa, los tipos de error que se estudian en los programas funcionales son semejantes a los que se presentan en los programas lógicos puros. Esto se debe a la naturaleza de los métodos de diagnóstico que hemos tratado, en los cuales se compara lo que el programa calcula con lo que el programa debe calcular. A pesar de la similitud, en la literatura relacionada no se plantea una semántica declarativa para programas funcionales de manera explícita, ni la correspondiente formulación del tipo de errores que se presentan. En lo que sigue, haremos referencia al computo de respuestas incorrectas al estilo de los programas lógicos puros también para el caso de programas funcionales con estrategia de evaluación perezosa, tal como se propone en [Naish, 1993], que constituye la referencia más cercana a nuestro trabajo.

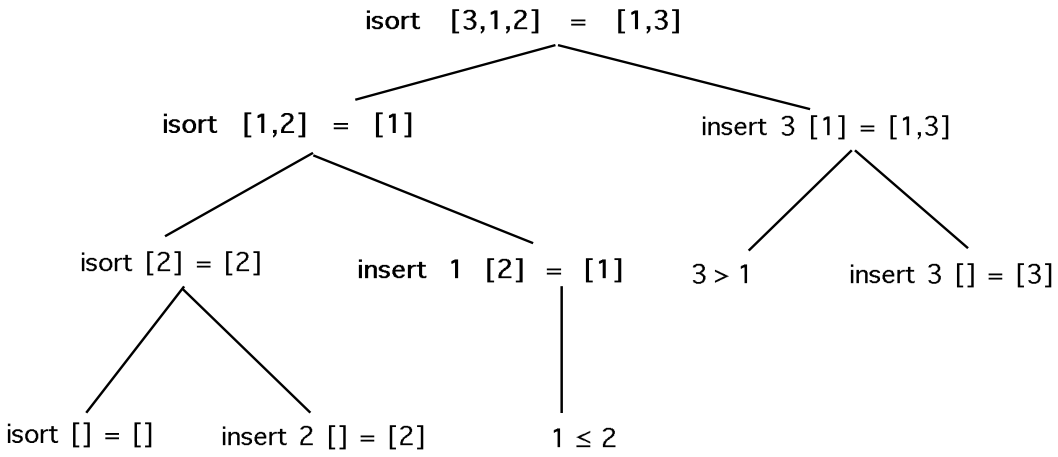


Figura 4.2: Árbol de computación para el objetivo $isort[3,1,2] = X$.

La evaluación de una expresión se representa en un árbol de demostración[13] de manera similar a la que se utiliza en los programas lógicos puros, con una interpretación diferente de los conceptos involucrados: las cláusulas son ecuaciones, los átomos son términos con un símbolo de función en el nivel superior, la verdad se interpreta como corrección de la evaluación y el éxito como evaluación de una función hasta un valor formado sólo por símbolos constructores [Naish, 1993]. La raíz contiene el término objetivo y su resultado. Cada nodo contiene una expresión a evaluar de la forma $f\ a_1 a_2 \ldots$, un resultado computado y un hijo por cada llamada a función que aparece en el lado derecho de la ecuación con la que empareja. Los argumentos se reducen para la evaluación del término objetivo [Naish y Barbour, 1995].

El siguiente ejemplo muestra el programa `isort` en versión funcional [Naish, 1997]. En la figura 4.2 se recoge una computación del objetivo `isort[3, 1, 2] = X`.

```
isort ([ ]) = [ ].
isort (N.Ns) = insert(N, isort(Ns)).


insert(N, [ ]) = [N]).
insert(N, S.Ss) =
          (N > S ? S.insert(N, Ss0).
               :  N.Ss0).          % debe ser N.S.Ss0
```

El siguiente esquema plantea el proceso de depuración de arriba hacia abajo para

[13] Llamado árbol de evaluación de dependencias en [Sparud y Nilsson, 1995]

detectar nodos *equívocos*[14] dado en [Naish y Barbour, 1995]. El predicado `erroneous` se utiliza para consultar al oráculo y `children` para la representación del árbol. Para ejecutarlo es necesario conocer un síntoma de incorrección, como en todas las propuestas revisadas.

```
debug ::  Node -> [Node]
debug n | not (erroneous n)
     = [ ]
debug n | erroneous n && bs /= [ ]
     = bs
        where bs = concat ( map debug (children n))
debug n | otherwise
     = [n]
erroneous ::  Node -> Bool
children ::  Node -> [Node]
```

4.2.2 Estrategias de evaluación y dificultades

En la aproximación de [Naish, 1993], un programa en código funcional que se escriba como un conjunto de ecuaciones mutuamente exclusivas, se puede traducir en Prolog puro mediante un proceso de "aplanamiento" que convierte las funciones en predicados. Una función evaluable con n argumentos, se expresa como un predicado con $n+1$ argumentos. Si los términos contienen varias ocurrencias de funciones evaluables, éstos se sustituyen por conjunciones de átomos. La depuración se aplica a la versión "aplanada" del programa [Naish, 1993].

Para representar el árbol de computación, se utiliza un argumento adicional al que se crea en el "aplanamiento" de la función. Si existe una función evaluable $f(\ldots)$ que se reescribe a una expresión $rhs(\ldots)$, entonces la representación de la computación en el argumento contiene un término de la forma $rewrite(f(\ldots), RHS, PG)$, donde RHS representa el valor de la computación de $rhs(\ldots)$ y PG es *true* si el término se reescribe con éxito en esta etapa [Naish, 1993]. En cierto sentido, el argumento almacena una traza de la computación del término objetivo que permite verificar cada subtérmino invocado con su respectivo valor. En la evaluación "perezosa" de funciones, que selecciona la expresión reducible más externa, se adiciona una función auxiliar para forzar un paso de evaluación cuando ocurre una llamada a función y se introducen predicados para tratar con expresiones que no son evaluadas. En la depuración, se manejan argumentos para dar cuenta tanto de los resultados de la evaluación como de las expresiones no evaluadas en el interior de la computación.

Para reducir la complejidad de las preguntas, se introduce el uso de cuantificadores. Si una expresión no evaluada se presenta en un argumento o en un argumento

[14] Un nodo es equívoco si el término asociado se reescribe de manera incorrecta y sus hijos no.

y el témino resultante, las expresiones se cambian por variables cuantificadas universalmente; esto equivale a tener objetivos Prolog que tienen éxito sin instanciar completamente sus variables. Si la expresión no evaluada ocurre en el resultado y no en los argumentos de la función, se sustituye por variables cuantificadas existencialmente; esto corresponde a funciones que no han sido evaluadas completamente.

Los problemas fundamentales en la depuración declarativa de programas funcionales se pueden resumir en los siguientes puntos:

- Las preguntas son de gran tamaño y complejas, debido al uso de expresiones que involucran estructuras de datos grandes. La expresión $rewrite(f(\ldots), RHS, PG)$ usada de manera recursiva es un ejemplo [Naish, 1993].
- Existen resultados intermedios con expresiones no evaluadas en los argumentos o en los resultados de las llamadas a funciones, debido a la estrategia "perezosa" [Naish, 1993].
- La construcción del árbol genera varias dificultades: consumo de memoria, representación de las expresiones no evaluadas, tipos de datos a utilizar para la representación del árbol, momento de construcción y estrategias para recorrerlo [Naish, 1993; Nilsson, 1994; Nilsson y Fritzson, 1994].
- Cuando el depurador interviene en el proceso de evaluación de una expresión, por ejemplo asignando valores a expresiones no evaluadas, puede verse afectada la propiedad de la transparencia referencial.
- La cantidad de preguntas realizadas al oráculo suele ser muy grande.

4.2.3 Estrategias de solución

Enfoques previos

Almacenar el árbol en un archivo después de la ejecución completa del programa, es costoso y poco útil. En [Naish, 1993] se propone modificar el programa fuente agregando argumentos adicionales, tal como se explicó anteriormente. La depuración se realiza durante la ejecución del programa y se genera un "árbol perezoso por niveles" a medida que ésta avanza; cada nodo es evaluado si se necesita. Para lograr este objetivo, se utiliza una primitiva que no cumple la propiedad de la transparencia referencial y, por supuesto, debe ser construida en otro lenguaje con algunas restricciones para no generar problemas con la portabilidad.

Una alternativa similar para manipular los datos en la depuración es la "estrictificación" (*strictification*), que consiste en ocultar el orden de la evaluación perezosa mediante la reestructuración del árbol de ejecución para obtener así un árbol de resultados y dar al usuario una impresión *estricta* de la ejecución perezosa [Nilsson y

Fritzson, 1994]; el árbol planteado en [Naish, 1993] es un ejemplo. En este enfoque, algunas de las expresiones no evaluadas se reemplazan por los valores que éstas representan, evitando reemplazos que contengan a su vez expresiones no evaluadas y sustituir expresiones que representen estructuras infinitas y computaciones que no terminan.

En resumen, la estrategia de evaluación perezosa, el conservar las propiedades del lenguaje y la construcción del árbol originan dificultades para la construcción del algoritmo de depuración y su aplicación práctica. En [Nilsson, 1994] se presentan algunos enfoques, los cuales difieren en tres aspectos fundamentales: la construcción del algoritmo en un lenguaje que puede ser diferente al del programa que se pretende corregir, la concepción del árbol y el permitir la mezcla de aspectos operacionales con el proceso de depuración.

4.3 Programas lógico funcionales

Las propuestas para la depuración de programas lógico funcionales son pocas y se dividen en dos grupos, de acuerdo con el criterio de observación del comportamiento del programa. En el primero, se realiza un seguimiento "procedural" de la ejecución del programa con base en su semántica operacional y utilizan como estrategia las *cajas de Byrd* [Hanus y Josephs, 1993; Arenas y Gil, 1994] propias de la depuración en el estilo de las implementaciones convencionales de Prolog. Una caja de Byrd es una estructura que representa un "paso" de la computación con cuatro puertos de comunicación –llamada, salida, fallo y rehacer–observables por el usuario, que se componen de manera recursiva y/o secuencial para reflejar el comportamiento de la ejecución del programa y poder determinar los posibles errores que se presentan. Los diferentes tipos de cajas precisan lo que antes hemos llamado "pasos" de manera ambigua. Para el lenguaje ALF con estrategia de "estrechamiento básico innermost con normalización" existen los siguientes tipos: literal, simplificación, rechazo, estrechamiento, reflexión, lado izquierdo, condición y simplificación innermost [Hanus y Josephs, 1993]. En el lenguaje BabLog con estrategia de "estrechamiento perezoso" basado en la demanda, existen los siguientes: ecuaciones, forma normal en cabeza, casos, enlace, unificación perezosa, unificación, aplicación de regla y condición [Arenas y Gil, 1994]. Estos enfoques no se relacionan con la semántica declarativa y quedan al margen de nuestra línea de trabajo.

El lenguaje GAPLog es un lenguaje integrado que se define como una extensión del Prolog. Las funciones se importan de programas externos (funcionales o estructurados) y se utilizan como cajas negras: reciben unos datos y entregan unos resultados. La igualdad se define en un procedimiento externo y se adopta la S-unificación como semántica operacional. Las interpretaciones son conjuntos de átomos y ecuaciones,

cuyos argumentos son términos constructores ground. A cada programa se le asocia el conjunto de todas las ecuaciones ground que generan las funciones. El mecanismo de derivación que se aplica es S-SLD-resolución, en el cual, cada derivación es una secuencia de átomos y restricciones, a partir del objetivo inicial. Los conceptos que fundamentan la depuración declarativa y los algoritmos son similares a los que se plantearon en la sección 4.1.3, con las respectivas modificaciones para manipular la igualdad y los errores en el resultado de la aplicación de una función. La depuración en esta concepción se relaciona directamente con la depuración de Programas Lógicos con Restricciones (CLP), tal como el mismo autor lo refiere [Eklund, 1997].

Escher es un lenguaje integrado visto como una extensión del lenguaje Haskell, su semántica declarativa es una extención de la teoría de tipos de Church y la semántica operacional se basa en la reescritura [Lloyd, 1999]. Una computación de un término objetivo t es una secuencia de términos $t_1, \ldots, t_n$ (traza), en donde t_1 denota el término t, cada t_i se obtiene mediante una regla de reescritura que se aplica al t_{i-1} con la estrategia que selecciona como expresión reducible la más externa a la izquierda y el t_n representa la respuesta. El algoritmo de depuración declarativa para Escher se aplica únicamente a computaciones que sean un síntoma de incorrección[15] para determinar una declaración (regla de reescritura–statement) incorrecta. El algoritmo encuentra un i tal que $t_i \neq t_{i-1}$ y muestra la declaración incorrecta asociada, utilizando como estrategia una variante del método de *divide y consulta* dado en [Shaphiro, 1982], que busca el punto medio de la secuencia de términos, determina la subsecuencia incorrecta y efectúa en ella la búsqueda de manera recursiva con el mismo criterio.

El algoritmo de depuración siempre termina y es correcto y completo. Pero presenta una dificultad para su implementación: un término en una pregunta que se realice al oráculo puede ser muy grande y/o complejo, lo que hace necesarias las siguientes mejoras:

- Computar subobjetivos más pequeños en donde se manifieste el error.
- Utilizar tipos de datos abstractos adecuados al usuario para mostrar subtérminos que involucran funciones ocultas en el interior de las consultas realizadas al oráculo.
- Construir de manera incremental el conocimiento de la interpretación deseada a partir de las respuestas a preguntas anteriores o de versiones previas del programa.

Una descripción detallada de este enfoque se encuentra en [Lloyd, 1998].

[15] Una computación C es un síntoma de incorrección si $t_1 = t_n$ no es válido en la interpretación deseada.

Por último, NUE-Prolog es una extensión de NU-prolog para incluir funciones evaluables, donde cada función se define como un conjunto de ecuaciones mutuamente exclusivas. Una ecuación es una cláusula que tiene una expresión de la forma $s = t$ en su cabeza y donde los predicados pueden tener símbolos de función en sus argumentos. Los programas son "traducidos" a NU-Prolog mediante el proceso de aplanamiento. Los algoritmos para la depuración de programas lógico funcionales resultan de la mezcla de los relacionados con programas lógicos puros [Naish, 1992, 1997] y con programas funcionales [Naish, 1993]. La representación del árbol de demostración permite mezclar código funcional y lógico para generar un árbol de demostración combinado; una descripción del enfoque se encuentra en [Naish y Barbour, 1994].

Un aspecto importante en este enfoque es el tratamiento de respuestas perdidas para programas funcionales y, por supuesto, aplicable a programas integrados. Una respuesta perdida en un programa funcional se debe a que un predicado no está bien definido para todas sus posibles entradas, denominadas llamadas no definidas. Para el diagnóstico de llamadas no definidas, se tienen en cuenta las siguientes características de los lenguajes funcionales: las funciones devuelven al menos un valor, los modos entrada-salida y el flujo de datos son simples, una llamada a función debe emparejar al menos con una ecuación y una llamada no definida siempre indica un error. Para la evaluación perezosa, una llamada no definida que obligue a un término a ser evaluado puede generar un error, llamado *demanda no razonable*. El algoritmo para diagnósticar llamadas no definidas en computaciones perezosas debe tener en cuenta lo siguiente: representar una computación que falla, involucrar preguntas con fórmulas cuantificadas para incluir los fallos y tratar las expresiones que nunca fueron evaluadas debido a una llamada no definida como una llamada que nunca fue evaluada. Una descripción completa se encuentra en [Naish y Barbour, 1994].

4.4 Programas lógico funcionales

Ahora introducimos alguna definiciones básicas sobre el diagnóstico declarativo de programas y adaptadas de [Comini *et al.*, 1995a]. Como semántica operacional consideramos la semántica del conjunto de éxitos. La Figura 4.3 ilustra la siguiente definición.

Definición 4.4.1 *Sea $\mathcal{I}$ la especificación de la semántica deseada de respuestas computadas para $\mathcal{R}$.*

1. *$\mathcal{R}$ es parcialmente correcto con respecto a $\mathcal{I}$, si $\mathcal{O}^{ca}_{\varphi}(\mathcal{R}) \subseteq \mathcal{I}$.*
2. *$\mathcal{R}$ es completo con respecto a $\mathcal{I}$, si $\mathcal{I} \subseteq \mathcal{O}^{ca}_{\varphi}(\mathcal{R})$.*
3. *$\mathcal{R}$ es totalmente correcto con respecto a $\mathcal{I}$, si $\mathcal{O}^{ca}_{\varphi}(\mathcal{R}) = \mathcal{I}$.*

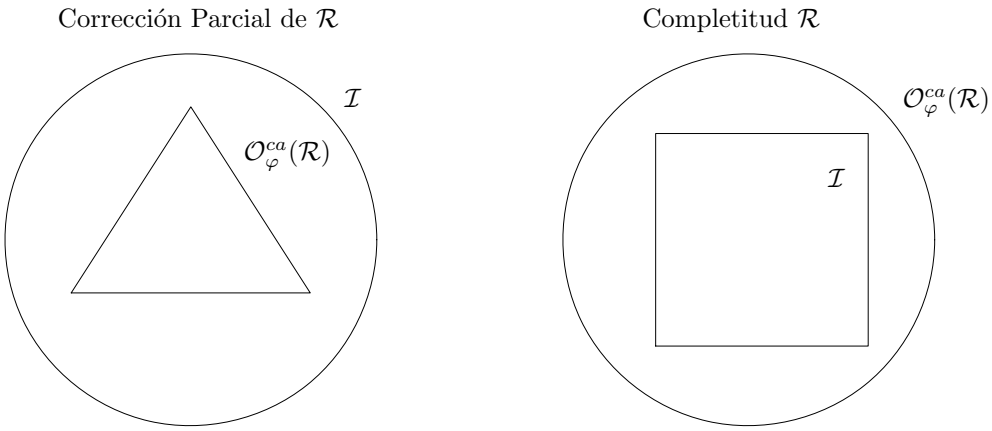


Figura 4.3: Corrección y completitud de $\mathcal{R}$ con respecto a las respuestas computadas

La *semántica deseada* la denotamos simplemente como $\mathcal{I}$, pero no asumimos que $\mathcal{I}$ sea única. Podría darse el caso de que diferentes especificaciones de la semántica deseada se den con diferentes propósitos. En particular, cuando tratamos con la corrección parcial, $\mathcal{I}$ describe una propiedad que debe ser satisfecha por todos los elementos de la semántica $\mathcal{O}^{ca}_{\varphi}(\mathcal{R})$. En otras palabras, $\mathcal{I}$ corresponde a las propiedades esperadas para todos los "resultados" o todos los "comportamientos" de el programa dependiendo de la clase de semántica. Cuando tratamos con la completitud, $\mathcal{I}$ caracteriza un conjunto de elementos que deberían estar en la semántica $\mathcal{O}^{ca}_{\varphi}(\mathcal{R})$, es decir, $\mathcal{I}$ describe algunos resultados o comportamientos esperados de $\mathcal{R}$ [Bueno *et al.*, 1997a].

La semánticas dadas por el conjunto $\mathcal{F}_{\varphi}(\mathcal{R})$ poseen más información que el conjunto $\mathcal{O}^{ca}_{\varphi}(\mathcal{R})$, pues para formarlo hemos descartado aquellas ecuaciones cuya lado derecho posee un símbolo de función definido. En el caso *out*, hemos descartado también aquellas ecuaciones que tienen en su lado derecho el símbolo $\perp$. En cierto sentido podemos decir que mientras en la semántica $\mathcal{O}^{ca}_{\varphi}(\mathcal{R})$, podemos encontrar el conjunto de respuestas computadas por el programa, en $\mathcal{F}^{ca}_{\varphi}(\mathcal{R})$ tenemos las respuestas computadas y adicionalmente una semántica capaz de dar cuenta de computaciones que no terminan, mediante el uso $\perp$. Por último en la semántica $\mathcal{F}_{\varphi}(\mathcal{R})$ tenemos toda la información anterior y adicionalmente la posibilidad de expresar computaciones intermedias.

Para definir una semántica deseada es necesario tener en cuenta que cuando el conjunto referente sea $\mathcal{O}^{ca}_{\varphi}(\mathcal{R})$, la semántica deseada debe ser del mismo estilo y con la información referida a dicha semántica, razón por la cual la llamaremos semántica deseada de respuestas computadas.

Con el uso del $T_{\mathcal{R}}^{\varphi}$ en nuestro marco de investigación, debemos usar una semántica más rica que de cuenta de computaciones intermedias e infinitas. Entonces la semántica deseada se plantea en términos de la semántica $\mathcal{F}_{\varphi}(\mathcal{R})$ y la llamaremos semántica deseada de punto fijo.

Si un programa contiene errores, estos son señalados por los correspondientes *síntomas*. Consideramos como primera semántica de referencia para el diagnóstico la 'semántica deseada del conjunto de éxitos', la cual nos permite establecer la validez de una ecuación atómica por un simple test "ser miembro de", en el estilo de la s-semántica.

Definición 4.4.2 *Sea $\mathcal{I}$ la especificación de la semántica deseada del conjunto de éxitos para $\mathcal{R}$.*

1. *Un* síntoma de incorrección *es una ecuación e tal que $e \in \mathcal{O}_{\varphi}^{ca}(\mathcal{R})$ y $e \notin \mathcal{I}$.*

2. *Un* síntoma de incompletitud *es una ecuación e tal que $e \in \mathcal{I}$ y $e \notin \mathcal{O}_{\varphi}^{ca}(\mathcal{R})$.*

Para el diagnóstico, es necesario considerar una semántica deseada "bien provista" $\mathcal{I}$, que modele tanto éxitos como computaciones en "progreso", y que cumpla las propiedades semánticas de la denotación -formalizadas en la Definición 3.2.3-, es decir, $\mathcal{I}$ debe corresponder a la semántica del punto fijo del programa correcto. En un sistema práctico, esta descripción no la entra manualmente el usuario, esta se infiere automáticamente de un conjunto finito de ecuaciones dadas. Éste es el enfoque que nosotros seguimos para la metodología de diagnóstico abstracto que nosotros presentamos en el Capítulo 6.

Para determinar las reglas defectuosas, damos las siguientes definiciones.

Definición 4.4.3 *Sea $\mathcal{I}$ la especificación de la semántica deseada de punto fijo para $\mathcal{R}$. Si existe una ecuación $e \in T_{\{r\}}^{\varphi}(\mathcal{I})$ y $e \notin \mathcal{I}$, entonces la regla $r \in \mathcal{R}$ es incorrecta sobre e.*

Por consiguiente, la incorrección de la regla r es señalada por una simple transformación de la semántica deseada $\mathcal{I}$.

Definición 4.4.4 *Sea $\mathcal{I}$ la especificación de la semántica deseada de punto fijo para $\mathcal{R}$. Una ecuación e es no cubierta si $e \in \mathcal{I}$ y $e \notin T_{\mathcal{R}}^{\varphi}(\mathcal{I})$.*

Por la definición anterior, una ecuación e es no–cubierta (descubierta) si no puede ser derivada por cualquier regla del programa que usa la semántica deseada de punto fijo.

Proposición 4.4.5 *Si no existen reglas incorrectas en $\mathcal{R}$ con respecto a la especificación de la semántica deseada de punto fijo, entonces $\mathcal{R}$ es parcialmente correcto con respecto a la semántica deseada de respuestas computadas.*

Demostración. Sea $\mathcal{I}_{\mathcal{F}}$ la especificación de la semántica deseada de punto fijo y sea $\mathcal{I}_{ca}$ la especificación de la semántica deseada del conjunto de éxitos de $\mathcal{R}$. Por hipótesis no hay ninguna regla incorrecta en $\mathcal{R}$ con respecto a $\mathcal{I}_{\mathcal{F}}$, entonces para todo $r \in R$ tal que $e \in T^{\varphi}_{\{r\}}(\mathcal{I}_{\mathcal{F}})$, $e \in \mathcal{I}_{\mathcal{F}}$ (Definición 4.4.3). Entonces $T^{\varphi}_{R}(\mathcal{I}_{\mathcal{F}}) \subseteq \mathcal{I}_{\mathcal{F}}$, por consiguiente $\mathcal{I}_{\mathcal{F}}$ es un pre-punto fijo de $T^{\varphi}_{R}(\mathcal{I}_{\mathcal{F}})$, por la continuidad de T^{φ}_{R}, se concluye que $T^{\varphi}_{R}(\mathcal{I}_{\mathcal{F}}) \uparrow \omega \subseteq \mathcal{I}_{\mathcal{F}}$.

Dado que $\mathcal{O}^{ca}_{\varphi}(\mathcal{R}) \subseteq \mathcal{F}^{ca}_{\varphi}(\mathcal{R}) \subseteq T^{\varphi}_{R}(\mathcal{I}_{\mathcal{F}}) \uparrow \omega$ entonces $\mathcal{O}^{ca}_{\varphi}(\mathcal{R}) \subseteq \mathcal{I}_{\mathcal{F}}$. Ahora bien, como $\mathcal{O}^{ca}_{\varphi}(\mathcal{R}) \cap inprogress(\mathcal{I}_{\mathcal{F}}) = \emptyset$, entonces para concluir la demostración tenemos que $\mathcal{O}^{ca}_{\varphi}(\mathcal{R}) \subseteq \mathcal{I}_{\mathcal{F}} - inprogress(\mathcal{I}_{\mathcal{F}})) = \mathcal{I}_{ca}$. □

La proposición 4.4.5 muestra una metodología simple para probar la corrección parcial. La completitud es más difícil: algunas incompletitudes no pueden ser detectadas comparando la especificación de la semántica deseada de punto fijo $\mathcal{I}$ y $T^{\varphi}_{\mathcal{R}}(\mathcal{I})$. Es decir, la ausencia de ecuaciones no cubiertas no nos permite afirmar que el programa es completo. consideremos el siguiente contraejemplo:

Ejemplo 19 *Consideremos la estrategia $\varphi = inn$, el programa*

$$\mathcal{R} = \{f(x) \rightarrow \mathtt{a} \Leftarrow \mathtt{f(x) = a}\}$$

y la especificación

$$\mathcal{I} = \{\mathtt{a = a},\ \mathtt{f(x) = f(x)},\ \mathtt{f(x) = a}\}$$

Entonces $\mathcal{R}$ no es completo debido a que $\mathcal{I} \nsubseteq \mathcal{O}^{ca}_{inn}(\mathcal{R}) = \{\}$. Sin embargo

$$\mathcal{I} \subseteq T^{inn}_{\mathcal{R}}(\mathcal{I}) = \{\ \mathtt{a = a},\ \mathtt{f(x) = f(x)},\ \mathtt{f(x) = a}\}$$

y no existen ecuaciones descubiertas.

El problema se debe a la carencia de un único punto fijo para el operador $T^{\varphi}_{\mathcal{R}}$ [Comini *et al.*, 1995a], como lo muestra el siguiente resultado:

Proposición 4.4.6 *Si $T^{\varphi}_{\mathcal{R}}$ tiene un único punto fijo y no existen ecuaciones no cubiertas, entonces $\mathcal{R}$ es completo con respecto a $\mathcal{I}$.*

Demostración. Sea $\mathcal{R}$ un programa, $\mathcal{I}_{\mathcal{F}}$ un semántica deseada de punto fijo y sea $\mathcal{I}_{ca}$ la correspondiente semántica deseada del conjunto de éxitos para $\mathcal{R}$.

Si $T^{\varphi}_{\mathcal{R}}$ tiene un único punto fijo y no existen ecuaciones no cubiertas con respecto a $\mathcal{I}_{\mathcal{F}}$. Entonces toda ecuación e tal que si $e \in \mathcal{I}$ entonces $e \in T^{\varphi}_{\mathcal{R}}(\mathcal{I}_{\mathcal{F}})$, es decir $\mathcal{I}_{\mathcal{F}} \subseteq T^{\varphi}_{\mathcal{R}}(\mathcal{I}_{\mathcal{F}})$, luego $\mathcal{I}_{\mathcal{F}}$ es un post punto fijo. Como existe un único punto fijo, y por la continuidad del operador, $\mathcal{I}_{\mathcal{F}} \subseteq T^{\varphi}_{\mathcal{R}} \uparrow \omega$ y como $\mathcal{I}_{ca} \subseteq \mathcal{I}_{\mathcal{F}}$ por lo tanto $\mathcal{I}_{ca} \subseteq T^{\varphi}_{\mathcal{R}} \uparrow \omega$

Se sabe que $\mathcal{O}^{ca}_{\varphi}(\mathcal{R}) \subseteq \mathcal{F}_{\varphi}(\mathcal{R})$. Dado que en $\mathcal{I}_{ca}$ no existen ecuaciones con símbolos de función definido en su lado derecho, entonces $\mathcal{I}_{ca} \subseteq \mathcal{F}^{ca}_{\varphi}(\mathcal{R})$ y por la construcción de $\mathcal{I}_{ca}$, se cumple que $I_{ca} \subseteq \mathcal{O}^{ca}_{\varphi}(\mathcal{R})$ ya que $I_{ca} \subseteq \mathcal{F}^{ca}_{\varphi}(\mathcal{R}) - inprogress(\mathcal{F}^{ca}_{\varphi}(\mathcal{R}))$, Teorema 3.3.4. □

Capítulo 5

Semántica abstracta

En este capítulo desarrollamos un marco para la depuración de programas lógico funcionales, basado en la idea de construir aproximaciones de la semántica deseada del programa. Formalizamos un esquema de depuración simple, uniforme y potente que admite instancias para diferentes relaciones de *narrowing* (impaciente y perezoso), y permite estudiar distintos tipos de propiedades (relacionadas con el conjunto de éxitos) de manera correcta. Calculamos una aproximación, computable en tiempo finito, que nos permite garantizar que el análisis termina, y es correcto y completo bajo ciertas condiciones. Para ello, determinamos una abstracción del sistema de reescritura de términos, con base en un *grafo de dependencia funcionales*, a fin de obtener una forma simplificada del programa para la cual se asegura la terminación. A continuación, se hará uso de un narrowing aproximado para llevar a cabo la reducción abstracta de un conjunto de ecuaciones.

5.1 Interpretación abstracta

Los análisis de flujo de datos son un componente esencial de muchas de las herramientas de la programación actual. Sin embargo, los análisis pueden ser muy complejos y, por tanto, difíciles de diseñar y de demostrar su corrección. La teoría de la interpretación abstracta [Cousot y Cousot, 1977] permite formalizar la relación entre análisis y semántica. La *teoría de la interpretación abstracta* provee un marco formal para desarrollar herramientas avanzadas para el análisis de flujo de datos, formalizando la idea de 'computación aproximada' la cual una computación se realiza con "descripciones de datos" en lugar de los datos mismos. Los operadores semánticos son reemplazados entonces por operadores abstractos que aproximan de forma 'segura' los operadores estándar. Partiendo de la semántica de punto fijo introducida en los Capítulo 3, en este capítulo desarrollamos una semántica abstracta que aproxima el comportamiento

observable del SRTC dado y es adecuada para modelar análisis de flujo de datos, tal como el análisis de insatisfacibilidad de un conjunto de ecuaciones o cualquier análisis que este basado en el conjunto de éxitos de un programa. Asumimos el marco de interpretación abstracta para el análisis de la insatisfacibilidad ecuacional tal como está definido en [Alpuente *et al.*, 1995]. Otro enfoque para construir un sistema de reescritura de términos abstracto se encuentra en [Bert *et al.*, 1993]. En este capítulo mostramos una aproximación diferente de la semántica de punto fijo dada en una de las secciones previas siguiendo un enfoque similar al que aparece en [Bert *et al.*, 1993]. Del mismo modo que en la semántica concreta, nosotros probamos que los resultados para las técnicas de aproximación son paramétricos con respecto a la estrategia de *narrowing* empleada $\varphi \in \{inn, out\}$. Comenzamos recordando algunas de las definiciones básicas sobre dominios abstractos y los operadores abstractos asociados; para una comprensión más amplia de nuestro enfoque ver [Alpuente *et al.*, 1995, 1996, 2001a,c]. Describimos un operador de consecuencias inmediatas nuevo $T_{\mathcal{R}}^{\sharp}$ capaz de aproximar el operador $T_{\mathcal{R}}$, y como consecuencia, la semántica de punto fijo abstracta, $\mathcal{F}^{\sharp}(\mathcal{R})$. En lo que sigue denotamos mediante $O^{\sharp}$ el objeto abstracto correspondiente a un objeto concreto O.

5.1.1 Dominios y operadores abstractos

Siguiendo una aproximación similar a la presentada en [Alpuente *et al.*, 1995; Codish *et al.*, 1994], definimos la *relación de aproximación* en términos de una *función de concretización*, que asigna a los elementos del dominio no estándar aquellos elementos del dominio estándar que están siendo descritos.

Definición 5.1.1 *Una* descripción γ *es una función anti–monótona con respecto a la relación de inclusión entre conjuntos de un* dominio abstracto $(D, \leq)$ *(un orden parcial entre conjuntos-poset) con* 2^E *donde* E *es un* dominio concreto $(E, \leq)$*(un poset).*

En la Definición 5.1.1, exigimos que la función de concretización γ sea anti-monótona. El motivo de esta restricción es simplemente reflejar la idea de que, cuanto menor es un elemento del dominio abstracto (menos preciso), mayor es el conjunto de elementos del dominio concreto que éste describe. Formalizamos a continuación nuestra noción de *aproximación*, basada en la función de concretización [Vidal, 1996].

Cuando $E = Eqn$, $E = Sub$ o $E = \tau(\mathcal{C} \cup \mathcal{V})$ la descripción se denomina *descripción de ecuación* o *descripción de sustitución* o *descripción de término* respectivamente. La correspondencia entre el dominio abstracto y el concreto se establece a través de una función de 'concretización' $\gamma : D \rightarrow 2^E$. Decimos que d *aproxima a* e, y lo escribimos $d \propto e$, sii $e \in \gamma(d)$. La relación de aproximación puede ser ampliada de manera

natural a relaciones y a dominios producto [Alpuente *et al.*, 1995].

Con el propósito de describir las sustituciones de respuesta computada para un objetivo dado, a continuación introducimos las sustituciones abstractas En nuestro enfoque, las ecuaciones abstractas y sustituciones abstractas corresponden a denotaciones de programas abstractos y propiedades observables abstractas, respectivamente. Los dominios para ecuaciones y sustituciones se definen como sigue. En donde $\mathcal{H}_V = (\tau(\Sigma \cup V), \leq)$ denota el dominio estándar de (clases de equivalencia de) términos ordenados por el orden parcial estándar $\leq$ inducida por el preorden sobre términos dado por la relación "es más general que".

Definición 5.1.2 (Universo Herbrand Abstracto) *Sea $\sharp$ un símbolo irreducible, donde $\sharp \notin \Sigma$. Decimos que $\mathcal{H}_V^\sharp = (\tau(\Sigma \cup V \cup \{\sharp\}), \preceq)$ es el dominio de términos sobre la signatura extendida con $\sharp$, donde el orden parcial $\preceq$ se define como sigue:*

i) $\forall t \in \mathcal{H}_V^\sharp . \sharp \preceq t \wedge t \preceq t$ y

ii) $\forall s_1, \ldots, s_n, s'_1, \ldots, s'_n \in \mathcal{H}_V^\sharp, \forall f/n \in \Sigma.\ s'_1 \preceq s_1 \wedge \ldots \wedge s'_n \preceq s_n \Rightarrow f(s'_1, \ldots, s'_n) \preceq f(s_1, \ldots, s_n)$.

Extendemos este orden a ecuaciones: $s' = t' \preceq s = t$ sii $s' \preceq s \wedge t' \preceq t$, y a un conjunto de ecuaciones S, S':

i) $S' \preceq S$ *sii* $\forall e' \in S'. \exists e \in S$ *tal que* $e' \preceq e$. *Observe que* $S' \preceq \{true\} \Rightarrow S' \equiv \{true\}$.

ii) $S' \sqsubseteq S$ *sii* $(S' \preceq S) \wedge (S \preceq S' \Rightarrow S' \subseteq S)$.

Intuitivamente, $S' \sqsubseteq S$ significa que S' contiene menos información que S o, si tienen la misma información, entonces S' expresa que lo hace con menos elementos.

En términos generales, hemos introducido un símbolo especial $\sharp$ en el dominio abstracto para representar cualquier término concreto. Desde el punto de vista lógico, $\sharp$ representa una variable cuantificada existencialmente [Alpuente *et al.*, 1995; Maher, 1988, 1990] y, desde el punto de vista operacional, se puede considerar como una aproximación de la variable "anónima" del Prolog. Definimos $[\![S]\!] = S'$, donde la n-tupla de ocurrencias de $\sharp$ en S se reemplaza en S' por una n-tupla de variables frescas cuantificadas existencialmente.

Definición 5.1.3 *Una sustitución abstracta es un conjunto de la forma $\{x_1/t_1, \ldots, x_n/t_n\}$ donde, para cada $i = 1, \ldots, n$, x_i es una variable distinta de V y no aparece en ninguno de los términos $t_1, \ldots, t_n$ y $t_i \in \tau(\Sigma \cup V \cup \{\sharp\})$. El orden sobre sustituciones abstractas está dado por la implicación lógica: Sean $\theta, \kappa \in Sub^\sharp, \kappa \preceq \theta$ sii $[\![\widehat{\theta}]\!] \Rightarrow [\![\widehat{\kappa}]\!]$.*

Introducimos ahora las descripciones de términos, sustituciones y ecuaciones.

Definición 5.1.4 *Sea* $\mathcal{H}_V = (\tau(\Sigma \cup V), \leq)$ *y* $\mathcal{H}_V^\sharp = (\tau(\Sigma \cup V \cup \{\sharp\}), \preceq)$. *Una* descripción de términos *es* $\langle \mathcal{H}_V^\sharp, \gamma, \mathcal{H}_V \rangle$ *donde* $\gamma : \mathcal{H}_V^\sharp \to 2^{\mathcal{H}_V}$ *es definida por:*

$$\gamma(t') = \{t \in \mathcal{H}_V | t' \preceq t\}.$$

Sea Eqn el conjunto de los conjuntos finitos de ecuaciones en $\tau(\Sigma \cup V)$ *y* $Eqn^\sharp$ *el conjunto de los conjuntos finitos de ecuaciones en* $\tau(\Sigma \cup V \cup \{\sharp\})$. *Una* descripción de ecuación *es* $\langle (Eqn^\sharp, \sqsubseteq), \gamma, (Eqn, \leq) \rangle$, *donde* $\gamma : Eqn^\sharp \to 2^{Eqn}$ *está definida por:*

$$\gamma(g') = \{g \in Eqn | g' \sqsubseteq g \text{ y } g \text{ no está cuantificada}\}.$$

Sea Sub el conjunto de sustituciones definidas en $\tau(\Sigma \cup V)$ *y* $Sub^\sharp$ *el conjunto de sustituciones en* $\tau(\Sigma \cup V \cup \{\sharp\})$. *Una* descripción de sustitución *es* $\langle (Sub^\sharp, \preceq), \gamma, (Sub, \leq) \rangle$, *donde* $\gamma : Sub^\sharp \to 2^{Sub}$ *está definida por:*

$$\gamma(\kappa) = \{\theta \in Sub | \kappa \preceq \theta\}.$$

5.1.2 Unificación abstracta

Para realizar computaciones sobre los dominios abstractos, recordamos la noción de *unificación abstracta* [Alpuente *et al.*, 1995]. El unificador abstracto más general para nuestro método es muy simple. Intuitivamente, obtiene la forma resuelta de un conjunto de ecuaciones con variables (posiblemente) cuantificadas existencialmente. Definimos el unificador abstracto más general para un conjunto de ecuaciones $S' \in Eqn^\sharp$ de la siguiente forma. Primero, reemplazamos todas ocurrencias de $\sharp$ en S' por variables frescas cuantificadas existencialmente. A continuación, computamos una forma resuelta del conjunto de ecuaciones cuantificadas resultante y, finalmente, reemplazamos las variables cuantificadas existencialmente de nuevo por $\sharp$. Formalmente:

Definición 5.1.5 (unificador abstracto más general)
Sea $\exists y_1 \ldots y_n . S = solve([\![S']\!])$ *y* $\kappa = \{y_1/\sharp, \ldots, y_n/\sharp\}$. *Entonces* $\widehat{mgu}^\sharp(S') = S\kappa$.

La siguiente proposición justifica el hecho de que utilicemos el calificativo 'más general' también en el caso abstracto (ver [Alpuente *et al.*, 1995] para más detalles).

Proposición 5.1.6 [Alpuente *et al.*, 1995] $\forall \theta \in unif([\![S]\!]).\ mgu^\sharp(S) \preceq \theta$.

5.1.3 Narrowing abstracto con estrategia

La relación de *narrowing* abstracto se define, en general, con respecto a un programa abstracto $\mathcal{R}_\sharp$ (que, en ocasiones, puede coincidir con el programa concreto $\mathcal{R}$). La decisión de abstraer el programa proviene de la necesidad de asegurar la terminación del análisis, en nuestro caso el proceso de depuración. Si para un objetivo dado, las computaciones sobre el programa concreto siempre son finitas, el programa abstracto puede coincidir con el concreto. En la práctica, existen varios métodos para estudiar

la terminación de *narrowing* respecto a un programa como, por ejemplo, los *grafos de dependencias funcionales* usados en [Dershowitz y Sivakumar, 1987], o los *grafos de términos* (dependientes del objetivo) usados en [Chabin y Réty, 1991; Réty *et al.*, 1985]. Posteriormente presentamos una técnica general, independiente del objetivo, tal como hemos planteado nuestro esquema para la depuración, y basada en la noción de grafo de "prevención de bucles" para obtener un programa abstracto cuyas computaciones son siempre finitas.

Definimos el *sistema de transición abstracto* que formaliza la relación de *narrowing abstracto*. El cálculo abstracto se define sobre los objetivos abstractos, y realiza las computaciones con respecto a un programa abstracto de $\mathcal{R}^\sharp$. El cálculo es equivalente al introducido en la Definición 2.4.1, reemplazando los dominios y operadores concretos por sus versiones abstractas. Al igual que en el caso concreto, la relación de *narrowing* abstracto se define de manera genérica respecto a una estrategia φ. Denotamos por $\mathcal{R}^\sharp_+$ a la correspondiente extensión del programa abstracto $\mathcal{R}^\sharp$ con las reglas para tratar la igualdad sintáctica, tal como lo hicimos en el caso concreto.

Definición 5.1.7 (narrowing abstracto con estrategia $\leadsto_{\sharp\varphi}$) *Sea $\mathcal{R}^\sharp \in I\!R_\varphi$ un programa abstracto, definimos la relación de narrowing condicional con estrategia φ, $\varphi \in \{inn, out\}$, como la menor relación $\leadsto_{\sharp\varphi}$ que satisface:*

$$\frac{u = \varphi(g) \ \wedge \ r \equiv (\lambda \to \rho \Leftarrow C) \ll \mathcal{R}^\sharp_+ \ \wedge \ \sigma = mgu^\sharp(\{g_{|u} = \lambda\})}{g \overset{\sigma}{\leadsto}_{\sharp\varphi} (C, g[\rho]_u)\sigma}$$

5.2 Terminación del análisis

En esta sección, pasamos a considerar el problema de garantizar que los análisis sean finitos, y las correspondientes consecuencias sobre el proceso de depuración. Esencialmente, dado un programa concreto $\mathcal{R}$, la terminación del análisis se asegura construyendo un programa aproximado $\mathcal{R}_\sharp$ que cumpla las dos condiciones siguientes:

- el programa abstracto es una aproximación correcta del programa concreto, (*corrección*); y
- toda derivación de *narrowing* abstracto usando las reglas de $\mathcal{R}_\sharp$ termina en un número finito de pasos (*terminación*).

En el siguiente apartado, se define un método general de abstracción de programas que, preservando la noción de aproximación, garantiza la terminación de las derivaciones aproximadas.

5.2.1 Abstracción de programas

En la literatura se proponen dos tipos de grafos para estudiar la terminación de un programa (ver, por ejemplo, [Bol, 1993]):

- *grafos de detección de bucles*: siempre que una derivación tiene asociado un bucle en el grafo, dicha derivación es infinita;
- *grafos de prevención de bucles*: toda derivación infinita tiene un bucle asociado en el grafo.

En ambos casos, el grafo se construye a partir del programa considerado y, dependiendo de tipo de grafo elegido, del propio objetivo a resolver. La información del grafo se puede usar de forma dinámica –en tiempo de ejecución– o de forma estática –en tiempo de compilación–.

El primer tipo de grafos sólo ayuda a eliminar algunas derivaciones infinitas, pero existe el riesgo de seguir explorando un espacio de búsqueda infinito (algunas derivaciones infinitas podrían no ser detectadas). Obviamente, este tipo de grafos no es interesante con respecto al análisis, ya que la terminación del mismo debe estar garantizada en cualquier caso. El segundo tipo, los grafos de prevención de bucles, pueden provocar la eliminación innecesaria de derivaciones finitas (con la correspondiente pérdida de precisión) pero, en cambio, se garantiza que todas las derivaciones infinitas son eliminadas. Este segundo tipo sí resulta útil para garantizar la terminación del análisis.

Nuestra noción de programa abstracto es paramétrica con respecto a un grafo de prevención de bucles estático, es decir un grafo finito de dependencias entre términos que se construye a partir del programa y es independiente del objetivo. Este grafo permite reconocer aquellas derivaciones que con toda seguridad terminan. Nuestro propósito es usarlo para transformar un espacio de búsqueda posiblemente infinito en uno finito (aproximado). Con la información del grafo, generamos una versión compilada del programa original (programa abstracto), que aún produce un conjunto completo de respuestas aproximadas pero que, en ningún caso, da lugar a una derivación infinita. La siguiente definición formaliza nuestra noción de grafo de prevención de bucles. Dos instancias diferentes pueden ser encontradas en [Alpuente *et al.*, 1995, 1996].

Definición 5.2.1 (grafo de prevención de bucles) *[Alpuente* et al.*, 1996] Un* grafo de prevención de bucles *es un grafo $\mathcal{G}_\mathcal{R}$ asociado con un programa $\mathcal{R}$ (es decir, una relación que consiste en un conjunto finito de pares de términos) tal que:*

(1) El cierre transitivo $\mathcal{G}^+_\mathcal{R}$ es decidible y

(2) Existe una función $\overset{\circ}{t} = t'$ que asigna a un término t algún nodo t' en $\mathcal{G}_\mathcal{R}$. Si existe una secuencia infinita:

$g_0 \overset{\theta_0}{\leadsto}_\varphi g_1 \overset{\theta_1}{\leadsto}_\varphi \ldots$

entonces $\exists i \geq 0.\ \langle \overset{\circ}{t}_i, \overset{\circ}{t}_i \rangle \in \mathcal{G}^+_\mathcal{R},$ *donde* $t_i = g_{i|u}, \wedge u \in \overline{O}(g_i)$.

En adelante diremos que $\langle \overset{\circ}{t}_i, \overset{\circ}{t}_i \rangle$ *es un 'bucle' de* $\mathcal{G}_\mathcal{R}$.

Informalmente, el grafo de prevención de bucles definido garantiza que, si existe alguna derivación infinita de *narrowing*, entonces alguno de los *redexes* seleccionados en dicha derivación tiene asociado un término en el grafo que está contenido en un bucle. Un grafo de prevención de bucles se puede ver como un tipo de "oráculo", cuya utilidad para demostrar la terminación de las derivaciones de *narrowing* se establece en el siguiente lema.

Lema 5.2.2 *Sea* $\mathcal{R}$ *un SRTC y* $\mathcal{G}_\mathcal{R}$ *un grafo de prevención de bucles para* $\mathcal{R}$. *Si* $\mathcal{G}_\mathcal{R}$ *no contiene bucles, entonces toda derivación de narrowing para* $\mathcal{R}$ *termina.*

Demostración. Inmediata a partir de la Definición 5.2.1 de grafo de prevención de bucles. □

Ejemplo 20 *Consideremos el programa* $\mathcal{R}$

$$\begin{array}{rcl} \{\mathtt{g(0)} & \rightarrow & \mathtt{0} \\ \mathtt{g(c(x))} & \rightarrow & \mathtt{c(g(x))} \\ \mathtt{f(0)} & \rightarrow & \mathtt{0} \\ \mathtt{f(c(x))} & \rightarrow & \mathtt{x} \Leftarrow \mathtt{g(c(x))} = \mathtt{c(x)} \end{array}$$

Dada la función (parcial) $\overset{\circ}{t}$ *tal que* $\overset{\circ}{\mathtt{t}} = \mathtt{t'}$ *asigna a un término* $\mathtt{t} \in \tau(\Sigma \cup \mathcal{V})$ *algún nodo* t' *en el grafo tal que* $\mathtt{t'}$ *unifica con* $\mathtt{t}$, *si tal nodo* t' *existe.*

Note que, el grafo definido por

$$\mathcal{G}_\mathcal{R} = \{\langle \mathtt{g(x)}, \mathtt{g(x)} \rangle\}$$

es un grafo de prevención de bucles, ya que cualquier derivación infinita, usando $\mathcal{R}$, *contiene necesariamente algún término que unifique con* $\mathtt{g(x)}$.

Muchos de los trabajos sobre detección y prevención de bucles consideran la aplicación de grafos de detección de bucles en tiempo de ejecución. Los grafos de prevención de bucles estáticos, sin embargo, no han sido estudiados con tanta atención.

En el caso de los programas funcionales, existen varias aproximaciones que encontramos relacionadas. [Chabin y Réty, 1991; Réty *et al.*, 1985] consideran un grafo de términos que permite la detección de algunos bucles en el árbol de búsqueda que no contribuyen a ninguna solución. Los grafos se construyen usando la información tanto de las ecuaciones a ser reducidas, como de las reglas de programa usadas por *narrowing*.

En el caso de los programas lógicos, la noción estándar de grafo de dependencias entre los símbolos de predicado de un programa lógico [Lloyd, 1987a] se puede ver

también como un tipo de grafo de detección de bucles. Una extensión de estos grafos a un dominio de "llamadas" es la noción de grafo-U (grafo de Unificación, *U-graph* [Wang y Shyamasundar, 1990]), el cual tiene en cuenta además la unificabilidad entre los átomos del programa.

A continuación mostramos una posible instancia de la Definición 5.2.1, siguiendo una idea comparable a la de los grafos-U de los programas lógicos (convenientemente extendida para tratar el anidamiento de funciones). Primero, necesitamos la siguiente definición auxiliar $\lfloor t \rfloor$, que reemplaza inductivamente cualquier subtérmino de t, cuyo símbolo de función más externo no sea constructor, por una variable nueva.

Definición 5.2.3 *Dado un término t, la función $\lfloor t \rfloor$ se define como sigue:*

$$\lfloor t \rfloor = \begin{cases} c(\lfloor t_1 \rfloor, \ldots, \lfloor t_k \rfloor) & \text{si } t = c(t_1, \ldots, t_k) \text{ y } c \in \mathcal{C} \\ y & \text{en otro caso, donde } y \text{ es una variable nueva.} \end{cases}$$

La siguiente definición formaliza un caso particular de grafo de dependencias entre términos $\mathcal{I}_\mathcal{R}$, "inducido" por un SRTC $\mathcal{R}$. Posteriormente, demostramos que $\mathcal{I}_\mathcal{R}$ es un grafo de prevención de bucles (i.e. cumple las condiciones de la Definición 5.2.1). Otro posible grafo de prevención de bucles diferente (pero menos preciso) se puede encontrar en [Alpuente *et al.*, 1995].

Definición 5.2.4 (grafo de dependencias $\mathcal{I}_\mathcal{R}$) *Sea $\mathcal{R}$ un SRTC. La siguiente transformación define un grafo dirigido $\mathcal{I}_\mathcal{R}$ de dependencias entre términos inducido por $\mathcal{R}$. Definimos la función auxiliar: $\bar{t} = f(\lfloor t_1 \rfloor, \ldots, \lfloor t_n \rfloor)$ si $t = f(t_1, \ldots, t_n)$. Para construir $\mathcal{I}_\mathcal{R}$, el algoritmo comienza con un estado $\langle \mathcal{R}, \rangle$ y aplica las siguientes reglas de transición mientras éstas añadan nuevos arcos*[1]*:*

$$(1) \quad \frac{r \equiv (\lambda \to \rho \Leftarrow C) \ll \mathcal{R}}{\langle \mathcal{R}, \mathcal{I}_\mathcal{R} \rangle \;\longmapsto\; \langle \mathcal{R} - \{r\}, \mathcal{I}_\mathcal{R} \cup \{\lambda \xrightarrow{\mathcal{R}} \bar{t} \mid (t = \rho_{|u} \wedge u \in \bar{O}(\rho)) \vee (t = e_{|u} \wedge e \in C \wedge u \in (\bar{O}(e) - \{\Lambda\}))\}\rangle}$$

$$(2) \quad \frac{(\lambda \xrightarrow{\mathcal{R}} \rho) \in \mathcal{I}_\mathcal{R} \;\wedge\; (\lambda' \xrightarrow{\mathcal{R}} \rho') \in \mathcal{I}_\mathcal{R} \;\wedge\; \rho \stackrel{?}{=} \lambda'}{\langle \mathcal{R}, \mathcal{I}_\mathcal{R} \rangle \;\longmapsto\; \langle \mathcal{R}, \mathcal{I}_\mathcal{R} \cup \{\rho \xrightarrow{u} \lambda'\}\rangle}$$

Dado un término $t \in \tau(\Sigma \cup V)$, definimos la función $\overset{\circ}{t}$ que asocia a t un nodo t' en el grafo, como sigue: $\overset{\circ}{t} = t'$ si
1) t' aparece en la parte izquierda de un arco $\xrightarrow{\mathcal{R}}$, y
2) t' unifica con $\bar{t}$.
En el caso de que haya más de un término que cumpla las dos condiciones anteriores, se selecciona aquél que posea un camino en el grafo $\mathcal{I}_\mathcal{R}$ de mayor longitud (si no hay ninguno que cumpla las condiciones, la función está indefinida para t).

[1] Los nodos (términos) del grafo se consideran módulo renombramiento de variables.

La terminación de este cálculo está asegurada, ya que el número de términos que aparecen en las reglas de $\mathcal{R}$ es finito. Informalmente, el algoritmo procede como sigue. Para cada regla $(\lambda \to \rho \Leftarrow C)$ de $\mathcal{R}$ y para cada término $f(t_1, \ldots, t_n)$ que aparezca en ρ o en C, la regla (1) añade un arco $\lambda \stackrel{\mathcal{R}}{\to} f(\lfloor t_1 \rfloor, \ldots, \lfloor t_n \rfloor)$ a $\mathcal{I}_\mathcal{R}$. La regla (2) añade un arco $\rho \stackrel{u}{\to} \lambda'$ entre la parte derecha ρ de un arco $\lambda \stackrel{\mathcal{R}}{\to} \rho$ de $\mathcal{I}_\mathcal{R}$ y la parte izquierda λ' de cada regla $\lambda' \stackrel{\mathcal{R}}{\to} \rho'$ con la que ρ unifica sintácticamente. En lo que sigue, $\to^*$ denota un camino en el grafo formado indistintamente por arcos $\stackrel{\mathcal{R}}{\to}$ ó $\stackrel{u}{\to}$. Por supuesto, teniendo en cuenta una estrategia φ particular, sería posible definir instancias más precisas del grafo, ya que no todos los subtérminos de las partes derechas de las cabezas y de las condiciones de las reglas pueden ser explotados por una determinada estrategia.

El grafo de términos $\mathcal{I}_\mathcal{R}$ es equivalente al grafo de símbolos más externos definido en [Alpuente *et al.*, 1995], cuando la función $\bar{t}$ se define como sigue: $\bar{t} = f$ si $t \equiv f(t_1, \ldots, t_n)$ (la función $\mathring{t}$ se define de idéntica forma). En general, $\mathcal{I}_\mathcal{R}$ permite representaciones más precisas de la estructura recursiva de un programa. Nuestra noción de grafo de prevención de bucles es similar a la de los grafos-U [Wang y Shyamasundar, 1990] de la programación lógica pura, en cuanto que éstos contienen un nodo por cada átomo (cada llamada) del programa y usan dos tipos de arcos: arcos de cláusulas y arcos de unificación. Existe un arco de cláusula de un átomo H a un átomo B_i si existe una cláusula en el programa de la forma $H \leftarrow B_1, \ldots, B_i, \ldots, B_n$, $n \geq 1$. Existe un arco de unificación entre un átomo B y un átomo H si B es un átomo del cuerpo de una cláusula, y unifica (módulo renombramiento) con la cabeza H de otra (o la misma) cláusula.

La siguiente proposición muestra que el grafo $\mathcal{I}_\mathcal{R}$ de dependencias entre términos inducido es, de hecho, un grafo de prevención de bucles.

Proposición 5.2.5 *[Alpuente* et al.*, 1996] Sea $\mathcal{R}$ un SRTC e $\mathcal{I}_\mathcal{R}$ el grafo de dependencias entre términos inducido por $\mathcal{R}$. Entonces, $\mathcal{I}_\mathcal{R}$ es un grafo de prevención de bucles para $\mathcal{R}$.*

Ejemplo 21 *Consideremos de nuevo el programa $\mathcal{R}$ del ejemplo 20*

$$\begin{array}{rcl} \{\mathtt{g(0)} & \to & \mathtt{0} \\ \mathtt{g(c(x))} & \to & \mathtt{c(g(x))} \\ \mathtt{f(0)} & \to & \mathtt{0} \\ \mathtt{f(c(x))} & \to & \mathtt{x} \Leftarrow \mathtt{g(c(x))} = \mathtt{c(x)} \end{array}$$

El correspondiente grafo de dependencias entre términos [Alpuente et al.*, 1996] está dado por el grafo $\mathcal{G}_\mathcal{R}$ que se muestra en la Figura 5.1.*

Se debe notar que existen dos bucles en el grafo, concretamente $\{\langle \mathtt{g(x)}, \mathtt{g(x)} \rangle$, $\langle \mathtt{g(c(x))}, \mathtt{g(c(x))} \rangle\}$.

A continuación, se describe un método para obtener un programa abstracto (compilado) a partir de la información recogida por un grafo de prevención de bucles. El

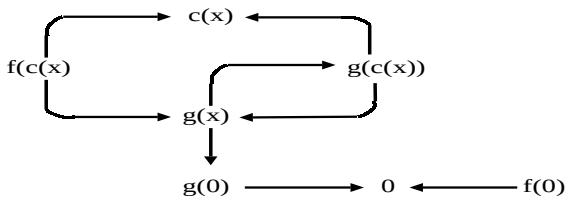


Figura 5.1: Grafo de prevención de bucles, Ejemplo 21

método no depende del objetivo a resolver (sólo del programa), y permite obtener un programa abstracto para el cual todas las derivaciones de *narrowing* aproximado terminan. Más aún, independientemente del grafo de prevención de bucles usado, la semántica de un objetivo puede ser aproximada correctamente usando las reglas del programa abstracto. Informalmente, un programa se abstrae simplificando las partes derechas de las cabezas de las reglas, así como las ecuaciones en el cuerpo de las mismas. La definición es inductiva sobre la estructura de los términos y ecuaciones. La idea principal consiste en que, aquellos términos cuyo nodo asociado en el grafo posee un bucle, son eliminados del programa y sustituidos por $\sharp$.

Definición 5.2.6 (programa abstracto) *Sea $\mathcal{R}$ un SRTC y $\mathcal{G}_{\mathcal{R}}$ un grafo de prevención de bucles para $\mathcal{R}$. Definimos la abstracción de $\mathcal{R}$ usando $\mathcal{G}_{\mathcal{R}}$ como sigue:*

$$\mathcal{R}^{\sharp} = \{\lambda \rightarrow sh(\rho) \Leftarrow sh(C) \mid (\lambda \rightarrow \rho \Leftarrow C) \in \mathcal{R}\}$$

donde la función $sh(o)$ para un objeto sintáctico o se define inductivamente de la forma:

$$sh(o) = \begin{cases} o & si\ o \in V \\ f(sh(t_1), \ldots, sh(t_k)) & si\ o \equiv f(t_1, \ldots, t_k)\ y\ \langle \mathring{o}, \mathring{o} \rangle \notin \mathcal{G}_{\mathcal{R}}^{+} \\ sh(l) = sh(r) & si\ o \equiv (l = r) \\ sh(e_1), \ldots, sh(e_n) & si\ o \equiv e_1, \ldots, e_n \\ \sharp & en\ otro\ caso. \end{cases}$$

El siguiente ejemplo ilustra la definición.

Ejemplo 22 *Consideremos de nuevo el programa $\mathcal{R}$ del ejemplo 20*

$$\begin{array}{rcl} \{\mathtt{g(0)} & \rightarrow & \mathtt{0} \\ \mathtt{g(c(x))} & \rightarrow & \mathtt{c(g(x))} \\ \mathtt{f(0)} & \rightarrow & \mathtt{0} \\ \mathtt{f(c(x))} & \rightarrow & \mathtt{x} \Leftarrow \mathtt{g(c(x))} = \mathtt{c(x)} \end{array}$$

La abstracción de $\mathcal{R}$, usando el grafo de prevención de bucles $\mathcal{G}_{\mathcal{R}}$, es el siguiente

programa abstracto $\mathcal{R}^\sharp =:$

$$\begin{array}{rcl} \{\mathtt{g(0)} & \rightarrow & \mathtt{0} \\ \mathtt{g(c(x))} & \rightarrow & \mathtt{c(\sharp)} \\ \mathtt{f(0)} & \rightarrow & \mathtt{0} \\ \mathtt{f(c(x))} & \rightarrow & \mathtt{x} \Leftarrow \sharp = \mathtt{c(x)} \end{array}$$

en el que la parte derecha de la segunda y cuarta reglas han sido drásticamente simplificada introduciendo un $\sharp$.

Una dificultad que observamos en el método de abstracción anterior es que la perdida de información en el proceso es muy alta. En la búsqueda de un método más refinado y que presente una abstracción con mejores propiedades semánticas, proponemos la siguiente definición. Primero abstraemos una regla r obteniendo $r^\sharp$ (seleccionamos primero las reglas recursivas, si las hay; de otro modo seleccionamos cualquier regla del programa). Entonces, reemplazamos r por $r^\sharp$ en $\mathcal{R}$ y recomputamos nuevamente el grafo de prevención de bucles, antes de proceder con la abstracción de la siguiente regla.

Definición 5.2.7 (programa abstracto mejorado) *Sea* $\mathcal{R}$ *un programa,* $r = (\lambda \rightarrow \rho \Leftarrow C) \in \mathcal{R}$ *y* $\mathcal{G}_\mathcal{R}$ *un grafo de prevención de bucles para* $\mathcal{R}$. *Definimos la abstracción de* r *como sigue:*

$$r^\sharp = \lambda \rightarrow sh(\rho, \mathcal{G}_\mathcal{R}) \Leftarrow sh(C, \mathcal{G}_\mathcal{R})$$

donde la función $sh(x, \mathcal{G})$ *de una expresión* x *de acuerdo con el grafo de prevención de bucles* $\mathcal{G}$ *está definida inductivamente*

$$sh(x, \mathcal{G}) = \begin{cases} x & si\ x \in V \\ f(sh(t_1, \mathcal{G}), \ldots, sh(t_k, \mathcal{G})) & si\ x \equiv f(t_1, \ldots, t_k)\ y\ \langle \mathring{x}, \mathring{x} \rangle \notin \mathcal{G}^+ \\ sh(l, \mathcal{G}) = sh(r, \mathcal{G}) & si\ x \equiv (l = r) \\ sh(e_1, \mathcal{G}), \ldots, sh(e_n, \mathcal{G}) & si\ x \equiv e_1, \ldots, e_n \\ \sharp & en\ otro\ caso \end{cases}$$

Ejemplo 23 *Continuando con el programa planteado en el Ejemplo 21 y teniendo en cuenta el grafo de prevención de bucles de la Figura 5.1, la abstracción de programa* $\mathcal{R}$ *es el programa* $\mathcal{R}^\sharp$

$$\begin{array}{rcl} \{\mathtt{g(0)} & \rightarrow & \mathtt{0} \\ \mathtt{g(c(x))} & \rightarrow & \mathtt{c(\sharp)} \\ \mathtt{f(0)} & \rightarrow & \mathtt{0} \\ \mathtt{f(c(x))} & \rightarrow & \mathtt{x} \Leftarrow \mathtt{g(c(x))} = \mathtt{c(x)} \end{array}$$

5.3 Semántica abstracta bottom-up genérica

En esta sección definimos una semántica abstracta de punto fijo en términos del menor punto fijo de una transformación $T_{\mathcal{R}}^{\varphi\sharp}$ basada en unificación abstracta y en la operación de abstracción de un programa, la cual es paramétrica con respecto a la estrategia de *narrowing*. La idea es determinar una aproximación, computable finitamente, de la denotación concreta del programa $\mathcal{R}$. En lo que sigue, definimos la transformación abstracta $T_{\mathcal{R}}^{\varphi\sharp}$.

Definición 5.3.1 (base de Herbrand abstracta, interpretación de Herbrand abstracta) *La base de Herbrand abstracta $\mathcal{B}_V^\sharp$ se define como el conjunto de ecuaciones sobre el universo de Herbrand abstracto $\mathcal{H}_V^\sharp$. Una interpretación de Herbrand abstracta es cualquier elemento de $2^{\mathcal{B}_V^\sharp}$. Un orden parcial $\subseteq^\sharp$ sobre interpretaciones abstractas puede estar definido de manera similar al orden $\subseteq$ sobre interpretaciones.*

De un modo simple, se demuestra que el conjunto de interpretaciones abstractas es un retículo completo con respecto a $\subseteq^\sharp$. Una ecuación trivial abstracta es una ecuación de la forma $\sharp = X$, $X = \sharp$ o $\sharp = \sharp$.

Definición 5.3.2 *Sea $\mathcal{R}$ un programa en $I\!R_\varphi$, $\mathcal{G}_\mathcal{R}$ un grafo de prevención de bucles para $\mathcal{R}$ y $\mathcal{R}^\sharp$ la abstracción de $\mathcal{R}$ utilizando $\mathcal{G}_\mathcal{R}$ donde se han eliminado las ecuaciones triviales abstractas del cuerpo de las reglas, si existen. Si $\mathcal{I}$ es una interpretación abstracta. Entonces,*

$$T_{\mathcal{R}}^{\sharp\varphi}(\mathcal{I}) = \Phi_\mathcal{R} \cup \Im_\mathcal{R}^\varphi \cup \{\ e \in \mathcal{B}_V^\sharp \mid \begin{array}{l}(\lambda \to \rho \Leftarrow C) \ll \mathcal{R}^{\sharp\varphi},\\ \{l = r\} \cup C' \subseteq \mathcal{I},\\ mgu^\sharp(flat_\varphi(C), C') = \sigma,\\ mgu^\sharp(\{\lambda = (r_{|u})\}\sigma) = \theta,\\ u \in \overline{O}^\varphi(r),\\ e = (l = r[\rho]_u)\sigma\theta\ \}.\end{array}$$

donde $\mathcal{R}^{\sharp\varphi} = \mathcal{R}_+^\sharp$ si $\varphi = inn$, mientras que $\mathcal{R}^{\sharp\varphi} = \mathcal{R}_+^\sharp \cup \{f(x_1,\ldots,x_n) \to \bot \mid f/n \in \mathcal{D}$ y $x_1,\ldots,x_n$ son variables$\}$ si $\varphi = out$

Proposición 5.3.3 *El operador $T_{\mathcal{R}}^{\sharp\varphi}$ es continuo sobre el retículo completo de interpretaciones abstractas, $\varphi \in \{inn, out\}$.*

Demostración. Análoga a la Proposición 3.2.2.

Vamos a probar que, para cualquier secuencia infinita $I_1 \subseteq I_2 \subseteq \ldots$

$$T_{\mathcal{R}}^{\sharp\varphi}(\cup_{n=1}^\infty \mathcal{I}_n) = \cup_{n=1}^\infty T_{\mathcal{R}}^{\sharp\varphi}(\mathcal{I}_n)$$

Si $e \in T_{\mathcal{R}}^{\sharp\varphi}(\cup_{n=1}^{\infty}\mathcal{I}_n)$ por definición de $T_{\mathcal{R}}^{\sharp\varphi}$ equivale a

$$e \in \Phi_{\mathcal{R}} \cup \Im_{\mathcal{R}}^{\varphi} \cup \{e' \in \mathcal{B}_V^{\sharp} \mid \begin{array}{l}(\lambda = \rho \Leftarrow C) \ll \mathcal{R}^{\sharp\varphi},\ \ \{l = r\} \cup C' \subseteq \cup_{n=1}^{\infty}\mathcal{I}_n, \\ mgu^{\sharp}(flat_{\varphi}(C), C') = \sigma,\ \ mgu^{\sharp}(\{\lambda = r_{|u}\}\sigma) = \theta, \\ u \in \overline{O}^{\varphi}(r),\ \ e' = (l = r[\rho]_u)\sigma\theta\ \}\end{array}$$

si y solo si existe i, $1 \leq i \leq n$ (definición de $\cup_{n=1}^{\infty}\mathcal{I}_n$)

$$e \in \Phi_{\mathcal{R}} \cup \Im_{\mathcal{R}}^{\varphi} \cup \{e' \in \mathcal{B}_V^{\sharp} \mid \begin{array}{l}(\lambda = \rho \Leftarrow C) \ll \mathcal{R}^{\sharp\varphi},\ \ \{l = r\} \cup C' \subseteq \mathcal{I}_i, \\ mgu^{\sharp}(flat_{\varphi}(C), C') = \sigma,\ \ mgu^{\sharp}(\{\lambda = r_{|u}\}\sigma) = \theta, \\ u \in \overline{O}^{\varphi}(r),\ \ e' = (l = r[\rho]_u)\sigma\theta\ \}\end{array}$$

si y solo si para algún i, $1 \leq i \leq n$, tenemos que $e \in T_{\mathcal{R}}^{\sharp\varphi}(\mathcal{I}_i)$
si y solo si $e \in \cup_{n=1}^{\infty} T_{\mathcal{R}}^{\sharp\varphi}(\mathcal{I}_n)$.

Para completar la demostración solo queda decir que, debido a la continuidad del operador de consecuencias inmediatas, el menor punto fijo existe y lo denotamos por $lfp(T_{\mathcal{R}}^{\sharp\varphi})$. □

Definimos $\mathcal{F}_{\varphi}^{\sharp}(\mathcal{R})$ y $\mathcal{F}_{\varphi}^{ca\sharp}(\mathcal{R})$ del mismo modo que lo hacemos con las construcciones concretas de $\mathcal{F}_{\varphi}(\mathcal{R})$ y $\mathcal{F}_{\varphi}^{ca}(\mathcal{R})$, siguiendo un procedimiento análogo al de la Sección 5.2.

Definición 5.3.4 (semántica de menor punto fijo abstracta) *Sea $\mathcal{F}_{\varphi}^{\sharp}(\mathcal{R}) = lfp(T_{\mathcal{R}}^{\sharp})$. La semántica de menor punto fijo abstracta de un programa $\mathcal{R}$ es $\mathcal{F}_{\varphi}^{ca\sharp}(\mathcal{R}) = \{l = r \in \mathcal{F}_{\varphi}^{\sharp}(\mathcal{R}) \mid r$ no contiene ningún símbolo de función definido $f/n \in \mathcal{D}\}$*

Lema 5.3.5 *Sea $\mathcal{R}$ un programa en $I\!R_{\varphi}$ y $\mathcal{R}^{\sharp}$ la correspondiente abstracción del programa $\mathcal{R}$. Para todo $k \geq 0$ se cumple que*

$$(f(\bar{x})\theta = t) \in T_{\mathcal{R}}^{\sharp\varphi} \uparrow k \quad sii \quad f(\bar{x}) = y \overset{\mathcal{R}^{\sharp}, \theta_*}{\leadsto}_{\sharp\varphi} t = y$$

en donde $\perp$ no ocurre en t.

Demostración. Análogo al Lema 3.2.5, reemplazando unificación mgu, sistemas de reescritura de términos STR, narrowing $\overset{\sigma}{\leadsto}_{\varphi}$ y operador de consecuencias inmediatos $T_{\mathcal{R}}^{\varphi}$ por unificación abstracta $mgu^{\sharp}$, sistemas de reescritura de términos abstracto $STR^{\sharp}$, narrowing abstracto $\overset{\sigma}{\leadsto}_{\sharp\varphi}$ y operador de consecuencias inmediatos abstracto $T_{\mathcal{R}}^{\sharp\varphi}$.

□

El siguiente teorema establece que $\mathcal{F}_{\varphi}^{\sharp}(\mathcal{R})$ y $\mathcal{F}_{\varphi}^{ca\sharp}(\mathcal{R})$ son computables finitamente.

Teorema 5.3.6 *Existe un número finito positivo k tal que $\mathcal{F}_{\varphi}^{\sharp}(\mathcal{R}) = T_{\mathcal{R}}^{\sharp\varphi} \uparrow k$, donde $\varphi = inn,\ out$.*

Demostración. Probemos que existe un número finito positivo k tal que $\mathcal{F}^{\sharp}_{\varphi}(\mathcal{R}) = T^{\sharp\varphi}_{\mathcal{R}} \uparrow k$ es el punto fijo del operador de consecuencias inmediatas abstracto. Para ello, probemos que dicho conjunto es finito.

Razonemos por reducción al absurdo, supongamos que es un conjunto infinito. Esto significa que para cada $k > 0$ existe al menos una ecuación $e \notin T^{\sharp\varphi}_{\mathcal{R}} \uparrow k$ y $e \in T^{\sharp\varphi}_{\mathcal{R}} \uparrow (k+1)$ generada a partir de una ecuación $e' \in T^{\sharp\varphi}_{\mathcal{R}} \uparrow k$.

Entonces existe una secuencia infinita:

$$\begin{array}{lcll} f(\bar{x}) & = & f(\bar{x}) \in T^{\sharp\varphi}_{\mathcal{R}} \uparrow 0 \\ f(\bar{x})\theta_1 & = & t_1 \quad \in T^{\sharp\varphi}_{\mathcal{R}} \uparrow 1 \\ f(\bar{x})\theta_1\theta_2 & = & t_2 \quad \in T^{\sharp\varphi}_{\mathcal{R}} \uparrow 2 \\ \vdots & & & \\ f(\bar{x})\theta_1\theta_2 \ldots \theta_{k+1} & = & t_{k+1} \quad \in T^{\sharp\varphi}_{\mathcal{R}} \uparrow (k+1) \\ \vdots & & & \end{array} \tag{5.1}$$

De acuerdo con el Lema 5.3.5, existe una derivación infinita de narrowing abstracto $\leadsto_{\sharp\varphi}$ a partir del objetivo $f(\bar{x}) = y$ y el correspondiente grafo de prevención de bucles para $\mathcal{R}^{\sharp}$ no contiene bucles. Por el Lema 5.2.2 obtenemos una contradicción.

Se concluye que existe un número finito positivo k tal que $\mathcal{F}^{\sharp}_{\varphi}(\mathcal{R}) = T^{\sharp\varphi}_{\mathcal{R}} \uparrow k$ □

Ejemplo 24 *Continuando con el programa planteado en el Ejemplo 20 y teniendo en cuenta el grafo de prevención de bucles de la Figura 5.1, la abstracción de programa $\mathcal{R}$ con respecto a la estrategia $\varphi = inn$ es $\mathcal{R}^{\sharp} =$*

$$\begin{array}{rcl} \{\mathtt{g(0)} & \to & \mathtt{0} \\ \mathtt{g(c(x))} & \to & \mathtt{c(\sharp)} \\ \mathtt{f(0)} & \to & \mathtt{0} \\ \mathtt{f(c(x))} & \to & \mathtt{x} \Leftarrow \mathtt{g(c(x))} = \mathtt{c(x)} \end{array}$$

La semántica del menor punto fijo es

$$\begin{array}{rcl} \mathcal{F}_{inn}(\mathcal{R}) & = & \{\mathtt{0} = \mathtt{0},\ \mathtt{g(x)} = \mathtt{g(x)},\ \mathtt{f(x)} = \mathtt{f(x)},\ \mathtt{c(x)} = \mathtt{c(x)},\ \mathtt{g(0)} = \mathtt{0}, \\ & & \mathtt{f(0)} = \mathtt{0},\ \mathtt{g(c(x))} = \mathtt{c(g(x))}, \ldots,\ \mathtt{g(c^n(x))} = \mathtt{c^n(g(x))}, \ldots, \\ & & \mathtt{g(c(0))} = \mathtt{c(0)}, \ldots,\ \mathtt{g(c^n(0))} = \mathtt{c^n(0)}, \ldots,\ \mathtt{f(c(0))} = \mathtt{0}, \ldots, \\ & & \mathtt{f(c^n(0))} = \mathtt{c^{n-1}(0)}, \ldots\} \end{array}$$

La correspondiente semántica abstracta de punto fijo es el conjunto es

$$\begin{array}{rcl} \mathcal{F}^{\sharp}_{inn}(\mathcal{R}) & = & \{\mathtt{0} = \mathtt{0},\ \mathtt{g(x)} = \mathtt{g(x)},\ \mathtt{f(x)} = \mathtt{f(x)},\ \mathtt{c(x)} = \mathtt{c(x)},\ \mathtt{g(0)} = \mathtt{0}, \\ & & \mathtt{f(0)} = \mathtt{0},\ \mathtt{g(c(x))} = \mathtt{c(\sharp)},\ \mathtt{f(c(x))} = \mathtt{x}\} \end{array}$$

y la correspondiente semántica de respuestas computadas

$$\begin{array}{rcl} \mathcal{F}^{ca\sharp}_{inn}(\mathcal{R}) & = & \{\mathtt{0} = \mathtt{0},\ \mathtt{c(x)} = \mathtt{c(x)},\ \mathtt{g(0)} = \mathtt{0},\ \mathtt{f(0)} = \mathtt{0},\ \mathtt{g(c(x))} = \mathtt{c(\sharp)}, \\ & & \mathtt{f(c(x))} = \mathtt{x}\} \end{array}$$

que aproxima de forma segura al conjunto de éxitos del programa.

Desde el punto de vista semántico, dado un programa $\mathcal{R}$, la aproximación a la semántica de punto fijo $\mathcal{F}_\varphi(\mathcal{R})$ (resp. $\mathcal{F}_\varphi{}^{ca}(\mathcal{R})$) es la correspondiente semántica de punto fijo abstracta $\mathcal{F}_\varphi{}^\sharp(\mathcal{R})$ (resp. $\mathcal{F}_\varphi{}^{ca\sharp}(\mathcal{R})$). Es decir, podemos computar una aproximación abstracta de la semántica concreta en un número finito de pasos. La corrección de la semántica de punto fijo abstracta con respecto a la semántica concreta se establece en el siguiente teorema:

Lema 5.3.7 $T_{\mathcal{R}}^{\sharp\varphi} \propto T_{\mathcal{R}}^{\varphi}$

Demostración. Esquema de la demostración. Debemos probar que, para todo I, I' tal que $I' \propto I$, $T_{\mathcal{R}}^{\sharp\varphi}(I') \propto T_{\mathcal{R}}^{\varphi}(I)$, es decir $T_{\mathcal{R}}(I) \in \gamma(T_{\mathcal{R}}^{\sharp\varphi}(I'))$. La demostración se sigue inmediatamente del Lema 3.2.5, Lema 5.3.5 y Lema 4.2.18 de [Vidal, 1996], en donde se probó que $\overset{\mathcal{R}'}{\leadsto}_{\sharp\varphi} \propto \overset{\mathcal{R}}{\leadsto}_{\varphi}$ siempre que $\mathcal{R}' \propto \mathcal{R}$, para toda estrategia independiente del entorno φ. □

Teorema 5.3.8 $\mathcal{F}_\varphi^\sharp(\mathcal{R}) \propto \mathcal{F}_\varphi(\mathcal{R})$ *y* $\mathcal{F}_\varphi^{ca\sharp}(\mathcal{R}) \propto \mathcal{F}_\varphi^{ca}(\mathcal{R})$.

Demostración. El resultado se sigue del Lema 5.3.7. □

La semántica $\mathcal{F}_\varphi^\sharp(\mathcal{R})$ recoge información (de forma independiente del objetivo) acerca de los patrones de éxito de un programa dado. La relación entre la semántica de punto fijo abstracta y la semántica operacional concreta (sustitutiones de respuestas computada) está dada por el siguiente teorema. Intuitivamente, dado un objetivo g, obtenemos una descripción del conjunto de respuestas computadas de g por unificación abstracta de las ecuaciones de $flat_\varphi(g)$ con las ecuaciones de la semántica aproximada $\mathcal{F}_\varphi^\sharp(\mathcal{R})$.

Teorema 5.3.9 (completitud fuerte) *Sea $\mathcal{R}$ un programa en $\mathbb{R}_\varphi$ y g un objetivo. Si θ es una sustitución de respuesta computada para g en $\mathcal{R}$ con respecto a φ, entonces existe $g' \equiv e_1, \ldots, e_m \ll \mathcal{F}^\sharp(\mathcal{R})$ tal que $\theta' = mgu^\sharp(flat(g), g')$ y $(\theta' \preceq \theta)_{|\mathcal{V}ar(g)}$.*

Demostración. (Esquema). Es análoga a la demostración para Teorema 3.2.6 usando Lema 5.3.5 en lugar del Lema 3.2.5, y el Teorema 5.3.8. □

Ejemplo 25 *Continuamos con los resultados obtenidos en los Ejemplos 20 y 24. Dado el objetivo $g \equiv$* $\mathtt{f(g(x))} = \mathtt{y}$, narrowing *condicional innermost computa el conjunto infinito de sustituciones*

$$\{\{x/\mathtt{0},\ \mathtt{y/0}\}, \{\mathtt{x/c(0)},\ \mathtt{y/0}\}, \{\mathtt{x/c^2(0)},\ \mathtt{y/c(0)}\}, \ldots, \{\mathtt{x/c^n(0)},\ \mathtt{y/c^{n-1}(0)}\}\}$$

Las sustituciones abstractas calculadas por el proceso de unificación abstracta del objetivo ecuacional $\mathtt{f(z)} = \mathtt{y},\ \mathtt{g(x)} = \mathtt{z}$ *en forma plana, con $\mathcal{F}_{inn}^{ca\sharp}(\mathcal{R})$ son*

$$\{\{x/\mathtt{0},\ \mathtt{y/0}\}, \{\mathtt{x/c(x')}, \mathtt{y/\sharp}\}\}$$

las cuales aproximan las respuestas computadas del objetivo g.

Ejemplo 26 *consideremos el siguiente programa, construido con partes de código de otros ejemplos previos* [2]

$$\begin{array}{rcl}
\texttt{from}(\texttt{x}) & \to & [\texttt{x}|\texttt{from}(\texttt{s}(\texttt{x}))]. \\
\texttt{first}([\texttt{x}|\texttt{y}])) & \to & \texttt{x}. \\
\texttt{add}(0,\texttt{x}) & \to & \texttt{x}. \\
\texttt{add}(\texttt{s}(\texttt{x}),\texttt{y}) & \to & \texttt{s}(\texttt{add}(\texttt{x},\texttt{y})). \\
\texttt{double}(\texttt{x}) & \to & \texttt{add}(\texttt{x},\texttt{x}).
\end{array}$$

Consideremos la estrategia outermost*, es decir $\varphi = out$. Consideremos el "grafo de prevención de bucles" [Alpuente* et al.*, 1996]*

$$\mathcal{G}_{\mathcal{R}} = \{\langle \texttt{from}(\texttt{x}), \texttt{from}(\texttt{x})\rangle, \langle \texttt{add}(\texttt{x},\texttt{y}), \texttt{add}(\texttt{x},\texttt{y})\rangle\}$$

y sea $\overset{\circ}{t}= t'$ una función (parcial) que, dado un grafo, asigna a un término t algún nodo t' en el grafo, tal que t' unifica con t, si tal nodo t' existe. Entonces, la abstracción del programa $\mathcal{R}$ es $\mathcal{R}^\sharp$:

$$\begin{array}{rcl}
\texttt{from}(\texttt{x}) & \to & [\texttt{x}|\sharp]. \\
\texttt{first}([\texttt{x}|\texttt{y}]) & \to & \texttt{x}. \\
\texttt{add}(0,\texttt{x}) & \to & \texttt{x}. \\
\texttt{add}(\texttt{s}(\texttt{x}),\texttt{y}) & \to & \texttt{s}(\sharp). \\
\texttt{double}(\texttt{x}) & \to & \texttt{add}(\texttt{x},\texttt{x}).
\end{array}$$

La correspondiente semántica de punto fijo abstracta es el conjunto finito

$$\begin{array}{rcl}
\mathcal{F}^\sharp_{out}(\mathcal{R}) & = & \{0 \approx 0,\ \texttt{s}(\texttt{x}) \approx \texttt{s}(\texttt{x}),\ [\texttt{x}|\texttt{y}] \approx [\texttt{x}|\texttt{y}],\ \texttt{add}(\texttt{x},\texttt{y}) = \texttt{add}(\texttt{x},\texttt{y}),\ \texttt{from}(\texttt{x}) = \texttt{from}(\texttt{x}), \\
& & \texttt{double}(\texttt{x}) = \texttt{double}(\texttt{x}),\ \texttt{first}(\texttt{x}) = \texttt{first}(\texttt{x}),\ (\texttt{x} \approx \texttt{y}) = (\texttt{x} \approx \texttt{y}), \\
& & \texttt{and}(\texttt{x},\texttt{y}) = \texttt{and}(\texttt{x},\texttt{y}),\ ([\texttt{x}|\texttt{y}] \approx [\texttt{x}|\texttt{y}]) = \sharp,\ ([\texttt{x}|\texttt{y}] \approx [\texttt{x}|\texttt{y}]) = \bot,\ (0 \approx 0) = \texttt{true}, \\
& & (\texttt{x} \approx \texttt{y}) = \bot,\ (\texttt{s}(\texttt{x}) \approx \texttt{s}(\texttt{y})) = \sharp,\ ([\texttt{x}|\texttt{y}] \approx [\texttt{x}|\texttt{y}]) = \texttt{and}(\sharp,\sharp),\ \texttt{and}(\texttt{true},\texttt{x}) = \texttt{x}, \\
& & \texttt{and}(\texttt{x},\texttt{y}) = \bot,\ \texttt{from}(\texttt{x}) = \bot,\ \texttt{from}(\texttt{x}) = [\texttt{x}|\sharp],\ \texttt{first}(\texttt{x}) = \bot,\ \texttt{first}([\texttt{x}|\texttt{y}]) = \texttt{x}, \\
& & \texttt{add}(\texttt{x},\texttt{y}) = \bot,\ \texttt{add}(0,\texttt{x}) = \texttt{x},\ \texttt{add}(\texttt{s}(\texttt{x}),\texttt{y}) = \texttt{s}(\sharp),\ \texttt{double}(0) = 0, \\
& & \texttt{double}(\texttt{x}) = \bot,\ \texttt{double}(\texttt{x}) = \texttt{add}(\texttt{x},\texttt{x}),\ \texttt{double}(\texttt{s}(\texttt{x})) = \texttt{s}(\sharp)\}
\end{array}$$

la que aproxima el conjunto de éxitos del programa. y la semántica de punto fijo concreta es

[2] Por simplicidad, en el ejemplo ignoramos la condición de que cada función debe ser totalmente definida, lo cual es necesario para la completitud de la estrategia $\varphi = out$ [Echahed, 1992; Padawitz, 1988], pero tal condición se puede lograr fácilmente con la introducción de *generos* o *sorts*.

$$
\begin{aligned}
\mathcal{F}_{out}(\mathcal{R}) \;=\; & \{\mathtt{0} \approx \mathtt{0},\ \mathtt{s(x)} \approx \mathtt{s(x)},\ \mathtt{[x|y]} \approx \mathtt{[x|y]},\ \mathtt{double(x)} = \mathtt{double(x)},\\
& \mathtt{add(x,y)} = \mathtt{add(x,y)},\ \mathtt{from(x)} = \mathtt{from(x)},\ \mathtt{first(x)} = \mathtt{first(x)},\\
& \mathtt{and(x,y)} = \mathtt{and(x,y)},\ (\mathtt{x} \approx \mathtt{y}) = (\mathtt{x} \approx \mathtt{y}),\ \mathtt{and(true,x)} = \mathtt{x},\\
& \mathtt{and(x,y)} = \bot,\ (\mathtt{[x|y]} \approx \mathtt{[x_1|y_1]}) = \mathtt{and}(\mathtt{x} \approx \mathtt{x_1}, \mathtt{y} \approx \mathtt{y_1}),\\
& (\mathtt{[x|y]} \approx \mathtt{[x_1|y_1]}) = \bot,\ (\mathtt{s(x)} \approx \mathtt{s(y)}) = (\mathtt{x} \approx \mathtt{y}),\ \mathtt{from(x)} = \bot,\\
& \mathtt{from(x)} = \mathtt{[x|from(s(x))]},\ \mathtt{from(x)} = \mathtt{[x|}\bot\mathtt{]}, \ldots,\\
& \mathtt{from(x)} = \mathtt{[x|[s(x)|\ldots[s^n(x)|from(s^{n+1}(x))]]]}, \ldots,\\
& \mathtt{from(x)} = \mathtt{[x|[s(x)|\ldots[s^n(x)|}\bot\mathtt{]]]}, \ldots, \mathtt{first(x)} = \bot,\ \mathtt{first([x|y])} = \mathtt{x},\\
& (\mathtt{x} \approx \mathtt{y}) = \bot,\ (\mathtt{0} \approx \mathtt{0}) = \mathtt{true}, (\mathtt{s(x)} \approx \mathtt{s(y)}) = \bot, \ldots, (\mathtt{s^n(x)} \approx \mathtt{s^n(y)}) = \bot,\\
& \ldots, (\mathtt{s(x)} \approx \mathtt{s(y)}) = (\mathtt{x} \approx \mathtt{y}), \ldots, (\mathtt{s^n(x)} \approx \mathtt{s^n(y)}) = (\mathtt{x} \approx \mathtt{y}), \ldots,\\
& (\mathtt{s(0)} \approx \mathtt{s(0)}) = \mathtt{true}, \ldots, (\mathtt{s^n(0)} \approx \mathtt{s^n(0)}) = \mathtt{true}, \ldots \mathtt{add(x,y)} = \bot,\\
& \mathtt{add(s(x),y)} = \mathtt{s}(\bot), \ldots, \mathtt{add(s^n(x),y)} = \mathtt{s^n}(\bot), \ldots, \mathtt{add(0,x)} = \mathtt{x},\\
& \mathtt{add(s(0),x)} = \mathtt{s(x)}, \ldots, \mathtt{add(s^n(0),x)} = \mathtt{s^n(x)}, \ldots,\\
& \mathtt{add(s(x),y)} = \mathtt{s(add(x,y))}, \ldots, \mathtt{add(s^n(x),y)} = \mathtt{s^n(add(x,y))}, \ldots,\\
& \mathtt{double(x)} = \bot, \ldots, \mathtt{double(s^n(x))} = \mathtt{s^n}(\bot), \ldots, \mathtt{double(x)} = \mathtt{add(x,x)},\\
& \mathtt{double(s(x))} = \mathtt{s(add(x,s(x)))}, \ldots, \mathtt{double(s^n(x))} = \mathtt{s^n(add(x,s^n(x)))},\\
& \ldots, \mathtt{double(0)} = \mathtt{0},\ \mathtt{double(s(0))} = \mathtt{s^2(0)}, \ldots,\\
& \mathtt{double(s^n(0))} = \mathtt{s^{2*n}(0)}, \ldots\}
\end{aligned}
$$

y Dado el objetivo $g \equiv\ \mathtt{first([double(x)|from(x)])} \approx \mathtt{y}$, narrowing outermost *computa el conjunto finito de sustituciones*

$$\{\{\mathtt{x/0},\ \mathtt{y/0}\}, \{\mathtt{x/s(0)},\ \mathtt{y/s^2(0)}\}, \ldots, \{\mathtt{x/s^n(0)},\ \mathtt{y/s^{2*n}(0)}\}\}$$

Ahora bien, las sustituciones abstractas computadas por la unificación abstracta en $\mathcal{F}^{ca\sharp}_{out}(\mathcal{R})$ *de las ecuaciones del objetivo plano* $flat_{out}(g) =$

$$(\mathtt{double(x)} = \mathtt{z_1},\ \mathtt{from(x)} = \mathtt{z_2},\ \mathtt{first([z_1|z_2])} = \mathtt{z_3},\ \mathtt{z_3} \approx \mathtt{y})$$

son

$$\{\{\mathtt{x/0},\ \mathtt{y/0}\}, \{\ \mathtt{x/s(x_1)},\ \mathtt{y/s}(\sharp)\}, \{\mathtt{x/x'}, \mathtt{y/}\bot\}\},$$

las cuales aproximan las respuestas computadas de g.

Es importante notar que los dos símbolos $\sharp$ y $\bot$ relacionados, en nuestro marco, con la "falta de información" pueden aparecer simultáneamente en la semántica $\mathcal{F}^{\sharp}_{out}(\mathcal{R})$ y también por dentro de una sesión de depuración(ver apéndice). Así pues debemos diferenciarlos porque estos símbolos presentan comportamientos diferentes. Por ejemplo $\sharp$ unifica abstractamente con todos los términos mientras que $\bot$ no. $\bot$ es un símbolo especial constructor de datos(constante) que tiene la interpretación de "valor por defecto" de una función. Este símbolo nunca ocurre en un programa pero puede ocurrir en una sesión de depuración. Por otro lado, el símbolo $\sharp$ ocurre en los programas (abstractos) y también durante las sesiones de depuración, por consiguiente, es necesario diferenciarlos sintácticamente para evitar confusiones.

Capítulo 6

Diagnóstico abstracto

En este capítulo, aplicamos la teoría de diagnóstico abstracto, referida en el capítulo anterior, al desarrollo de un depurador eficiente que se basa en las nociones de sobreaproximación y subaproximación [Bueno *et al.*, 1997a; Comini *et al.*, 1999] de la semántica de punto fijo deseada. La idea básica es considerar dos conjuntos para la verificación parcial de la corrección: un conjunto $\mathcal{I}^+$ que aproxima por exceso la semántica concreta deseada $\mathcal{I}$ (esto es, $\mathcal{I} \subseteq \gamma(\mathcal{I}^+)$) y otro conjunto $\mathcal{I}^-$ que aproxima por defecto $\mathcal{I}$ (esto es, $\mathcal{I} \supseteq \gamma(\mathcal{I}^-)$). Utilizamos tales aproximaciones para verificar la corrección (resp. completitud) con respecto a $(\mathcal{I}^+, \mathcal{I}^-)$ mediante la aplicación, una vez, del operador de consecuencias inmediatas con respecto al programa $\mathcal{R}$, a $\mathcal{I}^-$ (resp. $\mathcal{I}^+$), tal como se puede ver en lo que sigue.

6.1 Depuración declarativa

Los siguientes resultados son la base para construir el depurador.

Proposición 6.1.1 *Si existe una ecuación e tal que $e \notin \gamma(\mathcal{I}^+)$ y $e \in T^{\varphi}_{\{r\}}([\![\mathcal{I}^-]\!])$, entonces la regla $r \in \mathcal{R}$ es incorrecta sobre e.*

Demostración. Supongamos que existe una ecuación e tal que $e \notin \gamma(\mathcal{I}^+)$ y $e \in T^{\varphi}_{\{r\}}([\![\mathcal{I}^-]\!])$ y demostremos que la regla $r \in \mathcal{R}$ es incorrecta sobre e según la Definición 4.4.3. Para ello probaremos que $e \notin \mathcal{I}$ y $e \in T^{\varphi}_{\{r\}}(\mathcal{I})$.

Primero probemos que $e \in T^{\varphi}_{\{r\}}(\mathcal{I})$. Ya que $e \in T^{\varphi}_{\{r\}}([\![\mathcal{I}^-]\!])$, y por la definición del operador $[\![\ ]\!]$ se cumple que $[\![\mathcal{I}^-]\!] \subseteq \gamma(\mathcal{I}^-)$. Entonces, por la definición de $\mathcal{I}^-$ tenemos que $\gamma(\mathcal{I}^-) \subseteq \mathcal{I}$, y por la propiedad de la monotonía del operador $T^{\varphi}_{\mathcal{R}}$, tenemos que $e \in T^{\varphi}_{\{r\}}(\mathcal{I})$.

Por otro lado, como $e \notin \gamma(\mathcal{I}^+)$, y por la definición de $\mathcal{I}^+$ se tiene que $\gamma(\mathcal{I}^+) \supseteq \mathcal{I}$, concluimos que $e \notin \mathcal{I}$ y con esto finaliza la demostración.

□

Proposición 6.1.2 *Si existe una ecuación (abstracta) e tal que* $[\![e]\!] \notin \gamma(T_{\mathcal{R}}^{\sharp\varphi}([\![\mathcal{I}^+]\!]))$ *y* $e \in \mathcal{I}^-$*, entonces la ecuación* $[\![e]\!]$ *es no cubierta.*

Demostración. Asumamos que $e \in \mathcal{I}^-$ y $[\![e]\!] \notin \gamma(T_{\mathcal{R}}^{\sharp\varphi}(\mathcal{I}^+))$ y demostremos que la ecuación $[\![e]\!]$ es no cubierta, de acuerdo con la Definición 4.4.4, es decir, $[\![e]\!] \in \mathcal{I}$ y $[\![e]\!] \notin T_{\mathcal{R}}^{\varphi}(\mathcal{I})$.

Debido a que $e \in \mathcal{I}^-$ entonces (salvo renombramientos de variables) $[\![e]\!] \in [\![\mathcal{I}^-]\!]$ y por la definición de $[\![\ ]\!]$ se da $[\![\mathcal{I}^-]\!] \subseteq \gamma(\mathcal{I}^-)$. Ahora bien, como $\gamma(\mathcal{I}^-) \subseteq \mathcal{I}$, concluimos que $[\![e]\!] \in \mathcal{I}$.

Por otro lado, como $T_{\mathcal{R}}^{\sharp\varphi} \propto T_{\mathcal{R}}^{\varphi}$ (Lema 5.3.7) entonces $T_{\mathcal{R}}^{\varphi}(\gamma(\mathcal{I}^+)) \subseteq \gamma(T_{\mathcal{R}}^{\sharp\varphi}(\mathcal{I}^+)$. Ahora bien, ya que $[\![e]\!] \notin \gamma(T_{\mathcal{R}}^{\sharp\varphi}(\mathcal{I}^+))$ entonces $[\![e]\!] \notin T_{\mathcal{R}}^{\varphi}(\gamma(\mathcal{I}^+))$. Finalmente, ya que $\gamma(\mathcal{I}^+) \supseteq \mathcal{I}$, y obtenemos el resultado deseado $[\![e]\!] \notin T_{\mathcal{R}}^{\varphi}(\mathcal{I})$.

□

Es importante clarificar que el uso de γ en los resultados anteriores [Alpuente *et al.*, 2001a] no nos impide tener una metodología finita para la depuración, puesto que la comprobación de que $e([\![e]\!])$ no está en $\gamma(\mathcal{I}^+)$ (resp. $\gamma(T_{\mathcal{R}}^{\sharp\varphi}(\mathcal{I}^+))$ puede ser implementado de forma eficaz y segura, mediante un simple test, verificar si la ecuación considerada unifica abstractamente con algún elemento de $\mathcal{I}^+$o $T_{\mathcal{R}}^{\sharp\varphi}(\mathcal{I}^+)$, y realizando a continuación unas comprobaciones triviales en el *mgu* abstracto.

Denotamos por $\mathcal{I}$ el programa que especifica la semántica deseada. En adelante, consideramos $\mathcal{I}^+ = lfp(T_{\mathcal{I}}^{\sharp})$, es decir consideramos el conjunto de éxitos abstractos que definimos en la sección previa como la sobreaproximación del conjunto de éxitos del programa. Para definir la subaproximación de $\mathcal{I}$, podemos simplemente quedarnos con el conjunto que resulta de un número finito de iteraciones del operador $T_{\mathcal{I}}$ concreto. Esto proporciona un esquema simple para la depuración que es satisfactorio en la práctica y lo fijamos así para efectos de implementación y a manera de proporcionar una instancia de nuestro marco. El siguiente ejemplo muestra nuestra técnica. Dicho ejemplo se ha obtenido en una sesión de depuración real con el sistema experimental BUGGY [Alpuente *et al.*, 2001b].

Ejemplo 27 *Consideremos el programa* $\mathcal{R} =$

$$\begin{array}{lcl}
\{\mathtt{g(0)} & \rightarrow & \mathtt{0}, \\
\mathtt{f(0)} & \rightarrow & \mathtt{0}, \\
\mathtt{g(c(x))} & \rightarrow & \mathtt{g(x)}, \\
\mathtt{f(c(x))} & \rightarrow & \mathtt{x} \Leftarrow \mathtt{g(c(x))} = \mathtt{c(x)}\}
\end{array}$$

incorrecto al considerar como especificación de la semántica deseada el siguiente programa $\mathcal{I} =$

$$\begin{array}{lcl}
\{\mathtt{g(0)} & \to & \mathtt{0}, \\
\mathtt{f(0)} & \to & \mathtt{0}, \\
\mathtt{g(c(x))} & \to & \mathtt{c(g(x))}, \\
\mathtt{f(c(x))} & \to & \mathtt{g(x)}\}
\end{array}$$

Entonces, por la utilización del grafo de dependencias funcionales del Ejemplo 26 tenemos $\mathcal{I}^\sharp =$

$$\begin{array}{lcl}
\{\mathtt{g(0)} & \to & \mathtt{0}, \\
\mathtt{f(0)} & \to & \mathtt{0}, \\
\mathtt{g(c(x))} & \to & \mathtt{c(\sharp)}, \\
\mathtt{f(c(x))} & \to & \mathtt{g(x)}\}
\end{array}$$

Después de tres iteraciones del operador concreto $T_{\mathcal{I}}$*, tenemos que:*

$$\begin{array}{rcl}
\mathcal{I}^- & = & \{\mathtt{0=0},\ \mathtt{c(x)=c(x)},\ \mathtt{f(x)=f(x)},\ \mathtt{g(x)=g(x)},\ \mathtt{f(0)=0},\ \mathtt{g(0)=0}, \\
& & \mathtt{f(c(x))=g(x)},\ \mathtt{g(c(x))=c(g(x))},\ \mathtt{f(c(0))=0},\ \mathtt{g(c(0))=c(0)}, \\
& & \mathtt{f(c^2(x))=c(g(x))},\ \mathtt{g(c^2(x))=c^2(g(x))},\ \mathtt{f(c^2(0))=c(0)}, \\
& & \mathtt{f(c^3(x))=c^2(g(x))},\ \mathtt{g(c^2(0))=c^2(0)},\ \mathtt{g(c^3(x))=c^3(g(x))}\}
\end{array}$$

Después de dos iteraciones del operador $T_{\mathcal{I}}^\sharp$*, obtenemos el punto fijo:*

$$\begin{array}{rcl}
\mathcal{I}^+ = \mathcal{F}^\sharp(\mathcal{I}) = lfp(\mathcal{T}_{\mathcal{I}}^\sharp) & = & \{\mathtt{0=0},\ \mathtt{c(x)=c(x)},\ \mathtt{f(x)=f(x)},\ \mathtt{g(x)=g(x)},\ \mathtt{f(0)=0}, \\
& & \mathtt{g(0)=0},\ \mathtt{f(c(x))=g(x)},\ \mathtt{g(c(x))=c(\sharp)},\ \mathtt{f(c(0))=0}, \\
& & \mathtt{f(c^2(x))=c(\sharp)}\}
\end{array}$$

y para determinar las ecuaciones incorrectas tenemos

$$\begin{array}{rcl}
T_{\mathcal{R}}(\mathcal{I}^-) & = & \{\mathtt{0=0},\ \mathtt{c(x)=c(x)},\ \mathtt{f(x)=f(x)},\ \mathtt{g(x)=g(x)},\ \mathtt{f(0)=0},\ \mathtt{g(0)=0}, \\
& & \mathtt{g(c(x))=g(x)},\ \mathtt{f(c(0))=0},\ \mathtt{f(c^2(x))=g(x)},\ \mathtt{g(c(0))=c(0)}, \\
& & \mathtt{g(c^2(x))=c(g(x))},\ \mathtt{f(c^2(0))=c(0)},\ \mathtt{f(c^3(x))=c(g(x))}, \\
& & \mathtt{g(c^2(0))=c^2(0)},\ \mathtt{f(c^3(0))=c^2(0)}, \mathtt{f(c^4(x))=c^2(g(x))}, \\
& & \mathtt{g(c^3(x))=c^2(g(x))},\ \mathtt{g(c^3(0))=c^3(0)},\ \mathtt{g(c^4(x))=c^3(g(x))}\}.
\end{array}$$

Se observa que $\mathtt{g(c(x))=g(x)} \in \mathcal{T}_{\mathcal{R}}(\mathcal{I}^-)$*, mientras* $\mathtt{g(c(x))=g(x)} \notin \mathcal{I}^+$*. Por lo tanto, la correspondiente regla* $\mathtt{g(c(x))} \to \mathtt{g(x)}$ *es incorrecta.*

Ejemplo 28 *Consideremos ahora el siguiente programa (incorrecto)* $\mathcal{R}$ *para calcular el doble.*

$$\begin{array}{rcl}
\mathtt{add(0,x)} & \to & \mathtt{x}. \\
\mathtt{add(s(x),y)} & \to & \mathtt{s(add(x,y))}. \\
\mathtt{double(0)} & \to & \mathtt{0}. \\
\mathtt{double(s(x))} & \to & \mathtt{double(x)}.
\end{array}$$

La especificación de la semántica deseada está dada por el siguiente programa que utiliza la suma de naturales para calcular el doble:

$$\begin{array}{rcl} \mathtt{add}(\mathtt{0},\mathtt{x}) & \to & \mathtt{x}. \\ \mathtt{add}(\mathtt{s}(\mathtt{x}),\mathtt{y}) & \to & \mathtt{s}(\mathtt{add}(\mathtt{x},\mathtt{y})). \\ \mathtt{double}(\mathtt{x}) & \to & \mathtt{add}(\mathtt{x},\mathtt{x}). \end{array}$$

Sea $\varphi = out$; entonces $\mathcal{I}^\sharp =$

$$\begin{array}{rcl} \mathtt{add}(\mathtt{0},\mathtt{x}) & \to & \mathtt{x}. \\ \mathtt{add}(\mathtt{s}(\mathtt{x}),\mathtt{y}) & \to & \mathtt{s}(\sharp). \\ \mathtt{double}(\mathtt{x}) & \to & \mathtt{add}(\mathtt{x},\mathtt{x}). \\ \mathtt{0} \approx \mathtt{0} & \to & \mathtt{true}. \\ \mathtt{s}(\mathtt{x}) \approx \mathtt{s}(\mathtt{y}) & \to & \sharp. \\ \mathtt{add}(\mathtt{x},\mathtt{y}) & \to & \bot. \\ \mathtt{double}(\mathtt{x}) & \to & \bot. \\ \mathtt{x} \approx \mathtt{y} & \to & \bot. \end{array}$$

La sobreaproximación $\mathcal{I}^+$ esta dada por el siguiente conjunto de ecuaciones (después de tres iteraciones del operador $T_{\mathcal{I}}^{out\sharp}$, obtenemos el punto fijo):

$$\begin{array}{rcl} \mathcal{I}^+ = \mathcal{F}_{out}^\sharp(\mathcal{I}) = lfp(T_{\mathcal{I}}^{out\sharp}) & = & \{\mathtt{double}(\mathtt{0}) = \mathtt{0},\ \mathtt{double}(\mathtt{s}(\mathtt{x})) = \mathtt{s}(\sharp),\ \mathtt{add}(\mathtt{x},\mathtt{y}) = \bot, \\ & & \mathtt{double}(\mathtt{x}) = \bot,\ (\mathtt{x} \approx \mathtt{y}) = \bot,\ (\mathtt{s}(\mathtt{x}) \approx \mathtt{s}(\mathtt{y})) = \sharp, \\ & & (\mathtt{0} \approx \mathtt{0}) = \mathtt{true},\ \mathtt{add}(\mathtt{0},\mathtt{x}) = \mathtt{x}, \mathtt{double}(\mathtt{x}) = \mathtt{add}(\mathtt{x},\mathtt{x}), \\ & & \mathtt{add}(\mathtt{s}(\mathtt{x}),\mathtt{y}) = \mathtt{s}(\sharp),\ \mathtt{add}(\mathtt{x},\mathtt{y}) = \mathtt{add}(\mathtt{x},\mathtt{y}),\ \mathtt{0} \approx \mathtt{0}, \\ & & \mathtt{s}(\mathtt{x}) \approx \mathtt{s}(\mathtt{x}), \mathtt{double}(\mathtt{x}) = \mathtt{double}(\mathtt{x}),\ (\mathtt{x} \approx \mathtt{y}) = (\mathtt{x} \approx \mathtt{y})\} \end{array}$$

Después de cuatro iteraciones del operador $T_{\mathcal{I}}^{out}$, obtenemos la subaproximación

$$\begin{array}{rcl} \mathcal{I}^- & = & \{\mathtt{add}(\mathtt{x},\mathtt{y}) = \mathtt{add}(\mathtt{x},\mathtt{y}),\ \mathtt{0} \approx \mathtt{0},\ \mathtt{s}(\mathtt{x}) \approx \mathtt{s}(\mathtt{x}),\ \mathtt{double}(\mathtt{x}) = \mathtt{double}(\mathtt{x}), \\ & & (\mathtt{x} \approx \mathtt{y}) = (\mathtt{x} \approx \mathtt{y}),\ (\mathtt{x} \approx \mathtt{y}) = \bot,\ (\mathtt{s}(\mathtt{x}) \approx \mathtt{s}(\mathtt{y})) = \bot,\ (\mathtt{s}^2(\mathtt{x}) \approx \mathtt{s}^2(\mathtt{y})) = \bot, \\ & & (\mathtt{s}^3(\mathtt{x}) \approx \mathtt{s}^3(\mathtt{y})) = \bot,\ (\mathtt{s}(\mathtt{x}) \approx \mathtt{s}(\mathtt{y})) = (\mathtt{x} \approx \mathtt{y}),\ (\mathtt{s}^2(\mathtt{x}) \approx \mathtt{s}^2(\mathtt{y})) = (\mathtt{x} \approx \mathtt{y}), \\ & & (\mathtt{s}^3(\mathtt{x}) \approx \mathtt{s}^3(\mathtt{y})) = (\mathtt{x} \approx \mathtt{y}),\ (\mathtt{s}^4(\mathtt{x}) \approx \mathtt{s}^4(\mathtt{y})) = (\mathtt{x} \approx \mathtt{y}),\ (\mathtt{0} \approx \mathtt{0}) = \mathtt{true}, \\ & & (\mathtt{s}(\mathtt{0}) \approx \mathtt{s}(\mathtt{0})) = \mathtt{true},\ (\mathtt{s}^2(\mathtt{0}) \approx \mathtt{s}^2(\mathtt{0})) = \mathtt{true},\ (\mathtt{s}^3(\mathtt{0}) \approx \mathtt{s}^3(\mathtt{0})) = \mathtt{true}, \\ & & \mathtt{add}(\mathtt{x},\mathtt{y}) = \bot,\ \mathtt{add}(\mathtt{s}(\mathtt{x}),\mathtt{y}) = \mathtt{s}(\bot),\ \mathtt{add}(\mathtt{s}^2(\mathtt{x}),\mathtt{y}) = \mathtt{s}^2(\bot), \mathtt{add}(\mathtt{s}^3(\mathtt{x}),\mathtt{y}) = \mathtt{s}^3(\bot), \\ & & \mathtt{add}(\mathtt{0},\mathtt{x}) = \mathtt{x},\ \mathtt{add}(\mathtt{s}(\mathtt{0}),\mathtt{x}) = \mathtt{s}(\mathtt{x}),\ \mathtt{add}(\mathtt{s}^2(\mathtt{0}),\mathtt{x}) = \mathtt{s}^2(\mathtt{x}),\ \mathtt{add}(\mathtt{s}^3(\mathtt{0}),\mathtt{y}) = \mathtt{s}^3(\mathtt{y}), \\ & & \mathtt{add}(\mathtt{s}(\mathtt{x}),\mathtt{y}) = \mathtt{s}(\mathtt{add}(\mathtt{x},\mathtt{y})),\ \mathtt{add}(\mathtt{s}^2(\mathtt{x}),\mathtt{y}) = \mathtt{s}^2(\mathtt{add}(\mathtt{x},\mathtt{y})), \\ & & \mathtt{add}(\mathtt{s}^3(\mathtt{x}),\mathtt{y}) = \mathtt{s}^3(\mathtt{add}(\mathtt{x},\mathtt{y})),\ \mathtt{add}(\mathtt{s}^4(\mathtt{x}),\mathtt{y}) = \mathtt{s}^4(\mathtt{add}(\mathtt{x},\mathtt{y})),\ \mathtt{double}(\mathtt{x}) = \bot, \\ & & \mathtt{double}(\mathtt{s}(\mathtt{x})) = \mathtt{s}(\bot),\ \mathtt{double}(\mathtt{s}^2(\mathtt{x})) = \mathtt{s}^2(\bot),\ \mathtt{double}(\mathtt{x}) = \mathtt{add}(\mathtt{x},\mathtt{x}), \\ & & \mathtt{double}(\mathtt{s}(\mathtt{x})) = \mathtt{s}(\mathtt{add}(\mathtt{x},\mathtt{s}(\mathtt{x}))),\ \mathtt{double}(\mathtt{s}^2(\mathtt{x})) = \mathtt{s}^2(\mathtt{add}(\mathtt{x},\mathtt{s}^2(\mathtt{x}))), \\ & & \mathtt{double}(\mathtt{s}^3(\mathtt{x})) = \mathtt{s}^3(\mathtt{add}(\mathtt{x},\mathtt{s}^3(\mathtt{x}))), \mathtt{double}(\mathtt{0}) = \mathtt{0},\ \mathtt{double}(\mathtt{s}(\mathtt{0})) = \mathtt{s}(\mathtt{s}(\mathtt{0})), \\ & & \mathtt{double}(\mathtt{s}^2(\mathtt{0})) = \mathtt{s}^4(\mathtt{0})\} \end{array}$$

Entonces,

$$
\begin{aligned}
T_{\mathcal{R}}(\mathcal{I}^-) \;=\; & \{\mathtt{0} \approx \mathtt{0},\ \mathtt{s(x)} \approx \mathtt{s(x)},\ \mathtt{add(x,y)} = \mathtt{add(x,y)},\ \mathtt{double(x)} = \mathtt{double(x)},\\
& \mathtt{aprox(x,y)} = \mathtt{aprox(x,y)},\ (\mathtt{0} \approx \mathtt{0}) = \mathtt{true},\ (\mathtt{s(0)} \approx \mathtt{s(0)}) = \mathtt{true},\\
& (\mathtt{s^2(0)} \approx \mathtt{s^2(0)}) = \mathtt{true},\ (\mathtt{s^3(0)} \approx \mathtt{s^3(0)}) = \mathtt{true},\ (\mathtt{s^4(0)} \approx \mathtt{s^4(0)}) = \mathtt{true},\\
& (\mathtt{s(x)} \approx \mathtt{s(y)}) = (\mathtt{x} \approx \mathtt{y}),\ (\mathtt{s^2(x)} \approx \mathtt{s^2(y)}) = (\mathtt{x} \approx \mathtt{y}),\\
& (\mathtt{s^3(x)} \approx \mathtt{s^3(y)}) = (\mathtt{x} \approx \mathtt{y}),\ (\mathtt{s^4(x)} \approx \mathtt{s^4(y)}) = (\mathtt{x} \approx \mathtt{y}),\\
& (\mathtt{s^5(x)} \approx \mathtt{s^5(y)}) = (\mathtt{x} \approx \mathtt{y}),\ \mathtt{double(s(x))} = \mathtt{double(x)},\\
& \mathtt{double(s(x))} = \mathtt{s(add(x,s(x)))},\ \mathtt{double(s^2(x))} = \mathtt{s^2(add(x,s^2(x)))},\\
& \mathtt{double(s^3(x))} = \mathtt{s^3(add(x,s^3(x)))},\ \mathtt{double(s^4(x))} = \mathtt{s^4(add(x,s^4(x)))},\\
& \mathtt{double(0)} = \mathtt{0},\ \mathtt{double(s(0))} = \mathtt{s^2(0)},\ \mathtt{double(s^2(0))} = \mathtt{s^4(0)},\\
& \mathtt{double(s^3(0))} = \mathtt{s^6(0)},\ \mathtt{add(s(x),y)} = \mathtt{s(add(x,y))},\\
& \mathtt{add(s^2(x),y)} = \mathtt{s^2(add(x,y))},\ \mathtt{add(s^3(x),y)} = \mathtt{s^3(add(x,y))},\\
& \mathtt{add(s^4(x),y)} = \mathtt{s^4(add(x,y))},\ \mathtt{add(s^5(x),y)} = \mathtt{s^5(add(x,y))},\\
& \mathtt{add(0,x)} = \mathtt{x},\ \mathtt{add(s(0),x)} = \mathtt{s(x)},\ \mathtt{add(s^2(0),x)} = \mathtt{s^2(x)},\\
& \mathtt{add(s^3(0),y)} = \mathtt{s^3(y)},\ \mathtt{add(s^4(0),y)} = \mathtt{s^4(y)}\}
\end{aligned}
$$

El sistema detecta que la regla del programa $\mathtt{double(s(x))} = \mathtt{double(x)}$ *es incorrecta. Si el programador la reemplaza por la regla correcta* $\mathtt{double(s(x))} = \mathtt{s(s(double(x)))}$*, obtenemos el programa correcto y completo, de acuerdo con nuestras condiciones.*

Ejemplo 29 *Consideremos el programa que genera la lista con las constantes a, b de longitud menor o igual que N:*

$$
\begin{aligned}
\mathtt{gen(N,[a])} &\rightarrow \mathtt{[]}.\\
\mathtt{gen(N,L)} &\rightarrow \mathtt{L} \Leftarrow \mathtt{L} = \mathtt{[elem(X)|L1]}, \mathtt{N} = \mathtt{s(N1)}, \mathtt{gen(N1,L1)} = \mathtt{L1}.\\
\mathtt{element(X)} &\rightarrow \mathtt{a}.
\end{aligned}
$$

Es incorrecto en la primera cláusula y existe una ecuación no cubierta. La especificación $\mathcal{I}$ *está dada por el siguiente programa:*

$$
\begin{aligned}
\mathtt{gen(N,[])} &\rightarrow \mathtt{[]}.\\
\mathtt{gen(N,L)} &\rightarrow \mathtt{L} \Leftarrow \mathtt{L} = \mathtt{[elem(X)|L1]}, \mathtt{N} = \mathtt{s(N1)}, \mathtt{gen(N1,L1)} = \mathtt{L1}.\\
\mathtt{element(X)} &\rightarrow \mathtt{a}.\\
\mathtt{element(X)} &\rightarrow \mathtt{b}.
\end{aligned}
$$

Al ejecutar "BUGGY" con dos iteracciones obtenemos los siguientes resultados. En primer lugar, la sobreaproximación $\mathcal{I}^+$ *está dada por el siguiente conjunto de ecuaciones:*

$$\begin{aligned}\mathcal{I}^{+} \;=\; & \{\texttt{gen(s(N),[a|X])} = \texttt{[a|X]},\ \texttt{gen(s(N),[b|Y])} = \texttt{[b|Y]},\ \texttt{elem(X)} = \texttt{b},\\ & \texttt{elem(X)} = \texttt{a},\ \texttt{gen(s(N),[elem(Y)|Z])} = \texttt{[elem(Y)|Z]},\ \texttt{gen(N,[\,])} = [\,],\ \texttt{b} = \texttt{b},\\ & \texttt{a} = \texttt{a},\ \texttt{s(X)} = \texttt{s(X)},\ \texttt{elem(X)} = \texttt{elem(X)},\ \texttt{[X|Y]} = \texttt{[X|Y]},\ \texttt{gen(X,Y)} = \texttt{gen(X,Y)},\\ & [\,] = [\,]\}.\end{aligned}$$

La correspondiente subespecificación $\mathcal{I}^{-}$:

$$\begin{aligned}\mathcal{I}^{-} \;=\; & \{\texttt{gen(s(N),[elem(X)])} = \texttt{[elem(X)]},\ \texttt{gen(s(N),[a])} = \texttt{[a]},\ \texttt{gen(s(N),[b])} = \texttt{[b]},\\ & \texttt{elem(X)} = \texttt{b},\ \texttt{elem(X)} = \texttt{a},\ \texttt{gen(N,[\,])} = [\,],\ \texttt{b} = \texttt{b},\ \texttt{a} = \texttt{a},\ \texttt{s(X)} = \texttt{s(X)},\\ & \texttt{elem(X)} = \texttt{elem(X)},\ \texttt{[X|Y]} = \texttt{[X|Y]},\ \texttt{gen(X,Y)} = \texttt{gen(X,Y)},\ [\,] = [\,]\}.\end{aligned}$$

El sistema detecta que la regla $\texttt{gen(N,[a])} = [\,]$ *es incorrecta. Si el programador la reemplaza por la regla* $\texttt{gen(N,[a])} = [\,]$*, el programa es correcto y ahora el sistema detecta que la ecuación* $\texttt{gen(s(N),[b])} = \texttt{[b]}$ *es no cubierta. Ahora introducimos la regla* $\texttt{elem(X)} = \texttt{b}$ *y el programa es correcto y completo.*

En la siguiente sección, presentamos una estrategia para la corrección de errores que intenta modificar los componentes erróneas del código inicial e inferir el programa correcto. De este modo, mostramos como este mecanismo puede incorporarse dentro de nuestro método de diagnóstico para formar un sistema de depuración práctico.

6.2 Corrección de programas

La Inducción de Programas Lógicos (IPL) es el campo del *Aprendizaje Automático* (conocido habitualmente como *Machine Learning* dentro de la Inteligencia Artificial), relacionado con las tarea del aprendizaje de programas lógicos a partir de ejemplos positivos y negativos, generalmente literales básicos [Muggleton y de Raedt, 1994]. El prometedor campo de la Inducción de Programas Lógicos (IPL) conocido como *revisión inductiva de teorías* se relaciona directamente con la depuración de programas bajo la asunción del *programador competente* de [Shaphiro, 1982]. En general, se asume que el programa inicial está escrito con la intención de ser correcto y, si no lo es, entonces una variante suya lo es. La técnica de depuración que desarrollamos en este capítulo intenta encontrar dicha variante.

Formalmente, en nuestro enfoque de ILP para la depuración, se parte de una hipótesis inicial $\mathcal{R}$, bajo la restricción de que la hipótesis final $\mathcal{R}^c$ se aproxima tanto al programa pretendido como sea posible, en el sentido de que sólo los errores de $\mathcal{R}$ deben ser detectados, localizados y corregidos, para obtener $\mathcal{R}^c$. De este modo, asumimos que cada componente del programa aparece allí por alguna razón. Esto implica que si una parte del código se detecta como incorrecta, no podemos simplemente suprimirla; por lo contrario, la corregimos sin perjudicar la parte del código que es correcta. Se debe notar que nuestro enfoque tiene una diferencia importante

con respecto a [Shaphiro, 1982] en el sentido en que no le exigimos al usuario que interactúe con el depurador para proporcionar ejemplos de evidencias, ni responder preguntas sobre la corrección, establecer las clases de la equivalencia entre las reglas, o corregir manualmente el código.

La búsqueda automática de una nueva regla en el proceso de la inducción puede realizarse de forma ascendente (es decir, de una regla más específica a una más general) o de forma descendente (es decir, de una regla más general a una más específica). Nosotros seguimos un enfoque deductivo que se conoce como el *desplegado guiado por ejemplos* [Bostrom y Idestam-Alquist, 1999], que usa el desplegado como operador de especialización, orientada a la discriminación entre los ejemplos positivos y negativos.

El problema puede formularse como sigue:

Dado : un programa $\mathcal{R}$, tal que $\mathcal{R}' \subseteq \mathcal{R}$ es incorrecto con respecto a la especificación deseada $\mathcal{I}$.

Hallar : un conjunto de reglas $\mathcal{X}$ tal que el programa revisado $\mathcal{R}^c = (\mathcal{R} \setminus \mathcal{R}') \cup \mathcal{X}$ es (parcialmente) correcto con respecto a $(\mathcal{I}^+, \mathcal{I}^-)$, donde $\mathcal{I}^+$ e $\mathcal{I}^-$ son respectivamente una sobreaproximación y subaproximación de la semántica deseada para $\mathcal{R}$. El programa revisado $\mathcal{R}^c$ se denomina programa *correcto*.

6.2.1 Desplegado guiado por ejemplos

Una regla incorrecta de un programa $\mathcal{R}$ cubre un conjunto E^+ de ejemplos positivos (ecuaciones que pertenecen a la semántica deseada $\mathcal{I}$) y un conjunto E^- de ejemplos negativos (ecuaciones que no pertenecen a $\mathcal{I}$). Lo que necesitamos es alguna forma de *adivinar* un conjunto de reglas correctas que cubran las refutaciones de todas las ecuaciones de E^+ y ninguna ecuación de E^-. Por ejemplo, sea E^+ el conjunto formado por

$$\{\texttt{even(0)} = \texttt{true},\ \texttt{even(s}^2\texttt{(0))} = \texttt{true},\ \texttt{even(s}^4\texttt{(0))} = \texttt{true}, \ldots\}$$

y sea E^- el conjunto que contiene

$$\{\texttt{even(s(0))} = \texttt{true},\ \texttt{even(s}^3\texttt{(0))} = \texttt{true},\ \texttt{even(s}^5\texttt{(0))} = \texttt{true}, \ldots\}.$$

Entonces el programa

$$\{\texttt{even(0)} = \texttt{true},\ \texttt{even(s}^2\texttt{(x))} = \texttt{even(x)}\}$$

debe considerarse correcto.

El desplegado guiado por ejemplos [Alexin *et al.*, 1996; Bostrom y Idestam-Alquist, 1999] normalmente se aplica para especializar el programa incorrecto $\mathcal{R}$ de tal manera que se excluyan los ejemplos negativos sin excluir los positivos [Bostrom y Idestam-Alquist, 1999]. La idea fundamental del método es como sigue. Especializamos una regla incorrecta r de $\mathcal{R}$ desplegando uno de las llamadas en la parte derecha de la cabeza o del cuerpo de la regla. Posteriormente, eliminamos algunas reglas nuevas

del programa que tienen que ver únicamente con los ejemplos negativos. Esperamos que después de repetir varias veces los dos pasos previos, podamos encontrar las reglas que pueden eliminarse del programa sin destruir el comportamiento sobre los ejemplos positivos. La motivación fundamental del método, formalmente introducido en [Bostrom y Idestam-Alquist, 1994], es la siguiente:

- El desplegado elimina algunos nodos internos del árbol de especialización. Con ello procuramos separar los ejemplos positivos de los negativos, y también acercarlos más a la raíz del árbol.
- Si un ejemplo negativo cuelga directamente de la raíz del árbol de especialización, y su correspondiente regla de entrada r no se usa en otra parte del árbol para un ejemplo positivo, entonces el programa puede especializarse anulando r

Ilustramos esto con un ejemplo.

Ejemplo 30 *Consideremos el programa $\mathcal{R}$, que contiene las siguientes reglas*

$$\mathcal{R} = \{\mathtt{f(x)} = \mathtt{g(x)},\ \mathtt{g(a)} = \mathtt{a},\ \mathtt{g(b)} = \mathtt{a}\}.$$

Sea la especificación de la semántica deseada

$$\mathcal{I} = \{\mathtt{f(a)} = \mathtt{a},\ \mathtt{g(a)} = \mathtt{a},\ \mathtt{g(b)} = \mathtt{a}\}.$$

Por lo tanto, de acuerdo con nuestro método de diagnóstico abstracto del Capítulo 6, la regla $r : \mathtt{f(x)} = \mathtt{g(x)}$ es incorrecta. Entonces podemos fijar $E^+ = \{\mathtt{f(a)} = \mathtt{a}\}$, y $E^- = \{\mathtt{f(b)} = \mathtt{a}\}$. Como r cubre tanto ejemplos positivos como negativos, aplicamos un paso de desplegado. Desplegando r sobre $\mathtt{g(x)}$ reemplazamos la regla con el siguiente conjunto de reglas:

$$\mathcal{X} = \{\mathtt{f(a)} = \mathtt{a},\ \mathtt{f(b)} = \mathtt{a}\}.$$

La primera regla cubre el ejemplo positivo únicamente, mientras que la segunda cubre un ejemplo negativo. Ya que el ejemplo negativo esta conectado a la raíz del árbol de narrowing para $\mathtt{f(x)}$ en el programa $\mathcal{R} \setminus \{r\} \cup \mathcal{X}$, puede eliminarse borrando, simplemente, la regla $\mathtt{f(b)} = \mathtt{a}$ del programa que no afecta el ejemplo positivo.

En el resto de esta sección particularizamos este método a nuestro marco de la depuración.

6.2.2 Selección del conjunto de ejemplos

Cuando un ejemplo negativo está cubierto por la versión más reciente del programa, hay al menos una regla que es responsable del cubrimiento incorrecto. Para motivar nuestro método, supongamos que, en el primer paso, eliminamos las reglas incorrectas de $\mathcal{R}$. Mediante esto, obtenemos de manera inmediata un programa parcialmente

correcto $\mathcal{R}^-$, que no contiene reglas incorrectas. Sin embargo, podría ser incompleto con respecto a $(\mathcal{I}^+, \mathcal{I}^-)$ ya que pueden existir ecuaciones cubiertas por $\mathcal{I}$, pero no por $\mathcal{R}^-$. Las ecuaciones de este tipo son ejemplos positivos razonables, debido a que la corrección computada tiene que cubrirlos. Por consiguiente definimos el conjunto finito

$$E^+ = \{e \mid e \in \mathcal{I}^- \text{ y } [\![e]\!] \notin \gamma(T^{\sharp\varphi}_{\mathcal{R}^-}(\mathcal{I}^+)\}.$$

Similarmente, E^- define el conjunto de ecuaciones e el cual, de acuerdo con la Proposición 6.1.1, permite al depurador probar que cada regla r de $\mathcal{R}'$ es incorrecta (es decir, todas las ecuaciones e tal que $e \in T^{\varphi}_{\{r\}}([\![\mathcal{I}^-]\!])$ y $e \notin \gamma(\mathcal{I}^+)$)

6.2.3 Algoritmo de corrección

En el contexto de los programas lógico funcionales, una forma natural de especializar los programas es usar una forma de narrowing conducido por desplegado, es decir la expansión, por medio de *narrowing*, de las subexpresiones del programa utilizando las correspondientes definiciones (ver [Alpuente *et al.*, 1998] para una descripción completa). Una caracterización completa del desplegado con respecto a las respuestas computadas en los lenguajes lógico funcionales con semántica voraz o perezosa puede encontrarse en [Alpuente *et al.*, 1997, 1999a]. Siguiendo a [Bostrom y Idestam-Alquist, 1999], utilizamos el operador del desplegado para la corrección del programa como sigue.

Presentamos la base de un algoritmo que especializa los programas con respecto a los conjuntos de ejemplos positivos y negativos, aplicando la transformación de la *regla de desplegado* junto con la *regla de borrado*. El (esquema principal del) algoritmo, para la corrección de programas, se describe en la Figura 6.1. El procedimiento, inspirado en [Alexin *et al.*, 1996], se utiliza para producir (con algún conjunto extra necesario para especializar las definiciones recursivas [Bostrom y Idestam-Alquist, 1999]) una especialización correcta, cuando el programa cubre originalmente todos los ejemplos positivos y algunos incorrectos. El algoritmo contiene un ciclo principal que continúa hasta que ningún ejemplo negativo esté cubierto incorrectamente por las reglas del programa. Informalmente, tan pronto como se encuentre una regla en el programa que cubre un ejemplo negativo, se verifica si cubre algún ejemplo positivo o no. Si no cubre ningún ejemplo positivo, entonces la regla se elimina. De otro modo, se especializa con respecto al conjunto de los ejemplos positivos y negativos que se construyen tal como hemos descrito anteriormente. De este modo, las reglas incorrectas se dividen en varias reglas, que al unirlas, son equivalentes a la regla que se ha dividido. Entonces, el proceso se repite para aquellas reglas que aún cubren los ejemplos negativos, hasta que se alcanza una condición de terminación.

Ejemplo 31 *Supongamos que el programa* `even/1` *se define del siguiente modo (no-*

Entrada: $\mathcal{R}$, $\mathcal{I}$, y los conjuntos de ejemplos E^+, E^-
Salida: programa $\mathcal{R}_c$ (una especialización de $\mathcal{R}$ con respecto a E^+ y E^-).
Inicialización $k := 0$; $\mathcal{R}_k := \mathcal{R}$.
Mientras que exista $e^- \in E^-$ tal que $\mathcal{R}_k$ no falla sobre e^- **hacer**
    Desplegar la regla incorrecta $r \in \mathcal{R}_k$
    Sea $\mathcal{X}$ el conjunto de resultantes de un paso de r;
        Eliminar de $\mathcal{X}$ todas aquellas reglas que no ocurran
        en refutaciones de las ecuaciones de E^+
    Sea $\mathcal{R}_{k+1} = \mathcal{R}_k \setminus \{r\} \cup \mathcal{X}$
    Sea $k := k + 1$
Fin Mientras
Devolver $\mathcal{R}_k$

Figura 6.1: Algoritmo base para la corrección de programas

te que el error en este fragmento causa que el programa compute cualquier número natural en lugar de un número par):

$$\texttt{even(0)} = \texttt{true}. \qquad \texttt{even(s(x))} = \texttt{even(x)}.$$

junto con el los siguientes ejemplos positivos y negativos:

$$\begin{array}{rcl} E^+ & = & \{\texttt{even(0)} = \texttt{true},\ \texttt{even(s(s(0)))} = \texttt{true}\} \\ E^- & = & \{\texttt{even(s(0))} = \texttt{true}\} \end{array}$$

La regla incorrecta que origina el ejemplo negativo es la segunda regla en la definición. Según nuestro algoritmo de corrección, la regla se despliega y luego es eliminada, ya que no se usa para refutar ningún otro ejemplo negativo. Obtenemos lo que deseamos, el programa correcto

$$\texttt{even(0)} = \texttt{true} \qquad \texttt{even(s(s(x))} = \texttt{even(x)}$$

Desafortunadamente, el desplegado y la regla de borrado no son generalmente suficientes para un método de corrección de programas completo: algunos problemas de especialización no pueden resolverse de este modo debido a que los ejemplos positivos y negativos no se diferencian fácilmente, tal como se ilustra en el siguiente ejemplo.

Ejemplo 32 *Consideremos de nuevo el programa $\mathcal{R}$, que consiste en el conjunto de reglas que definen la adición de números naturales junto con la regla*

$$\{\texttt{double(x)} = \texttt{x} + \texttt{0}\}.$$

Sea la especificación deseada $\mathcal{I}$, el programa que contiene las reglas para la adición junto con las reglas

$$\{\texttt{double(0)} = \texttt{0},\ \ \texttt{double(s(x))} = \texttt{s(s(double(x)))}\}.$$

La regla $\{\texttt{double(x)} = \texttt{x} + \texttt{0}\}$ *cubre un ejemplo positivo* $\texttt{double(0)} = \texttt{0}$ *(note que no cubre otros), y un ejemplo negativo:* $\texttt{double(s(x))} = \texttt{s(x)}$. *Al desplegar, obtenemos la especialización*

$$\{\texttt{double(0)} = \texttt{0}, \quad \texttt{double(s(x))} = \texttt{s(x + 0)}\}.$$

Ahora, la segunda regla cubre únicamente el ejemplo negativo, entonces es eliminada; como consecuencia, no es posible inferir la regla correcta $\texttt{double(x)} = \texttt{x} + \texttt{x}$ *mediante el método de desplegado guiado por ejemplos.*

Análogamente a lo que es habitual en otras metodologías basadas en plegado y desplegado, es necesario algún mecanismo de generalización de las llamadas para mejorar la potencia del método. Esto ya se había planteado en [Bostrom y Idestam-Alquist, 1994, 1999], dónde se propuso una solución a este problema. En el contexto de los lenguajes lógico funcionales, una definición formal de estas reglas, así como las demostraciones de la propiedad de corrección y completitud (cuando se combinan las reglas) con respecto a la semántica de respuestas computadas se encuentra en [Alpuente *et al.*, 1997, 1999a].

6.3 Implementación del depurador

En esta sección describimos el sistema Buggy, un depurador declarativo para programas lógico funcionales. El programa incluye cuatro fases. La primera corresponde a la introducción del programa incorrecto y la especificación de la semántica deseada que, para efectos de implementación, también es un programa. La segunda fase es la búsqueda de los errores en el programa y sus correcciones. La tercera fase es la búsqueda de la ecuaciones no cubiertas y sus correcciones. La última fase es la inducción de reglas para reparar los errores detectados.

El prototipo del sistema Buggy incluye un analizador para un lenguaje lógico funcional (condicional), cuya semántica está basada en: *narrowing innermost* con selección de redex más a la izquierda (*leftmost*), *narrowing básico innermost* (una estrategia mejor que no requiere que las funciones estén completamente definidas [Bosco *et al.*, 1988; Hölldobler, 1989]), *narrowing leftmost outermost* y *narrowing necesario*. Hemos supuesto que cada programa satisface los requerimientos para la corrección y completitud de la estrategia de narrowing correspondiente. La implementación también incluye un módulo para computar una abstracción del programa basada en la construcción de un grafo de prevención de bucles (en particular, el grafo de dependencias funcionales), y un depurador automático que requiere que el usuario indique algunos parámetros, tales como la estrategia de narrowing y el número n de iteraciones para aproximar el conjunto de éxitos. De este modo, los errores se encuentran automáticamente ejecutando el depurador. El usuario puede realizar la corrección

de forma automática mediante el modulo de especialización o indicar manualmente las correcciones a realizar sobre las reglas detectadas como incorrectas. En la implementación actual del sistema BUGGY, la semántica deseada $\mathcal{I}$ se considera como un programa (es decir, una especificación ejecutable). Aunque esto es simplemente una decisión de implementación que apunta a facilitar la experimentación y de ninguna manera puede ser considerado como un inconveniente del enfoque teórico. Una extensión posible es ayudar al usuario en la tarea de entrar manualmente la subaproximación $\mathcal{I}^-$ y la sobreaproximación $\mathcal{I}^+$.

Para depurar los programas bajo la estrategia de narrowing necesario seguimos la aproximación transformacional basada en Hanus y Prehofer [1999]. Esto es, los anidamientos funcionales en la partes izquierdas, lhs's, de las reglas se eliminan reemplazándolos por una "distinción de casos" en las partes derechas, rhs, de las reglas. Entonces, el programa traducido se ejecuta usando la estrategia de narrowing leftmost outermost que es fuertemente equivalente al narrowing necesario sobre el programa traducido, como lo mencionamos en la Sección 2.5.

Las características principales del sistema BUGGY son las siguientes:

1. Incluye un **analizador** que verifica y adecúa la sintaxis de las reglas del programa y la especificación relativo a la representación interna utilizada.

2. Determina una **aproximación correcta** $\mathcal{I}^+$ de la especificación del sistema de reescritura de términos.

3. Permite computar el **operador de consecuencias inmediatas** $\mathcal{T}_{\mathcal{I}}^{\sharp}$ asociado con la especificación abstracta.

4. La **sobreaproximación** $\mathcal{I}^+$ se encuentra computando el menor punto fijo del operador abstracto $\mathcal{T}_{\mathcal{I}}^{\sharp}$.

5. La **subaproximación** $\mathcal{I}^-$ se encuentra computando un número finito iteraciones del operador de consecuencias inmediato $\mathcal{T}_{\mathcal{I}}$.

6. Determina las **ecuaciones incorrectas** mediante un simple test de comparación entre los conjuntos $T_{\mathcal{R}}^{\varphi}(\mathcal{I}^-)$ e $\mathcal{I}^+$.

7. Determina las **ecuaciones no cubiertas** comparando los conjuntos $T_{\mathcal{R}}^{\sharp\varphi}(\mathcal{I}^+)$ e $\mathcal{I}^-$.

8. Para una lista dada de constructores, el sistema calcula dos conjuntos: uno con **ejemplos positivos** y el otro con **ejemplos negativos**, que permiten permiten inducir la reglas mediante el modulo para realizar la síntesis de programas.

6.3.1 Sistema BUGGY

Características del sistema

El sistema BUGGY está escrito en SICStus Prolog v3.8.1. La aplicación completa contiene aproximadamente 400 cláusulas (2100 líneas de código). Antes de comenzar, es necesario ubicar las librerías 'sweden.pl' y 'sweden.plo' en el lugar dispuesto para las librerías del Prolog. El sistema BUGGY no necesita ninguna instalación previa. Al ejecutar, el programa muestra la siguiente pantalla:

```
**********************************************************************
*** help.      --> Display menu                                      ***
*** clean.     --> Clean database of the program and specification ***
*** loadspe.   --> load specification  of the semantics from file  ***
*** listprg.   --> Display program                                   ***
*** listspe.   --> Display specification                             ***
*** debugger.  --> run debugger                                      ***
*** save.      --> Save program in a file                            ***
*** exit.      --> Leave program                                     ***
**********************************************************************
```

Su funcionalidad se describe a continuación:

- *help.* Visualiza el menú visto anteriormente.
- *clean.* Borra de la memoria del sistema los predicados dinámicos que almacenan las reglas del programa, de la especificación, las ecuaciones de $\mathcal{I}^+$ e $\mathcal{I}^-$ y otras utilizadas por el sistema.
- *loadprg.* Dado el nombre del programa a depurar, realiza una llamada al 'analizador', que transforma las reglas del programa a la representación interna. Las reglas transformadas se insertan usando predicados dinámicos.
- *loadspe.* Análogo a *loadprg*, permite introducir la especificación deseada del programa.
- *listprg.* Visualiza las reglas del programa cargado.
- *listspe.* Visualiza las reglas de la especificación deseada.
- *debugger.* Realiza el proceso de depuración. Pregunta al usuario el tipo de estrategia a utilizar entre las cuatro posibles. Seguidamente, realiza el cálculo de $\mathcal{I}^+$, muestra sus elementos y el número de iteraciones para calcularlo, seguidamente pregunta al usuario el número de iteraciones que desea para el cálculo de $\mathcal{I}^-$. Acontinuación aplica la corrección o de inducción según desee el usuario. Al finalizar del proceso de corrección visualiza el programa obtenido.

- *save*. Guarda el programa actual.
- *exit*. Provoca la salida del sistema.

La explicación detallada de los diferentes comandos y pasos para ejecutar el sistema se encuentra en el manual de usuario disponible en:

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

Analizador

En el modulo *analizador* se transforman las reglas ecuacionales del tipo $\lambda = \rho$:- $s_1 = t_1$,...,$s_n = t_n$ a reglas de la forma $equa(\lambda,rho) :- equa(s_1,t_1),\ldots,equa(s_n,t_n)$. El predicado $equa(s,t)$ representa la ecuación s=t.

Las reglas del programa transformado se insertan en un fichero que será utilizado en la siguiente fase del *analizador*, en donde cada regla del tipo $equa(\lambda,\rho) :- equa(s_1,t_1),\ldots,equa(s_n,t_n)$ se expresa como una estructura del tipo $rule(equa(\lambda,\rho), [equa(s_1,t_1),\ldots,equa(s_n,t_n)])$.

Adicionalmente, el *analizador* se utiliza también en la fase de corrección de errores del depurador, en la que el usuario tiene la posibilidad de añadir nuevas reglas. La sintaxis utilizada hace más fácil la inserción de las reglas para el usuario.

Abstracción de la especificación deseada

El grafo de dependencias funcionales $\mathcal{G}_\mathcal{R}$ se representa como una lista dinámica de pares de símbolos de función: [[f_1,g_1],...,[f_n,g_n]]. Cada [f_i,g_i] es un arco orientado del nodo f_i al nodo g_i. El arco [f_i,g_i] significa que la función f_i está definida por medio de la función la función g_i. Cada nodo se representa a su vez como una lista de dos elementos, que son el correspondiente símbolo de función y su respectiva aridad.

Para establecer si exite un ciclo (o bucle) que parte de *f*, es necesario controlar recursivamente si existe un camino *path* (*f*,*f*) que parte de *f* y, siguiendo los arcos de $\mathcal{G}_\mathcal{R}$, vuelve a *f*.

$$path(f,f) = \begin{cases} [f,f] \in \mathcal{G}_\mathcal{R} & \text{o bien} \\ [f,g] \in \mathcal{G}_\mathcal{R} & \text{si path } (g,f) \end{cases}$$

El símbolo ♯, introducido en la abstracción de la especificación, se representa con una variable *fresca*.

La idea principal es eliminar del programa aquellos términos cuyo nodo en el grafo $\mathcal{G}_\mathcal{R}$ es un bucle, reemplazándolos por ♯. Un problema en la abstracción es que, según el método que se utilice para abstraer, se puede perder más información o menos en la semántica. Nuestro algoritmo de abstracción, en el momento de eliminar los ciclos del grafo de prevención de bucles, trata de mantener la máxima información posible en la semántica. A continuación se presenta dicho algoritmo:

1. Calcular el grafo de prevención de bucles inicial.
2. Recorrer todas las reglas y abstraer solamente las recursivas.
3. Construir un nuevo grafo de prevención de bucles, en el que cada nodo contendrá, además del símbolo de función y la aridad, el número de regla en la cual aparece dicho símbolo de función.
4. Recorrer el grafo de prevención de bucles.
   (a) Si se encuentra un ciclo:
      i. Eliminar dicho ciclo, abstrayendo la correspondiente regla.
      ii. Volver al punto 3.
   (b) Si no hay ciclos, temina el algoritmo.

Las reglas recursivas son aquéllas en las que el símbolo de función definido en la parte izquierda de la ecuación de la cabeza de la regla, también aparece o bien en la parte derecha de la ecuación de la cabeza de la regla, o bien en el cuerpo. Para abstraer estas reglas (paso 2), se aplica, en función del grafo de prevención de bucles, la función *sh* (definida en el Capítulo 5). La función *sh* recibe como entrada un término concreto y, para cada símbolo de función interno del término, averigua si existe una función definida de la que parte un ciclo, y solamente la sustituye por una variable *fresca* si el símbolo de función de dicha función definida se corresponde con el símbolo de función de la parte izquierda de la ecuación de la cabeza de la regla en proceso.

Después de eliminar las funciones recursivas, se construye un nuevo grafo (paso 3), en el que cada nodo se representa con una terna [símbolo de función, aridad, numRegla], donde numRegla es el numero de regla en la especificación en el que aparece tal sı́mbolo de función. Esto hace más fácil la tarea en el momento de abstraer la regla (paso 4.*i*).

En los siguientes pasos del algoritmo, primero se recorre el nuevo grafo calculado en el paso anterior para detectar si hay ciclos. Si encuentra un ciclo, entonces busca en la especificación la regla que se corresponde con el número de la regla del nodo y la abstrae, eliminando así el ciclo encontrado. Después modifica el conjunto de reglas y vuelve a recalcular el grafo de prevención de bucles (vuelve al paso 3). Estos pasos se repiten hasta que no se encuentren más ciclos en el grafo de prevención de bucles.

6.3.2 Cálculo del *mgu*

Tal como establecimos en el Capítulo 6, si existe una ecuación e tal que $e \notin \gamma(\mathcal{I}^+)$ y $e \in T^{\varphi}_{\{r\}}([\![\mathcal{I}^-]\!])$, entonces la regla $r \in \mathcal{R}$ es incorrecta sobre e. La ecuación e generada en el cálculo de $T^{\varphi}_{\{r\}}([\![\mathcal{I}^-]\!])$ se compara con las que conforman el conjunto $\gamma(\mathcal{I}^+)$,

con el fin de establecer si e es un síntoma de incorrección. En primer lugar, dado que el conjunto $\gamma(\mathcal{I}^+)$ es infinito, nuestro test se reduce a comparar los conjuntos $\mathcal{I}^+$ y $T^{\varphi}_{\{r\}}([\![\mathcal{I}^-]\!])$. No se trata de una simple "pertenencia de conjuntos", ($e \notin \mathcal{I}^+$), ya que $\mathcal{I}^+$ es una sobreaproximación de la especificación de la semántica deseada (en general, puede contener ecuaciones donde están presentes los símbolos $\perp$ o $\sharp$), sino de observar si la ecuación e es o no una instancia de alguna ecuación de $\mathcal{I}^+$. Para esto se usa el procedimiento de unificación más general y se analiza adecuadamente si hubo instanciación de variables de la ecuación o no.

Capítulo 7

Conclusiones y trabajo futuro

En esta tesis presentamos una metodología para la depuración declarativa de programas lógico funcionales con respecto al observable de respuestas computadas. Nuestra aproximación se basa en extender los métodos de diagnóstico declarativo "ascendente" (bottom–up) a un contexto multiparadigma (lógico funcional integrado) y generalizarlos para que se apliquen tanto a lenguajes impacientes como perezosos. Para ello presentamos una semántica de punto fijo para un lenguaje lógico funcional, paramétrica con respecto a la estrategia de narrowing, que posteriormente concretamos sobre varios modelos computacionales, incluyendo la estrategia conocida como *narrowing necesario*.

También presentamos un método para la depuración basado en interpretación abstracta que usa una sobreaproximación y subaproximación de la semántica deseada. Mediante la utilización de un operador apropiado de consecuencias inmediato abstracto obtenemos una descripción (finita e independiente del objetivo) del conjunto de los éxitos del programa que, usada en combinación con varias aproximaciones de la especificación de la semántica deseada, el programa a depurar y el correspondiente operador de consecuencias inmediatas, es la base para producir un sistema de depuración finito y efectivo. Nuestra metodología no requiere que el usuario provea síntomas para comenzar (ni guiar) el proceso de depuración ya que es totalmente automática.

Hemos propuesto dos técnicas diferentes de abstracción de los programas que conducen a una semántica finita y que, por consiguiente, hacen que el proceso de depuración sea finito también. No obstante, la formulación es fácilmente extensible para hacerla paramétrica con respecto a la aproximación. Esto permitiría desarrollar y experimentar con varias versiones del depurador, lo cual nos proponemos abordar en un futuro inmediato.

Hemos iniciado investigaciones en el campo del aprendizaje automático de reglas que aprovechan los resultados del proceso de depuración para sintetizar una versión

correcta del programa. La idea básica es calcular un conjunto de ejemplos positivos y negativos que utilizamos como datos de entrada para las técnicas de aprendizaje automático. Hemos explorado el camino de aprendizaje de reglas siguiendo el enfoque *backward* (mediante la inversión del operador de *narrowing*) o *forward* (utilizando los métodos de plegado y desplegado de reglas) [Alpuente *et al.*, 2002b,a]. Actualmente disponemos de una ampliación, en fase de prueba, de nuestro sistema experimental BUGGY que realiza este proceso [Alpuente *et al.*, 2002c].

Una de las expectativas importantes al trabajar en integración de paradigmas, además de preguntarse por las diferentes propiedades que se heredan de cada uno de los paradigmas a integrar, es determinar qué ventajas adicionales surgen de esa integración y, a manera de retorno, qué podemos aplicar de los resultados obtenidos en un entorno multiparadigma a los paradigmas componentes. En este sentido, en [Alpuente *et al.*, 2002b] hemos propuesto algunas ideas para la corrección automática de programas funcionales: nuestro enfoque se hereda en gran parte de la programación lógica y derivamos un método automático que se aplica al caso de programas funcionales, en donde según nuestro conocimiento no existe una herramienta con características similares.

A lo largo de nuestra investigación, y en particular para efectos de implementación, hemos supuesto que la especificación de la semántica deseada es un programa (una especificación ejecutable). Una alternativa a explorar es desarrollar otras formas de codificar la semántica deseada. Dado que pretendemos aproximar por defecto y por exceso la semántica deseada mediante conjuntos abstractos ($\mathcal{I}^-$ e $\mathcal{I}^+$) y, que éstos son la base para realizar la depuración, es interesante experimentar con otras posibilidades para el tratamiento de la semántica deseada que resultan más flexibles para el usuario. Algunas ideas ya investigadas en otros paradigmas, como el uso de "aserciones" (*assertions*), pueden ser incorporadas como solución en nuestro contexto.

Un camino futuro por explorar es el de la aplicación de los principios de la depuración declarativa a sistemas reales, dotando al sistema con la potencia y robustez necesaria para manejarlos. Esto permitirá extender el sistema BUGGY de forma que sea posible depurar directamente programas escritos en Toy, Curry o Escher.

Bibliografía

Aït-Kaci H. y Nasr R., 1989. Integrating Logic and Functional Programming. *Lisp and Symbolic Computation*, 2(1):51–89.

Alexin Z., Gyimóthy T. y Bostrom H., 1996. Integrating Algorithmic Debugging and Unfolding Transformation in an Interactive Learner. En *Proc. ECAI'96*. John Wiley & Sons.

Alpuente M., Ballis D., Correa F.J. y Falaschi M., 2002a. Automated Correction of Functional Logic Programs. Submitted for publication.

Alpuente M., Ballis D., Correa F.J. y Falaschi M., 2002b. A Multi Paradigm Automatic Correction Schem. En M. Falaschi, ed., *Proc. WFLP'02*. To appear.

Alpuente M., Ballis D., Correa F.J. y Falaschi M., 2002c. The System for Automatic Correction NOBug. Informe técnico, UPV. Available at URL: `http://www.dsic.upv.es/users/elp/papers.html`.

Alpuente M., Correa F. y Falaschi M., 2001a. Declarative Debugging of Functional Logic Programs. En B. Gramlich y S. Lucas, eds., *Proc. of the International Workshop on Reduction Strategies in Rewriting and Programming (WRS 2001)*, volumen 57 de *Electronic Notes in Theoretical Computer Science*. Elsevier Science Publishers.

Alpuente M., Correa F., Falaschi M. y Marson S., 2001b. The Debugging System Buggy. Informe técnico, UPV. Available at URL: `http://www.dsic.upv.es/users/elp/papers.html`.

Alpuente M., Correa F.J. y Falaschi M., 2001c. Debugging Scheme of Funtional Logic Programs. En M. Hanus, ed., *Proc. of International Workshop on Functional and (Constraint) Logic Programming, WFLP'01*, volumen 2017 de *Technical Report*, págs. 59–72. Christian-Albrechts-Universitaet zu Kiel, Kiel, Germany. Submitted to ENTCS.

Alpuente M., Correa F.J. y Falaschi M., 2002d. Abstract Debugging of Functional Logic Programs. *Journal of Symbolic Computation*. Special issue on Reduction Strategies in Rewriting and Programming. Submitted as invited contribution.

Alpuente M., Falaschi M. y Manzo F., 1995. Analyses of Unsatisfiability for Equational Logic Programming. *Journal of Logic Programming*, 22(3):221–252.

Alpuente M., Falaschi M., Moreno G. y Vidal G., 1997. Safe Folding/Unfolding with Conditional Narrowing. En H.H. M. Hanus y K. Meinke, eds., *Proc. of the International Conference on Algebraic and Logic Programming, ALP'97, Southampton (England)*, págs. 1–15. Springer LNCS 1298.

Alpuente M., Falaschi M., Moreno G. y Vidal G., 1999a. A Transformation System for Lazy Functional Logic Programs. En A. Middeldorp y T. Sato, eds., *Proc. of the 4th Fuji International Symposyum on Functional and Logic Programming, FLOPS'99, Tsukuba (Japan)*, págs. 147–162. Springer LNCS 1722.

Alpuente M., Falaschi M., Ramis M. y Vidal G., 1994. A Compositional Semantics for Conditional Term Rewriting Systems. En H. Bal, ed., *Proc. of 6th IEEE Int'l Conf. on Computer Languages, ICCL'94*, págs. 171–182. IEEE, New York.

Alpuente M., Falaschi M. y Vidal G., 1996. A Compositional Semantic Basis for the Analysis of Equational Horn Programs. *Theoretical Computer Science*, 165(1):97–131.

Alpuente M., Falaschi M. y Vidal G., 1998. Partial Evaluation of Functional Logic Programs. *ACM Transactions on Programming Languages and Systems*, 20(4):768–844.

Alpuente M., Hanus M., Lucas S. y Vidal G., 1999b. Specialization of Functional Logic Programs Based on Needed Narrowing. Informe Técnico 99-4, RWTH Aachen. Accesible en URL: `http://www.dsic.upv.es/users/elp/papers.html`.

Antoy S., 1992. Definitional Trees. En *Proc. of the 3rd Int'l Conference on Algebraic and Logic Programming, ALP'92*, págs. 143–157. Springer LNCS 632.

Antoy S., 1997. Optimal Non-Deterministic Functional Logic Computations. En *Proc. of the Int'l Conference on Algebraic and Logic Programming, ALP'97*, págs. 16–30. Springer LNCS 1298.

Antoy S., 2001. Evaluation Strategies for Functional Logic Programming. En *Int'l Workshop on Reduction Strategies in Rewriting and Programming WRS'01*, volumen 57. Electronic Notes in Theoretical Computer Science.

Antoy S., Echahed R. y Hanus M., 1994. A Needed Narrowing Strategy. En *Proc. 21st ACM Symp. on Principles of Programming Languages, Portland*, págs. 268–279. ACM Press, New York.

Antoy S., Echahed R. y Hanus M., 2000. A Needed Narrowing Strategy. *Journal of the ACM*, 47(4):776–822. Preliminary version appeared in Proc. 21st ACM Symp. on Principles of Programming Languages, pages 268-279. ACM Press, New York, 1994.

Apt K., 1990. Introduction to Logic Programming. En J. van Leeuwen, ed., *Handbook of Theoretical Computer Science*, volumen B: Formal Models and Semantics. Elsevier, Amsterdam and The MIT Press, Cambridge, Mass.

Arenas P. y Gil A., 1994. A Debugging Model for Lazy Functional Languages. Informe Técnico DIA 94/6, Universidad Complutense de Madrid.

Arenas P. y Gil A., 1995. A Debugging Model for Lazy Narrowing. En *Proc. of PLILP'95*, volumen 982 de *Lecture Notes in Computer Science*, págs. 453–454. Springer.

Bellia M. y Levi G., 1986. The relation between logic and functional languages. *Journal of Logic Programming*, 3:217–236.

Bert D. y Echahed R., 1986. Design and implementation of a generic, logic and functional programming language. En *Proc. of First European Symp. on Programming, ESOP'86*, págs. 119–132. Springer LNCS 213.

Bert D. y Echahed R., 1994. Integrating Disequations in the Algebraic and Logic Programming Language LPG. En *Proc. of the ICLP'94 Post-Conference Workshop on Integration of Declarative Paradigms*. MPI-I-94-224, Saarbrucken.

Bert D., Echahed R. y Østvold B., 1993. Abstract Rewriting. En *Proc. of Third Int'l Workshop on Static Analysis, WSA'93*, págs. 178–192. Springer LNCS 724.

Bol R., 1993. Loop Checking in Partial Deduction. *Journal of Logic Programming*, 16(1&2):25–46.

Bosco P., Giovannetti E. y Moiso C., 1988. Narrowing vs. SLD-resolution. *Theoretical Computer Science*, 59:3–23.

Bostrom H. y Idestam-Alquist P., 1994. Specialization of Logic Programs by Pruning SLD-trees. En *Proc of 4th Int'l Workshop on Inductive Logic Programming (ILP-94)*, págs. 31–47.

Bostrom H. y Idestam-Alquist P., 1999. Induction of Logic Programs by Example–guided Unfolding. *Journal of Logic Programming*, 40:159–183.

Boye J., Drabent W. y Maluszyński J., 1997. Declarative diagnosis of constraint programs: an assertion-based approach. En *Proc. of the 3rd Int'l Workshop on Automated Debugging - AADEBUG'97*, págs. 123–141. U. of Linkoping Press, Linkoping, Sweden.

Bueno F., Deransart P., Drabent W., Ferrand G., Hermenegildo M., Maluszyński J. y Puebla G., 1997a. On the Role of Semantic Approximations in Validation and Diagnosis of Constraint Logic Programs. En *Proc. of the 3rd. Int'l Workshop on Automated Debugging-AADEBUG'97*, págs. 155–170. U. of Linkoping Press.

Bueno F., Puebla G. y Hermenegildo M., 1997b. An Assertion Language for Debugging of Constraint Logic Programs. En *Proceedings of the ILPS'97 Workshop on Tools and Environments for (Constraint) Logic Programming*.

Byrd L., 1980. Understanding the Control Flow of Prolog Programs. En *In Proc. Logic Programming Workshop,Debrecen, Hungary*, págs. 127–138. Also Univ. of Edinburgh Dept. of Artificial Intelligence Research Paper 151.

Caballero-Roldán R., López-Fraguas F. y Artalejo M.R., 2001. Theoretical Foundations for the Declarative Debugging of Lazy Functional Logic Programs. En *Fifth International Symposium on Functional and Logic Programming*, Lecture Notes in Computer Science. Springer-Verlag, Berlin. To appear.

Chabin J. y Réty P., 1991. Narrowing directed by a graph of terms. En G. Goos y J. Hartmanis, eds., *Proc. of RTA'91*, págs. 112–123. Springer LNCS 488.

Cheong P. y Fribourg L., 1992. A survey of the implementations of narrowing. En J. Darlington y R. Dietrich, eds., *Declarative Programming. Workshops in Computing*, págs. 177–187. Springer-Verlag and BCS.

Codish M., Falaschi M. y Marriott K., 1991. Suspension Analysis for Concurrent Logic Programs. En K. Furukawa, ed., *Proc. of Eighth Int'l Conf. on Logic Programming*, págs. 331–345. The MIT Press, Cambridge, MA.

Codish M., Falaschi M. y Marriott K., 1994. Suspension Analyses for Concurrent Logic Programs. *ACM Transactions on Programming Languages and Systems*, 16(3):649–686.

Comini M., Gori R. y Levi G., 2000. Logic programs as specifications in the inductive verification of logic programs. En *Proceeding of Appia-Gulp-Prode'00, Joint Conference on Declarative Programming*. URL: `http://nutella.di.unipi.it/~agp00/AccptList.html`.

Comini M., Gori R. y Levi G., 2001. Assertion based Inductive Verification Methods for Logic Programs. En A.K. Seda, ed., *Proceedings of MFCSIT'2000*, volumen 40 de *Electronic Notes in Theoretical Computer Science*. Elsevier Science Publishers.

Comini M. y Levi G., 1994. An algebraic theory of observables. En M. Bruynooghe, ed., *Proceedings of the 1994 Int'l Symposium on Logic Programming*, págs. 172–186. The MIT Press.

Comini M., Levi G., Meo M. y Vitiello G., 1996. Proving Properties of Logic Programs by Abstract Diagnosis. En M. Dams, ed., *Analysis and Verification of Multiple-Agent Languages, 5th LOMAPS Workshop*, volumen 1192 de *Lecture Notes in Computer Science*, págs. 22–50. Springer-Verlag.

Comini M., Levi G., Meo M.C. y Vitiello G., 1999. Abstract Diagnosis. *Journal of Logic Programming*, 39(1-3):43–93.

Comini M., Levi G. y Vitiello G., 1994. Abstract Debugging of Logic Programs. En L. Fribourg y F. Turini, eds., *Proc. Logic Program Synthesis and Transformation and Metaprogramming in Logic 1994*, volumen 883 de *Lecture Notes in Computer Science*, págs. 440–450. Springer-Verlag, Berlin.

Comini M., Levi G. y Vitiello G., 1995a. Declarative Diagnosis Revisited. En J.W. Lloyd, ed., *Proceedings of the 1995 Int'l Symposium on Logic Programming*, págs. 275–287. The MIT Press.

Comini M., Levi G. y Vitiello G., 1995b. Efficient detection of incompleteness errors in the abstract debugging of logic programs. En M. Ducassé, ed., *Proc. 2nd International Workshop on Automated and Algorithmic Debugging, AADEBUG'95*.

Cousot P. y Cousot R., 1977. Abstract Interpretation: A Unified Lattice Model for Static Analysis of Programs by Construction or Approximation of Fixpoints. En *Proc. of Fourth ACM Symp. on Principles of Programming Languages*, págs. 238–252.

Cousot P. y Cousot R., 1979. Systematic Design of Program Analysis Frameworks. En *Proc. of Sixth ACM Symp. on Principles of Programming Languages*, págs. 269–282.

Cousot P. y Halbwachs N., 1978. Automatic Discovery of Linear Restraints Among Variables of a Program. En *Proc. of Fifth ACM Symp. on Principles of Programming Languages*, págs. 84–96.

Darlington J., Guo Y. y Pull H., 1991. Constraints unify functional and logic programming. Informe técnico, Department of Computing, Imperial College, London.

DeGroot D. y Lindstrom G., 1986. *Logic Programming, Functions, Relations and Equations*. Prentice Hall, Englewood Cliffs, NJ.

Dershowitz N. y Jouannaud J.P., 1990. Rewrite Systems. En J. van Leeuwen, ed., *Handbook of Theoretical Computer Science*, volumen B: Formal Models and Semantics, págs. 243–320. Elsevier, Amsterdam.

Dershowitz N. y Lee Y., 1987. Deductive Debugging. En *Proceedings of the Fourth IEEE Symposium on Logic Programming*, págs. 298–306. San Francisco, California.

Dershowitz N. y Lee Y., 1992. Logical Debugging. Informe técnico, Naval Postgraduate School and University of Illinois.

Dershowitz N. y Plaisted A., 1988. Equational Programming. *Machine Intelligence*, 11:21–56.

Dershowitz N. y Sivakumar G., 1987. Solving Goals in Equational Languages. En S. Kaplan y J. Joaunnaud, eds., *Proc. of First Int'l Workshop on Conditional Term Rewriting*, págs. 45–55. Springer LNCS 308.

Drabent W., Nadjim-Tehrani S. y Maluszynski J., 1988. The Use of Assertions in Algorithmic Debugging. En *Proceedings of the 1988 Iinternational Conference on Fifth Generation Computer Systems*, págs. 573–581. Tokyo, Japan.

Ducassé M., 1999a. Abstract views of Prolog executions with Opium. En B.d.B. P. Brna, ed., *Learning to Build and Comprehend Complex Information Structures: Prolog as a Case Study, Cognitive Science and Technology, chapter 10*, págs. 223–243.

Ducassé M., 1999b. Opium: An extendable trace analyser for Prolog. *Journal of Logic Programming*, 39:177–223. Special issue on Synthesis, Transformation and Analysis of Logic Programs.

Echahed R., 1988. On completeness of narrowing strategies. En *Proc. of CAAP'88*, págs. 89–101. Springer LNCS 299.

Echahed R., 1992. Uniform narrowing strategies. En *Proc. of ICALP'92*, págs. 259–275. Springer LNCS 632.

Edman A. y Tärnlund S., 1983. Mechanization of an Oracle in a Debugging System. En *Proceedings of Eighth IJCAI*, págs. 553–555. Karlsruhe, Germany.

Eklund F., 1997. *Declarative Error Diagnosis of GAPLog Programs*. Tesis Doctoral, Linköping University.

Falaschi M., Levi G., Martelli M. y Palamidessi C., 1989. Declarative Modeling of the Operational Behavior of Logic Languages. *Theoretical Computer Science*, 69(3):289–318.

Falaschi M., Levi G., Martelli M. y Palamidessi C., 1993. A Model-Theoretic Reconstruction of the Operational Semantics of Logic Programs. *Information and Computation*, 103(1):86–113.

Fay M., 1979. First Order Unification in an Equational Theory. En *Proc of 4th Int'l Conf. on Automated Deduction*, págs. 161–167.

Ferrand G., 1987. Error Diagnosis in Logic Programming, and Adaptation of E.Y.Shapiro's Method. *Journal of Logic Programming*, 4(3):177–198.

Ferrand G. y Tessier A., 1996. Declarative Debugging. *The Newsletter of the European Network in Computational Logic*, 3(1):71–76.

Fribourg L., 1985. SLOG: a logic programming language interpreter based on clausal superposition and rewriting. En *Proc. of Second IEEE Int'l Symp. on Logic Programming*, págs. 172–185. IEEE, New York.

Frutos-Escrig D. y Fernández-Camacho M., 1991. On Narrowing Strategies for Partial Non-Strict Functions. En S. Abramsky y T. Maibaum, eds., *Proc. TAPSOFT'01*, págs. 416–437. Springer LNCS 493.

Giovannetti E., Levi G., Moiso C. y Palamidessi C., 1991. Kernel Leaf: A Logic plus Functional Language. *Journal of Computer and System Sciences*, 42:363–377.

Hanus M., 1990. Compiling Logic Programs with Equality. En *Proc. of 2nd Int'l Workshop on Programming Language Implementation and Logic Programming*, págs. 387–401. Springer LNCS 456.

Hanus M., 1991. Efficient Implementation of Narrowing and Rewriting. En *Proc. Int'l Workshop on Processing Declarative Knowledge*, págs. 344–365. Springer LNAI 567.

Hanus M., 1992. Improving Control of Logic Programs by Using Functional Logic Languages. En M. Bruynooghe y M. Wirsing, eds., *Proc. of PLILP'92, Leuven (Belgium)*, volumen 631, págs. 1–23. Springer LNCS 631.

Hanus M., 1994a. Combining Lazy Narrowing with Simplification. En *Proc. of 6th Int'l Symp. on Programming Language Implementation and Logic Programming*, págs. 370–384. Springer LNCS 844.

Hanus M., 1994b. The Integration of Functions into Logic Programming: From Theory to Practice. *Journal of Logic Programming*, 19&20:583–628.

Hanus M. y Josephs B., 1993. A Debugging Model for Functional Logic Programs. En M. Bruynooghe y J. Penjam, eds., *Proc. of 5th Int'l Symp. on Programming Language Implementation and Logic Programming*, volumen 714 de *Lecture Notes in Computer Science*, págs. 28–43. Springer.

Hanus M., Kuchen H. y Moreno-Navarro J., 1995. Curry: A Truly Functional Logic Language. En *Proc. ILPS'95 Workshop on Visions for the Future of Logic Programming*, págs. 95–107.

Hanus M. y Prehofer C., 1999. Higher-Order Narrowing with Definitional Trees. *Journal of Functional Programming*, 9(1):33–75. Preliminary version in Proc. RTA'96, pages 138-152. Springer LNCS 1103.

Hermenegildo M. y Rossi F., 1989. On the Correctness and Efficiency of Independent And-Parallelism in Logic Programs. En E. Lusk y R. Overbeck, eds., *Proc. of the 1989 North American Conf. on Logic Programming*, págs. 369–389. The MIT Press, Cambridge, MA.

Heyer T., 1995. Debugging in GAPLog: a model for execution of logic programs with delayed function calls. En M. Ducassé, ed., *Proceedings Workshop on Automated and Algorithmic Debugging (AADEBUG '95)*.

Hölldobler S., 1989. *Foundations of Equational Logic Programming*. Springer LNAI 353.

Hudak P., 1989. Conception, Evolution and Application of Functional Programming Languages. ACM Computing Surveys, 21(3):359–411.

Hullot J., 1980. Canonical Forms and Unification. En *Proc of 5th Int'l Conf. on Automated Deduction*, págs. 318–334. Springer LNCS 87.

Hussman H., 1985. Unification in Conditional-Equational Theories. En *Proc. European Conf. on Computer Algebra EUROCAL'85*, págs. 543–553. Springer LNCS 204.

Jacquet J.M., 1989. *Conclog: A Methodological Approach to Concurrent Logic Programming*. Tesis Doctoral, University of Namur, Belgium.

Julián P. y Moreno G., 1996. Nuevas Semánticas para Programas Lógico–Ecuacionales: Teoría e Implementación. Informe Técnico DSIC-II/13/96, UPV.

Kanamori T., Kawamura T., Maeji M. y Horiuchi K., 1989. Logic Program Diagnosis Specification. Icot technical report tr-447, Institute for New Generation Computer Technology, Tokyo, Japan.

Kaplan S., 1987. Simplifying Conditional Term Rewriting Systems: Unification, Termination and Confluence. *Journal of Symbolic Computation*, 4:295–334.

Klop J., 1992. Term Rewriting Systems. En S. Abramsky, D. Gabbay y T. Maibaum, eds., *Handbook of Logic in Computer Science*, volumen I, págs. 1–112. Oxford University Press.

Klop J. y Middeldorp A., 1989. Sequentiality in Orthogonal Term Rewriting Systems. *Journal of Symbolic Computation*, págs. 161–195.

Knill E., Cox P. y Pietrzykowski T., 1993. Equality and abductive residua for Horn Clauses. *Theoretical Computer Science*, 120:1–44.

Kowalski R., 1974. Predicate Logic as a Programming Language. En *Information Processing 74*, págs. 569–574. North-Holland.

Kowalski R., 1979. *Logic for Problem Solving*. North-Holland.

Kuchen H., Loogen R., Moreno-Navarro J. y Rodríguez-Artalejo M., 1990a. Graph-based Implementation of a Functional Logic Language. En *Proc. of ESOP'90*, págs. 279–290. Springer LNCS 432.

Kuchen H., Loogen R., Moreno-Navarro J. y Rodríguez-Artalejo M., 1990b. Lazy Narrowing in a Graph Machine. En *Proc. of the Int'l Conf. on Algebraic and Logic Programming*, págs. 298–317. Springer LNCS 463.

Lassez J.L., Maher M.J. y Marriott K., 1988. Unification Revisited. En J. Minker, ed., *Foundations of Deductive Databases and Logic Programming*, págs. 587–625. Morgan Kaufmann, Los Altos, Ca.

Lloyd J., 1987a. *Foundations of Logic Programming*. Springer-Verlag, Berlin. Second edition.

Lloyd J.W., 1987b. Declarative Error Diagnosis. *New Generation Computing*, 5(2):133–154.

Lloyd J.W., 1998. Debugging for a declarative programming language. *Machina Intelligence 15*. In press.

Lloyd J.W., 1999. Programming in an integrated functional and logic language. *Journal of Functional and Logic Programming*, 3.

Lu L., 1994a. Abstract Interpretation, Bug Detection and Bug Diagnosis in Normal Logic Programs. PhD thesis, University of Birmingham.

Lu L., 1994b. A Generic Declarative Diagnoser for Normal Logic Programs. *Lecture Notes in Artificial Intelligence*, 822:290–304.

Maher M., 1988. Complete Axiomatizations of the Algebras of Finite, Rational and Infinite Trees. En *Proc. of Third IEEE Symp. on Logic In Computer Science*, págs. 348–357. Computer Science Press, New York.

Maher M., 1990. On Parameterized Substitutions. Informe Técnico RC 16042, IBM - T.J. Watson Research Center, Yorktown Heights, NY.

Maneth S. y Vogler H., 1995. Calculi and Efficiency Measures for Narrowing Techniques. Informe técnico, Technical University of Dresden. Technical Report TUD/FI95/13.

Middeldorp A. y Hamoen E., 1994. Completeness Results for Basic Narrowing. *Applicable Algebra in Engineering, Communication and Computing*, 5:213–253.

Moreno J.J., 1994. Expressivity of Functional-logic Languajes and their Implementation. En *GULP-PRODE'94*, págs. 11–42. Invited Tutorial.

Moreno-Navarro J. y Rodríguez-Artalejo M., 1992. Logic Programming with Functions and Predicates: The language Babel. *Journal of Logic Programming*, 12(3):191–224.

Muggleton S. y de Raedt L., 1994. Inductive Logic Programming: Theory and Methods. *Journal of Logic Programming*, 19,20:629–679.

Naish L., 1992. Declarative Diagnosis of Missing Answer. *New Generation Computing*, 10(3):255–285. Technical Report 88/9 (revised), Department of Computer Science, University of Melbourne.

Naish L., 1993. Declarative Debugging of Lazy Functional Programs. *Australian Computer Science Communications*, 15(1):287–294.

Naish L., 1997. A Declarative Debugging Scheme. *Journal of Functional and Logic Programming*, 1997(3).

Naish L. y Barbour T., 1994. Declarative Debugging of a Logical-Functional Language. Technical report 94/30, Department of Computer Science, University of Melbourne, Melbourne, Australia.

Naish L. y Barbour T., 1995. Towards a portable lazy functional declarative debugger. Informe técnico, Department of Computer Science, University of Melbourne, Melbourne, Australia.

Nilsson H., 1994. A declarative approach to debugging for lazy functional languages. Licentiate Thesis.

Nilsson H. y Fritzson P., 1994. Algoritmic debugging for lazy functional languages. *Journal of Functional Programming*, 4(1):337–370.

O'Donnell M., 1977. *Computing in Systems Described by Equations*. Springer LNCS 58.

Padawitz P., 1988. *Computing in Horn Clause Theories*, volumen 16 de *EATCS Monographs on Theoretical Computer Science*. Springer-Verlag, Berlin.

Palamidessi C., 1988. A fixpoint semantics for Guarded Horn Clauses. Informe Técnico CS-R8833, Centre for Mathematics and Computer Science, Amsterdam.

Palamidessi C., 1990. Algebraic Properties of Idempotent Substitutions. En M. Paterson, ed., *Proc. of 17th Int'l Colloquium on Automata, Languages and Programming*, págs. 386–399. Springer LNCS 443.

Pereira L.M., 1986. Rational Debugging in Logic Programming. En E. Shapiro, ed., *Proceedings of the Third International Conference on Logic Programming*, págs. 203–210. Springer LNCS 528, London, England.

Plasmeijer R. y van Eekelen M., 1993. *Functional Programming and Parallel Graph Rewriting.*. Addison Wesley.

Plotkin G., 1981. A structured approach to operational semantics. Informe Técnico DAIMI FN-19, Computer Science Department, Aarhus University.

Pope B., 1998. Buddha: A declarative debugger for Haskell.

Reade C., 1993. *Elements of Functional Programming*. Addison-Wesley, Reading, MA.

Reddy U., 1985. Narrowing as the Operational Semantics of Functional Languages. En *Proc. of Second IEEE Int'l Symp. on Logic Programming*, págs. 138–151. IEEE, New York.

Réty P., 1987. Improving basic narrowing techniques. En *Proc. of the Conf. on Rewriting Techniques and Applications*, págs. 228–241. Springer LNCS 256.

Réty P., Kirchner C., Kirchner H. y Lescanne P., 1985. NARROWER: A new algorithm for unification and its applications to logic programming. En *Proc. of RTA'85*, págs. 141–157. Springer LNCS 202.

Safra S. y Shapiro E., 1986. Meta Interpreters for Real. En H.J. Kugler, ed., *Information Processing 86, Dublin, Ireland*, págs. 271–278. North-Holland, Amsterdam.

Shaphiro E.Y., 1982. *Algorithmic Program Debugging*. The MIT Press, Cambridge, Massachusetts. ACM Distinguished Dissertation.

Slagle J., 1974. Automated Theorem-Proving for Theories with Simplifiers, Commutativity and Associativity. *Journal of the ACM*, 21(4):622–642.

Sparud J. y Nilsson H., 1995. The architecture of a debugger for lazy functional languages. En M. Ducassé, ed., *Proceedings of AADEBUG'95*. Saint-Malo, France.

Vidal G., 1996. *Semantics-Based Analysis and Transformation of Functional Logic Programs*. Tesis Doctoral, DSIC-UPV. En español.

Wang B. y Shyamasundar R., 1990. Methodology for Proving the Termination of Logic Programs. En *Proc. of PLILP'90*, págs. 204–221. Springer LNCS 456.

You Y., 1989. Enumerating outer narrowing derivations for constructor-based term rewriting systems. *Journal of Symbolic Computation*, 7:319–343.

Zartmann F., 1997. Denotational Abstract Interpretation of Functional Logic Programs. En P.V. Hentenryck, ed., *Proc. of the 4th Int'l Static Analysis Symposium, SAS'97*, págs. 141–159. Springer LNCS 1302.

Apéndice A

Una sesión de BUGGY

Mostramos una sesión BUGGY para el diagnóstico y corrección de dos programas lógico funcionales.

Una vez el programa se ejecuta se presenta el siguiente menu

```
*********************************************************************
       DECLARATIVE DEBUGGER FOR FUNCTIONAL LOGIC PROGRAMS.
*********************************************************************
*********************************************************************
*** help.      -> display menu                                     ***
*** clean.     -> clean program and specification                  ***
*** loadprg.   -> read in program to be corrected from file        ***
*** loadspe.   -> load specification of the semantics from file    ***
*** listprg.   -> display program                                  ***
*** listspe.   -> display specification                            ***
*** debugger.  -> run debugger                                     ***
*** save.      -> save program to file                             ***
*** exit.      -> leave program                                    ***
*********************************************************************
>> clean.
>> loadprg.
Write the name of the program: 'evenprg.pl'.
Program loaded.

>> loadspe.
Write the name of the program: 'evenspe.pl'.
Specification loaded.
```

```
>> listprg.
even(0)=true.
even(s(X))=even(X).

>> listspe.
even(0)=true.
even(s(X))=odd(X).
odd(s(0))=true.
odd(s(s(X)))=odd(X).

>> debugger.
************************************************
          NARROWING STRATEGY  OPTIONS
************************************************
    1. Basic Narrowing.
    2. Leftmost-Innermost Narrowing.
    3. Leftmost-Outermost Narrowing.
    4. Needed Narrowing.
    Introduce the option:4.

Translating program to case version:
even(X)  =  case_5(X,0,true,s(X_1),even(X_1)).
            case_5(s(X),_,_,s(X),T) = T.
            case_5(0,0,T,_,_) = T.
(X ≈ Y) =  case_4(X,true,case_1(Y,true,true),0,case_2(Y,0,true),
                  s(X_1),case_3(Y,s(Y_1),(X_1 ≈ Y_1))).
            case_4(true,true,T,_,_,_,_) = T.
            case_4(0,_,_,0,T,_,_) = T.
            case_4(s(X),_,_,_,_,_,s(X),T) = T.
            case_1(true,true,T) = T.
            case_2(0,0,T) = T.
            case_3(s(X),s(X),T) = T.

Translating intended specification to case version:
even(X)  =  case_5(X,0,true,s(X_1),odd(X_1)).
            case_5(0,0,T,_,_) = T.
            case_5(s(X),_,_,s(X),T) = T.
odd(X)   =  case_7(X,s(X_1),case_6(X_1,0,true,s(X_2),odd(X_2))).
            case_7(s(X),s(X),T) = T.
```

```
            case_6(0,0,T,_,_) = T.
            case_6(s(X),_,_,s(X),T) = T.
(X ≈ Y) =  case_4(X,true,case_1(Y,true,true),0,case_2(Y,0,true),
                 s(X_1),case_3(Y,s(Y_1),(X_1 ≈ Y_1))).
            case_4(true,true,T,_,_,_,_) = T.
            case_4(0,_,_,0,T,_,_) = T.
            case_4(s(X),_,_,_,_,s(X),T) = T.
            case_1(true,true,T) = T.
            case_2(0,0,T) = T.
            case_3(s(X),s(X),T) = T.
************************************************
***          OVER APPROXIMATION   I+          ***
************************************************
true ≈ true.          0 ≈ 0.                    (X ≈ Y) = (X ≈ Y).
s(X) ≈ s(X).          odd(X) = odd(X).           even(X) = even(X).
case_1(X,Y,Z) = case_1(X,Y,Z).
                   %Iteration 0, omitting equations:
                   %case_n(X,...,Z) = case_n(X,...,Z), with 1 < n
even(X) = case_5(X,0,true,s(X_1),odd(X_1)).
case_5(0,0,T,_,_) = T.
case_5(s(X),_,_,s(X),T) = T.
odd(X) = case_7(X,s(X_1),case_6(X_1,0,true,s(X_2),♯))).
case_7(s(X),s(X),T) = T.
case_6(0,0,T,_,_) = T.
case_6(s(X),_,_,s(X),T) = T.
(X ≈ Y) = case_4(X,true,case_1(Y,true,true),0,case_2(Y,0,true),
                s(X_1),case_3(Y,s(Y_1),♯)).
case_4(true,true,T,_,_,_,_) = T.
case_4(0,_,_,0,T,_,_) = T.
case_4(s(X),_,_,_,_,s(X),T) = T.
case_1(true,true,T) = T.
case_2(0,0,T) = T.
case_3(s(X),s(X),T) = T.
case_1(_,_,_) = bottom.          %Iteration 1, omitting equations:
                                 %case_n(_,..,_) = bottom.  1 < n
even(X) = bottom.
odd(X) = bottom.
(X ≈ Y) = bottom.
odd(s(X)) = case_6(X,0,true,s(X_1),♯).
```

```
even(s(X)) = odd(X).
even(0) = true.
(true ≈ Y) = case_1(Y,true,true).
(0 ≈ Y) = case_2(Y,0,true).
(s(X) ≈ Y) = case_3(Y,s(Y_1),♯).                        % Iteration 2.
odd(s(0)) = true.
odd(s(s(X))) = ♯.
odd(s(X)) = bottom.
even(s(X)) = case_7(X,s(X_1),case_6(X_1,0,true,s(X_2),♯))).
even(s(X)) = bottom.
(true ≈ true) = true.
(true ≈ Y) = bottom.
(0 ≈ 0) = true.
(0 ≈ Y) = bottom.
(s(X) ≈ s(Y) = ♯.
(s(X) ≈ Y) = bottom.                                    % Iteration 3.
even(s(s(X))) = case_6(X,0,true,s(X_1),♯).              % Iteration 4.
even(s(s(0))) = true.
even(s(s(s(X)))) = ♯.
even(s(s(X))) = bottom.                                 % Iteration 5.
************************************************
The Over Approximation  I+ has been computed in 5 iterations.
Enter the number K of iterations to compute  I- : 4.
************************************************
***          UNDER APPROXIMATION  I-          ***
************************************************
true ≈ true.          0 ≈ 0.                    (X ≈ Y) = (X ≈ Y).
s(X) ≈ s(X).          odd(X) = odd(X).          even(X) = even(X).
case_1(X,Y,Z) = case_1(X,Y,Z).
                   %Iteration 0, omitting equations:
                   %case_n(X,...,Z) = case_n(X,...,Z), with 1 < n
even(X) = case_5(X,0,true,s(X_1),odd(X_1)).
case_5(0,0,T,_,_) = T.
case_5(s(X),_,_,s(X),T) = T.
odd(X) = case_7(X,s(X_1),case_6(X_1,0,true,s(X_2),odd(X_2)))).
case_7(s(X),s(X),T) = T.
case_6(0,0,T,_,_) = T.
case_6(s(X),_,_,s(X),T) = T.
(X ≈ Y) = case_4(X,true,case_1(Y,true,true),0,case_2(Y,0,true),
```

```
                s(X_1),case_3(Y,s(Y_1),(X_1 ≈ Y_1))).
case_4(true,true,T,_,_,_,_) = T.
case_4(0,_,_,0,T,_,_) = T.
case_4(s(X),_,_,_,_,_,s(X),T) = T.
case_1(true,true,T) = T.
case_2(0,0,T) = T.
case_3(s(X),s(X),T) = T.
odd(X) = bottom.
even(X) = bottom.
(X ≈ Y) = bottom.
case_1(_,_,_) = bottom.          %Iteration 1, omitting equations:
                                 %case_n(_,..,_) = bottom.  1 < n
odd(s(X)) = case_6(X,0,true,s(X_1),odd(X_1)).
even(s(X)) = odd(X).
even(0) = true.
(true ≈ Y) = case_1(Y,true,true).
(0 ≈ Y) = case_2(Y,0,true).
(s(X) ≈ Y) = case_3(Y,s(Y_1),(X ≈ Y_1)).          % Iteration 2.
odd(s(0)) = true.
odd(s(s(X))) = odd(X).
odd(s(X)) = bottom.
even(s(X)) = case_7(X,s(X_1),case_6(X_1,0,true,s(X_2),odd(X_2))).
even(s(X)) = bottom.
(true ≈ true) = true.
(true ≈ Y) = bottom.
(0 ≈ 0) = true.
(0 ≈ Y) = bottom.
(s(X) ≈ s(Y) = (X ≈ Y).
(s(X) ≈ Y) = bottom.                                  % Iteration 3.
odd(s(s(X))) = case_7(X,s(X_1),case_6(X_1,0,true,s(X_2),odd(X_2))).
odd(s(s(X))) = bottom.
even(s(s(X))) = case_6(X,0,true,s(X_1),odd(X_1)).
(s(X) ≈ s(Y)) = case_4(X,true,case_1(Y,true,true),0,case_2(Y,0,true),
                    s(X_1),case_3(Y,s(Y_1),(X_1 ≈ Y_1))).
(s(X) ≈ s(Y)) = bottom.                               % Iteration 4.
***********************************************
***              T_R(I^-)                   ***
***********************************************
true ≈ true.          0 ≈ 0.                    (X ≈ Y) = (X ≈ Y).
```

```
s(X) ≈ s(X).            even(X) = even(X).
case_1(X,Y,Z) = case_1(X,Y,Z).
                        %Iteration 0, omitting equations:
                        %case_n(X,...,Z) = case_n(X,...,Z), with 1 < n
even(X) = case_5(X,0,true,s(X_1),even(X_1)).
case_5(s(X),_,_,s(X),T) = T.
case_5(0,0,T,_,_) = T.
even(0) = true.
even(s(X)) = even(X).
(X ≈ Y) = case_4(X,true,case_1(Y,true,true),0,case_2(Y,0,true),
                 s(X_1),case_3(Y,s(Y_1),(X_1 ≈ Y_1))).
case_4(true,true,T,_,_,_,_) = T.
case_4(0,_,_,0,T,_,_) = T.
case_4(s(X),_,_,_,_,s(X),T) = T.
case_1(true,true,T) = T.
case_2(0,0,T) = T.
case_3(s(X),s(X),T) = T.
(X ≈ Y) = bottom.
(true ≈ Y) = case_1(Y,true,true).
(0 ≈ Y) = case_2(Y,0,true).
(s(X) ≈ Y) = case_3(Y,s(Y_1),(X ≈ Y_1)).
(true ≈ true) = true.
(true ≈ Y) = bottom.
(0 ≈ 0) = true.
(0 ≈ Y) = bottom.
(s(X) ≈ s(Y) = (X ≈ Y).
(s(X) ≈ Y) = bottom.
(s(X) ≈ s(Y)) = case_4(X,true,case_1(Y,true,true),0,case_2(Y,0,true),
                       s(X_1),case_3(Y,s(Y_1),(X_1 ≈ Y_1))).
(s(X) ≈ s(Y)) = bottom.
(s(0) ≈ s(Y)) = case_2(Y,0,true).
(s(s(X)) ≈ s(Y)) = case_3(Y,s(Y_1),(X ≈ Y_1)).
************************************************
***         DIAGNOSIS:  ERROR FOUND          ***
************************************************
The equation: even(s(X)) = even(X) in T_R(I^-) is incorrect.
The incorrect rule is: even(s(X)) = even(X).

            CORRECTION OPTIONS
```

```
***********************************************
1. Automatic correction.
2. Manual correction.

Enter option: 1.
***********************************************
***           GENERATING EXAMPLES            ***
***********************************************
Positive examples: even(s(s(0)) = true.
Negative examples: even(s(0)) = true.
***********************************************
***           CORRECTED PROGRAM              ***
***********************************************
even(0) = true.
even(s(s(X))) = even(X).

>> clean.
>> loadprg.
Write the name of the program: 'lastprg.pl'.
Program loaded.

>> loadspe.
Write the name of the program: 'lastspe.pl'.
Specification loaded.

>> listprg.
last([])=[].
last([X|Y])=last(Y).

>> listspe.
last(X)=Y :- append(Z,[Y])=X.
append([],B)=B.
append([A|B],C)=[A|append(B,C)].

>> debugger.
***********************************************
          NARROWING STRATEGY  OPTIONS
***********************************************
    1. Basic Narrowing.
```

```
    2. Leftmost-Innermost Narrowing.
    3. Leftmost-Outermost Narrowing.
    4. Needed Narrowing.
    Introduce the option:2.

************************************************
***          OVER APPROXIMATION  I+          ***
************************************************
[] = [].                        [A|B] = [A|B].
append(A,B) = append(A,B).      last(X) = last(X).      % Iteration 0
append([],B) = B.
last(append(A,[B])) = B.
append([A|B],C) = [A|#].                                % Iteration 1
last([X]) = X.                                          % Iteration 2
last([X|Y]) = #.                                        % Iteration 3
************************************************
The Over Approximation  I+ has been computed in 3 iterations.
Enter the number K of iterations to compute  I- : 4.
************************************************
***          UNDER APPROXIMATION  I-         ***
************************************************
[] = [].                        [A|B] = [A|B].
append(A,B) = append(A,B).      last(X) = last(X).       % Iteration 0
append([],B) = B.
last(append(A,[B])) = B.
append([A|B],C) = [A|append(B,C)].                       % Iteration 1
last([A]) = A.
append([A],B) = [A|B].
last([A|append(B,[C])]) = C.
last([A,B]) = B.
append([A,B|C],D) = [A,B|append(C,D)].                   % Iteration 2
last([A,B|append(C,[D])]) = D.
append([A,B],C) = [A,B|C].
last([A,B,C]) = C.
append([A,B,C|D],E) = [A,B,C|append(D,E)].               % Iteration 3
last([A,B,C|append(D,[E])]) = E.
append([A,B,C],D) = [A,B,C|D].
append([A,B,C,D|E],F) = [A,B,C,D|append(E,F)].           % Iteration 4
************************************************
```

```
***                T_R(I^-)                    ***
************************************************
[] = [].                                [A|B] = [A|B].
append(A,B) = append(A,B).              last(X) = last(X).
last([]) = [].
************************************************
***         DIAGNOSIS:  ERROR FOUND          ***
************************************************
The equation: last([]) = []. in T_R(I^-) is incorrect.
The incorrect rule is: last([]) = [].

            CORRECTION OPTIONS
************************************************
1. Automatic correction.
2. Manual correction.

Enter option:  2.
Enter the position of the rule: 1.
Enter the correct clause: last([X])=X.
************************************************
***                   T_R^♯(I^+)                ***
************************************************
last([A|B]) = last(B).
last([A,B,C|D]) = last(D).
last([A,B,C,D|E]) = last(E).
last([A,B|C]) = last(C).
last([A]) = A.
last([A,B,C]) = C.
last([A,B,C,D]) = D.
last([A,B]) = B.
[] = [].
[A|B] = [A|B].
last(X) = last(X).
************************************************
>> Analyzing completeness...
************************************************
***            UNCOVERED EQUATIONS          ****
************************************************
append([A,B,C,D|E],F) = [A,B,C,D|append(E,F)].
```

```
append([A,B,C],D) = [A,B,C|D].
append([A,B,C|D],E) = [A,B,C|append(D,E)].
append([A,B],C) = [A,B|C].
append([A,B|C],D) = [A,B|append(C,D)].
append([A],B) = [A|B].
append([A|B],C) = [A|append(B,C)].
append([],B) = B.

                    OPTION
************************************************
1. Adding one rule
2. Correcting one rule

Introduce the option: 1.
Introduce the position of the rule: 3.
Introduce the new rule: append([],B)=B.
...
************************************************
1. Adding one rule
2. Correcting one rule

Introduce the option: 1.
Introduce the position of the rule: 4.
Introduce the new rule: append([A|B],C)=[A|append(B,C)].
...
************************************************
>> Analyzing completeness...
************************************************
        ... the program is totally correct.
```

Índice de Materias