Un Depurador Abstracto, Inductivo y Paramétrico para Programas Multiparadigma*

María Alpuente Frasnedo ** Francisco José Correa Zabala ***

Resumen

Presentamos un marco general para el diagnóstico abstracto de programas lógico-funcionales, válido para diferentes estrategias de estrechamiento. Asociamos a cada programa una semántica por punto fijo que modela las respuestas computadas. Nuestra metodología está basada en la interpretación abstracta y es paramétrica con respecto a la estrategia de cómputo. Gracias a que la aproximación del conjunto de éxitos que presentamos es finita, la metodología de diagnóstico que se propone puede ser usada de manera estática. Una implementación de nuestro sistema de depuración "BUGGY" demuestra experimentalmente que el método permite encontrar algunos errores comunes sobre una muestra amplia de programas.

Palabras Claves: *depuración declarativa, diagnóstico abstracto, interpretación abstracta, lenguaje lógico funcional, programación multiparadigma, semántica operacional, semántica de punto fijo.*

Abstract

We present a general framework for the abstract debugging of functional logic programs which is valid for different narrowing strategies. We associate to our programs a fixpoint semantics which models computed answers. Our methodology is based on abstract interpretation and it is parametric with respect to the computation strategy. By approximating the intended specification of the success set we derive a finitely terminating debugging method, which can be used statically. We use an implementation of our debugging system "BUGGY" which shows experimentally on a wide set of benchmarks that we are able to find some common errors in the user programs.

Keywords: *abstract diagnosis, abstract interpretation, declarative debugging, functional logic programs, multiparadigm programming, Operational and fixpoint semantics.*

1. Introducción

El problema de la integración entre la programación lógica y la programación funcional está considerado como uno de los más importantes en el área de investigación sobre programación declarativa [33]. Para que los lenguajes declarativos sean útiles y puedan utilizarse en aplicaciones reales, es necesario que el grado de eficiencia de su ejecución se aproxime al de los lenguajes imperativos, tal y como se ha conseguido con el lenguaje Prolog. Para ello, es imprescindible el desarrollo de herramientas potentes para la manipulación de programas, capaces de optimizar las implementaciones realizadas. En general, es deseable sustituir las aproximaciones ad-hoc por tratamientos más sistemáticos para los problemas

*Este trabajo ha sido parcialmente financiado por el MCYT proyecto TIC2001-2705-C03-01, HU2003-0003, por Generalitat Valenciana proyecto GR03/025, por ICT for EU-India Cross Cultural Dissemination Project proyecto ALA/95/23/2003/077-054 y por la Universidad EAFIT proyecto PY0223.

**Departamento de Sistemas Informáticos y Computación-DSIC, Universidad Politécnica de Valencia, Camino de Vera s/n, 46022, alpuente@dsic.upv.es

***Departamento de Informática y Sistemas–DIS, Grupo de Lógica y Computación, Universidad EAFIT, Carrera 49 7 Sur 50, A.A. 3300, fcorrea@eafit.edu.co

relacionados con la manipulación de programas. Puesto que la semántica de los lenguajes lógico-funcionales ha sido objeto de numerosos estudios y está matemáticamente bien formalizada, surge el interés por el desarrollo de métodos y técnicas formales para la formulación de optimizaciones, basadas en la semántica, que preserven las propiedades computacionales del programa. Este artículo presenta técnicas y herramientas formales que ayudan al programador a manipular y optimizar sus programas, en lo que respecta a la depuración declarativa de programas lógico-funcionales.

A pesar de que, potencialmente, los distintos paradigmas de programación declarativa 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, al carácter experimental de las implementaciones existentes 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.

El presente artículo se organiza como sigue. Las tres primeras secciones permiten al lector ubicarse en el contexto general de nuestra línea de investigación. En la sección 2 haremos una revisión de algunos conceptos preliminares del área; desarrollamos ideas generales sobre semánticas declarativas y operacionales, programas lógicos, programas funcionales y programas lógico-funcionales con sus correspondientes métodos de cálculo. En la sección 3 planteamos algunos de los elementos correspondientes a la depuración declarativa de programas lógicos puros y programas funcionales. En la sección 4 hacemos una descripción global de nuestro trabajo de investigación. Finalmente, en la sección 5 presentamos nuestras conclusiones y trabajo futuro.

2. Preliminares

En esta sección resumimos, brevemente, algunos de los elementos fundamentales del marco teórico, [5, 6, 4]. En este artículo, $\mathcal{V}$ denota un conjunto infinito enumerable de *variables* y Σ denota el conjunto de símbolos de *función* (o *signatura*), cada uno de las cuales tiene una aridad previamente fijada. $\mathcal{T}(\Sigma \cup \mathcal{V})$ denota el conjunto de *términos* construidos usando los símbolos de Σ y $\mathcal{V}$. $\mathcal{T}(\Sigma)$ denota el conjunto de *términos básicos ("ground")*[1] y es el habitual *universo de Herbrand* sobre Σ y lo denotaremos por $\mathcal{H}$. Una Σ-*ecuación* tiene la forma $s = t$ donde $s, t \in \mathcal{T}(\Sigma \cup \mathcal{V})$. El conjunto de todas las *ecuaciones básicas* que pueden ser construidas con elementos de $\mathcal{H}$ se denota por $\mathcal{B}$ y se denomina *base de Herbrand*. De manera similar, nos referimos a $\mathcal{H}_\mathcal{V}$ para denotar el universo de Herbrand con variables, es decir, donde se permiten variables como componentes de los términos y a $\mathcal{B}_\mathcal{V}$ la base de Herbrand con variables sobre $\mathcal{T}(\Sigma \cup \mathcal{V})$.

Una *expresión* es 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 $e \in \mathcal{B}_\mathcal{V}$. El conjunto de variables que aparecen en una expresión e se denota por $\mathcal{V}ar(e)$. Como consecuencia, una expresión t es *básica* si $\mathcal{V}ar(t) = \emptyset$.

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. $O(t)$ denota el conjunto de posiciones del término t y $\overline{O}(t)$ denota el conjunto de posiciones no variables del término t. $t|_p$ es el

[1] Sin variables.

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: $\overline{O}(g) = \{i.u \mid u \in \overline{O}(e_i), i = 1, \ldots, n\}$.

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 aún, si dos términos son unificables entonces existe un unificador más general idempotente entre ellos. Estas nociones pueden ser extendidas a otros objetos sintácticos. 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 [40].

Describimos el retículo de ecuaciones sintácticas [14]. Eqn denota el conjunto finito de ecuaciones sobre términos posiblemente cuantificadas existencialmente [43]. 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 resuelta 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 [40], mientras que para el conjunto de ecuaciones cuantificadas esto no es cierto en general [44]. Por abuso en ocasiones trataremos los conjuntos de ecuaciones como secuencias.

2.1. 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 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 si su modelo se corresponde con las intenciones del programador, para validar su corrección (y otras propiedades) respecto de la funcionalidad esperada.

2.1.1. 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 [51] para una clasificación). Algunos de los más interesantes 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 es necesario diferenciarlos debido a que la semántica por punto fijo se aplica también a otro tipo de aproximaciones.

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 constante, de función y de predicado (símbolos de base del lenguaje) a elementos, funciones y relaciones del 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, es decir, 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 construido de un modelo totalmente sintáctico), el teorema de completitud de Gödel (equivalencia entre derivabilidad e implicación semántica) y el teorema de Gödel sobre incompletitud 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, es utilizado para caracterizar el significado de este tipo de programas.

En la aproximación por punto fijo se define el significado de un programa P como un determinado punto fijo T_P de una transformación T, continua, asociada a dicho programa.[2] La continuidad de T implica la existencia de un punto fijo mínimo, $lfp(T)$, 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 punto fijo mínimo.

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 programa 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 hechos nuevos. El menor punto fijo así obtenido coincide con el menor modelo [38]. Así pues, el punto fijo 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 pruebas de equivalencia entre ambas.

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, 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.

2.1.2. Semántica Operacional

La semántica operacional es, posiblemente, la más antigua de las definiciones semánticas. 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 de la semántica operacional de un lenguaje de programación equivale a la definición de un intérprete para el lenguaje, 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 implementación concreta de un lenguaje no tiene por qué seguir los mismos mecanismos operacionales de su definición semántica, pero sí ha de obtener los

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

mismos resultados. Una de las aproximaciones más relevantes a la definición de semántica operacional es la definida por Plotkin [53] como "Structural Operational Semantics" (SOS), basada en un concepto de máquina verdaderamente abstracta y muy simple: el formalismo de los *Sistemas de Transición*.

2.2. 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 [38, 39], *"programa = lógica + control"*. 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. Según J. A. Robinson, la programación declarativa utiliza la lógica como lenguaje de programación,[3] lo cual puede conceptualizarse 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". Los principales requisitos que debe cumplir la lógica empleada son: disponer de un lenguaje que sea suficientemente expresivo, disponer de una semántica operacional, disponer de una semántica declarativa y obtener resultados de corrección y completitud que aseguren 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). La programación declarativa incluye tanto la programación lógica como la funcional, cuyas principales características se resumen en los próximos apartados.

2.3. Programación Lógica

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

La programación lógica [11, 38, 42] se define tomando como base subconjuntos de la lógica de predicados, siendo la más popular la lógica de cláusulas de Horn (HCL, del inglés Horn clause logic), que pueden emplearse en un lenguaje de programación al poseer una semántica operacional susceptible de una implementación eficiente, como es el caso de la resolución SLD. Como semántica declarativa se utiliza una semántica por teoría de modelos que toma 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 el enlace de un valor a una variable lógica. La resolución SLD es un método de prueba correcto y completo para la lógica HCL.

Los lenguajes declarativos proporcionan una gestión automática de la memoria, evitando una de las mayores fuentes de error en los programas implementados en otros lenguajes. Además, el programa puede responder, haciendo uso del mecanismo de resolución SLD, a diferentes consultas (objetivos) sin necesidad de efectuar ningún cambio en el programa, gracias al hecho de emplearse un mecanismo de cómputo que permite una búsqueda indeterminista (built-in search) de soluciones. Esto permite computar con datos parcialmente definidos y hace posible que la relación de entrada/salida no esté fijada de antemano.

[3] Ver [61] para una discusión completa.

2.4. Programación Funcional

El fundamento esencial de los *lenguajes funcionales* es el concepto de *función* (matemática). Los programas son conjuntos de ecuaciones (generalmente recursivas) que definen funciones. 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*, 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 sustitución 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 mismos.

Una de las características de los lenguajes funcionales es el empleo de las funciones como "ciudadanos de primera clase" dentro del lenguaje, de forma que éstas puedan almacenarse en estructuras de datos, pasarse como argumento a otras funciones y devolverse como resultados. Estas características se referencian como *orden superior*. De otro modo, 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* [52, 54]. La ejecución del programa consiste en la evaluación del objetivo de acuerdo con las ecuaciones y alguna forma de *reducción*. La reducción es una secuencia de pasos que transforma la expresión inicial, compuesta de símbolos de función y 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. A manera de ejemplo, podemos distinguir dos estrategias de reducción, la *impaciente* y la *perezosa*. La estrategia impaciente evalúa primero los argumentos de la 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 liberan al programador de la preocupación por el orden de evaluación, y las ventajas expresivas que proporciona, como la habilidad de computar con estructuras infinitas, han hecho que el modo evaluación perezosa sea muy apreciado en los lenguajes funcionales modernos.

2.5. 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 como por ejemplo determinismo y evaluación perezosa.

2.6. 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 en donde la reescritura o reducción juega el papel de intérprete completo y una construcción algebraica apropiada refleja el modelo estándar. La relación de reescritura se entiende como la deducción ecuacional que usa las ecuaciones orientadas 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 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[4] y el razonamiento ecuacional mediante un sistema de deducción ecuacional que plasma 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.

Un *sistema de reescritura de términos condicional* (SRTC para abreviar) es un par $(\Sigma, \mathcal{R})$, donde $\mathcal{R}$ es un conjunto finito de (esquemas de) reglas de reescritura (o reglas de reducción) de la forma $\lambda \to \rho \Leftarrow C$, donde $\lambda, \rho \in \mathcal{T}(\Sigma \cup \mathcal{R})$ y $\lambda \notin \mathcal{V}$ y $\mathcal{V}ar(\lambda) \subseteq \mathcal{V}ar(\rho)$. La condición C es una secuencia (posiblemente vacía) de ecuaciones $e_1, \ldots, e_n$, tal que $n \geq 0$. Las variables en ρ ó C que no aparecen en λ reciben el nombre de *variables extras.* Cuando una regla de reescritura no posee condición, escribimos simplemente $\lambda \to \rho$. 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 encuentran previamente durante la computación. Dado un SRTC $(\Sigma, \mathcal{R})$, asumimos que la signatura Σ está particionada en dos conjuntos disjuntos $\Sigma = \mathcal{C} \cup \mathcal{D}$, donde $\mathcal{D} = \{f | (f(t_1, \ldots, t_n) \Leftarrow C) \in R\}$ y $\mathcal{C} = \Sigma - \mathcal{D}$. Los elementos de $\mathcal{C}$ se denominan *constructores* y los de $\mathcal{D}$ *funciones definidas.* Un *objetivo ecuacional* es una regla sin cabeza o, equivalentemente, una secuencia de ecuaciones. Denotamos por *Goal* el conjunto de objetivos ecuacionales $\Leftarrow g$ que, en lo que sigue, denotamos simplemente por g.

3. Depuración de Programas

Nosotros consideramos la siguiente noción general de denotación de un programa.

Definición 3.1 *Una semántica $S(\mathcal{R})$ para un programa $\mathcal{R}$ es un subconjunto de $\mathcal{B}_\mathcal{V}$, (es decir, $S(\mathcal{R}) \subseteq \mathcal{B}_\mathcal{V}$).*

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

La depuración es el proceso mediante el cual los programas se corrigen para eliminar las discrepancias entre lo que el programa calcula realmente y lo que desea el programador. La búsqueda de estos errores 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 [47]. Aunque en [58, 30] el diagnóstico se refiere a la identificación de un error en un programa que se comporta incorrectamente y la depuración a la identificación, ubicación y corrección del error [15]. El objetivo principal de la depuración es eliminar los errores que se encuentran en este proceso. El diagnóstico declarativo se define como sigue [15], en concordancia con los enfoques dados en este campo por [58, 42, 29].

Definición 3.2 *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 $\mathcal{M}$, 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. Las semánticas declarativas exploradas más comúnmente son: el modelo mínimo de Herbrand [58], el conjunto de modelos de la compleción del programa [41] y el conjunto de consecuencias lógicas atómicas del programa [29, 30].

Adicionalmente, es posible enfocar los métodos de diagnóstico de acuerdo con una propiedad que se quiera 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, terminación, exitos y fallos (definidos en [17]). La siguiente definición formaliza el diagnóstico con respecto al observable α [17] y es una extensión para el diagnóstico de respuestas computadas de las definiciones dadas en [58, 41, 29].

Definición 3.3 *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 $[\mathcal{P}]_\alpha$ la semántica operacional de $\mathcal{P}$ con respecto a α, entonces*

- *$\mathcal{P}$ es **parcialmente correcto** con respecto a $\mathcal{M}_\alpha$ si $[\mathcal{P}]_\alpha \subseteq \mathcal{M}_\alpha$,*
- *$\mathcal{P}$ es **completo** con respecto a $\mathcal{M}_\alpha$ si $\mathcal{M}_\alpha \subseteq [\mathcal{P}]_\alpha$ y*
- *$\mathcal{P}$ es **totalmente correcto** con respecto a $\mathcal{M}_\alpha$ si $\mathcal{M}_\alpha = [\mathcal{P}]_\alpha$.*

Un síntoma es la aparición de una anomalía durante la ejecución de un programa [30], que resulta de la comparación de la semántica del programa con la semántica deseada por el programador, como lo precisa la siguiente definición [18].

Definición 3.4 *Sea $\mathcal{P}$ un programa, $S(\mathcal{P})$ la semántica de $\mathcal{P}$ e $\mathcal{I}$ la semántica deseada de $\mathcal{P}$.*

- *Un **síntoma de incorrección** es una ecuación e tal que $e \in S(\mathcal{P})$ y $e \notin \mathcal{I}$.*
- *Un **síntoma de incompletitud** es una ecuación e' tal que $e' \in \mathcal{I}$ y $e' \notin S(\mathcal{P})$*

Desde el punto de vista de la ejecución del programa, un síntoma de incorrección es un resultado que no está en las expectativas del programador y un síntoma de incompletitud se presenta cuando el programa es incapaz de computar algún resultado esperado por el programador.

Los algoritmos de depuración declarativa clásicos exigen que la semántica deseada $\mathcal{M}$ sea declarada extensionalmente. Sin embargo, en general $\mathcal{M}$ es infinita. Los algoritmos pueden manejar este tipo de semánticas trabajando con síntomas que se obtienen usando técnicas de generación de tests. Estos algoritmos son llamados "algoritmos dirigidos por síntomas". Otra forma de manejar semánticas infinitas es utilizar aproximaciones de la semántica, que generalmente se basan en la teoría de *Interpretación Abstracta*.

Las técnicas de *diagnóstico abstracto* aparecen como una combinación de tres técnicas bien conocidas: la depuración declarativa [58, 41], la aproximación conocida como s-semántica (o semántica de respuestas computadas) para la definición de programas que modelen varios comportamientos observables y la teoría de la interpretación abstracta. El diagnóstico abstracto es una generalización de la técnica de diagnóstico declarativo en la que se consideran propiedades relacionadas con la semántica operacional y de observables, en lugar de las propiedades basadas en la semántica declarativa. De esta forma, se obtiene un marco más potente con el que desarrollar herramientas para el diagnóstico y la verificación de los programas.

Por otro lado, 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 las ecuaciones [58, 41, 48]. El depurador establece un diálogo con el usuario, que actúa como oráculo e identifica síntomas de error (respuestas incorrectas o respuestas perdidas), para que el sistema localice la causa del fallo y emita el diagnóstico. También se puede describir mediante una especificación ejecutable [22, 25, 23, 37], la cual puede resultar particularmente útil cuando las estructuras de datos son complejas. Una estrategia adicional es el método de aserciones, donde el oráculo se reemplaza por anotaciones en el programa (aserciones) que reflejen adecuadamente el significado deseado del programa. Las técnicas de interpretación abstracta son también usadas para determinar una representación adecuada de la semántica deseada y son el fundamento de nuestro enfoque.

3.1. Programas Lógicos Puros

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 *modelo mínimo de Herbrand* del programa, la cual es adecuada para modelar las características de los programas lógicos en el caso en el que no se incluyen variables en las expresiones. Las s-semánticas, al permitir variables en la especificación, constituyen una caracterización más cercana al comportamiento del programa y, consecuentemente, constituyen una herramienta más potente de ayuda a las técnicas de depuración declarativa. En ellas la noción de interpretación [27, 26] es un subconjunto de la base de Herbrand extendida, $\mathcal{B}_\mathcal{V}$, formada por todos los átomos, posiblemente con variables, inducidos por el programa $\mathcal{P}$ (módulo la varianza inducida por el cambio de nombre).

En el enfoque de las s-semánticas, se establece la equivalencia entre la semántica operacional $\mathcal{O}^s$, con respecto a un programa $\mathcal{P}$, y el menor punto fijo del operador de consecuencias inmediatas $T^s_\mathcal{P}$, con respecto a un programa $\mathcal{P}$, para el observable de respuestas computadas y, en consonancia, permite generar algoritmos para el diagnóstico de programas lógicos cuya estrategia de búsqueda puede ser descendente o ascendente. En [18], se propone un algoritmo de diagnóstico cuya estrategia es descendente y se diferencia de las que se basan en árboles de computación [58] o demostración [48, 41, 12] en los que la estrategia de búsqueda se realiza con objetivos atómicos más generales, que no precisan síntomas como entradas y que se introduce la posibilidad de simular el oráculo. La diferencia radica en que en el proceso de diagnóstico se conoce la s-semántica deseada $\mathcal{I}$ y el $T^s_\mathcal{P}(\mathcal{I})$ y, al compararlos, se determinan las cláusulas incorrectas del programa en el caso en que el operador $T^s_\mathcal{P}$ tiene un único punto fijo. La simulación del oráculo también se basa en este argumento.

La generación de respuestas computadas, en programas lógicos puros, también puede

ser descrita usando *árboles de computación o demostración* y 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 [57], por una transformación del programa [49] o se puede usar el átomo para representar de manera implícita su propio árbol de demostración [48].

La depuración declarativa para respuestas computadas incorrectas consiste en la búsqueda de un *nodo equívoco* (nodo no válido con hijos válidos). Para ello, el árbol de computación se recorre preguntando al oráculo por la validez de cada nodo. En [48], se prueba 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 esquema general para algoritmos de diagnóstico declarativo de respuestas incorrectas, que se aplica al caso de árboles de computación finitos que tengan nodos equívocos más altos.

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. Las diferentes aproximaciones varían según 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. 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 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, dado que, existen programas que no tienen átomos que no están cubiertos y no son completos [18, 19].

3.2. Programas Funcionales

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.

La evaluación de una expresión se representa en un *árbol de demostración* (árbol de evaluación de dependencias en [60]) 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 [47]. 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 [50].

4. Depuración de Programas Lógico–Funcionales

Muchas propuestas para la integración usan sistemas de reescritura de términos condicionales (SRTC) como programas y alguna variante del *estrechamiento* (*narrowing*) como mecanismo operacional, soportando así unificación y variables lógicas en un contexto funcional. El procedimiento de *estrechamiento* puede verse como una extensión de la reescritura que puede ser implementado eficientemente sustituyendo emparejamiento por unificación en el procedimiento de reducción. En un paso de estrechamiento 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 las correspondientes instancias de las condiciones de la regla utilizada. El uso de estrechamiento 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 estrechamiento, como medio para la obtención de un conjunto de soluciones a un problema de unificación, fue descrita en los trabajos pioneros de [59] y [28]. La relación definida por Fay en [28], y que denominó estrechamiento (*narrowing*), es una variante que posteriormente ha recibido el nombre de *estrechamiento normalizante* [56, 34]. El estrechamiento normalizante se caracteriza por la normalización del término a evaluar previa a la aplicación de cada paso de estrechamiento.

En general, el procedimiento de estrechamiento es indeterminista, debido a la existencia de dos grados de libertad: la elección del subtérmino a reducir y la elección de la regla. Esto conduce a un espacio de búsqueda muy amplio. Se han diseñado muchas *estrategias* para reducir el tamaño del espacio de búsqueda, eliminando algunas derivaciones inútiles, entre las que podemos citar: *estrechamiento el más interno a la izquierda* [31], *estrechamiento básico* [36, 45], *estrechamiento básico el más interno* [35], *estrechamiento el más externo a la izquierda* [24], *estrechamiento perezoso* [10, 32, 46, 55] y *estrechamiento necesario* [10] entre otros. Cada una de estas estrategias sigue manteniendo, bajo determinadas condiciones, la completitud del cálculo.

Más formalmente, en [5, 6, 4] formulamos una definición general y paramétrica de estrechamiento con estrategia donde $inn(g)$ (resp. $out(g)$) denota la estrategia de estrechamiento que selecciona el "redex"[5] de g ubicado más a la izquierda más interno (respectivamente más a la izquierda más externo). Además, el método se define para reglas condicionales con respecto a la estrategia φ, $\varphi \in \{inn, out\}$

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

Si $\mathcal{R}$ es un programa, para $\varphi \in \{inn, out\}$, definimos $\mathcal{R}_+ = \mathcal{R} \cup \{Eq^{\varphi}\}$, donde Eq^{φ} son las reglas que modelan la igualdad sobre los términos constructores. Sea $c/n \in \mathcal{C}, 0 \leq n$.

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

Cuando $\varphi = inn$ es la igualdad estándar "=" de términos, mientras que para el caso en que φ es *out* y además se consideren STRC no terminantes es necesario diferenciar la igualdad estándar (no estricta) "=", que está definida sobre estructuras de datos parcialmente determinadas o infinitas, de la igualdad estricta "≈", definida para estructuras de datos finitas y completamente determinadas, que brinda a la igualdad el significado más

[5] Subtérmino reducible por la estrátegia.

débil de "identidad entre objetos finitos". Adicionalmente, asumimos que las ecuaciones en el objetivo g y en la condición C tienen la forma $s = t$ cuando consideramos $\varphi = inn$, mientras que las ecuaciones tienen la forma $s \approx t$ cuando consideramos $\varphi = out$.

4.1. Denotación de un Programa Lógico Funcional

En [5, 6] definimos la semántica $\mathcal{F}^{ca}_{\varphi}(\mathcal{R})$ para el programa $\mathcal{R}$ tal que las sustituciones de respuestas computadas por estrechamiento 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.

Cualquier secuencia de ecuaciones $\mathcal{E}$ puede ser transformada en una secuencia de ecuaciones planas equivalente, $flat_{\varphi}(\mathcal{E})$. El procedimiento de aplanamiento para conjuntos de ecuaciones que produce objetivos planos es paramétrico con respecto a la estrategia *inn* y *out*.

Definición 4.1 (Objetivo plano con respecto a φ) *Una ecuación plana es una ecuación de la forma $f(d_1, \ldots, d_n) = d$ ó $d_l =_{\varphi} d_m$, donde $d, d_1, \ldots, d_n, d_l, d_m$ son términos constructores. Un objetivo plano es un conjunto de ecuaciones planas.*

Semánticas de Punto Fijo. La formulación en [5, 6] se basa en la definición operador de consecuencias inmediatas $T^{\varphi}_{\mathcal{R}}$ que modela las respuestas computadas con respecto a la estrategia φ. Para ello se adicionan los axiomas reflexivos entre los símbolos de función constructores y definidas de la forma $f(x_1, \ldots, x_n) = f(x_1, \ldots, x_n)$ donde f es de aridad n. El hecho de que $T^{\varphi}_{\mathcal{R}}$ sea continuo nos permite definir una semántica de punto fijo como sigue.

Definición 4.2 *La semántica del menor punto fijo de un programa $\mathcal{R}$, está definida por $\mathcal{F}_{\varphi}(\mathcal{R}) = lfp(T^{\varphi}_{\mathcal{R}}), \varphi \in \{inn, out\}$. $\mathcal{F}^{ca}(\mathcal{R})$ denota el conjunto $\{l =_{\varphi} r \in lfp(T^{\varphi}_{\mathcal{R}}) | r$ no contiene símbolos de función definido $f/n \in \mathcal{D}\}$, $\varphi \in \{inn, out\}$.*

Ejemplo 1 *Sea $R = \{\mathtt{g(x)} \rightarrow \mathtt{0},\ \mathtt{f(0)} \rightarrow \mathtt{0},\ \mathtt{f(s(x))} \rightarrow \mathtt{f(x)}\}$ entonces, las continuas aplicaciones del operador $T^{\varphi}_{\mathcal{R}}$ con respecto a $\varphi = inn$, son:*

$$
\begin{array}{ll}
T^{inn}_{\mathcal{R}} \uparrow 0 = & \{\mathtt{0} = \mathtt{0},\ \mathtt{s(x)} = \mathtt{s(x)},\ \mathtt{g(x)} = \mathtt{g(x)},\ \mathtt{f(x)} = \mathtt{f(x)}\} \\
T^{inn}_{\mathcal{R}} \uparrow 1 = & T^{inn}_{\mathcal{R}} \uparrow 0 \cup \{\mathtt{g(x)} = \mathtt{0},\ \mathtt{f(0)} = \mathtt{0},\ \mathtt{f(s(x))} = \mathtt{f(x)}\} \\
T^{inn}_{\mathcal{R}} \uparrow 2 = & T^{inn}_{\mathcal{R}} \uparrow 1 \cup \{\mathtt{f(s(0))} = \mathtt{0},\ \mathtt{f(s^2(x))} = \mathtt{f(x)}\} \\
\vdots & \\
lfp(T^{inn}_{\mathcal{R}}) = & \{\mathtt{0} = \mathtt{0},\ \mathtt{s(x)} = \mathtt{s(x)},\ \mathtt{g(x)} = \mathtt{g(x)}, \mathtt{f(x)} = \mathtt{f(x)},\ \mathtt{g(x)} = \mathtt{0}, \\
 & \mathtt{f(0)} = \mathtt{0}, \mathtt{f(s(0))} = \mathtt{0}, \ldots, \mathtt{f(s^n(0))} = \mathtt{0}, \ldots, \mathtt{f(s(x))} = \mathtt{f(x)}, \\
 & \ldots, \mathtt{f(s^n(x))} = \mathtt{f(x)}, \ldots\}
\end{array}
$$

Sea $I\!R_{\varphi}$ la clase de programas para los que la estrategia φ de *estrechamiento* es completa con respecto al computo de respuestas. El siguiente teorema relaciona las respuestas computadas por *estrechamiento* con respecto a φ y las substituciones calculadas por unificación con la semántica de respuestas computadas de punto fijo [20].

Teorema 4.3 (Corrección y completitud fuertes) *Sea $\mathcal{R}$ un programa en $I\!R_{\varphi}$, y g un objetivo (no trivial[6]) de acuerdo con φ. Entonces θ es una sustitución de respuesta computada por la estrategia de estrechamiento φ, para g en $\mathcal{R}$ si y solo si existe $g' \ll \mathcal{F}^{ca}_{\varphi}(\mathcal{R})$ tal que $\theta = mgu(flat_{\varphi}(g), g')_{|var(g)}$*

[6] Un objetivo es trivial si es de la forma $x = y$ con $x, y \in \mathcal{V}$.

De acuerdo con el Teorema 4.3, $\mathcal{F}^{ca}_{\varphi}(\mathcal{R})$ 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 $flat_{\varphi}(g)$ en el programa, $\mathcal{F}^{ca}_{\varphi}(\mathcal{R})$ por unificación estándar, como si el símbolo de igualdad fuera un predicado ordinario.

Ejemplo 2 *Consideremos de nuevo el programa R del Ejemplo 1. De acuerdo con la Definición 4.2,* $\mathcal{F}^{ca}_{inn}(\mathcal{R}) = \{\mathtt{0} = \mathtt{0},\ \mathtt{s(X)} = \mathtt{s(X)},\ \mathtt{g(X)} = \mathtt{0},\ \mathtt{f(0)} = \mathtt{0},\ \mathtt{f(s(0))} = \mathtt{0}, \ldots, \mathtt{f(s^n(0))} = \mathtt{0}, \ldots\}$. *Dado el objetivo* $g \equiv\ (\mathtt{y} = \mathtt{f(z)})$, *estrechamiento impaciente calcula las respuestas computadas* $\{\ \{y/\mathtt{0}, \mathtt{z/0}\},\ \{\mathtt{y/0, z/s(0)}\}, \ldots, \{\mathtt{y/0, z/s^n(0)}\}, \ldots\}$ *en* $\mathcal{R}$, *que coinciden exactamente con el conjunto de substituciones computadas por unificación del objetivo plano* $\mathtt{f(z)} = \mathtt{y}$ *con las ecuaciones en* $\mathcal{F}^{ca}_{inn}(\mathcal{R})$.

Semántica de respuestas computadas. La semántica operacional del conjunto de éxitos $\mathcal{O}^{ca}_{\varphi}(\mathcal{R})$ de un programa $\mathcal{R}$ con respecto a la semántica de estrechamiento φ se define considerando las respuestas computadas para "llamadas más generales" (objetivos planos cuyos argumentos son variables) [5].

Definición 4.4 *Sea* $\mathcal{R} \in I\!R_{\varphi}$ *y sea* $\Im^{\varphi}_{\mathcal{R}}$ *el conjunto de los axiomas reflexivos para los símbolos definidos de* $\mathcal{R}$. *Entonces,* $\mathcal{O}^{ca}_{\varphi}(\mathcal{R}) = \Im^{\varphi}_{\mathcal{R}} \cup \{(f(x_1, \ldots, x_n) = x_{n+1})\theta \mid$ *con el objetivo* $(f(x_1, \ldots, x_n) =_{\varphi} x_{n+1})$ *se calcula* θ *por medio de la estrategia* φ *de estrechamiento y* $x_1, \ldots, x_n \in \mathcal{V}\}$

La equivalencia entre la semántica operacional y la del menor punto fijo queda establecida por el siguiente teorema [20].

Teorema 4.5 *Sea* $\mathcal{R} \in I\!R_{inn}$ *entonces* $\mathcal{O}^{ca}_{inn}(\mathcal{R}) = \mathcal{F}_{inn}(\mathcal{R})$. *Si* $\mathcal{R} \in I\!R_{out}$ *entonces* $\mathcal{O}^{ca}_{out}(\mathcal{R}) = \{l = r \in \mathcal{F}^{ca}_{out}(\mathcal{R}) | \bot$ *no ocurre en* $r\}$

4.2. Conjunto de Éxitos Abstracto

Dado que la denotación de un programa generalmente consiste de un número infinito de ecuaciones, las condiciones para la corrección y completitud de un programa con respecto a una especificación dada no pueden ser computadas efectivamente. Los análisis de flujo de datos son un componente escencial 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 [21] proporciona ayudas para 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" en la cual una computación se realiza con "descripciones de datos" en lugar de los datos mismos. Los operadores semánticos son reemplazados por operadores abstractos que aproximan de forma "segura" los operadores estándar. Teniendo en cuenta la teoría de interpretación abstracta, el diagnóstico abstracto es una aproximación correcta de la metodología de diagnóstico presentada anteriormente, en donde los dominios semánticos y los operadores son reemplazados por sus correspondientes versiones abstractas.

Nuestro análisis está basado en una simplificación (abstracción) del programa que aproxima las computaciones del programa original, de modo que, las computaciones en el programa aproximado siempre terminan y los objetivos pueden ser ejecutados eficientemente. Nuestra noción de programa abstracto es paramétrica con respecto a un *grafo de prevención de bucles*, Figura 1. Este es un grafo finito de dependencias entre términos que se construye a partir del programa y es independiente del objetivo. Permite reconocer aquellas derivaciones que con toda seguridad terminan. Dos instancias diferentes pueden ser encontradas en [8, 9].

Definición 4.6 *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 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 exite una secuencia infinita:

$\langle \Leftarrow G_0, \theta_0 \rangle \rightsquigarrow \langle \Leftarrow G_1, \theta_1 \rangle \rightsquigarrow \ldots$

entonces $\exists i \geq 0.\ \langle \overset{\circ}{t_i}, \overset{\circ}{t_i} \rangle \in \mathcal{G}_\mathcal{R}^+,$ donde $t_i = e_{|u}\theta_i,\ e \in G_i \wedge u \in \bar{O}(e)$. (En adelante diremos que $\langle \overset{\circ}{t_i}, \overset{\circ}{t_i} \rangle$ es un 'bucle' de $\mathcal{G}_\mathcal{R}$.)

Un programa abstracto se obtiene por simplificación del lado derecho de las cabezas de las reglas y las ecuaciones en el cuerpo de las mismas. La definición está dada inductivamente sobre la estructura de los términos y ecuaciones. La idea principal es eliminar del programa aquellos términos cuyo nodo correspondiente en el grafo $\mathcal{G}_\mathcal{R}$ es un bucle, reemplazándolos por $\sharp$. Primero abstraemos una regla r obteniendo $r^\sharp$ (seleccionamos primero las reglas recursivas, reemplazamos r por $r^\sharp$ en $\mathcal{R}$ y recomputamos finalmente el grafo de prevención de bucles, antes de proceder con la abstracción de la siguiente regla).

Definición 4.7 (Programa abstracto) *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*

$$sh(x,\mathcal{G}) = \begin{cases} x & si\ x \in V \\ f(sh(t_1,\mathcal{G}),\ldots,sh(t_k,\mathcal{G})) & si\ x \equiv f(t_1,\ldots,t_k)\ y\ \langle \overset{\circ}{x}, \overset{\circ}{x} \rangle \notin \mathcal{G}^+ \\ sh(l,\mathcal{G}) = sh(r,\mathcal{G}) & si\ x \equiv (l = r) \\ sh(e_1,\mathcal{G}),\ldots,sh(e_n,\mathcal{G}) & si\ x \equiv e_1,\ldots,e_n \\ \sharp & en\ otro\ caso \end{cases}$$

Ejemplo 3 *Consideremos el programa* $\mathcal{R} = \{\mathtt{g(0)} \rightarrow \mathtt{0},\ \mathtt{g(c(x))} \rightarrow \mathtt{c(g(x))},\ \mathtt{f(0)} \rightarrow \mathtt{0},\ \mathtt{f(c(x))} \rightarrow \mathtt{x} \Leftarrow \mathtt{g(c(x))} = \mathtt{c(x)}\}$. *El correspondiente grafo de dependencias entre términos [9] está dado por el grafo $\mathcal{G}_\mathcal{R}$ que se muestra en la Figura 1.*

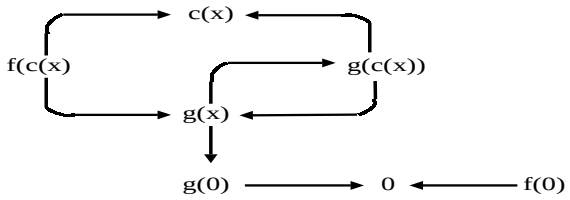

Figura 1: Grafo de prevención de bucles

Se debe notar que existen dos bucles en el grafo, concretamente $\{\langle \mathtt{g(x)}, \mathtt{g(x)} \rangle, \langle \mathtt{g(c(x))}, \mathtt{g(c(x))} \rangle\}$. *Podemos definir la función (parcial) $\overset{\circ}{t}$ como sigue: $\overset{\circ}{t}= t'$ asigna a un término t algún nodo t' en el grafo tal que t' unifica con t, si tal nodo t' existe.*

Partiendo de la semántica de punto fijo, introducida en [5, 6] 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. Los resultados para las técnicas de aproximación son paramétricos con respecto a la estrategia de estrechamiento (similar a la semántica concreta) y a la noción de programa abstracto. El correspondiente operador de consecuencias inmediatas abstracto $T_\mathcal{R}^{\sharp\varphi}$ aproxima

el operador $T_{\mathcal{R}}^{\varphi}$ y con él se obtiene la correspondiente semántica de punto fijo abstracta $\mathcal{F}_{\varphi}^{\sharp}(\mathcal{R})$ y $\mathcal{F}_{\varphi}^{ca\sharp}(\mathcal{R})$. 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. $\mathcal{F}_{\varphi}^{ca\sharp}(\mathcal{R})$), con la propiedad de que podemos computar una aproximación abstracta de la semántica concreta en un número finito de pasos y aproxima adecuadamente a la semántica concreta. Adicionalmente, la semántica de punto fijo abstracta recoge información (de forma independiente del objetivo) acerca de los patrones de éxito de un programa dado. La relación entre semántica de punto fijo abstracta y la semántica operacional concreta (sustitutiones de respuestas computadas) está dada por un teorema similar al Teorema 4.3. Intuitivamente, dado un objetivo g y un programa $\mathcal{R}$, 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 $\mathcal{F}_{\varphi}^{ca\sharp}(\mathcal{R})$.

4.3. Diagnóstico Abstracto

Aplicamos la teoría de interpretación abstracta para el desarrollo de un depurador eficiente que se basa en una noción de aproximación por exceso y por defecto 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 específica deseada $\mathcal{I}$ y otro conjunto $\mathcal{I}^{-}$ que aproxima por defecto $\mathcal{I}$.

Los siguientes resultados son la base para la construcción del depurador.

Proposición 4.8 *Si existe una ecuación e tal que $e \notin \mathcal{I}^{-}$ y $e \in T_r(\mathcal{I}^{-})$, entonces la regla $r \in \mathcal{R}$ es incorrecta sobre e.*

Proposición 4.9 *Si existe una ecuación e tal que $e \notin T_{\mathcal{R}}(\mathcal{I}^{+})$ y $e \in \mathcal{I}^{-}$, entonces la ecuación e es no cubierta.*

En una primera aproximación definimos $\mathcal{I}$ el programa que especifica la semántica deseada. Consideramos $\mathcal{I}^{+} = lfp(T_{\mathcal{I}}^{\varphi\sharp})$, es decir consideramos el conjunto de éxitos abstractos como la aproximación por exceso del conjunto de éxitos del programa. Para definir la aproximación por defecto de $\mathcal{I}$, podemos simplemente quedarnos con el conjunto que resulta de un número finito de iteraciones del operador $T_{\mathcal{I}}^{\varphi}$ concreto. Con este argumento construimos nuestro sistema experimental "BUGGY".

Ejemplo 4 *Consideremos como estrategia $\varphi = inn$ y un programa $\mathcal{R}$ (incorrecto) el cual debería computar el último elemento de una lista dada.*

$$\begin{array}{ll} 1)\ \texttt{last([X])} & \rightarrow\ \texttt{[]}. \\ 2)\ \texttt{last([X | Y])} & \rightarrow\ \texttt{last(Y)}. \end{array}$$

Consideremos la especificación $\mathcal{I}$ de la semántica deseada:

$$\begin{array}{lrcl} 0) & \texttt{last(X)} & \rightarrow & \texttt{Y} \Leftarrow \texttt{append(Z,[Y])} = \texttt{X}. \\ 1) & \texttt{append([],B)} & \rightarrow & \texttt{B}. \\ 2) & \texttt{append([A | B],C)} & \rightarrow & \texttt{[A | append(B,C)]}. \end{array}$$

Entonces $\mathcal{I}^{\sharp}$ =

$$\begin{array}{lrcl} \{\ 0) & \texttt{last(X)} & \rightarrow & Y \Leftarrow \sharp = \texttt{X}. \\ 1) & \texttt{append([],B)} & \rightarrow & \texttt{B}. \\ 2) & \texttt{append([A | B],C)} & \rightarrow & \texttt{[A | D]} \Leftarrow \sharp = \texttt{D}.\} \end{array}$$

Después de tres iteraciones del operador $T_{\mathcal{I}}$ tenemos:
$\mathcal{I}^{-} = \{\texttt{[]} = \texttt{[]},\ \texttt{[A|B]} = \texttt{[A|B]},\ \texttt{last([Y])} = \texttt{last([Y])},\ \texttt{append(A,B)} = \texttt{append(A,B)},$

$\mathtt{last}([\mathtt{Y}]) = \mathtt{Y}$, $\mathtt{append}([],\mathtt{B}) = \mathtt{B}$, $\mathtt{append}([\mathtt{A}|\mathtt{B}],\mathtt{C}) = [\mathtt{A}|\mathtt{append}(\mathtt{B},\mathtt{C})]$, $\mathtt{append}([\mathtt{A}],\mathtt{B}) = [\mathtt{A} \mid \mathtt{B}]$, $\mathtt{append}([\mathtt{A},\mathtt{A}',\mathtt{B}'],\mathtt{C}) = [\mathtt{A}|[\mathtt{A}'|\mathtt{append}(\mathtt{B}',\mathtt{C})]]$, $\mathtt{last}([\mathtt{A},\mathtt{B}]) = \mathtt{B}\}$

En una iteración del operador $T_{\mathcal{I}}^{\sharp}$, obtenemos el punto fijo:
$\mathcal{I}^{\sharp} = \mathcal{I}^{+} = \{\ [] = [],\ [\mathtt{A}|\mathtt{B}] = [\mathtt{A}|\mathtt{B}],\ \mathtt{last}([\mathtt{Y}]) = \mathtt{last}([\mathtt{Y}]),\ \mathtt{append}(\mathtt{A},\mathtt{B}) = \mathtt{append}(\mathtt{A},\mathtt{B}),$
$\mathtt{append}([],\mathtt{B}) = \mathtt{B},\ \mathtt{append}([\mathtt{A}|\mathtt{B}],\mathtt{C}) = [\mathtt{A}|\sharp],\ \mathtt{last}(\mathtt{X}) = \mathtt{X},\ \mathtt{last}([\mathtt{X}|\mathtt{Y}]) = \sharp\}$

Ahora, $T_{\mathcal{R}}(\mathcal{I}^{-}) = \{\ [] = [],\ [\mathtt{A}|\mathtt{B}] = [\mathtt{A}|\mathtt{B}],\ \mathtt{last}([\mathtt{Y}]) = \mathtt{last}([\mathtt{Y}]),\ \mathtt{last}([]) = []\}$

Dado que $\mathtt{last}([]) = [] \in \mathtt{T}_{\mathcal{R}}(\mathcal{I}^{-})$*, y la ecuación* $\mathtt{last}([\mathtt{X}]) = [] \notin \mathcal{I}^{+}$ *la técnica concluye automáticamente que la correspondiente regla es incorrecta.*

5. Conclusiones

Nuestra propuesta [5, 6, 4] 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) estrechamiento condicional. En esta investigación presentamos un esquema genérico para la depuración abstracta de programas lógico funcionales que extiende los métodos propuestos en [13, 16] al caso de los lenguajes multiparadigma. Dichas extensiones distan mucho de ser inmediatas y en particular nuestro aporte está en plantear un método de diagnóstico paramétrico con respecto a la estrategia de evaluación del lenguaje: perezosa, voraz, básica y necesaria. Presentamos una semántica de punto fijo abstracta, la cual es paramétrica con respecto a la estrategia de estrechamiento, que caracteriza el conjunto de respuestas computadas abstractas de manera ascendente e independiente del objetivo. Utilizamos una técnica de aproximación de la semántica deseada del conjunto de éxitos, planteamos los conceptos de especificación por exceso $\mathcal{I}^{+}$ y especificación por defecto $\mathcal{I}^{-}$ para aproximar correctamente por exceso (respectivamente 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 [7] de nuestro sistema de depuración "BUGGY" que demuestra experimentalmente que el método permite encontrar algunos errores comunes sobre una muestra amplia de programas. Disponible en `http://www.dsic.upv.es/users/elp/soft.html`. Hemos aplicado nuestros resultados al desarrollo de técnicas para la inducción de programas lógico–funcionales [1, 2, 3], 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. En la actualidad estamos investigando técnicas para la aplicación de nuestras métodos a programas con gran cantidad de código y de aplicación real.

Referencias

[1] M. Alpuente, D. Ballis, F. Correa, and M. Falaschi. A Multi Paradigm Automatic Correction Scheme. In "Proc. of 11th Int'l Workshop on Functional and (Constraint) Logic Programming" (2002).

[2] M. Alpuente, D. Ballis, F. Correa, and M. Falaschi. The System for Automatic Correction nobug. Technical Report, UPV (2002). Available at URL: http://www.dsic.upv.es/users/elp/papers.html.

[3] M. Alpuente, D. Ballis, F. Correa, and M. Falaschi. Correction of functional logic program. In P. Degano, editor, "Programming Languages and Systems: 12th European Symposium on Programming, ESOP 2003, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2003, Warsaw, Poland, April 7-11, 2003.", volume 2618 / 2003 of "Lecture Notes in Computer Science". Springer-Verlag Heidelberg (2003).

[4] M. Alpuente, F. Correa, and M. Falaschi. Declarative debugging framework for functional logic programs.

[5] M. Alpuente, F. Correa, and M. Falaschi. Declarative Debugging of Funtional Logic Programs. In B. Gramlich and S. Lucas, editors, "Proc. of the International Workshop on Reduction Strategies in Rewriting and Programming (WRS 2001)", volume 57 of "Electronic Notes in Theoretical Computer Science". Elsevier Science Publishers (2001).

[6] M. Alpuente, F. Correa, and M. Falaschi. Debugging Scheme of Functional Logic Programs. In M. Hanus, editor, "Proc. of International Workshop on Functional and (Constraint) Logic Programming, WFLP'01", volume 64 of "Electronic Notes in Theoretical Computer Science". Elsevier Science Publishers (2002).

[7] M. Alpuente, F. Correa, M. Falaschi, and S. Marson. The Debugging System buggy. Technical Report, UPV (2001). Available at URL: http://www.dsic.upv.es/users/elp/soft.html.

[8] M. Alpuente, M. Falaschi, and F. Manzo. Analyses of Unsatisfiability for Equational Logic Programming. *Journal of Logic Programming* **22**(3), 221–252 (1995).

[9] M. Alpuente, M. Falaschi, and G. Vidal. A Compositional Semantic Basis for the Analysis of Equational Horn Programs. *Theoretical Computer Science* **165**(1), 97–131 (1996).

[10] S. Antoy, R. Echahed, and M. Hanus. A Needed Narrowing Strategy. In "Proc. 21st ACM Symp. on Principles of Programming Languages, Portland", pages 268–279, New York (1994). ACM Press.

[11] K. R. Apt. Introduction to Logic Programming. In J. van Leeuwen, editor, "Handbook of Theoretical Computer Science", volume B: Formal Models and Semantics. Elsevier, Amsterdam and The MIT Press, Cambridge, Mass. (1990).

[12] J. Boye, W. Drabent, and J. Maluszyński. Declarative diagnosis of constraint programs: an assertion-based approach. In "Proc. of the 3rd Int'l Workshop on Automated Debugging - AADEBUG'97", pages 123–141, Linkoping, Sweden (May 1997). U. of Linkoping Press.

[13] F. Bueno, P. Deransart, W. Drabent, G. Ferrand, M Hermenegildo, J. Maluszyński, and G. Puebla. On the role of semantic approximations in validation and diagnosis of constraint logic programs. In "Proc. of the 3rd. Int'l Workshop on Automated Debugging-AADEBUG'97", pages 155–170. U. of Linkoping Press (1997).

[14] M. Codish, M. Falaschi, and K. Marriott. Suspension Analysis for Concurrent Logic Programs. In K. Furukawa, editor, "Proc. of Eighth Int'l Conf. on Logic Programming", pages 331–345. The MIT Press, Cambridge, MA (1991).

[15] M. Comini, G. Levi, M. Meo, and G. Vitiello. Proving Properties of Logic Programs by Abstract Diagnosis. In M. Dams, editor, "Analysis and Verification of Multiple-Agent Languages, 5th LOMAPS Workshop", volume 1192 of "Lecture Notes in Computer Science", pages 22–50. Springer-Verlag (1996).

[16] M. Comini, G. Levi, M. C. Meo, and G. Vitiello. Abstract diagnosis. *Journal of Logic Programming* **39**(1-3), 43–93 (1999).

[17] M. Comini, G. Levi, and G. Vitiello. Abstract Debugging of Logic Programs. In L. Fribourg and F. Turini, editors, "Proc. Logic Program Synthesis and Transformation and Metaprogramming in Logic 1994", volume 883 of "Lecture Notes in Computer Science", pages 440–450. Springer-Verlag, Berlin (1994).

[18] M. Comini, G. Levi, and G. Vitiello. Declarative Diagnosis Revisited. In John W. Lloyd, editor, "Proceedings of the 1995 Int'l Symposium on Logic Programming", pages 275–287. The MIT Press (1995).

[19] M. Comini, G. Levi, and G. Vitiello. Efficient detection of incompleteness errors in the abstract debugging of logic programs. In M. Ducassé, editor, "Proc. 2nd International Workshop on Automated and Algorithmic Debugging, AADEBUG'95" (1995).

[20] F. Correa. "Depuración Declarativa de Programas Lógico Funcionales". PhD thesis, Universidad Politécnica de Valencia (2002). ISBN 84-688-5981-8.

[21] P. Cousot and R. Cousot. Abstract Interpretation: A Unified Lattice Model for Static Analysis of Programs by Construction or Approximation of Fixpoints. In "Proc. of Fourth ACM Symp. on Principles of Programming Languages", pages 238–252 (1977).

[22] N. Dershowitz and Y. Lee. Deductive Debugging. In "Proceedings of the Fourth IEEE Symposium on Logic Programming", pages 298–306, San Francisco, California (August 1987).

[23] W. Drabent, S.Ñadjim-Tehrani, and J. Maluszynski. The use of assertions in algorithmic debugging. In "Proceedings of the 1988 Iinternational Conference on Fifth Generation Computer Systems", pages 573–581, Tokyo, Japan (December 1988).

[24] R. Echahed. On completeness of narrowing strategies. In "Proc. of CAAP'88", pages 89–101. Springer LNCS 299 (1988).

[25] A. Edman and S. Tärnlund. Mechanization of an Oracle in a Debugging System. In "Proceedings of Eighth IJCAI", pages 553–555, Karlsruhe, Germany (August 1983).

[26] M. Falaschi, G. Levi, M. Martelli, and C. Palamidessi. Declarative Modeling of the Operational Behavior of Logic Languages. *Theoretical Computer Science* **69**(3), 289–318 (1989).

[27] M. Falaschi, G. Levi, M. Martelli, and C. Palamidessi. A Model-Theoretic Reconstruction of the Operational Semantics of Logic Programs. *Information and Computation* **103**(1), 86–113 (1993).

[28] M. Fay. First Order Unification in an Equational Theory. In "Proc of 4th Int'l Conf. on Automated Deduction", pages 161–167 (1979).

[29] G. Ferrand. Error Diagnosis in Logic Programming, and Adaptation of E.Y.Shapiro's Method. *Journal of Logic Programming* **4**(3), 177–198 (1987).

[30] G. Ferrand and A. Tessier. Declarative Debugging. *The Newsletter of the European Network in Computational Logic* **3**(1), 71–76 (1996).

[31] L. Fribourg. SLOG: a logic programming language interpreter based on clausal superposition and rewriting. In "Proc. of Second IEEE Int'l Symp. on Logic Programming", pages 172–185. IEEE, New York (1985).

[32] M. Hanus. Combining Lazy Narrowing with Simplification. In "Proc. of 6th Int'l Symp. on Programming Language Implementation and Logic Programming", pages 370–384. Springer LNCS 844 (1994).

[33] M. Hanus. The Integration of Functions into Logic Programming: From Theory to Practice. *Journal of Logic Programming* **19&20**, 583–628 (1994).

[34] M. Hanus. Towards the Global Optimization of Functional Logic Programs. In "Proc. of Fifth Int'l Conf. on Compiler Construction", pages 68–82. Springer LNCS 786 (1994).

[35] S. Hölldobler. "Foundations of Equational Logic Programming". Springer LNAI 353 (1989).

[36] J. M. Hullot. Canonical Forms and Unification. In "Proc of 5th Int'l Conf. on Automated Deduction", pages 318–334. Springer LNCS 87 (1980).

[37] T. Kanamori, T. Kawamura, M. Maeji, and K. Horiuchi. Logic Program Diagnosis Specification. Icot technical report tr-447, Institute for New Generation Computer Technology, Tokyo, Japan (March 1989).

[38] R. A. Kowalski. Predicate Logic as a Programming Language. In "Information Processing 74", pages 569–574. North-Holland (1974).

[39] R. A. Kowalski. "Logic for Problem Solving". North-Holland (1979).

[40] J. L. Lassez, M. J. Maher, and K. Marriott. Unification Revisited. In J. Minker, editor, "Foundations of Deductive Databases and Logic Programming", pages 587–625. Morgan Kaufmann, Los Altos, Ca. (1988).

[41] J. W. Lloyd. Declarative error diagnosis. *New Generation Computing* **2**(2), 133–154 (1987).

[42] J. W. Lloyd. "Foundations of Logic Programming". Springer-Verlag, Berlin (1987). Second edition.

[43] M. J. Maher. Complete Axiomatizations of the Algebras of Finite, Rational and Infinite Trees. In "Proc. of Third IEEE Symp. on Logic In Computer Science", pages 348–357. Computer Science Press, New York (1988).

[44] M. J. Maher. On Parameterized Substitutions. Technical Report RC 16042, IBM - T.J. Watson Research Center, Yorktown Heights, NY (1990).

[45] A. Middeldorp and E. Hamoen. Completeness Results for Basic Narrowing. *Applicable Algebra in Engineering, Communication and Computing* **5**, 213–253 (1994).

[46] J.J. Moreno-Navarro and M. Rodríguez-Artalejo. Logic Programming with Functions and Predicates: The language Babel. *Journal of Logic Programming* **12**(3), 191–224 (1992).

[47] L. Naish. Declarative Debugging of Lazy Functional Programs. *Australian Computer Science Communications* **15**(1), 287–294 (1993).

[48] L. Naish. A Declarative Debugging Scheme. Technical report 95/1, Department of Computer Science, University of Melbourne, Melbourne, Australia (February 1995).

[49] L. Naish and T. Barbour. Declarative Debugging of a Logical-Functional Language. Technical report 94/30, Department of Computer Science, University of Melbourne, Melbourne, Australia (December 1994).

[50] L. Naish and T. Barbour. Towards a portable lazy functional declarative debugger. Technical Report, Department of Computer Science, University of Melbourne, Melbourne, Australia (1995).

[51] C. Palamidessi. A fixpoint semantics for Guarded Horn Clauses. Technical Report CS-R8833, Centre for Mathematics and Computer Science, Amsterdam (1988).

[52] R. Plasmeijer and M. van Eekelen. "Functional Programming and Parallel Graph Rewriting." Addison Wesley (1993).

[53] G. Plotkin. A structured approach to operational semantics. Technical Report DAIMI FN-19, Computer Science Department, Aarhus University (1981).

[54] C. Reade. "Elements of Functional Programming". Addison-Wesley Publishing Company (1993).

[55] U.S. Reddy. Narrowing as the Operational Semantics of Functional Languages. In "Proc. of Second IEEE Int'l Symp. on Logic Programming", pages 138–151. IEEE, New York (1985).

[56] P. Réty. Improving basic narrowing techniques. In "Proc. of the Conf. on Rewriting Techniques and Applications", pages 228–241. Springer LNCS 256 (1987).

[57] S. Safra and E. Shapiro. Meta Interpreters for Real. In H.-J. Kugler, editor, "Information Processing 86, Dublin, Ireland", pages 271–278. North-Holland, Amsterdam (1986).

[58] E. Y. Shaphiro. "Algorithmic Program Debugging". The MIT Press, Cambridge, Massachusetts (1982). ACM Distinguished Dissertation.

[59] J.R. Slagle. Automated Theorem-Proving for Theories with Simplifiers, Commutativity and Associativity. *Journal of the ACM* **21**(4), 622–642 (1974).

[60] J. Sparud and H. Nilsson. The architecture of a debugger for lazy functional languages. In M. Ducassé, editor, "Proceedings of AADEBUG'95", Saint-Malo, France (May 1995).

[61] Peter Van Roy and Seif Haridi. "Concepts, Techniques, and Models of Computer Programming". MIT Press (2004). ISBN 0-262-22069-5.