Desarrollo Modular de Procesadores de Lenguajes a partir de Especificaciones Semánticas Reutilizables

Jose Emilio Labra Gayo

26 de junio de 2001

Tabla de Contenidos

Resumen

En este documento se realiza un estudio de las principales técnicas de especificación semántica de lenguajes de programación y se propone una nueva técnica. En la realización del estudio se ha realizado la especificación de un mismo lenguaje de programación en cada una de las técnicas y se han valorado las siguientes características de cada técnica: no ambigüedad, modularidad semántica, componentes semánticos reutilizables, facilidades de demostración de propiedades, posibilidad de desarrollo de prototipos, legibilidad, flexibilidad ante diferentes paradigmas y experiencia en especificación de lenguajes reales.

La técnica propuesta es una combinación de la semántica monádica modular y de los avances desarrollados en el campo de la programación genérica. Las ventajas de la nueva técnica son la modularidad semántica, componentes semánticos reutilizables, la obtención de prototipos de forma automática y la demostrabilidad de propiedades.

Para la obtención de prototipos ejecutables se ha implementado un *Sistema de prototipado de lenguajes*. El sistema contiene un metalenguaje de descripciones semánticas y permite chequear y ejecutar los lenguajes especificados. El sistema se ha implementado como un lenguaje de dominio específico empotrado en Haskell, un lenguaje de propósito general puramente funcional.

Con el fin de demostrar la flexibilidad de la técnica propuesta, se han desarrollado las especificaciones de cuatro ejemplos de lenguajes siguiendo los paradigmas de programación lógica, funcional, imperativa y Orientada a Objetos. Además, todas las especificaciones se han desarrollado de forma modular, reutilizando y combinando bloques comunes.

Abstract

In this document, we study the main semantic specification approaches of programming languages and we propose a new approach.

The study has been made specifying the same language in the different approaches and assessing the following features: non-ambiguity, semantic modularity, reusable semantic building blocks, verification facilities, prototype development, legibility, flexibility for different paradigms and experience in the specification of real languages.

The proposed method is a combination of modular monadic semantics and generic programming concepts. The advantages of this approach are: semantic modularity, reusable semantic building blocks, automatic protoype generation and property verification.

In order to obtain executable prototypes a *Language Prototyping System* has been implemented. The system contains a semantic description metalanguage and allows to test and execute the specified languages. It has been implemented as a domain specific language embedded in Haskell, a general purpose functional programming language.

In order to demonstrate the flexibility of our approach we have developed the specification of four languages based on the imperative, functional, object-oriented and logic programming paradigms. Furthermore, all the specifications have been made in a modular way reusing and combining common blocks.

Agradecimientos

A la memoria de mi padre

Muchas personas han contribuido de alguna forma al desarrollo de la presente tesis, bien respondiendo alguna de mis preguntas en listas de correo, bien charlando conmigo en alguna conferencia o simplemente dándome ánimos para que siguiese con la tesis. La siguiente lista refleja por orden alfabético algunas de estas personas: L. S. Barbosa, L. Duponcheel, Anton Ertl, J. Goodman, P. R. Henriques, J. Heering, R. Hinze, J. Hughes, J. Jeuring, Mark P. Jones, R. Kieburtz, R. Lämmel, E. Meijer, M. Mernik, , P. Mosses, A. Mycroft A. Pardo, F. Rubio, J. Saraiva, E. Visser, K. Wansbrouhgh, etc.

La principal fuente de consulta ha sido *Internet*. Aprovecho estas líneas para reconocer las enormes posibilidades que este medio ofrece para la investigación y agradecer a todas las personas que colaboran incluyendo sus artículos en la red. Otra fuente de consulta han sido las bibliotecas de diversos centros en las que he encontrado referencias que se me resistían en *Internet*, entre ellas, las bibliotecas de las universidades de Génova, Buffalo, Toronto, Canisius College, etc.

También quiero agradecer las bolsas de viaje para asistir a congresos, así como la subvención de la organización del *3rd International Summer School on Advanced Functional Programming*, la invitación de Rafael Lins para participar en la reunión del Grupo *IFIP 2.8 – Programación Funcional* en Recife, y la beca TMR de la comunidad europea para participar en el congreso *PLI'99* en París.

El director de esta tesis J. M. Cueva ha sido un apoyo constante, así como todos los del grupo Oviedo3 y allegados, cuyas actividades académico-festivas forman un ambiente de trabajo especialmente agradable.

También aprovecho para agradecer a mi familia por apoyarme y comprender mis ausencias, y especialmente a 'Ana' por estar siempre conmigo en los momentos buenos y malos.

Capítulo 1

Introducción

However, the fact that it is possible to push a pea up a mountain with your nose does not mean that this is a sensible way of getting it there
C. Strachey [221]

1.1 Introducción

Los lenguajes de programación constituyen una herramienta fundamental de la informática. De hecho, cualquier producto *software* es desarrollado utilizando uno o varios lenguajes de programación y la elección de lenguajes adecuados puede ser la clave del éxito de muchos proyectos informáticos.

En la actualidad, existe una enorme variedad de lenguajes de programación[1]. La descripción de estos lenguajes requiere identificar el formato de los programas que se pueden escribir (sintaxis) así como su comportamiento (semántica). Mientras que para la descripción sintáctica, la notación BNF se utiliza de forma prácticamente universal, para la descripción semántica, no existe un formalismo comúnmente aceptado. Por el contrario, la semántica se especifica habitualmente en lenguaje natural [242], lo cual supone un problema clave en el desarrollo de productos informáticos fiables. Si un lenguaje de programación no tiene una semántica claramente definida, es imposible garantizar un comportamiento adecuado de los programas escritos en él.

Se han propuesto diversos formalismos para la descripción semántica de lenguajes de programación, como la semántica operacional estructurada, natural, denotacional, algebraica, axiomática, de acción, etc. Sin embargo, en la práctica, estos formalismos son escasamente utilizados en el diseño y especificación de lenguajes de programación reales.

A continuación se enumeran una serie de características que podrían considerarse *ideales*. Estas características servirán de criterios a la hora de evaluar los diferentes formalismos en la sección 10.1.

- *No ambigüedad.* La herramienta no debe admitir descripciones de características que puedan interpretarse de forma ambigüa al implementar el lenguaje.

[1] En [227] se incluye una lista de más de 2000 lenguajes de programación

- *Modularidad semántica.* La descripción de un lenguaje debe poder realizarse de forma incremental. Al añadir nuevas características computacionales a un lenguaje, no debe ser necesario retocar las definiciones que no dependan de dichas características. Por ejemplo, al añadir variables a un lenguaje de expresiones aritméticas, las especificaciones realizadas para las expresiones aritméticas no deberían verse afectadas.

- *Reusabilidad.* La herramienta debe facilitar la reutilización de descripciones en diferentes lenguajes. Debería ser posible disponer de una librería de descripciones semánticas de características que pudiesen añadirse o eliminarse a un lenguaje de forma sencilla. Este tipo de descripciones se denominarán *componentes semánticos reutilizables.*

  Un ejemplo sería la descripción semántica de expresiones aritméticas. Existen multitud de lenguajes que admiten este tipo de expresiones y la descripción de estos lenguajes debería permitir incorporar tal componente de forma sencilla.

- *Demostración.* La herramienta debe tener una base matemática que permita la posterior demostración de propiedades de los programas escritos en el lenguaje especificado.

- *Prototipo.* Debería ser posible obtener prototipos ejecutables de los lenguajes que se están diseñando durante el proceso de especificación de forma automática.

- *Legibilidad.* Las especificaciones deben ser legibles por personas con una formación heterogénea. La descripción del comportamiento de un lenguaje afecta a todas las personas que intervienen en el proceso de desarrollo de software y lo ideal es que todas esas personas pudiesen comprender la especificación.

- *Flexibilidad.* La herramienta descriptiva debe adaptarse a la gran variedad de lenguajes y familias de lenguajes existentes.

- *Experiencia.* La herramienta debe ser capaz de describir lenguajes reales. Algunas de las técnicas existentes son aplicables a lenguajes sencillos pero se resienten al ser aplicadas a algunos de los complejos lenguajes existentes en la actualidad. Debe existir, por tanto, cierta experiencia en la aplicación de la técnica de especificación a lenguajes de programación reales.

El fracaso de las técnicas existentes puede deberse a que no cumplen algunas de las características anteriores. En esta tesis se realiza un estudio comparativo de las principales técnicas en base a las características anteriores y se propone una nueva técnica. La técnica propuesta se basa en la integración de la semántica monádica modular [145, 146, 147] con los nuevos desarrollos en el campo de la programación genérica [14]. En dicha técnica, la descripción semántica de un lenguaje se divide en las siguientes partes

- Definición de la sintaxis abstracta como el punto fijo de un functor no recursivo que captura la forma del lenguaje. Cuando el lenguaje está formado por varias categorías sintácticas, puede ser necesario definir n-functores, cuyo punto fijo da lugar a tipos mutuamente recursivos.

- Especificación de la *estructura computacional* mediante una mónada M que se obtiene mediante la aplicación de varios transformadores de mónadas a una mónada base. Cada transformador monádico añadirá una determinada noción computacional como puede ser el acceso a un entorno, modificación de un estado, *backtracking*, etc.

- Descripción del *dominio de valores* mediante uniones extensibles de dominios.

- Especificación de *álgebras* sobre los functores que definen la sintaxis abstracta que toman como portadora la estructura computacional. En ocasiones es posible definir *álgebras monádicas* que permiten separar la evaluación recursiva de subcomponentes de la propia especificación semántica. De la misma forma, si se han definido n-functores, las especificaciones semánticas estarán formadas por n-*sorted*-álgebras.

Una vez realizadas las especificaciones anteriores, el intérprete se obtiene de forma automática mediante catamorfismos, catamorfismos monádicos, n-catamorfismos o combinaciones entre ambos.

Con el fin de comprobar la aplicabilidad de la técnica propuesta, se ha desarrollado un *Sistema de Prototipado de Lenguajes* [3] que permite la especificación modular de lenguajes obteniendo de forma automática el intérprete de los lenguajes especificados. La técnica utilizada para la implementación del sistema ha sido la utilización de un metalenguaje de dominio específico empotrado en el lenguaje *Haskell* [92, 118]. De esta forma se consigue una mayor facilidad de desarrollo a la vez que se incorpora el potente sistema de inferencia y comprobación estática de tipos de Haskell. El sistema desarrollado incluye un marco interactivo de interpretación de lenguajes que permite elegir qué lenguaje interpretar y facilita un entorno común de chequeo de características.

Mediante el Sistema de Prototipado de Lenguajes, se presenta la especificación de diversos lenguajes de programación en los diferentes paradigmas: un lenguaje imperativo, un lenguaje funcional, un lenguaje orientado a objetos y un lenguaje de programación lógica. Estas modelizaciones reutilizan bloques semánticos comunes y ofrecen un marco común que permite identificar y comparar las características fundamentales de cada paradigma.

1.2 Estructura de la tesis

Comienza la tesis con un repaso de los conceptos básicos sobre lenguajes de programación y con una valoración de la creciente importancia de los lenguajes de dominio específico en el capítulo 2.

Posteriormente, en el capítulo 3 se describen y valoran las principales técnicas de especificación semántica de lenguajes de programación. Para ello se describe un mismo lenguaje de forma incremental en cada formalismo y se valoran los criterios ideales mencionados en la sección anterior.

En el capítulo 4 se describe la semántica monádica modular y se realiza una valoración de la misma mediante el lenguaje ejemplo. En el capítulo 5 se describen los principales conceptos de programación genérica que se han adaptado para su incorporación en el Sistema de Prototipado de Lenguajes que se presenta en el capítulo 6.

A continuación se presentan las especificaciones semánticas de lenguajes característicos de programación funcional (capítulo 7), programación orientada a objetos (capítulo 8) y programación lógica (capítulo 9).

En el capítulo de conclusiones (capítulo 10) se valoran y comparan las diferentes técnicas de especificación semántica, se plantea una discusión de las ventajas e inconvenientes del sistema propuesto y se describen las principales aportaciones realizadas en esta tesis doctoral. Finaliza el capítulo con una indicación de futuras líneas de investigación a considerar.

El anexo A incluye una introducción básica a la teoría de categorías que incluye una definición de varios conceptos como functor, transformación natural, mónada, etc. en dicho contexto. En el anexo B se presenta una breve introducción al lenguaje *Haskell* presentando las principales características del lenguaje así como otras características no estándard como las clases de tipos con múltiples parámetros, el polimorfismo de primera clase, los registros extensibles, las dependencias funcionales, etc. En el anexo C se presenta una tabla de conversión de términos en inglés cuya castellanización parecía conflictiva y en el anexo D se resume la notación empleada.

Cada capítulo comienza con una breve cita. Las citas se han incluido deliberadamente en lengua original y con ellas se ha pretendido realizar un breve homenaje a algunos pioneros en el desarrollo de los lenguajes de programación.

Capítulo 2

Lenguajes de Programación

> *It may be that a hundred years from now there will be a programming language that by then has stood the test of time, needs no more changes for most uses, and is used by all persons who write programs because each child learns it in school. But that is not where we are now*
> Guy L. Steele Jr. [70]

2.1 Aspectos lingüísticos

2.1.1 Lenguajes como Instrumentos de Comunicación

Un *lenguaje natural* es un instrumento de comunicación utilizado de forma común entre personas. Para que exista comunicación, debe existir una comprensión mutua de cierto conjunto de símbolos y reglas del lenguaje.

Los lenguajes de programación tienen como objetivo la construcción de programas, normalmente escritos por personas humanas. Estos programas se ejecutarán por un computador que realizará las tareas descritas. El *programa* debe ser comprendido tanto por personas como por computadores. La utilización de un *lenguaje de programación* requiere, por tanto, una comprensión mutua por parte de personas y máquinas. Este objetivo es difícil de alcanzar debido a la naturaleza diferente de ambos.

En un lenguaje natural, el significado de los símbolos se establece por la costumbre y se aprende mediante la experiencia. Sin embargo, los lenguajes de programación se definen habitualmente por una autoridad, que puede ser el diseñador individual del lenguaje o un determinado comité.

Para que el computador pueda comprender un lenguaje humano, es necesario diseñar métodos que traduzcan tanto la estructura de las frases como su significado a código máquina. Los diseñadores de lenguajes de programación construyen lenguajes que saben cómo traducir o que creen que serán capaces de traducir. Si los computadores fuesen la única audiencia de los programas, éstos se escribirían directamente en código máquina o en lenguajes mucho más mecánicos. Por otro lado, el programador debe ser capaz de leer y comprender el programa que está construyendo y las personas humanas no son capaces de procesar información con el mismo nivel de detalle que las máquinas.

Los lenguajes de programación son, por tanto, una solución de compromiso entre las necesidades del emisor (programador – persona) y del receptor (computador – máquina).

De esa forma, las declaraciones, tipos, nombres simbólicos, etc. son concesiones de los diseñadores de lenguajes para que los humanos podamos entender mejor lo que se ha escrito en un programa. Por otro lado, la utilización de un vocabulario limitado y de unas reglas estrictas son concesiones para facilitar el proceso de traducción.

En 1938, C. Morris [174] realiza una división del estudio de los signos (semiótica) en tres partes:

- Sintaxis: relación de los signos entre sí
- Semántica: Relación de los signos con los objetos a los que se aplican
- Pragmática: Relación de los signos con sus intérpretes

Adaptando dichas definiciones al caso particular de lenguajes de programación, la sintaxis se refiere al formato de los programas del lenguaje, la semántica estudia el comportamiento de los programas y la pragmática estudia aspectos relacionados con las técnicas empleadas para la construcción de programas.

2.1.2 Sintaxis

La sintaxis de un lenguaje de programación determina el conjunto de programas legales y la relación entre los símbolos y frases que en ellos aparecen.

Es posible dividir la sintaxis en *sintaxis concreta* y *sintaxis abstracta*. La *sintaxis concreta* se refiere al *análisis*: el reconocimiento de programas legales a partir de textos (normalmente secuencias de caracteres) y su *análisis sintáctico* no ambigüo para obtener frases.

La *sintaxis abstracta* se refiere a la síntesis y se centra en la *estructura* que permite componer frases de un programa, ignorando qué metodos se han utilizado para su reconocimiento. En general, las definiciones semánticas de lenguajes son más sencillas si se realizan a partir de la sintaxis abstracta, en lugar de la sintaxis concreta.

En las descripciones de lenguajes de programación son necesarios ambos tipos de sintaxis, junto con una indicación de cómo se relacionan. Esta tesis, sin embargo, se centra en la especificación semántica de lenguajes que se obtendrá a partir de la sintaxis abstracta. De esta forma, a la sintaxis concreta se le dedicará únicamente la siguiente sección introductoria y la sección 6.6.1 sobre combinadores de analizadores sintácticos.

2.1.2.1 Sintaxis Concreta

Convencionalmente, la sintaxis concreta se separa en *análisis léxico* y *análisis sintáctico* propiamente dicho o *parsing*.

El objetivo del análisis léxico es agrupar los caracteres del texto de un programa en un conjunto de símbolos legales o *tokens*. Con el análisis sintáctico se intenta agrupar estos símbolos en frases, construyendo un *árbol sintáctico* o *árbol de derivación* que tiene a los símbolos como hojas. Los principales tipos de símbolos reconocidos en el análisis léxico suelen ser:

- Delimitadores: marcas de puntuación, signos matemáticos, etc.
- Palabras: secuencias alfanuméricas, las cuales pueden subdividirse en un conjunto de *palabras reservadas* como *begin* o *end*, y un conjunto de *identificadores* utilizados para dar nombre a entidades del programa.
- Literales numéricos: secuencias de dígitos, puntos decimales y signos indicando bases y/o exponentes.
- Literales que representan caracteres o cadenas de caracteres: suelen utilizarse comillas simples o dobles.
- Comentarios: secuencias arbitrarias de caracteres, delimitados por secuencias especiales que se utilizan para ayudar a la comprensión del programa por parte de los programadores, pero que son ignorados por el computador.
- Separadores: espacios, caracteres *fin de línea*, etc.

Los comentarios y los separadores aparecen generalmente de forma libre entre el resto de símbolos léxicos. Resulta tedioso describir dicha libertad como parte de la estructura de las frases, por lo que, en general, una vez reconocidos por el analizador léxico, son eliminados de la secuencia de símbolos que se obtiene del análisis léxico.

El análisis sintáctico obtiene a partir del conjunto de símbolos anteriores una estructura de las frases que forman el programa. Habitualmente, se utilizan varias categorías:

- Declaraciones: Permiten introducir identificadores que dan nombre a entidades del programa.
- Expresiones: Se asemejan a términos matemáticos.
- Enunciados: simples o compuestos, que pueden contener a su vez declaraciones o expresiones.
- Programas: combinaciones adecuadas de declaraciones, expresiones y enunciados.
- etc.

La clasificación anterior puede realizarse mediante lo que se denomina *análisis libre de contexto*, ya que el sistema puede reconocer las diferentes categorías independientemente del contexto en el que aparezcan.

Para describir la sintaxis libre de contexto suele utilizarse la notación *BNF* (*Backus-Naur Form*), introducida para la descripción del lenguaje Algol-60 [55].

Ejemplo 2.1 *La siguiente gramática describe la sintaxis concreta de expresiones aritméticas simples. Para resolver precedencia y asociatividad de los operadores se utilizan 3 categorías sintácticas.*

$$
\begin{array}{ll}
expr & : term \ + \ expr \\
 & \mid term \ - \ expr \\
 & \mid term \\
term & : factor \ * \ term
\end{array}
$$

$$
\begin{array}{rl}
 & \mid factor \ / \ term \\
 & \mid factor \\
factor & : number \\
 & \mid (\ expr\ )
\end{array}
$$

A la expresión $3 + 4 * 5$ *le correspondería el siguiente árbol sintéctico de la figura 2.1*

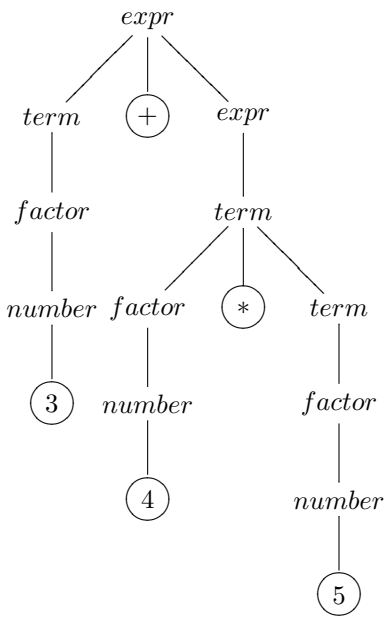


Figura 2.1: Árbol sintáctico concreto de la expresión $3 + 4 * 5$

2.1.2.2 Sintaxis Abstracta

La sintaxis abstracta [155] proporciona un interfaz adecuado entre la sintaxis concreta y la semántica. Habitualmente se obtiene de forma simple, ignorando aquellos detalles del árbol sintáctico que no tienen relevancia semántica, dejando árboles sintácticos que representan únicamente la estructura composicional esencial de los programas.

Ejemplo 2.2 *La siguiente gramática define la sintaxis abstracta del mismo lenguaje formado por expresiones aritméticas simples del ejemplo 2.1.*

$$
\begin{array}{rl}
expr & : expr\ +\ expr \\
 & \mid expr\ -\ expr \\
 & \mid expr\ *\ expr \\
 & \mid expr\ /\ expr \\
 & \mid number
\end{array}
$$

A partir de dicha gramática, el árbol de la sintaxis abstracta de la expresión $3 + 4 * 5$ *se representa en la figura 2.2*

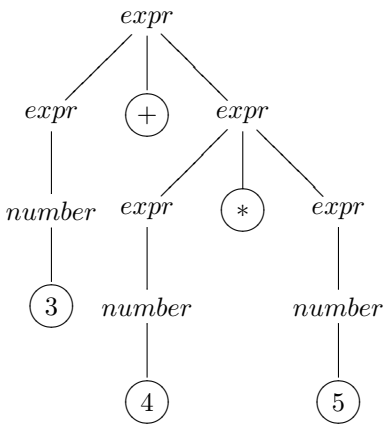


Figura 2.2: Árbol sintáctico concreto de la expresión $3 + 4 * 5$

2.1.2.3 Sintaxis sensible al contexto

La *sintaxis sensible al contexto* se enfrenta a quellos aspectos que no pueden ser tratados por la sintaxis libre de contexto, como la *declaración de identificadores antes de su utilización*, las *expresiones con el tipo adecuado*, etc.

Habitualmente la sintaxis sensible al contexto se encarga del chequeo estático de tipos. Los sistemas de tipos [33, 192] han alcanzado una enorme popularidad ya que permiten al programador especificar restricciones sobre el comportamiento de los programas que pueden verificarse antes de la ejecución. Los beneficios principales son

- *Seguridad.* Los programas que superan el chequeo de tipos no producen errores de tipos en tiempo de ejecución
- *Eficiencia.* Durante la ejecución no es necesario realizar comprobaciones de tipos, ya que éstas han sido realizadas previamente por el sistema de tipos.

Al realizar el análisis de lenguajes con sintaxis sensible al contexto, es necesario encajar partes de un programa distantes entre sí. Existen dos alternativas:

- Las *gramáticas de atributos* o *attribute grammars* introducidas por D. Knuth en 1968 [127] toman una gramática libre de contexto y le añaden un conjunto de atributos y una serie de ecuaciones. De esta forma es posible enviar información sobre los valores de los atributos entre los diferentes nodos del árbol sintáctico, facilitando el análisis contextual del lenguaje. Las *gramáticas de atributos* tradicionales tienen algunos problemas de modularidad aunque se han desarrollado técnicas para su solución [208].
- Otra posibilidad es considerar la sintaxis sensible al contexto como un tipo especial de semántica, conocida como *semántica estática.* Se denomina así porque depende únicamente de la estructura del programa, y es independiente de la entrada del usuario en una ejecución particular. El

comportamiento de un programa que depende de la entrada se conoce como *semántica dinámica*, o simplemente, su semántica cuando no hay confusión[1].

La forma más sencilla de resolver la semántica estática es considerarla como una *restricción* sobre los programas. Si un programa no cumple la restricción, entonces se considera ilegal. Tal semántica podría considerarse como una función que asignase valores booleanos a los programas[2]. De esta forma, las técnicas utilizadas para la semántica dinámica pueden utilizarse para la semántica estática. Por este motivo, no se desarrollarán descripciones de la semántica estática de lenguajes en el resto de esta tesis.

2.1.3 Semántica

Dado un programa en un lenguaje determinado, ¿cuál es la naturaleza de su semántica?

En primer lugar, deben apartarse los efectos que el programa pueda tener sobre un lector humano, evocando, por ejemplo, sentimientos de admiración, o quizás, y más a menudo, de disgusto. A diferencia de la filología, la lingüística de la programación, no trabaja con cualidades subjetivas. El significado de un programa depende solamente del *comportamiento* objetivo que el programa cause cuando es ejecutado por un computador.

Los computadores son sistemas complejos y, cuando se ejecuta un programa, pueden observarse efectos muy diversos: movimientos de cabezas del disco, flujos de corrientes en los circuitos, caracteres que aparecen en una pantalla o en una impresora, etc. Con el fin de considerar programas que controlan de forma específica tales comportamientos físicos, es necesario afrontar estos fenómenos en su semántica.

Sin embargo, se considerarán programas en *lenguajes de alto nivel*, los cuales no realizan un control directo sobre detalles de comportamiento físico del computador. La semántica apropiada de estos programas es *independiente de la implementación*, que consiste únicamente en aquellas características que son comunes a todas las implementaciones. Esto incluye habitualmente propiedades sobre la terminación de un programa, pero ignora consideraciones electrónicas.

Puede utilizarse la siguiente definición (adaptada de [177]).

Definición 2.1 (Semántica) *La semántica de un programa P es una entidad abstracta que modeliza el comportamiento de forma independiente de la implementación particular. La semántica de P se denotará como $[\![P]\!]$.*

Una característica especialmente deseable de las descripciones semánticas es que sean composicionales.

Definición 2.2 (Composicionalidad) *Una descripción semántica es composicional si el significado de una frase compuesta viene determinado únicamente por el significado de sus componentes y la forma en que se combinan.*

[1] En algunos textos tradicionales que sólo se ocupan del desarrollo de procesadores de lenguaje, se denomina semántica a la semántica estática, lo cual puede provocar cierta confusión

[2] En la práctica, un tratamiento sistemático suele ser más complicado, ya que deben tenerse en cuenta los mensajes de error sobre los motivos del incumplimiento

La propiedad de *composicionalidad* fue enunciada por Frege para estudiar el significado del lenguaje natural [77, 205]. Su importancia radica en que permite descomponer el estudio semántico en las diferentes entidades que lo forman.

Definición 2.3 (Metalenguaje) *Un* metalenguaje *es un lenguaje empleado para la descripción de otro lenguaje.*

Es conveniente realizar una separación entre lenguaje y metalenguaje para evitar problemas de circularidad, en los que una entidad se define en función de sí misma.

2.1.4 Pragmática

La pragmática se refiere a la relación entre el lenguaje y sus usuarios. En el caso de lenguajes de programación la pragmática estudiará las técnicas y tecnologías empleadas en la construcción de programas.

A la hora de desarrollar grandes proyectos informáticos, aspectos pragmáticos como la organización y planificación, el análisis de requerimientos, la gestión de recursos humanos, el trabajo en equipo, etc. tienen una importancia incuestionable.

No obstante, se considera fundamental el conocimiento y comprensión de la herramienta básica de trabajo, en este caso, el lenguaje de programación. El comportamiento del software está determinado en último momento por el comportamiento especificado del lenguaje en el que se construye.

En este trabajo se prescinde de los detalles pragmáticos de la construcción de software y se tratan exclusivamente los aspectos de descripción semántica de lenguajes de programación.

2.2 Procesadores de Lenguaje

2.2.1 Intérpretes

Definición 2.4 (Intérprete) *Un intérprete es un programa que analiza y ejecuta de forma simultánea un programa.*

En la figura 2.3 se presenta el esquema de un intérprete. $P/\mathcal{L}$ representa un programa P escrito en el lenguaje $\mathcal{L}$.

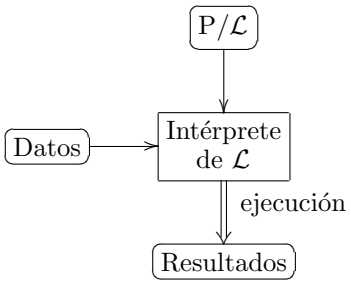


Figura 2.3: Esquema de un intérprete

Definición 2.5 (Corrección de interpretación) *Se dice que un intérprete $\mathcal{I}$ es correcto si su comportamiento al ejecutar un programa coincide con el comportamiento pretendido de dicho programa. Es decir, si para todo $P/\mathcal{L}$ se cumple que:*

$$[\![P/\mathcal{L}]\!] = [\![\mathcal{I}]\!](P/\mathcal{L})$$

Los intérpretes tienen las siguientes características

- La implementación de un intérprete es relativamente sencilla y flexible, siguiendo fielmente la semántica del lenguaje a interpretar y adaptándose a cambios con facilidad.
- En ocasiones, un intérprete puede utilizarse como modelo para la descripción semántica del lenguaje. Dicho intérprete se denomina *implementación prototipo* del lenguaje. La *semántica operacional* de un lenguaje puede considerarse un caso especial de implementación prototipo en el que el intérprete se desarrolla en un lenguaje suficientemente abstracto.
- El intérprete no requiere disponer de todo el código del programa a ejecutar al principio de la ejecución. Esta característica se aplica en los *intérpretes de comandos* en los que el código a interpretar es introducido de forma interactiva por el usuario. Otra aplicación son los sistemas distribuidos: al ejecutar desde un cliente programas que residen en un servidor no es necesario traer todo los programas completos, sino que es posible ir ejecutando porciones (véase 2.2.4).
- Los intérpretes permiten acceder al propio código del programa fuente durante la ejecución. Lenguajes como Lisp, Prolog o Smalltalk explotan esta característica permitiendo construir programas que se manipulan a sí mismos en tiempo de ejecución.

Definición 2.6 (Máquina abstracta) *Una* máquina abstracta *puede definirse como un procedimiento para ejecutar un conjunto de instrucciones en algún lenguaje formal.*

La definición de máquina abstracta no requiere que existe implementación de dicha máquina en Hardware, en cuyo caso sería una máquina concreta. De hecho, las máquinas de Turing son máquinas abstractas que ni siquiera pueden implementarse en Hardware.

Definición 2.7 (Máquina virtual) *Una máquina virtual es una máquina abstracta para la que existe un intérprete.*

Existen varios ejemplos de máquinas virtuales, como la máquina virtual de Java [148] o la máquina Carbayonia desarrollada para el Sistema Integral Orientado a Objetos Oviedo3 [9].

Definición 2.8 (Interpretación abstracta) *La interpretación abstracta estudia el comportamiento de un programa substituyendo los dominios concretos por dominios abstractos.*

La interpretación abstracta [111] realiza un análisis de los programas mediante una aproximación de la ejecución en unos dominios abstractos. Estos análisis son realizados antes de la ejecución efectiva del programa con el fin de garantizar determinadas propiedades de seguridad y se basan en la descripción semántica del lenguaje [244].

2.2.2 Traductores y compiladores

Definición 2.9 (Traductor) *Un* traductor *transforma un programa escrito en un lenguaje (denominado* lenguaje fuente*) en otro programa escrito en otro lenguaje (denominado* lenguaje objeto*).*

En la figura 2.4 se presenta el esquema de un traductor.

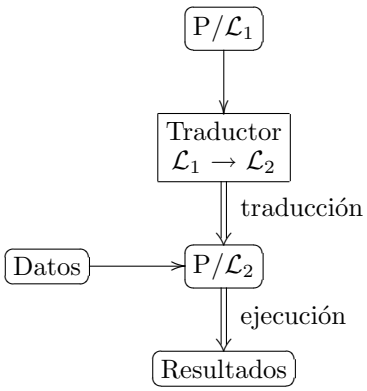


Figura 2.4: Esquema de un traductor

La ejecución de un programa mediante un traductor, requiere dos fases. Una primera fase de traducción al lenguaje objeto y una segunda fase de ejecución del programa resultante.

Definición 2.10 (Corrección de traducción) *Se dice que un compilador $\mathcal{C}$ es correcto si el comportamiento de todos los programas compilados es el mismo que el comportamiento pretendido de los programas sin compilar. Es decir, si para todo $P/\mathcal{L}_1$ se cumple que:*

$$[\![P/\mathcal{L}_1]\!] = [\![P/\mathcal{L}_2]\!]$$

donde $P/\mathcal{L}_2 = [\![\mathcal{C}]\!](P/\mathcal{L}_1)$

Un traductor en el que el lenguaje fuente es de alto nivel y el lenguaje objeto de bajo nivel se denomina *compilador*. La definición anterior es poco rigurosa, ya que el nivel de un lenguaje de programación es un concepto bastante discutible.

La principal ventaja de un compilador sobre un intérprete es la eficiencia.

2.2.3 Compilador incremental

Algunos lenguajes, como Lisp o Prolog, permiten realizar modificaciones del propio programa fuente durante su ejecución. Esta característica limita la

posibilidad de implementar un compilador íntegro ya que durante la ejecución de un programa no sería posible acceder al código fuente de dicho programa.

Los programas escritos en este tipo de lenguajes pueden dividirse en una parte estática, que no es susceptible de modificaciones durante la ejecución, y una parte dinámica, que puede sufrir modificaciones durante la ejecución.

La compilación incremental es una técnica que compila únicamente las partes estáticas del programa. Posteriormente, al ejecutar el programa, las partes dinámicas no compiladas son interpretadas.

2.2.4 Compilación *Just in time*

La compilación *Just in Time* es una técnica que se aplica fundamentalmente a entornos distribuidos en los que un cliente desea ejecutar un programa que reside en un servidor.

Con la compilación tradicional, sería necesario solicitar todo el programa fuente, compilarlo y ejecutarlo. Con la compilación *Just in time* se solicita un módulo del programa fuente, se compila y se ejecuta dicho módulo. Si el módulo realiza una llamada a otro módulo, se solicita entonces dicho módulo, se compila y se ejecuta, y así sucesivamente.

La ventaja para el usuario es que puede comenzar a obtener resultados parciales de la ejecución antes que con la compilación tradicional. Además, en el caso de programas grandes, no siempre es necesario compilar todos los módulos, ya que algunos pueden no utilizarse.

La *compilación continua* [194] es una mejora de la compilación *Just in time* en la que se solicitan módulos y se ejecutan de forma paralela.

2.2.5 Evaluación parcial

Los datos de entrada de muchos programas pueden clasificarse en dos grandes conjuntos, datos estáticos (Est) que no cambian de una ejecución a otra, y datos dinámicos (Din) que cambian entre ejecuciones.

Un *evaluador parcial* o *especializador* [185, 110] toma un programa $P/\mathcal{L}$, junto con sus datos estáticos Est, y devuelve un programa especializado para dichos datos estáticos, que se denota como $P_{Est}/\mathcal{L}'$. Normalmente, el evaluador parcial devuelve un programa especializado en el mismo lenguaje fuente, aunque no es obligatorio.

La utilización de evaluadores parciales se justifica por la mayor eficiencia del programa especializado. Existen múltiples aplicaciones de los evaluadores parciales, pero una de las más interesantes en lo que respecta a esta tesis doctoral es la posibilidad de obtener compiladores de forma automática a partir de un intérprete.

Supóngase que se dispone de un intérprete de un lenguaje fuente LF escrito en un lenguaje de transición $\mathcal{L}_T$, el cual se denota como $IntLF/\mathcal{L}_T$. Si el evaluador parcial recibe como entrada el propio intérprete junto con el programa a interpretar P/LF, el resultado, será un intérprete especializado para dicho programa. Por definición, el comportamiento del intérprete especializado es el mismo que el de P/LF, por lo que se ha transformado el programa original en un programa equivalente en otro lenguaje. En la figura 2.6 se muestra el proceso.

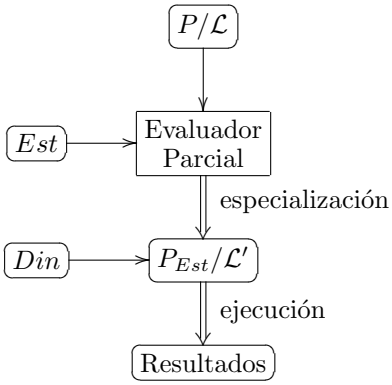


Figura 2.5: Esquema de un evaluador parcial

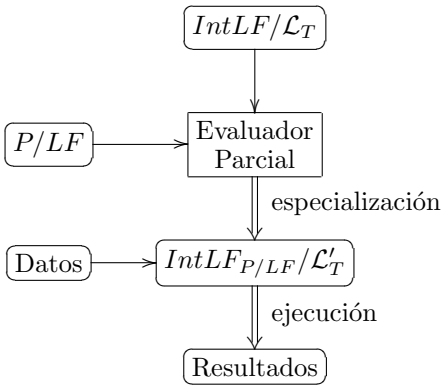


Figura 2.6: Esquema de un evaluador parcial

2.3 Diseño de Lenguajes

2.3.1 Principios de Diseño

Una pregunta natural al estudiar los lenguajes de programación es si existe un lenguaje perfecto. Si existiese tal lenguaje, entonces sería importante identificar sus características y no perder el tiempo utilizando lenguajes imperfectos. En este sentido, conviene recordar la reflexión de Umberto Eco sobre los lenguajes naturales [52].

> El tema de la confusión de lenguas, y el intento de remediarla mediante la recuperación o la invención de una lengua común a todo el género humano, aparece en la historia de todas las culturas. [...]
>
> La historia de los lenguajes perfectos es la historia de una utopía y de una serie de fracasos [...] ahora bien, aunque ésta será la historia de una serie de fracasos, veremos cómo a cada fracaso le ha seguido un efecto "colateral": los distintos proyectos no se han mantenido, pero han dejado como una estela de consecuencias benéficas.

Al diseñar lenguajes de programación a menudo es necesario tomar decisiones sobre las características que se incluyen de forma permanente, las características que no se incluyen pero que existen mecanismos que facilitan su inclusión y las que no se permiten. Estas decisiones pueden afectar al diseño final del lenguaje y a menudo entrar en conflicto con otros aspectos del lenguaje.

A continuación se resumen algunos principios de diseño de lenguajes de programación recogidos de diversas fuentes [224, 89, 156, 117, 197, 56].

- *Concisión notacional*: el lenguaje debe permitir describir algoritmos con el nivel de detalle adecuado.
- *Integridad conceptual*: se proporcionan un conjunto de conceptos simple, claro y unificado. Lo ideal es disponer de un conjunto de conceptos diferentes mínimo, con una reglas simples y regulares para su combinación.
- *Ortogonalidad*. Dos características de un lenguaje son ortogonales si pueden ser comprendidas y combinadas de forma independiente.
- *Generalidad*. Todas las características de un lenguaje son generadas a partir de conceptos básicos.
- *Abstracción*. Evitar que algo deba ser enunciado más de una vez. Permitiendo la factorización de patrones repetitivos.
- *Extensiblidad*. El lenguaje debe admitir la creación de nuevas características no previstas en el momento de su creación [70].
- *Seguridad*. Deben existir medios adecuados para comprobar cuándo los programas no cumplen con la definición del lenguaje o con la propia estructura pretendida por el programador. Por ejemplo, si el lenguaje admite declaraciones de tipos, debe ser posible comprobar que el tipo declarado de una función coincide con el tipo de su implementación.
- *Automatización*. El lenguaje debe admitir la automatización de tareas mecánicas, tediosas o susceptibles de producir errores.
- *Portabilidad*. La definición del lenguaje debe facilitar que sus programas funcionen en diferentes máquinas y clases de máquinas.
- *Eficiencia*. Es conveniente estudiar la eficiencia, tanto de los programas al ejecutarse, como de las herramientas de procesamiento del lenguaje.
- *Entorno*. Aunque el entorno no forma parte del lenguaje, muchos lenguajes débiles técnicamente son ampliamente utilizados debido a que disponen de un entorno de desarrollo potente. De la misma forma, la disposición de documentación, ejemplos de programas e incluso programadores pueden ser factores clave del desarrollo de un lenguaje de programación.

2.3.2 Familias de lenguajes

El concepto de *paradigma* fue utilizado por T. S. Kuhn [129] para justificar las revoluciones científicas. Un paradigma engloba un conjunto de normas, prácticas y creencias de una comunidad científica en un momento dado. Las

personas de esa comunidad aceptan y comparten dichas creencias sin discusión hasta que aparece un nuevo paradigma que rebate alguna característica fundamental.

En informática, también es posible observar varias comunidades de ese tipo, cada una hablando su propio lenguaje y utilizando sus propios paradigmas. De hecho los lenguajes de programación suelen fomentar el uso de ciertos paradigmas y disuadir el uso de otros.

A continuación se resumen los principales paradigmas de programación, aunque existen lenguajes de programación híbridos:

- La *programación imperativa* presta especial atención a la secuencia de órdenes que el programador debe comunicar para resolver un problema. *Cobol*, *Fortran*, *Pascal*, *C*, etc. son lenguajes comúnmente aceptados dentro de este paradigma. En oposición al paradigma imperativo, surgen los paradigmas declarativos.

- La *programación funcional* toma como elemento central las funciones que intervienen en el problema a resolver. Cuando se permiten exclusivamente funciones matemáticas sin efectos laterales, los lenguajes se denominan puramente funcionales. Una característica común de este paradigma es la utilización de *funciones de orden superior*, es decir, funciones que pueden tener como argumentos otras funciones y devolver funciones como resultados. *Lisp* [154], *Scheme* [216], *ML* [66, 189], *Haskell* [112] son ejemplos de lenguajes funcionales.

- La *programación lógica* se centra en la descripción de las relaciones que intervienen en el problema. El lenguaje más conocido sería el lenguaje *Prolog* [218], aunque existen otros lenguajes como *Mercury*, *Goedel*, *λProlog*, etc.

- En la *Programación Orientada a Objetos* se resuelve el problema definiendo los objetos que intervienen y el envío de mensajes entre ellos. Los lenguajes *Simula* [47], *Smalltalk* [64], *C++* [223], *Java* [113], etc. se consideran lenguajes Orientados a Objetos.

Existen otros paradigmas, como la programación dirigida por eventos, la programación visual, etc.

2.3.3 Lenguajes de Dominio Específico

En todas las ramas de la ingeniería, aparecen técnicas genéricas junto con técnicas específicas. La aplicación de técnicas genéricas proporciona soluciones generales para muchos problemas, aunque tales soluciones pueden no ser óptimas. Las técnicas específicas suelen estar optimizadas y aportan una solución mejor para un conjunto reducido de problemas.

En los lenguajes de programación también se produce esta dicotomía, lenguajes de *dominio específico* respecto a lenguajes de *propósito general*.

Recientemente, hay un interés creciente en el estudio de las técnicas de implementación de lenguajes de dominio específico. Una posible definición, adaptada de [231], podría ser.

Definición 2.11 (Lenguaje de dominio específico) *Un* Lenguaje de Dominio Específico *es un lenguaje de programación o un lenguaje de especificación ejecutable que ofrece potencia expresiva enfocada y restringida a un dominio de problema concreto.*

Entre las principales características de estos lenguajes están:

- Estos lenguajes suelen ser *pequeños*, ofreciendo un conjunto limitado de notaciones y abstracciones. En ocasiones, sin embargo, pueden contener un completo sublenguaje de propósito general. Proporcionando, además de las capacidades generales, las del dominio del problema concreto. Esta situación aparece cuando el lenguaje es empotrado en un lenguaje general.
- Suelen ser *lenguajes declarativos*, pudiendo considerarse lenguajes de especificación, además de lenguajes de programación. Muchos contienen un compilador que genera aplicaciones a partir de los programas, en cuyo caso el compilador se denomina *generador de aplicaciones*. Otros, como Yacc [20] o ASDL [238], no tienen como objetivo la generación o especificación de aplicaciones completas, sino generar librerías o componentes.
- Un objetivo común de muchos de estos lenguajes es la *programación realizada por el usuario final*, que ocurre cuando son los usuarios finales los que se encargan de desarrollar sus programas. Estos usuarios no suelen tener grandes conocimientos informáticos pero pueden realizar tareas de programación en dominios concretos con un vocabulario cercano a su especialización.
- Existe gran variedad de dominios en los que la utilización de Lenguajes de Dominio Específico ha supuesto un avance. En [231] se resumen algunos lenguajes de dominio específico clasificados por dominios.

2.3.3.1 Desarrollo de Lenguajes de Dominio Específico

Existen varias técnicas para desarrollar lenguajes de dominio específico.

- La primera posibilidad es crear un lenguaje específico independiente y procesarlo como cualquier lenguaje de programación ordinario. Esta opción requiere el diseño completo del lenguaje y la implementación de un *intérprete* o un compilador siguiendo las técnicas tradicionales.
- Otra posibilidad es *empotrar* el lenguaje específico en un lenguaje de propósito general. La versatilidad sintáctica y semántica de algunos lenguajes como Haskell, favorecen el empotramiento de lenguajes de dominio específico. Existen varios ejemplos, en [41] se describe el empotramiento del lenguaje *Lava* de descripción de Hardware, en [42] se describe cómo incrustar variables lógicas en Haskell, ofreciendo características de programación lógica. En [118] se resumen varios lenguajes empotrados de dominio específico
- Mediante *Preprocesamiento* o *proceso de macros* se empotra un lenguaje de propósito específico en otro lenguaje incluyendo una fase intermedia que convierte el lenguaje específico en el lenguaje anfitrión. Por ejemplo, el preprocesador de C++ puede utilizarse para crear un completo lenguaje específico para tareas concretas.

- *Intérprete extensible* Consiste en añadir elementos que permitan modificar el intérprete para que analize lenguajes diferentes.

2.4 Evolución

Las bases teóricas de los lenguajes de programación se emparejan con las de los lenguajes formales. Los cuales surgen a partir de los desarrollos de la lógica matemática de finales del siglo XIX.

La lógica fue planteada por Aristóteles (384/3–322a.C) con el fin de estudiar los procesos de razonamiento humano. Durante mucho tiempo, la lógica se desarrolla en el campo filosófico, utilizando el lenguaje natural como instrumento de comunicación.

G. W. Leibniz (1646–1716) propone dar un rigor matemático a la lógica que permita resolver las argumentaciones mediante procedimientos de cálculo. Sus ideas no tienen una influencia inmediata y es G. Boole en 1847 [25] quien propone un estudio algebraico de los razonamientos.

Durante la segunda mitad del s. XIX se produce un enorme interés en el estudio de la lógica, destacando los trabajos de G. Fregge que dan lugar a la lógica de predicados y el programa logicista que pretende demostrar que los teoremas y principios matemáticos pueden obtenerse por cálculos deductivos a partir de unos axiomas.

En 1910, Russell y Whitehead publican los Principia Mathematica, en los que recopilan los conocimientos de lógica matemática de la época. En su obra se hace mención a la paradoja de las clases cuya solución dará lugar a la teoría de tipos, base de los sistemas de tipos de los lenguajes de programación actuales.

A principios de los años 20, Hilbert desarrolla la teoría de la prueba con el objetivo de axiomatizar los conceptos matemáticos básicos y poder demostrar mediante métodos constructivos todos los teoremas matemáticos. En sus desarrollos aparecen las definiciones de lenguajes formales y de métodos constructivos que deben estar formados por una secuencia finita de pasos. Aunque el teorema de incompletud de Gödel da al traste con las expectativas de Hilbert, durante la primera mitad de los años 30 se desarrollaron métodos constructivos como el cálculo lambda de Church y las máquinas de Turing que formarán la base teórica de los lenguajes de programación actuales. En esa época, A. Tarski desarrolla la teoría de modelos que fundamentará el estudio semántico de lenguajes formales y, por tanto, de lenguajes de programación.

Por otro lado, durante la segunda guerra mundial K. Zuse, diseña el que se puede considerar primer lenguaje de programación [126] aunque no llega implementarlo.

Con la aparición de los primeros computadores, surge la necesidad de utilizar notaciones que faciliten la programación. El lenguaje ensamblador consiste en una serie de macros mnemotécnicas que se corresponden de forma más o menos directa con las instrucciones de la máquina. En 1954, J. Backus lidera un equipo para implementar el lenguaje *Fortran* que ofrece construcciones de control de alto nivel y se impondrá como lenguaje de cálculo científico.

En 1959, J. McCarthy [154] desarrolla el lenguaje *Lisp*, que incluye facilidades para tratamiento de listas y la utilización de funciones de orden superior. El lenguaje *Lisp* se convertirá en uno de los lenguajes más utilizados en aplicaciones de Inteligencia Artificial, considerándose el primer lenguaje funcional.

A finales de los años 50, existen numerosos lenguajes de programación para arquitecturas específicas e incompatibles entre sí. Se desarrolla un comité internacional para el desarrollo de un lenguaje algorítmico universal dirigido por P. Naur. El lenguaje desarrollado se denominará *Algol* y contiene las principales características conocidas en la época. El lenguaje tendrá cierto éxito en ambientes académicos como instrumento de comunicación de algoritmos, pero su complejidad impide la construcción de implementaciones y de aplicaciones prácticas.

En la descripción sintáctica del lenguaje *Algol*, P. Naur adopta una notación previamente utilizada por J. Backus, que posteriormente se denominará *notación BNF* (*Backus-Naur Form*). En la misma época, N. Chomsky propone las gramáticas libres de contexto para la descripción del lenguaje natural. El descubrimiento de la equivalencia descriptiva entre ambos formalismos favorece la posterior evolución de la lingüística computacional.

Algol marcó un hito en el diseño de lenguajes y tuvo una enorme influencia en los lenguajes posteriores. Esta influencia puede resumirse en tres líneas.

- Por un lado, N. Wirth desarrolla el lenguaje *Pascal* (1971) con propósitos educativos. Posteriormente, desarrollará los lenguajes *Modula-2* (1983) y *Oberon* (1988).

- Por otro lado, C. Strachey, pionero en la creación de la semántica denotacional de lenguajes, diseña el lenguaje *CPL* (*Combined Programming Language*) en 1966. A partir de dicho lenguaje, Richards diseña *BCPL* o *Basic CPL* (1966) y posteriormente, D. Ritchie diseña el lenguaje C (1972) como lenguaje de implantación de programas asociados al sistema operativo Unix. Este lenguaje tendrá una enorme popularidad por la combinación entre características de alto nivel con la libertad de acceso a elementos de bajo nivel.

- Finalmente, K. Nygaard y O. J. Dahl diseñan el lenguaje *Simula* [47] para el desarrollo de aplicaciones de simulación. La principal novedad del lenguaje es la combinación de datos y procedimientos en una entidad, dando lugar al concepto de objeto y clase. A partir de dicho lenguaje, A. C. Kay diseña *Smalltalk* como un sistema uniforme orientado a objetos. La programación Orientada a Objetos alcanzará gran popularidad con los lenguajes *C++* [223] y *Java* [113]. En el capítulo 8 se introducen los conceptos básicos del paradigma de la programación Orientada a Objetos.

Por otro lado, en 1972 A. Colmerauer desarrolla el primer intérprete de *Prolog* con el propósito de desarrollar sistemas de tratamiento de lenguaje natural. Este lenguaje dará lugar al paradigma de la programación lógica que se describe con más detalle en el capítulo 9.

En 1974, se produce un nuevo intento de desarrollo de lenguaje universal. El Departamento de Defensa de Estados Unidos realiza un concurso para el diseño de un nuevo lenguaje para sistemas empotrados que se denominará *Ada*. De la misma forma que en el caso de *Algol*, el lenguaje es excesivamente complejo aglutinando muchos de los conceptos de la ingeniería del software del momento.

En el campo de la programación funcional, la popularidad del lenguaje *Lisp* para aplicaciones de Inteligencia Artificial favoreció el crecimiento incontrolado

del lenguaje incorporando características imperativas. Como reacción, en 1975, G. Steele Jr. y Sussman diseñan Scheme, un lenguaje tipo *Lisp* más sencillo.

El sistema de demostración de teoremas *LCF* incluía el metalenguaje funcional *ML* para la programación de estrategias. *ML* contenía un sistema de chequeo e inferencia estática de tipos cuya utilidad le permitió independizarse y comenzar a ser utilizado como lenguaje de propósito general.

Los programas *ML* pueden ser compilados consiguiendo competir con programas imperativos en eficiencia. No obstante, el lenguaje *ML* no es puramente funcional.

Por otro lado, en 1976 D. Turner desarrolla el lenguaje *SASL* incluyendo evaluación perezosa. Este lenguaje dará lugar al lenguaje comercial *Miranda* y tendría cierta popularidad en ambientes académicos.

En 1989 se crea un comité internacional con el objetivo de diseñar un lenguaje puramente funcional que aglutine las características existentes. Este lenguaje se denominará *Haskell* y se describe con más detalle en el anexo B.

Capítulo 3

Descripción Semántica de Lenguajes de Programación

> *Discussions about programming languages often resemble medieval debates about the number of angels that can dance on the head of a pin instead of exciting contests between fundamentally differing concepts.*
> J. Backus [15]

En este capítulo se describen brevemente las principales técnicas de especificación semántica de lenguajes de programación y se realiza una valoración de las mismas. Para realizar la valoración, se especifica en cada formalismo el mismo lenguaje. Dicho lenguaje se obtiene de forma incremental en cinco etapas, en cada una de las cuales se añade una nueva característica al lenguaje anterior. El objetivo de tal presentación es observar la modularidad y capacidad de reusabilidad de cada formalismo.

Para obtener información más detallada de estas técnicas pueden consultarse [68, 123, 168, 182, 200, 209, 220, 225, 241, 245].

3.1 Justificación de las técnicas de especificación semántica

La *semántica* describe el significado de las sentencias del lenguaje. En el caso de un lenguaje de programación, las sentencias son los programas escritos en dicho lenguaje y el significado será el comportamiento de dichos programas al ejecutarse en una determinada máquina.

La notación *BNF* ha sido aceptada de forma prácticamente universal para la descripción sintáctica de lenguajes. Sin embargo, no existe una notación comúnmente aceptada para la especificación semántica. Por el contrario, se han desarrollado y siguen desarrollándose gran variedad de notaciones, ninguna de las cuales tiene una aceptación generalizada.

Las principales ventajas de la descripción semántica de lenguajes de programación son:

- Para el **diseño** de nuevos lenguajes de programación se requiere una técnica que permita registrar las decisiones sobre construcciones particulares de un lenguaje y descubrir posibles irregularidades u omisiones.

- Durante la **implementación** del lenguaje, la semántica puede ayudar a asegurar que la implementación se comporta de forma adecuada.
- La **estandarización** del lenguaje se debe realizar mediante la publicación de una semántica no ambigüa. Los programas deben poder transportarse de una implementación a otra exhibiendo el mismo comportamiento.
- La **comprensión** de un lenguaje por parte del programador requiere el aprendizaje de su comportamiento, es decir, de su semántica. La semántica debe aclarar el comportamiento del lenguaje y sus diversas construcciones en términos de conceptos familiares, haciendo aparente la relación entre el lenguaje considerado y otros lenguajes familiares para el programador.
- La semántica asiste al programador durante el **razonamiento** sobre el comportamiento de un programa: verificando que hace lo que se pretende. Esto requiere la manipulación matemática de programas y significados para poder demostrar que los programas cumplen ciertas condiciones.
- Finalmente, el estudio de las especificaciones semánticas ayudará a comprender las relaciones entre diferentes lenguajes de programación, aislando propiedades comunes que permitan avanzar la **investigación de nuevos lenguajes** de programación.

Lamentablemente, la especificación de la mayoría de los lenguajes, desde Fortran [44] hasta Java [**?**] se realiza en un lenguaje natural más o menos formal. Las descripciones en lenguaje natural acarrean gran cantidad de problemas como la falta de rigurosidad y ambigüedad.

En muchas ocasiones, se recurre a la utilización de *implementaciones prototipo*. Se define un intérprete estándar para el lenguaje que funciona en una determinada máquina y se indica que el comportamiento de un programa escrito en dicho lenguaje debe ser el mismo que el que se produzca al ejecutar dicho programa en el intérprete prototipo.

La utilización de implementaciones prototipo requiere decidir en qué maquina se va a ejecutar dicho intérprete y qué lenguaje se utiliza para su implementación. En la mayoría de las ocasiones se utilizan lenguajes ya implementados con el fin de ofrecer especificaciones ejecutables. En otras ocasiones, se utiliza el mismo lenguaje que se está definiendo. En todos los casos, se produce una *sobre-especificación*, no solamente es necesario especificar el lenguaje objeto, sino el lenguaje de implementación.

En las siguientes secciones se presentan las principales técnicas de descripción semántica. Para ello se describirán los mismos lenguajes en cada técnica.

3.2 Lenguaje Natural

3.2.1 Especificación del ejemplo

Como se ha indicado, la descripción de la mayoría de los lenguajes se realiza en un lenguaje natural más o menos riguroso. Para valorar la modularidad de las técnicas, se describirán cinco lenguajes de forma incremental, es decir, cada lenguaje añade nuevas características al lenguaje anterior.

3.2.1.1 Lenguaje $\mathcal{L}_0$

El lenguaje $\mathcal{L}_0$ consiste únicamente en expresiones aritméticas simples. La sintaxis abstracta viene dada por:

$$\begin{array}{lll} expr & : Const\ Integer & \text{— constante numérica} \\ & \mid expr\ +\ expr & \text{— suma} \\ & \mid expr\ -\ expr & \text{— resta} \end{array}$$

$\mathcal{L}_0$ tiene un único tipo primitivo $\mathbb{N}$ que representa los números enteros. Las expresiones de $\mathcal{L}_0$ denotan valores de $\mathbb{N}$.

Ejemplo 3.1 *La expresión Const* $3 + Const\ 4$ *es una sentencia de* $\mathcal{L}_0$ *que denota el valor* $7 \in \mathbb{N}$.

3.2.1.2 Lenguaje $\mathcal{L}_1$

El lenguaje $\mathcal{L}_1$ añade productos y divisiones a $\mathcal{L}_0$. La sintaxis será:

$$\begin{array}{lll} expr & : \ldots & \text{— definiciones anteriores de } \mathcal{L}_0 \\ & \mid expr\ *\ expr & \text{— producto} \\ & \mid expr\ /\ expr & \text{— división} \end{array}$$

Aunque la extensión realizada pueda parecer inocua, con la operación de división surgirá la posibilidad de que se produzcan errores (divisiones por cero). De esta forma, la expresión

$$Const\ 6\ /\ Const\ 0$$

no indica un valor que pertenezca a $\mathbb{N}$, sino un error. La descripción semántica del lenguaje deberá tener en cuenta esta posibilidad.

3.2.1.3 Lenguaje $\mathcal{L}_2$

$\mathcal{L}_2$ añade expresiones relacionales al lenguaje anterior. La sintaxis será

$$\begin{array}{lll} expr & : \ldots & \text{— definiciones anteriores de } \mathcal{L}_1 \\ & \mid expr\ =\ expr & \text{— relación de igualdad} \\ & \mid expr\ <\ expr & \text{— relación menor} \\ & \mid \mathsf{True} & \text{— verdadero} \\ & \mid \mathsf{False} & \text{— Falso} \end{array}$$

Semánticamente, al añadir expresiones relacionales, las expresiones del lenguaje $\mathcal{L}_2$ pueden denotar dos tipos de valores. O bien denotan un valor numérico perteneciente a $\mathbb{N}$, o un valor booleano perteneciente a $\mathbb{B}$.

3.2.1.4 Lenguaje $\mathcal{L}_3$

El lenguaje $\mathcal{L}_3$ añade variables o identificadores al lenguaje anterior. La sintaxis será:

$$\begin{array}{lll} expr & : \ldots & \text{— definiciones anteriores de } \mathcal{L}_2 \\ & \mid Var\ ident & \text{— identificador} \end{array}$$

Semánticamente, la evaluación de una expresión con identificadores del tipo

$$\mathit{Const}\ 3\ +\ \mathit{Var}\ x$$

depende del valor que el identificador x tenga en el momento de la evaluación.

Aunque la modelización del entorno difiere según el formalismo semántico utilizado. Para unificar las diferentes presentaciones, ς denotará un valor de tipo *State* que se utilizará para representar el contexto en el que se evalúan los identificadores.

Para unificar la presentación de las especificaciones se supone que se han definido las siguientes funciones:

- $\mathit{upd} : \mathit{State} \to \mathit{ident} \to \mathit{Value} \to \mathit{State}$, $\mathit{upd}\ \varsigma\ x\ v$ devuelve el estado resultante de asignar a x el valor v en el estdo ς.
- $\mathit{lkp} : \mathit{State} \to \mathit{ident} \to \mathit{Value}$, $\mathit{lkp}\ \varsigma\ x$ devuelve el valor de v en ς

3.2.1.5 Lenguaje $\mathcal{L}_4$

$\mathcal{L}_4$ incorpora comandos imperativos formados por secuencias y asignaciones con la siguiente sintaxis

$$\begin{array}{lll} \mathit{comm} & : \mathit{ident} := \mathit{expr} & \text{— asignación} \\ & \mid \mathit{comm}\ ;\ \mathit{comm} & \text{— secuencia} \\ & \mid \mathit{skip} & \text{— comando vacío} \end{array}$$

Los comandos imperativos denotan una actualización del entorno que modifica el valor de las variables.

Ejemplo 3.2 *La sentencia*

$$x\ :=\ \mathit{Const}\ 1\ ;\ y\ :=\ \mathit{Var}\ y\ +\ \mathit{Var}\ x$$

modifica el valor de x en el entorno (asignándole un 1) y el valor de y asignándole la suma del valor anterior de y con el nuevo valor de x.

Obsérvese que el operador de secuenciamiento requiere que la evaluación se realice de izquierda a derecha.

El comando *skip* no realiza ninguna modificación al entorno. Se incluye únicamente porque su utilización facilita la legibilidad de algunas descripciones semánticas.

3.2.1.6 Lenguaje $\mathcal{L}_5$

$\mathcal{L}_5$ añade sentencias condicionales y repetitivas a $\mathcal{L}_4$ con la siguiente sintaxis

$$\begin{array}{lll} \mathit{comm} & : \ldots & \text{— definiciones anteriores de } \mathcal{L}_4 \\ & \mid \textbf{if}\ \mathit{expr}\ \textbf{then}\ \mathit{comm}_1\ \textbf{else}\ \mathit{comm}_2 & \text{— condicional} \\ & \mid \textbf{while}\ \mathit{expr}\ \textbf{do}\ \mathit{comm} & \text{— repetición} \end{array}$$

Ejemplo 3.3 *El siguiente programa calcula el factorial de la variable almacenada en x*

$n := Var\ x;$
$fact := Const\ 1;$
while $Var\ n > Const\ 1$ do
  $fact := Var\ fact * Var\ n$
  $n := Var\ n - Const\ 1;$

3.2.2 Valoración

A continuación se realiza la valoración del lenguaje natural respecto a las características definidas en 1.1.

- No ambigüedad. Evidentemente, la descripción del comportamiento de un sistema en lenguaje natural es una puerta abierta a todo tipo de problemas de *ambigüedad* y falta de rigurosidad.

- Modularidad. Las descripciones en lenguaje natural permiten una descripción incremental de las características computacionales (debido precisamente a la falta de rigurosidad mencionada).

- Reusabilidad. De la misma forma, las descripciones son reutilizables. Así, la descripción de las expresiones aritméticas no ha sido modificada al añadir comandos.

- Demostración. Esa falta de rigurosidad es la causante de que no sea posible demostrar propiedades básicas entre los programas. Lo ideal sería que a partir de la especificación en lenguaje natural fuese posible desarrollar un compilador correcto, sin embargo, ésto no es posible en la actualidad.

- Prototipo. De la misma forma, actualmente es imposible transformar especificaciones en lenguaje natural en el intérprete de un lenguaje de forma automática.

- Legibilidad. Las descripciones en lenguaje natural son engañosamente legibles. En primer lugar, la legibilidad depende claramente de la capacidad redactora del especificador (lo cual, no suele cumplirse). El lenguaje natural ofrece una gran libertad para la especificación. En ocasiones las especificaciones son especialmente detalladas, formando grandes tomos de difícil lectura. Otras veces, las especificaciones son excesivamente resumidas, dejando detalles sin especificar que pueden dar lugar a diferentes interpretaciones.

- Flexibilidad. Evidentemente, el lenguaje natural tiene una enorme expresividad y flexibilidad, adaptándose a la especificación de todo tipo de lenguajes y paradigmas.

- Experiencia. La especificación de la mayoría de los lenguajes populares ha sido realizada en lenguaje natural.

3.3 Semántica Operacional

3.3.1 Descripción

La semántica operacional se centra en la descripción paso a paso del comportamiento de un programa al ejecutarse en una máquina abstracta. De esta forma, la construcción de un intérprete de un lenguaje puede considerarse un ejemplo de semántica operacional.

Uno de los primeros intentos de especificación formal fue la descripción del lenguaje *PL/I* en 1969 desarrollada en el laboratorio IBM de Viena. Se utilizaba una notación denominada *VDL* (*Viena Definition Language*). La especificación consistía en dos partes, un traductor indicaba cómo transformar programas PL/I en una sintaxis abstracta, y un intérprete especificaba cómo ejecutar dicha sintaxis abstracta.

En 1981, G. Plotkin [195] desarrolla la *semántica operacional estructurada* (*SOS*). Se especifican las transiciones elementales de un programa mediante reglas de inferencia definidas por inducción sobre su estructura, lo cual permite demostrar propiedades. Este formalismo se aplica habitualmente en la formalización de lenguajes concurrentes.

Recientemente, P. Mosses, inspirando por la semántica monádica modular, ha propuesto una semántica operacional estructurada modular [176]

Las reglas de inferencia tienen el siguiente formato:

$$\frac{P_1 \quad P_2 \quad \dots \quad P_n}{Q}$$

que equivale a: si se cumple P_1 y P_2 y $\dots P_n$ entonces se cumple Q.

3.3.2 Especificación del ejemplo

3.3.2.1 Lenguaje $\mathcal{L}_0$

El lenguaje $\mathcal{L}_0$ consta únicamente de expresiones aritméticas sencillas cuya evaluación producirá un valor de tipo $\mathbb{N}$.

La semántica operacional define, una función de evaluación $\overset{eval}{\rightsquigarrow}$: $Expr \to \mathbb{N}$ que relaciona una expresión con el valor que denota.

La notación $\langle e \rangle \overset{eval}{\rightsquigarrow} v$ indica que v es el resultado de evaluar la expresión e.

Las reglas de inferencia son las siguientes

$$\frac{}{\langle Const\, n \rangle \overset{eval}{\rightsquigarrow} n}$$

$$\frac{\langle E_1 \rangle \overset{eval}{\rightsquigarrow} v_1 \quad \langle E_2 \rangle \overset{eval}{\rightsquigarrow} v_2}{\langle E_1 + E_2 \rangle \overset{eval}{\rightsquigarrow} v_1 + v_2}$$

$$\frac{\langle E_1 \rangle \overset{eval}{\rightsquigarrow} v_1 \quad \langle E_2 \rangle \overset{eval}{\rightsquigarrow} v_2}{\langle E_1 - E_2 \rangle \overset{eval}{\rightsquigarrow} v_1 - v_2}$$

3.3.2.2 Lenguaje $\mathcal{L}_1$

En el lenguaje $\mathcal{L}_1$ se añadían multiplicaciones y divisiones. Como ya se ha mencionado, con las divisiones aparece la posibilidad de errores. Convencionalmente, se utiliza el símbolo $\perp$para denotar el resultado de una evaluación que ha producido un error.

Dado un conjunto S, se denotará por $S_\perp$ el conjunto $S \cup \{\perp\}$ De esta forma, el resultado de una evaluación será un valor del conjunto $\mathbb{N}_\perp$.

El tipo de la función de evaluación se modifica para contemplar la posibilidad de que el resultado de la evaluación sea un error. El tipo será $\overset{eval}{\rightsquigarrow}: \mathit{Expr} \to \mathbb{N}_\perp$.

Las funciones básicas, como la suma o la resta, también deben ser modificadas. Por ejemplo, la función $+ : \mathbb{N} \to \mathbb{N} \to \mathbb{N}$ debe ampliarse para que tenga el tipo $+ : \mathbb{N}_\perp \to \mathbb{N}_\perp \to \mathbb{N}_\perp$. Esta ampliación es trivial, aunque tediosa.

Las nuevas reglas de inferencia serán:

$$\frac{\langle E_1\rangle \overset{eval}{\rightsquigarrow} v_1 \quad \langle E_2\rangle \overset{eval}{\rightsquigarrow} v_2}{\langle E_1 * E_2\rangle \overset{eval}{\rightsquigarrow} v_1 * v_2}$$

$$\frac{\langle E_1\rangle \overset{eval}{\rightsquigarrow} v_1 \quad \langle E_2\rangle \overset{eval}{\rightsquigarrow} v_2 \quad v_2 \neq 0}{\langle E_1/E_2\rangle \overset{eval}{\rightsquigarrow} v_1/v_2}$$

$$\frac{\langle E_1\rangle \overset{eval}{\rightsquigarrow} v_1 \quad \langle E_2\rangle \overset{eval}{\rightsquigarrow} v_2 \quad v_2 = 0}{\langle E_1/E_2\rangle \overset{eval}{\rightsquigarrow} \perp}$$

3.3.2.3 Lenguaje $\mathcal{L}_2$

En $\mathcal{L}_2$ se añaden expresiones relacionales. Semánticamente, será necesario adaptar la función de evaluación para que el resultado de una evaluación sea, o bien un valor entero $n \in \mathbb{N}$, o un valor booleano $b \in \mathbb{B}$ o un error $\perp$. El conjunto de posibles valores será, por tanto $(\mathbb{N} \cup \mathbb{B})_\perp$ y las reglas de inferencia

$$\frac{}{\langle \mathsf{True}\rangle \overset{eval}{\rightsquigarrow} \mathsf{True}}$$

$$\frac{}{\langle \mathsf{False}\rangle \overset{eval}{\rightsquigarrow} \mathsf{False}}$$

$$\frac{\langle E_1\rangle \overset{eval}{\rightsquigarrow} v_1 \quad \langle E_2\rangle \overset{eval}{\rightsquigarrow} v_2}{\langle E_1 = E_2\rangle \overset{eval}{\rightsquigarrow} v_1 = v_2}$$

$$\frac{\langle E_1\rangle \overset{eval}{\rightsquigarrow} v_1 \quad \langle E_2\rangle \overset{eval}{\rightsquigarrow} v_2}{\langle E_1 < E_2\rangle \overset{eval}{\rightsquigarrow} v_1 < v_2}$$

3.3.2.4 Lenguaje $\mathcal{L}_3$

Al añadir variables al conjunto de expresiones, la función de evaluación debe modificarse para poder consultar el valor de las variables en un contexto ς.

De esta forma, es necesario realizar una nueva modificación en la función de evaluación. El nuevo tipo será $\overset{eval}{\rightsquigarrow}: \mathit{Expr} \to \mathit{State} \to (\mathbb{N} \cup \mathbb{B})_\perp$. Es decir, el

significado de una expresión ya no será un valor, sino una función que depende del contexto.

El nuevo formato será: $\langle e, \varsigma \rangle \overset{eval}{\rightsquigarrow} v$ indicando que v es el resultado de evaluar la expresión e en el contexto ς.

La regla de inferencia para la evaluación de variables consiste simplemente en buscar el valor de la variable en el contexto.

$$\frac{}{\langle Var\, v, \varsigma \rangle \overset{eval}{\rightsquigarrow} lkp\ v\ \varsigma}$$

Sin embargo, la modificación realizada afecta a las definiciones realizadas hasta el momento. Las cuales deben adaptarse al nuevo tipo de la función de evaluación. A continuación se presentan las dos primeras reglas de inferencia modificadas. El resto sufriría un tratamiento similar.

$$\frac{}{\langle Const\, n, \varsigma \rangle \overset{eval}{\rightsquigarrow} n}$$

$$\frac{\langle E_1, \varsigma \rangle \overset{eval}{\rightsquigarrow} v_1 \qquad \langle E_2, \varsigma \rangle \overset{eval}{\rightsquigarrow} v_2}{\langle E_1 + E_2, \varsigma \rangle \overset{eval}{\rightsquigarrow} v_1 + v_2}$$

3.3.2.5 Lenguaje $\mathcal{L}_4$

En $\mathcal{L}_4$ se añaden comandos imperativos de asignación y secuencia. Los comandos imperativos se caracterizan porque modifican un estado global $\varsigma \in State$.

El significado de un comandos imperativo es una función que transforma un estado en un nuevo estado. Se define $\overset{step}{\rightsquigarrow}: (Comm \times State) \to (Comm \times State)$.

La función de ejecución secuencial paso a paso tendrá el formato

$$\langle C_1, \varsigma \rangle \overset{step}{\rightsquigarrow} \langle C_2, \varsigma' \rangle$$

indicando que, tras la ejecución de C_1 en el estado ς debe continuarse ejecutando C_2 en el estado ς'.

Las reglas de inferencia serán

$$\frac{\langle E, \varsigma \rangle \overset{eval}{\rightsquigarrow} v}{\langle x := E, \varsigma \rangle \overset{step}{\rightsquigarrow} \langle skip, upd\ \varsigma\ x\ v \rangle}$$

$$\frac{}{\langle \mathsf{skip}; M, \varsigma \rangle \overset{step}{\rightsquigarrow} \langle M, \varsigma \rangle}$$

$$\frac{\langle M_1, \varsigma \rangle \overset{step}{\rightsquigarrow} \langle M_1', \varsigma' \rangle}{\langle M_1; M_2, \varsigma \rangle \overset{step}{\rightsquigarrow} \langle M_1'; M_2, \varsigma' \rangle}$$

3.3.2.6 Lenguaje $\mathcal{L}_5$

En $\mathcal{L}_5$ se añade el comando condicional y la sentencia repetitiva. La semántica operacional de ambos se define fácilmente:

$$\frac{\langle E, \varsigma \rangle \overset{eval}{\rightsquigarrow} \mathsf{True}}{\langle \mathsf{if}\ E\ \mathsf{then}\ M_1\ \mathsf{else}\ M_2, \varsigma \rangle \overset{step}{\rightsquigarrow} \langle M_1, \varsigma \rangle}$$

$$\frac{\langle E, \varsigma\rangle \overset{eval}{\rightsquigarrow} \mathsf{False}}{\langle \mathsf{if}\ E\ \mathsf{then}\ M_1\ \mathsf{else}\ M_2, \varsigma\rangle \overset{step}{\rightsquigarrow} \langle M_2, \varsigma\rangle}$$

$$\frac{}{\langle \mathsf{while}\ E\ \mathsf{do}\ M, \varsigma\rangle \overset{step}{\rightsquigarrow} \langle \mathsf{if}\ E\ \mathsf{then}\ (M; \mathsf{while}\ E\ \mathsf{do}\ M)\ \mathsf{else}\ \mathsf{skip}, \varsigma\rangle}$$

Obsérvese que la regla de inferencia que describe la sentencia while no es composicional ya que la definición del comando se hace en función de sí mismo. En el caso de la semántica operacional, este tipo de definiciones no suponen ningún problema, ya que las reglas de inferencia están definiendo la traza de la ejecución de un posible intérprete. Sin embargo, la solución de este tipo de definiciones en semántica denotacional va a requerir la utilización de puntos fijos.

La semántica operacional definida se denomina también semántica de paso simple (*single-step semantics*) ya que se especifica la traza de ejecución de cada comando paso a paso.

La semántica global de un comando puede obtenerse mediante

$$exec : (Comm \times State) \rightarrow State$$

que toma un comando y un estado y devuelve el estado final resultante de su ejecución.

$$exec(C, \varsigma) = \begin{cases} \varsigma' & \text{si } \langle c, \varsigma\rangle \overset{step^*}{\rightsquigarrow} \langle \mathsf{skip}, \varsigma'\rangle \\ \bot & \text{en caso contrario} \end{cases}$$

donde $\overset{step^*}{\rightsquigarrow}$ representa el cierre transitivo $\overset{step}{\rightsquigarrow}$

Definición 3.1 (Equivalencia observacional) *Dos programas P_1 y P_2 son* observacionalmente equivalentes *y se representa como $P_1 \approx P_2$ si para todo par de contextos ς,ς', $exec(P_1, \varsigma) = \varsigma'$ si y sólo si $exec(P_2, \varsigma) = \varsigma'$*

La equivalencia observacional entre dos programas indica que en todas las posibles ejecuciones, el resultado obtenido es el mismo.

3.3.3 Valoración

- *No ambigüedad.* La semántica operacional estructurada no es ambigüa.
- *Modularidad.* Los problemas de modularidad de este formalismo son los mismos que en semántica denotacional. Existe una propuesta de semántica operacional estructurada modular realizada por Mosses inspirándose en la semántica monádica [176].
- *Reusabilidad.* No es posible definir componentes semánticos reutilizables, ya que las definiciones se realizan por inducción directa sobre la estructura del lenguaje.
- *Demostración.* Es difícil realizar demostraciones de propiedades ya que este formalismo define una semántica de bajo nivel.

- *Prototipo.* Es relativamente sencillo construir un prototipo a partir de la semántica operacional estructurada, aunque no se conocen sistemas que obtengan esta transformación de forma automática.
- *Legibilidad.* Las definiciones utilizan reglas de inferencia que no son especialmente difíciles de comprender, aunque requieren cierta familiaridad con los rudimentos de algún sistema de inferencia lógica.
- *Flexibilidad.* Este formalismo se adapta fácilmente a diferentes lenguajes y paradigmas ya que sus descripciones se realizan a bajo nivel.
- *Experiencia.* Este formalismo se ha aplicado con éxito a la especificación de varios lenguajes, especialmente, para la descripción de sistemas concurrentes y distribuidos.

3.4 Semántica Natural

3.4.1 Descripción

La *semántica natural* [115] surge como una variación de la semántica operacional que define transiciones globales en lugar de transiciones elementales. También se denomina *big-step semantics* o *semántica de gran paso.* En algunos contextos se produce cierta confusión entre este formalismo y el anterior, denominando semántica operacional estructurada a lo que aquí se denomina semántica natural [90].

3.4.2 Especificación del ejemplo

La especificación de las expresiones aritméticas y relacionales (lenguajes $\mathcal{L}_0$ a $\mathcal{L}_3$) es idéntica a la sección anterior, por lo que se omite su presentación. Conviene indicar que los problemas de modularidad de este formalismo son los mismos que en el formalismo anterior.

A continuación se presenta la especificación de los comandos añadidos en los lenguajes $\mathcal{L}_4$ y $\mathcal{L}_5$. Se utiliza la función

$$\overset{exec}{\rightsquigarrow}\colon (Comm \times State) \to State$$

La definición $\langle c, \varsigma\rangle \overset{exec}{\rightsquigarrow} \varsigma'$ indica que al ejecutar el comando c en el estado inicial ς se obtiene el estado final ς'.

$$\frac{\langle E, \varsigma\rangle \overset{eval}{\rightsquigarrow} v}{\langle x := E, \varsigma\rangle \overset{exec}{\rightsquigarrow} upd\ \varsigma\ x\ v}$$

$$\frac{}{\langle \mathsf{skip}, \varsigma\rangle \overset{exec}{\rightsquigarrow} \varsigma}$$

$$\frac{\langle M_1, \varsigma\rangle \overset{exec}{\rightsquigarrow} \varsigma' \qquad \langle M_2, \varsigma'\rangle \overset{exec}{\rightsquigarrow} \varsigma''}{\langle M_1; M_2, \varsigma\rangle \overset{exec}{\rightsquigarrow} \varsigma''}$$

$$\frac{\langle E, \varsigma\rangle \overset{eval}{\rightsquigarrow} \mathsf{True} \qquad \langle M_1, \varsigma\rangle \overset{exec}{\rightsquigarrow} \varsigma'}{\langle \mathsf{if}\ E\ \mathsf{then}\ M_1\ \mathsf{else}\ M_2, \varsigma\rangle \overset{exec}{\rightsquigarrow} \varsigma'}$$

$$\frac{\langle E, \varsigma\rangle \overset{eval}{\rightsquigarrow} \mathsf{False} \quad \langle M_2, \varsigma\rangle \overset{exec}{\rightsquigarrow} \varsigma'}{\langle \mathsf{if}\ E\ \mathsf{then}\ M_1\ \mathsf{else}\ M_2, \varsigma\rangle \overset{exec}{\rightsquigarrow} \varsigma'}$$

$$\frac{\langle E, \varsigma\rangle \overset{eval}{\rightsquigarrow} \mathsf{True} \quad \langle M, \varsigma\rangle \overset{exec}{\rightsquigarrow} \varsigma' \quad \langle \mathsf{while}\ E\ \mathsf{do}\ M, \varsigma'\rangle \overset{exec}{\rightsquigarrow} \varsigma''}{\langle \mathsf{while}\ E\ \mathsf{do}\ M, \varsigma\rangle \overset{exec}{\rightsquigarrow} \varsigma''}$$

$$\frac{\langle E, \varsigma\rangle \overset{eval}{\rightsquigarrow} \mathsf{False}}{\langle \mathsf{while}\ E\ \mathsf{do}\ M, \varsigma\rangle \overset{exec}{\rightsquigarrow} \varsigma}$$

La semántica natural es una alternativa a la semántica operacional estructurada que se centra en el resultado final de la ejecución. Esta técnica se acerca de esta forma a la semántica denotacional. Sin embargo, obsérvese que la definición anterior de la sentencia *while* tampoco es composicional.

3.4.3 Valoración

- *No ambigüedad.* Este formalismo no contiene ambigüedad en sus especificaciones.

- *Modularidad.* Las descripciones en semántica natural tienen los mismos problemas de modularidad que las descripciones en semántica operacional estructurada.

- *Reusabilidad.* De la misma forma, tampoco es posible definir componentes semánticos reutilizables.

- *Demostración.* La semántica natural mejora la semántica operacional estructurada facilitando la demostración de propiedades.

- *Prototipo.* Existen varios sistemas para implementar la semántica natural. *Centaur* [29] utiliza un meta-lenguaje denominado *Typol.* cuyas definiciones son traducidas a Prolog. Posteriormente, el meta-lenguaje relacional *RML* [191] traduce sus definiciones al lenguaje C proporcionando una mayor eficiencia. En [226] se describe una codificación de especificaciones en semántica natural en el sistema de demostración de teoremas *Coq.*

- *Legibilidad.* La legibilidad de las especificaciones en semántica natural es similar a la de las especificaciones en semántica operacional.

- *Flexibilidad.* El formalismo se adapta a la descripción de diferentes paradigmas.

- *Experiencia.* La semántica natural se ha aplicado en la definición de la semántica estática y dinámica del lenguaje SML [166], y es uno de los formalismos más utilizados para la especificación de sistemas de tipos.

3.5 Semántica Denotacional

3.5.1 Descripción

En semántica denotacional el comportamiento del programa se describe modelizando los significados mediante entidades matemáticas básicas.

El estudio denotacional de lenguajes de programación fue propuesto originalmente por C. Strachey en 1967 [221] utilizando cálculo λ. Posteriormente D. Scott y C. Strachey colaboran en 1969 para establecer las bases teóricas utilizando teoría de deminios.

El estudio de la semántica denotacional de diferentes lenguajes de programación permite identificar los conceptos básicos subyacentes de dichos lenguajes. La semántica denotacional se ha aplicado a la modelización de varios lenguajes como Algol, Pascal, Scheme, Prolog [181], C [186] y más recientemente XPath [236]

VDM es en parte una variante de la semántica denotacional, que incluye algunas características propias para especificar lenguajes imperativos. VDM fue desarrollado en el laboratorio IBM de Viena y fue utilizado para la especificación formal de Ada, Algol-60 y Pascal.

Otros descendientes de la semántica denotacional serán la semántica monádica modular y la semántica de acción que se describirán posteriormente con más detalle.

3.5.2 Especificación del ejemplo

3.5.2.1 Lenguaje $\mathcal{L}_0$

En el caso del lenguaje $\mathcal{L}_0$, la evaluación de expresiones será una función $\mathcal{E} : Expr \to \mathbb{N}$.

$$\mathcal{E}[\![Const\, n]\!] = n$$

$$\mathcal{E}[\![E_1 + E_2]\!] = \mathcal{E}[\![E_1]\!] + \mathcal{E}[\![E_2]\!]$$

$$\mathcal{E}[\![E_1 - E_2]\!] = \mathcal{E}[\![E_1]\!] - \mathcal{E}[\![E_2]\!]$$

3.5.2.2 Lenguaje $\mathcal{L}_1$

De igual forma que con la semántica operacional, al añadir divisiones y aparecer errores, los dominios deben modificarse. La función de evaluación deberá modificarse para que tenga el tipo $\mathcal{E} : Expr \to \mathbb{N}_\perp$ y las funciones utilizadas para la suma, resta, multiplicación y división deben contemplar el caso particular en el que uno de los argumentos esté indefinido.

Por lo demás, las funciones semánticas que deben añadirse son simples

$$\mathcal{E}[\![E_1 * E_2]\!] = \mathcal{E}[\![E_1]\!] * \mathcal{E}[\![E_2]\!]$$

$$\mathcal{E}[\![E_1/E_2]\!] = \begin{cases} \mathcal{E}[\![E_1]\!]/\mathcal{E}[\![E_2]\!] & \text{si } \mathcal{E}[\![E_2]\!] \neq 0 \\ \perp & \text{en caso contrario} \end{cases}$$

3.5.2.3 Lenguaje $\mathcal{L}_2$

Al añadir expresiones relacionales, se modificará el dominio para incorporar valores booleanos. De esa forma, la función de evaluación debe modificarse para que tenga el tipo $\mathcal{E} : Expr \to (\mathbb{N} \cup \mathbb{B})_\perp$.

$$\mathcal{E}[\![\mathsf{True}]\!] = \mathsf{True}$$

$$\mathcal{E}[\![\mathsf{False}]\!] = \mathsf{False}$$

$$\mathcal{E}[\![E_1 = E_2]\!] = \mathcal{E}[\![E_1]\!] = \mathcal{E}[\![E_2]\!]$$

$$\mathcal{E}[\![E_1 < E_2]\!] = \mathcal{E}[\![E_1]\!] < \mathcal{E}[\![E_2]\!]$$

3.5.2.4 Lenguaje $\mathcal{L}_3$

Al añadir variables al lenguaje, es necesario modificar la función de evaluación para que el valor de las variables se obtenga del contexto ς. De esta forma, la función de evaluación será $\mathcal{E} : Expr \to State \to (\mathbb{N} \cup \mathbb{B})_\perp$.

$$\mathcal{E}[\![Var\, v]\!]\varsigma = lkp\ v\ \varsigma$$

Sin embargo, la modificación anterior obliga a modificar todas las definiciones anteriores para que tengan en cuenta el nuevo argumento. Las dos primeras definiciones deberían ser

$$\mathcal{E}[\![Const\, n]\!]\varsigma = n$$

$$\mathcal{E}[\![E_1 + E_2]\!]\varsigma = \mathcal{E}[\![E_1]\!]\varsigma + \mathcal{E}[\![E_2]\!]\varsigma$$

El resto de definiciones debería modificarse de forma similar (y tediosa).

3.5.2.5 Lenguaje $\mathcal{L}_4$

Para capturar la semántica denotacional de los comandos se utilizará una nueva función semántica $\mathcal{C} : Comm \to State \to State$. Es decir, la ejecución de un comando denota una función que, dado un estado inicial, devuelve un estado final.

$$\mathcal{C}[\![\mathsf{skip}]\!]\varsigma = \varsigma$$

$$\mathcal{C}[\![x := E]\!]\varsigma = upd\ \varsigma\ x\ (\mathcal{E}[\![E_1]\!]\varsigma)$$

$$\mathcal{C}[\![C_1; C_2]\!]\varsigma = \mathcal{C}[\![C_2]\!](\mathcal{C}[\![C_1]\!])$$

3.5.2.6 Lenguaje $\mathcal{L}_5$

$$\mathcal{C}[\![\mathsf{if}\ E\ \mathsf{then}\ C_1\ \mathsf{else}\ C_2]\!]\varsigma = \begin{cases} \mathcal{C}[\![C_1]\!]\varsigma & \text{si } \mathcal{E}[\![\mathtt{E}]\!]\varsigma = \mathsf{True} \\ \mathcal{C}[\![C_2]\!]\varsigma & \text{si } \mathcal{E}[\![\mathtt{E}]\!]\varsigma = \mathsf{False} \end{cases}$$

La definición de la sentencia *while* requiere la utilización de una función auxiliar recursiva.

$$\mathcal{C}[\![\mathsf{while}\ E\ \mathsf{do}\ C]\!]\varsigma = loop\ \varsigma$$

donde

$$loop\ \varsigma = \begin{cases} loop\,(\mathcal{C}[\![C]\!]\varsigma) & \text{si } \mathcal{E}[\![\mathtt{E}]\!]\varsigma = \mathsf{True} \\ \varsigma & \text{si } \mathcal{E}[\![\mathtt{E}]\!]\varsigma = \mathsf{False} \end{cases}$$

Aunque en un principio, la definición de la función *loop* puede parecer no composicional, en realidad puede definirse como el punto fijo de una función no recursiva.

Definición 3.2 (Adecuación y Completud de abstracción) *La semántica denotacional de un lenguaje es adecuada si para todo par de programas P_1 y P_2 se cumple que*

$$\mathcal{C}[\![P_1]\!] = \mathcal{C}[\![P_2]\!] \Longrightarrow P_1 \approx P_2$$

si además, la implicación se cumple en el otro sentido, entonces se dice que se cumple la semántica denotacional es completamente abstracta.

La completud de abstracción es una propiedad muy útil de las descripciones en semántica denotacional ya que los razonamientos realizados en semántica denotacional pueden incorporarse en semántica operacional. Esto es importante ya que realizar demostraciones directamente en semántica operacional suele ser complicado y, sin embargo, estudiar la equivalencia entre programas es muy importante para realizar transformaciones y optimizaciones. No obstante, la construcción de semánticas denotacionales completamente abstractas es un problema matemático difícil de resolver, en general [165, 219].

3.5.2.7 Continuaciones

Aunque los lenguajes $\mathcal{L}_0$ a $\mathcal{L}_5$ no requieren la utilización de continuaciones. En esta sección se introduce brevemente el concepto ya que se utilizará posteriormente en la sección 7.1.3.1.

Las continuaciones fueron propuestas a principios de los años 70 por C. Strachey y C. Wadsworth [222] como un mecanismo para la especificación semántica de programas que incluyen saltos.

La modelización de la ejecución secuencial de $\mathcal{L}_5$ era

$$\mathcal{C}[\![C_1; C_2]\!]\varsigma = \mathcal{C}[\![C_2]\!](\mathcal{C}[\![C_1]\!])$$

El problema surge cuando el comando C_1 contiene algún salto o interrumpe la ejecución normal.

Con la definición planteada, $\mathcal{C}[\![C_2]\!]$ sigue siempre a $\mathcal{C}[\![C_1]\!]$.

Para evitarlo, se utiliza un argumento extra (una función denominada *continuación*) que indicará cómo continúa la ejecución desde ese comando hasta el final del programa en caso de que dicho comando devuelva el control normalmente. La modelización de un comando sin saltos, implicará una llamada a la continuación (indicando que la ejecución continúa normalmente), mientras que un comando que no devuelva el control, no llamará a la continuación. De esta forma, el significado de un comando será de la forma $\mathcal{C} : Comm \times Cont \times State \to State$ donde $Cont : State \to State$ será la continuación.

La modelización de la ejecución secuencial sería entonces:

$$\mathcal{C}[\![C_1\,;C_2]\!]\kappa\varsigma = \mathcal{C}[\![C_1]\!](\mathcal{C}[\![C_2]\!]\kappa)\varsigma$$

3.5.3 Valoración

A continuación se realiza una valoración de la semántica denotacional siguiendo los criterios definidos en 1.1.

- *No ambigüedad.* Este formalismo no contiene ambigüedad en sus especificaciones.

- *Modularidad.* El principal problema de la semántica denotacional es la falta de modularidad. Al modificar la estructura computacional se requiere una modificación en las definiciones previas.

- *Reusabilidad.* No es posible definir componentes semánticos reutilizables.

- *Demostración.* Es posible la demostración de propiedades. Por ejemplo en [229] se toman descripciones en semántica denotacional y se transforman en descripciones en semántica axiomática y operacional estructurada, demostrando que la corrección de todo el proceso. Para ello se utiliza el sistema de demostración de teoremas PVS.

- *Prototipo.* No obstante, son escasos los sistemas que permitan transformar especificaciones en semántica denotacional en prototipos de lenguajes.

- *Legibilidad.* Las descripciones en semántica denotacional suelen utilizar notaciones con un sabor quizás demasiado matemático lo cual complica la comprensión de las ideas subyacentes. Además, algunos conceptos empleados en semántica denotacional no son precisamente obvios.

- *Flexibilidad.* La semántica denotacional ha sido aplicada a lenguajes de diferentes paradigmas: funcionales, imperativos, lógicos, orientados a objetos, etc.

- *Experiencia.* Existen algunos intentos de aplicación de semántica denotacional a lenguajes reales. Por ejemplo, en [186] se desarrolla la semántica denotacional del lenguaje C, aunque recurre a la utilización de mónadas para facilitar la modularidad.

3.6 Semántica Axiomática

3.6.1 Descripción

En la *Semántica Axiomática*, las características de un lenguaje se describen mediante los axiomas de un sistema lógico. Las raíces de este formalismo se encuentran en los trabajos de Floyd para razonar con diagramas de flujo en 1967. Posteriormente, Hoare extiende dichas reglas para la verificación de un lenguaje imperativo [88] y posteriormente se desarrolla la semántica axiomática del lenguaje Pascal.

Este formalismo ha alcanzado gran popularidad en medios académicos, existiendo numerosos libros de texto en los que se describe con mayor detalle [67, 12, 190]

La utilización de la semántica axiomática para la descripción de lenguajes de programación se presenta en [245, 114].

Definición 3.3 (Aserción) *Una* aserción *es una fórmula P de lógica de predicados que toma un valor booleano en un contexto ς, el cual se denota como* $[\![P]\!](\varsigma)$

Definición 3.4 (Terna de Hoare) *Una* terna de Hoare *es una expresión de la forma $\{P\}C\{Q\}$ donde P y Q son aserciones y C es una sentencia. P se denomina* precondición *y Q se denomina* postcondición.

Definición 3.5 (Corrección parcial y total) *La terna $\{P\}C\{Q\}$ es parcialmente correcta si toda ejecución de C en un contexto ς en el que $[\![P]\!](\varsigma) = \mathsf{True}$ cuando termina, cumple que $[\![Q]\!](\varsigma) = \mathsf{True}$. La corrección total se cumple cuando se certifica además que la ejecución termina.*

La semántica axiomática consiste en definir una serie de reglas de inferencia que caracterizan las propiedades de las diferentes construcciones del lenguaje. Un problema de este formalismo es que la identificación de estas propiedades no es una tarea sencilla. Normalmente, este formalismo se emplea a posteriori, es decir, una vez definido el lenguaje, se estudia su semántica axiomática.

3.6.2 Especificación del ejemplo

A continuación se exponen las reglas de inferencia de los comandos del lenguaje ejemplo.

$$\frac{}{\{P\}\mathsf{skip}\{P\}}$$

$$\frac{}{\{P(x/E)\}\mathsf{x} := \mathsf{E}\{P\}}$$

$P(x/E)$ denota la aserción obtenida al substituir todas las apariciones de x por E en P.

$$\frac{\{P\}M_1\{Q\} \quad \{Q\}M_2\{R\}}{\{P\}M_1; M_2\{R\}}$$

Condicional

$$\frac{\{P \wedge E\}M_1\{Q\} \quad \{P \wedge \neg E\}M_2\{Q\}}{\{P\}\mathsf{if}\ E\ \mathsf{then}\ M_1\ \mathsf{else}\ M_2\{Q\}}$$

Repetición

$$\frac{\{P \wedge E\}M\{P\}}{\{P\}\mathsf{while}\ E\ \mathsf{do}\ M\{P \wedge \neg E\}}$$

Para poder realizar verificaciones de algoritmos se incluyen varios axiomas generales, entre los que se pueden destacar.

- Reforzamiento de precondición

$$\frac{P_0 \rightarrow P_1 \quad \{P_1\}C\{Q\}}{\{P_0\}C\{Q\}}$$

- Debilitamiento de la postcondición

$$\frac{Q_0 \rightarrow Q_1 \quad \{P\}C\{Q_0\}}{\{P\}C\{Q_1\}}$$

- Disyunción de la precondición

$$\frac{\{P_0\}C\{Q\} \quad \{P_1\}C\{Q\}}{\{P_0 \vee P_1\}C\{Q\}}$$

- Conjunción de la postcondición

$$\frac{\{P\}C\{Q_0\} \quad \{P\}C\{Q_1\}}{\{P\}C\{Q_0 \wedge Q_1\}}$$

3.6.3 Valoración

- *No ambigüedad*. Aunque este formalismo no contiene ambigüedad en sus especificaciones. Suele utilizarse para la verificación de propiedades de algoritmos y se ha detectado que en varios libros de texto, la especificación axiomática de las expresiones básicas (números, booleanos, etc.) no se realiza de forma rigurosa ya que requiere utilizar teorías lógicas que se salen del ámbito de dichos textos.

- *Modularidad*. La especificación axiomática se centra principalmente en definir las propiedades que deben cumplir las características del lenguaje de forma bastante aislada. De esta forma, se resiente al modificar la estructura computacional subyacente.

- *Reusabilidad*. Las definiciones axiomáticas se realizan de una forma global. No se plantea la reutilización de especificaciones.

- *Demostración*. El principal beneficio de la semántica axiomática es precisamente permitir demostrar propiedades de los programas. Además de la verificación a posteriori, existen diversos trabajos que profundizan en la posibilidad de derivación de programas a partir de la especificación [116, 67]

- *Prototipo.* No se conocen sistemas que obtengan directamente prototipos a partir de especificaciones axiomátivas.
- *Legibilidad.* Las reglas de inferencia son relativamente comprensibles, aunque se requiere bastante familiaridad con este tipo de definiciones. La obtención de dichas reglas no es fácil.
- *Flexibilidad.* La semántica axiomática resulta, por tanto, poco flexible a cambios de paradigmas y su aportación al diseño de lenguajes debería más bien considerarse para formalizar especificaciones a posteriori.
- *Experiencia.* La semántica axiomática fue aplicada a posteriori para la especificación del lenguaje Pascal. El lenguaje *Euclid* fue diseñado utilizando semántica axiomática.

3.7 Semántica Algebraica

3.7.1 Descripción

La semántica algebraica aparece a finales de los años 70, a partir de los trabajos desarrollados para la especificación algebraica de tipos de datos. Matemáticamente, se basa en el álgebra universal, también llamada lógica ecuacional [163].

Una *especificación algebraica* permite definir una estructura matemática de forma abstracta junto con las propiedades que debe cumplir. Existen múltiples lenguajes de especificación algebraica como ASL [207], Larch [39], ACT ONE, Clear, OBJ [63], etc. Recientemente, se ha creado una iniciativa para el desarrolo de un marco común de especificación algebraica [5].

En esta sección se adopta una notación inspirada en la notación utilizada en [63] basada en el lenguaje OBJ. Una especificación algebraica consiste en dos partes:

- La *signatura* define los géneros (*sorts*) que se están especificando, así como los símbolos de operaciones sobre dichos géneros y sus funcionalidades.
- Los *axiomas* son sentencias lógicas que definen el comportamiento de las operaciones. Habitualmente se utilizan una serie de ecuaciones, estableciendo una lógica ecuacional.

Ejemplo 3.4 *El siguiente módulo define una específicación algebraica de valores booleanos. La palabra* **sort** *se utiliza para indicar que se define un género, en este caso Bool. La palabra* **op** *indica que se define un símbolo de operación, mientras que la palabra* **eq** *indica que se está definiendo un axioma mediante la ecuación correspondiente.*

obj *Bool* **is**
sort *Bool*
op *true* $: \rightarrow$ *Bool*
op *false* $: \rightarrow$ *Bool*
op *not* _ : *Bool* $\rightarrow$ *Bool*
op _ $\wedge$ _ : *Bool Bool* $\rightarrow$ *Bool*
op _ $\vee$ _ : *Bool Bool* $\rightarrow$ *Bool*

var b : *Bool*
eq *not true* = *false*
eq *not false* = *true*
eq $b \wedge$ *true* = b
eq $b \wedge$ *false* = *false*
eq $b \vee$ *true* = *true*
eq $b \vee$ *false* = b
endo

Ejemplo 3.5 *Puede definirse una especificación algebraica de los números enteros como:*

obj *Int* **is**
sort *Int*
op $0 : \rightarrow$ *Int*
op *succ* _ : *Int* $\rightarrow$ *Int*
op *pred* _ : *Int* $\rightarrow$ *Int*
op _ + _ : *Int Int* $\rightarrow$ *Int*
op _ − _ : *Int Int* $\rightarrow$ *Int*
op _ ∗ _ : *Int Int* $\rightarrow$ *Int*
vars x, y, z
eq *succ* (*pred* x) = x
eq *pred* (*succ* x) = x
eq $x + 0 = x$
eq $x +$ *succ* y = *succ* $(x + y)$
...
endo

Un *álgebra* consiste en uno o más conjuntos de valores (denominados portadores), junto con una colección de funciones sobre dichos conjuntos. Cuando un álgebra cumple los axiomas de una especificación algebraica, se dice que el álgebra satisface dicha especificación. En dicho caso, un álgebra puede considerarse como una implementación de la especificación.

En general, una especificación puede satisfacerse por muchas álgebras diferentes. Sin embargo, es conveniente utilizar una implementación *estándar* de la especificación. Un caso especial de implementación *estándar* es el *álgebra inicial*[1].

Al álgebra inicial sólo pertenecen las constantes de la especificación y la aplicación reiterada de los operadores de la especificación sobre ellas. En general, se elige el álgebra inicial como implementación estándar, debido a las facilidades que proporciona para la demostración de propiedades por inducción.

Existen desarrollos que promueven la elección de otras álgebras. Por ejemplo, el álgebra final ofrece intuitivamente una idea de encapsulación útil para la especificación de tipos abstractos de datos. Las demostraciones que utilizan álgebras finales suelen realizarse mediante bisimulación.

[1] Las definiciones de signatura, álgebra, álgebra inicial, etc. se simplifican considerablemente mediante el concepto de functor que se presenta en 5.2.

3.7.2 Especificación del ejemplo

3.7.2.1 Lenguaje $\mathcal{L}_0$

La especificación algebraica del lenguaje formado por espresiones aritméticas es similar a la semántica denotacional.

obj *Expr* **is pr** *Int*
  subsort *Int* $<$ *Expr*
  op _ + _ : *Expr Expr* $\rightarrow$ *Expr*
  op _ − _ : *Expr Expr* $\rightarrow$ *Expr*
  op *eval* _ : *Expr* $\rightarrow$ *Int*
  var n : *Expr*
  vars e_1 e_2 : *Expr*
  eq *eval* $n = n$
  eq *eval* $(e_1 + e_2) =$ *eval* $e_1 +$ *eval* e_2
  eq *eval* $(e_1 - e_2) =$ *eval* $e_1 -$ *eval* e_2
endo

La expresión **pr** *Int* indica que se toma como base la especificación de enteros definida en el ejemplo 3.5. La sentencia **subsort** *Int* $<$ *Expr* indica que los números enteros forman parte de las expresiones.

3.7.2.2 Lenguaje $\mathcal{L}_1$

Al añadir productos y divisiones a las expresiones del lenguaje anterior, aparece la posibilidad de errores. Los errores pueden tratarse mediante funciones parciales [190].

Otra solución para el tratamiento algebraico de computaciones parciales es la utilización de repliegues o *retracts*. Para ello se utilizan *order-sorted algebras* [62] que permiten definir estructuras algebraicas en las que las funciones parciales se tratan como funciones totales sobre subdominios. Por ejemplo, la función división es parcial sobre el dominio de valores enteros, pero es total sobre el subdominio de enteros distintos de cero.

La especificación algebraica de los enteros con la operación de división sería

obj *IntD* **is pr** *Int*
  subsort *NZInt* $<$ *Int*
  op _ / _ : *Int NZInt* $\rightarrow$ *Int*
  ...
endo

En el tipo de la operación de división se especifica que toma un entero como numerador y un entero distinto de cero como denominador.

El analizador del lenguaje OBJ añade un repliegue, que es una función $r : Int \rightarrow NZInt$ tal que para todo $n : NZInt$, $r\ n = n$.

Al intentar ejecutar $3/0$ el sistema devuelve $r(0)$ indicando que se ha producido un error.

Una vez incluidos los repliegues, la especificación de $\mathcal{L}_1$ es sencilla

obj *Expr* **is pr** *IntD*
...
op _ ∗ _ : *Expr Expr* → *Expr*
op _ / _ : *Expr Expr* → *Expr*
op *eval* _ : *Expr* → *Int*
eq *eval* $(e_1 * e_2)$ = *eval* e_1 ∗ *eval* e_2
eq *eval* (e_1 / e_2) = *eval* e_1 / *eval* e_2
endo

3.7.2.3 Lenguaje $\mathcal{L}_2$

La inclusión de expresiones relacionales se realiza mediante el siguiente módulo.

obj *BExpr* **is pr** *Expr*
subsort *Bool* < *Expr*.
op _ = _ : *Expr Expr* → *Bool*
op _ < _ : *Expr Expr* → *Bool*
var *b* : *Bool*
vars e_1 e_2 : *Expr*
op *eval* _ : *Expr* → *Bool*
eq *eval b* = *b*
eq *eval* $(e_1 = e_2)$ = *eval* e_1 = *eval* e_2
eq *eval* $(e_1 < e_2)$ = *eval* e_1 < *eval* e_2
endo

Se ha extendido la especificación algebraica de *Expr* añadiendo nuevas operaciones. De esa forma, es posible reutilizar las especificaciones anteriores. No obstante, la técnica de extensión deja abierta la posibilidad de modificar el comportamiento de la operación de evaluación, lo cual puede producir inconsistencias.

3.7.2.4 Lenguaje $\mathcal{L}_3$

En el lenguaje $\mathcal{L}_3$ se añadían variables cuyo valor dependía del contexto de evaluación. Para su modelización se define una especificación algebraica del contexto de evaluación. La axiomatización incluye las operaciones *initial* que devuelve el estado inicial, *lkp* que busca el valor de un identificador en un estado, y *upd* que actualiza el estado.

th *State* **is pr** *Int*, **pr** *Var*
sort *State*
op *initial* : → *State*
op *lkp* : *State Var* → *Int*
op *upd* : *State Var Int* → *State*
var ς : *State*
vars *x y* : *Var*
var *n* : *Int*
eq *lkp* $(upd\ \varsigma\ x\ n)\ x$ = *n*

$$\textbf{cq}\ lkp\ (upd\ \varsigma\ x\ n)\ y\ =\ lkp\ \varsigma\ y\ \textbf{if}\ x\ \neq\ y$$
endth

Las palabras clave **th** y **endth** indican que se está definiendo una especificación abstracta del estado *State*. Pueden existir múltiples estructuras que cumplan dicha especificación.

Se ha incluido el módulo *Var* que representa variables mediante identificadores (se supone que está predefinido).

La especificación de expresiones con variables será:

obj $VExpr$ **is pr** $State$, **pr** $Expr$
$\quad$ **subsort** $Var\ <\ Expr$
$\quad$ **op** $eval\ :\ State\ Var\ \rightarrow\ Int$
$\quad$ **var** $x\ :\ Var$
$\quad$ **var** $\varsigma\ :\ State$
$\quad$ **eq** $eval\ \varsigma\ x\ =\ lkp\ \varsigma\ x$
endo

Sin embargo, las funciones de evaluación anteriores ya no sirven, puesto que no tenían en cuenta el estado. Deben modificarse por tanto todas las definiciones anteriores para que tengan el siguiente formato

obj $Expr$ **is pr** $State$
$\quad$ **op** $eval\ :\ State\ Expr\ \rightarrow\ Int$
$\quad$ ...
$\quad$ **eq** $eval\ \varsigma\ (e_1\ +\ e_2)\ =\ eval\ \varsigma\ e_1\ +\ eval\ \varsigma\ e_2$
$\quad$ ...
endo

3.7.2.5 Lenguaje $\mathcal{L}_4$

La inclusión de comandos en $\mathcal{L}_4$ se modeliza de la siguiente forma

obj $Comm$ **is pr** $Expr$, **pr** $State$
$\quad$ **sort** $Comm$
$\quad$ **op** $\_\ :=\ \_\ :\ Var\ Expr\ \rightarrow\ Comm$
$\quad$ **op** $\_\ ;\ \_\ :\ Comm\ Comm\ \rightarrow\ Comm$
$\quad$ **op** $exec\ :\ State\ Comm\ \rightarrow\ State$
$\quad$ **var** $\varsigma\ :\ State$
$\quad$ **vars** $c_1\ c_2\ :\ Comm$
$\quad$ **var** $x\ :\ Var$, **var** $e\ :\ Expr$
$\quad$ **eq** $exec\ \varsigma\ (x\ :=\ e)\ =\ upd\ \varsigma\ x\ (eval\ \varsigma\ e)$
$\quad$ **eq** $exec\ \varsigma\ (c_1; c_2)\ =\ exec\ (exec\ \varsigma\ c_1)\ c_2$
endo

3.7.2.6 Lenguaje $\mathcal{L}_5$

Las sentencias repetitiva y condicional pueden modelizarse de la siguiente forma:

obj *EComm* **is pr** *Comm*
op *if_then_else_* : *Expr Comm Comm* → *Comm*
op *while_do_* : *Expr Comm* → *Comm*
var ς : *State*
vars c_1 c_2 : *Comm*
var e : *Expr*
cq $exec\ \varsigma\ (if\ e\ then\ c_1\ else\ c_2) = exec\ \varsigma\ c_1$ **if** $eval\ \varsigma\ e$
cq $exec\ \varsigma\ (if\ e\ then\ c_1\ else\ c_2) = exec\ \varsigma\ c_2$ **if** $not\ (eval\ \varsigma\ e)$
cq $exec\ \varsigma\ (while\ e\ do\ c) = exec\ \varsigma\ (c; while\ e\ do\ c)$ **if** $eval\ \varsigma\ b$
cq $exec\ \varsigma\ (while\ e\ do\ c) = \varsigma$ **if** $not(eval\ \varsigma\ b)$
endo

3.7.3 Valoración

La semántica algebraica proporciona una mezcla de la semántica operacional, denotacional y axiomática intentando obtener lo mejor de cada formalismo. Es denotacional porque se especifican las estructuras algebraicas que denotan las diferentes construcciones del programa. Es axiomática, ya que se define el comportamiento de estas estructuras mediante axiomas y, a la vez, es operacional, ya que se obtiene de forma automática un intérprete mediante un sistema de reescritura a partir de las ecuaciones.

- *No ambigüedad.* Este formalismo no contiene ambigüedad en sus especificaciones.
- *Modularidad.* Las definiciones no alcanzan la modularidad semántica, ya que las modificaciones en la estructura computacional alteran las especificaciones realizadas.
- *Reusabilidad.* A pesar de que las especificaciones algebraicas prestan gran atención a la reutilización de especificaciones. La falta de modularidad semántica limita la creación de componentes semánticos reutilizables.
- *Demostración.* El punto fuerte de la semántica algebraica es la verificación de programas y la corrección de implementaciones.
- *Prototipo.* Apoyándose en sistemas de especificación como OBJ, es posible construir prototipos ejecutables de los lenguajes. El entorno ASF+SDF [230, 75] permite compilar especificaciones algebraicas en prototipos de lenguajes.
- *Legibilidad.* La sintaxis del lenguaje OBJ permite cierta flexibilidad con el tratamiento de operadores, alcanzando mayor legibilidad que otros formalismos.

- *Flexibilidad.* El componente axiomático y operacional de este formalismo hace difícil su adaptación a la especificación de otros tipos de lenguajes. No obstante, existen especificaciones de lenguajes orientados a objetos y de programación lógica y funcional [72] utilizando semántica algebraica.

- *Experiencia.* Existen escasas especificaciones de lenguajes de programación reales mediante este formalismo. Se ha aplicado a subconjuntos de lenguajes imperativos [63] y lenguajes Orientados a Objetos mediante la denominada *álgebra oculta* ó *hidden algebra.*

3.8 Máquinas de Estado Abstracto

3.8.1 Descripción

Las máquinas de estado abstracto o *Abstract State Machines* (ASM) definen un algoritmo mediante una abstracción del estado sobre el que se trabaja y una serie de reglas de transición entre elementos de dicho estado. Originalmente, se denominaban *evolving algebras* y fueron desarrolladas por Y. Gurevich [69].

Las máquinas de estado abstracto son una evolución de la semántica operacional que trabajan con un nivel de abstracción cercano al algoritmo que se está definiendo, pudiendo considerarse como una especie de pseudo-código sobre datos abstractos.

3.8.2 Especificación del ejemplo

Para la descripción semántica del lenguaje imperativo, se toma como modelo la especificación de Java [27] simplificándola para el ejemplo.

La especificación parte de un árbol que representa la sintaxis abstracta del programa. La semántica se considera como un recorrido sobre dicho árbol. En cada nodo, se ejecuta la tarea especificada y luego, el flujo de control prosigue con la siguiente tarea.

Se utilizan una serie de funciones dinámmicas para representar el estado de la computación. Una función dinámica es una función que puede modificarse durante la ejecución.

La función dinámica *task* devuelve la frase del programa que hay que ejecutar en un momento dado. Las frases del programa se denotan como valores de tipo *Phrase* y serán elementos de la sintaxis abstracta (expresiones o comandos) a los que se añade el elemento *finished* que indica que ya no hay más tareas a ejecutar.

Para cada construcción del lenguaje, existen dos funciones dinámicas

$$fst, nxt : Phrase \rightarrow Phrase$$

Tales que *fst f* indica la primer subexpresión o primer subcomando a ejecutar en cada momento y *nxt f* indica la expresión o comando a ejecutar al finalizar.

El avance de una tarea a otra se realiza mediante la macro *proceed* definida como

$$proceed = nxt(task)$$

3.8.2.1 Lenguaje $\mathcal{L}_0$

Para la especificación del lenguaje de expresiones aritméticas se utiliza la función dinámica $val : Expr \rightarrow Int$ que almacena el valor intermedio a devolver.

El estado inicial se forma a partir de la expresión a evaluar *expr* que se almacena en la función *task* e indicando que al acabar la evaluación de *expr* se finaliza el programa.

$$task = fst(expr)$$
$$next(expr) = finished$$

Las reglas de transición serán las siguientes

if $task$ **is** num **then**
  $val(task) := num$
  proceed

if $task$ **is** $e_1 + e_2$ **then**
  $val(task) := val(e_1) + val(e_2)$
  proceed

if $task$ **is** $e_1 - e_2$ **then**
  $val(task) := val(e_1) - val(e_2)$
  proceed

3.8.2.2 Lenguaje $\mathcal{L}_1$

Como ya se ha indicado, la inclusión de divisiones requiere realizar un tratamiento de excepciones. Para ello se añade el siguiente universo.

$$Reason ::= DivisionByZero$$

y la función dinámica

$$mode : Reason$$

que registra si el modo es normal (la función está indefinida) o se ha producido un error. Para gestionar el error, se utilizará la función $up : Phrase \rightarrow Phrase$ que devuelve *finished*.

La técnica presentada puede generalizarse para realizar un tratamiento de excepciones más general, para lo cual, la función *mode* registraría el tipo de excepción y la función *up* devolvería la frase encargada de capturar y gestionar la excepción.

Las reglas de transición serán

if $task$ **is** $e_1 * e_2$ **then**
  $val(task) := val(e_1) * val(e_2)$
  proceed

if $task$ **is** $e_1 / e_2 \wedge e_2 \neq 0$ **then**

$val(task) := val(e_1) \,/\, val(e_2)$
$proceed$

if $task$ **is** $e_1 \,/\, e_2 \wedge e_2 = 0$ **then**
$mode := DivisionByZero$
$up(task)$

3.8.2.3 Lenguaje $\mathcal{L}_2$

La inclusión de expresiones relacionales supone redefinir el dominio de valores incluyendo booleanos

$$Value ::= Integer \mid Bool$$

Las reglas de transición son similares a las anteriores

if $task$ **is** $e_1 = e_2$ **then**
$val(task) := val(e_1) = val(e_2)$
$proceed$

if $task$ **is** $e_1 < e_2$ **then**
$val(task) := val(e_1) < val(e_2)$
$proceed$

3.8.2.4 Lenguaje $\mathcal{L}_3$

Para la inclusión de variables, se añade la función dinámica $loc : Var \rightarrow Value$ que devuelve el valor actual de una variable. La regla de transición será:

if $task$ **is** var **then**
$val(task) := loc(var)$
$proceed$

3.8.2.5 Lenguaje $\mathcal{L}_4$

La regla de transición para los comandos serán

if $task$ **is** $skip$ **then**
$proceed$

if $task$ **is** $x := e$ **then**
$loc(x) := val(e)$
$proceed$

En el comando secuencial, se define la función nxt de la siguiente forma

let $c = c_1; c_2$ **in**
$nxt(c_1) = c_2$
$nxt(c_2) = nxt(c)$

y la regla de transición será

if $task$ **is** $c_1; c_2$ **then**
$task := fst(c_1)$

3.8.2.6 Lenguaje $\mathcal{L}_5$

La inclusión del condicional y la repetición requiere indicar cuáles son los valores *fst* y *nxt* en cada caso

let $c = \mathit{if}\ e\ \mathit{then}\ c_1\ \mathit{else}\ c_2$ **in**
$\mathit{fst}(c) = \mathit{fst}(e)$
$\mathit{nxt}(e) = c$
$\mathit{nxt}(c_1) = \mathit{nxt}(c)$
$\mathit{nxt}(c_2) = \mathit{nxt}(c)$

if *task* **is** $\mathit{if}\ e\ \mathit{then}\ c_1\ \mathit{else}\ c_2$ **then**
if $\mathit{val}(e)$ **then** $\mathit{task} := \mathit{fst}(c_1)$
else $\mathit{task} := \mathit{fst}(c_2)$

Para el bucle *while* se realizan las asignaciones para que el siguiente comando a ejecutar una vez acabado el cuerpo del bucle sea la expresión booleana, y que, alcabar dicha expresión booleana comprueba si continúa en el bucle o no.

let $c = \mathit{while}\ e\ \mathit{do}\ c_1$ **in**
$\mathit{fst}(c) = \mathit{fst}(e)$
$\mathit{nxt}(e) = c$
$\mathit{nxt}(c_1) = \mathit{fst}(e)$

if *task* **is** $\mathit{while}\ e\ \mathit{do}\ c$ **then**
if $\mathit{val}(e)$ **then** $\mathit{task} := \mathit{fst}(c)$
else $\mathit{task} := \mathit{nxt}(c)$

3.8.3 Valoración

- *No ambigüedad.* Este formalismo no contiene ambigüedad.
- *Modularidad.* La semántica se realiza de forma modular añadiendo nuevos elementos a la estructura abstracta del estado. Las definiciones previas no es necesario modificarlas al añadir nuevas características al lenguaje. De hecho, en [215] se realiza una definición del lenguaje Java de forma modular, reutilizando las definiciones previas.
- *Reusabilidad.* La especificación permite definir componentes semánticos reutilizables. Esta posibilidad se alcanza mediante modificaciones en el nivel de abstracción.
- *Demostración.* El carácter fuertemente operacional dificulta la realización de demostraciones sobre las especificaciones realizadas. No obstante, los artículos sobre el tema ponen especial énfasis en la demostración de propiedades.
- *Prototipo.* Existen varios sistemas que permiten obtener prototipos a partir de las especificaciones semánticas.

- *Legibilidad.* La legibilidad de las especificaciones parece bastante discutible dado su carácter fuertemente operacional.

- *Flexibilidad.* De la misma forma, la flexibilidad para describir lenguajes no imperativos también es discutible. A pesar de que una aplicación importante de este formalismo ha sido la descripción del lenguaje Prolog (aunque debe recordarse que el lenguaje Prolog no es puramente declarativo).

- *Experiencia.* Este formalismo ha sido aplicado con éxito a varios lenguajes reales como Prolog [26], Modula 2, Java [215], etc. También se ha aplicado a la especificación de protocolos, sistemas de hardware, etc.

3.9 Semántica de Acción

3.9.1 Descripción

La *semántica de acción* fue propuesta por Mosses [177] con el fin de crear especificaciones semánticas más legibles, modulares y utilizables. Para más información sobre la semántica de acción, puede consultarse [178, 241, 114].

La semántica de un lenguaje se especifica mediante *acciones*, que expresan computaciones. Sintácticamente, la notación de acciones tiene bastante libertad simulando un lenguaje natural restringido. Internamente, la notación de acciones se transforma en semántica operacional estructurada.

Una acción es una entidad que puede ser ejecutada, utilizando los datos que le pasan otras acciones. Los datos a los que se haga referencia en las acciones se modelizan mediante *productores* o *yielders*. Cuando se ejecuta, una acción produce un resultado. La acción puede completarse (finalizar normalmente) proporcionando sus datos a otras acciones, fallar o divergir (no terminar). En general, el resultado de una acción depende de los datos que se le pasen.

Existen dos tipos de acciones, primitivas y compuestas. Algunos ejemplos de acciones primitivas son

- *complete* finaliza de forma normal

- *fail* finaliza de forma anormal

- *check d* finaliza con éxito si d produce el valor True, en caso contrario, falla.

Las acciones compuestas se forman mediante combinadores de acciones. Un combinador de acciones toma una o más acciones y genera una acción más compleja, dirigiendo el flujo de datos y control de sus subacciones.

Algunos combinadores de acción son

- A_1 *or* A_2 ejecuta A_1 ó A_2, pero si la subacción elegida falla, entonces ejecuta la otra subacción.

- A_1 *and* A_2 ejecuta A_1 y A_2 en un orden cualquiera y, si ambas convergen, mezcla sus datos transitorios.

- A_1 *and then* A_2 es similar a A_1 *and* A_2 salvo que ejecuta primero A_1 y luego A_2.

- A_1 *then* A_2 hace que los datos transitorios de A_1 sean tomados por A_2.
- *unfolding* A ejecuta la acción A y, si durante la ejecución de A se encuentra con la acción *unfold*, entonces substituye *unfold* por *unfolding* A.

  La acción *unfolding* A permite construir sentencias repetitivas.

En la notación de acciones, los valores enteros, booleanos, listas, y otros datos, así como las acciones propiamente dichas, son clasificados en géneros (*sorts*)

Un género es un conjunto de elementos junto con una serie de operaciones especificadas mediante semántica operacional estructurada. Por ejemplo, *Boolean*, *Integer* son géneros predefinidos con algunas operaciones como:

$$sum, difference, product, quotient : (Integer, Integer) \rightarrow Integer$$
$$is, less : (Integer, Integer) \rightarrow Boolean$$

La notación de acciones también incluye la noción de subgéneros (subsorts). Si S es un subgénero de S', entonces todo elemento de S está también incluido en S'.

Por ejemplo *Datum* representa todos los valores, *Integer* y *Boolean* son subgéneros de *Datum*. La notación $S_1 \| S_2$ indica la unión de dos géneros.

Los elementos del género *Action* son acciones. En general, las acciones pueden tener muchos resultados posibles. Para restringir el rango de resultados se utilizan subgéneros de *Action*. Por ejemplo, $Action[complete]$ representa una acción que termina sin tener ningún otro efecto.

La modelización de los diferentes tipos de acciones se realiza mediante *facetas*. Existen varios tipos de facetas

- La *faceta básica* se refiere al flujo de control ordinario.
- La *faceta funcional* captura la idea de información transitoria, es decir, datos transferidos entre acciones sucesivas. Añade las siguientes acciones
  - *give* d devuelve el dato d
  - *the given* S produce el dato que se le pasa (el dato debe pertenecer al género S, en caso contrario, produce el valor *nothing*.
  - *the given* $S\#n$ produce el dato n-ésimo de la tupla que se le pasa (el dato debe pertenecer al género S, en caso contrario devuelve *nothing*)
- La *faceta declarativa* se refiere a la información dependiente de un entorno que liga identificadores con datos. Añade las siguientes acciones.
  - *bind* x *to* d liga el identificador x al dato d
  - *the* S *bound to* x produce el dato con el que está ligado el identificador x en el entorno (si x no está ligado, o está ligado a un valor cuyo género no es S, entonces devuelve *nothing*).
- *faceta imperativa* se refiere a un estado global modificable. El estado contiene un número arbitrario de celdas. Cada celda tiene un estado actual que puede contener: un valor del género *Datum*; un valor *undefined* (indicando que ha sido reservado pero no contiene valor) o un valor *unused* (no contiene datos ni ha sido reservada)

El estado de una celda puede cambiarse en cualquier momento, pero los datos almacenados en las celdas son estables, es decir, permanecen inalterados hasta que las celdas son liberadas o se almacenan nuevos datos. Se añaden las siguientes acciones

  - *store d in c* almacena el dato d en la celda c.
  - *allocate c* reserva la celda c.
  - *deallocate c* libera la celda c
  - *the S stored in c* devuelve el valor almacenado en c

- Existen otras facetas, como la *faceta comunicativa*, la *faceta reflectiva* y la *faceta directiva* para modelizar aspectos de concurrencia, reflexividad y de indirección que no se tratan en esta presentación.

La especificación del lenguaje se realiza mediante tres apartados: sintaxis abstracta, funciones semánticas y entidades semánticas.

La *sintaxis abstracta* se describe mediante una gramática libre de contexto aumentada con expresiones regulares. Las *funciones semánticas* consisten en conjuntos de ecuaciones. Cada ecuación define la semántica de un tipo de frase particular a partir de la semántica de sus subcomponentes. Estas ecuaciones pueden considerarse como una definición inductiva que transforma la sintaxis abstracta en las entidades semánticas. Las *entidades semánticas* se describen mediante la notación de acciones, la cual puede ser extendida para incorporar nuevas acciones o tipos de datos.

3.9.2 Especificación del ejemplo

En la especificación de los lenguajes $\mathcal{L}_0$ a $\mathcal{L}_5$ se especificarán únicamente las funciones semánticas y se mencionan las entidades semánticas. La especificación utilizada para la sintaxis abstracta es similar a la sintaxis presentada en 3.2.

3.9.2.1 Lenguaje $\mathcal{L}_0$

- evaluate _ :: Expr → action [giving Integer]

(1) evaluate ⟦ n:Numeral ⟧ =
    give n

(2) evaluate ⟦ E_1:Expr "+" E_2:Expr ⟧ =
    | evaluate E_1
    | and
    | evaluate E_2
    then give sum(given Integer#1,given Integer#2)

(3) evaluate ⟦ E_1:Expr "-" E_2:Expr ⟧ =
    | evaluate E_1
    | and
    | evaluate E_2
    then give difference(given Integer#1,given Integer#2)

3.9.2.2 Lenguaje $\mathcal{L}_1$

La incorporación de divisiones en el lenguaje $\mathcal{L}_1$ supone modificar el tipo de la función de evaluación para indicar que la acción puede fallar. La extensión modifica la función *evaluate* añadiendo dos nuevos casos.

- evaluate _ :: Expr → Action [giving Integer | failing]

```
(1) evaluate ⟦ E1:Expr "*" E2:Expr ⟧ =
        | evaluate E1
        | and
        | evaluate E2
      then give product(given Integer#1,given Integer#2)

(2) evaluate ⟦ E1:Expr "/" E2:Expr ⟧ =
        | evaluate E1
        | and
        | evaluate E2
      then
        | | check given Integer#2 is 0 and then fail
        | or
        | | give quotient(given value#1,given value#2)
```

3.9.2.3 Lenguaje $\mathcal{L}_2$

En $\mathcal{L}_2$ se añaden expresiones relacionales. El dominio de valores debe extenderse para devolver números o valores booleanos

```
(1) value = Integer | Boolean
```

- evaluate _ :: Expr → action [giving Value | . . .]

`evaluate` se modifica para indicar que debe devolver un valor en lugar de un número.

```
(1) evaluate ⟦ E1:Expr "=" E2:Expr ⟧ =
        | evaluate E1
        | and
        | evaluate E2
      then give is(given Integer#1,given Integer#2)

(2) evaluate ⟦ E1:Expr "<" E2:Expr ⟧ =
        | evaluate E1
        | and
        | evaluate E2
      then give less(given Integer#1,given Integer#2)
```

3.9.2.4 Lenguaje $\mathcal{L}_3$

En el lenguaje $\mathcal{L}_3$ se añaden identificadores cuyo valor depende del contexto.

```
(1) evaluate ⟦ x : Ident ⟧ =
        | give value stored in cell bound to x
```

3.9.2.5 Lenguaje $\mathcal{L}_4$

- execute _ :: Comm → Action [completing | diverging | storing | failing]

(1) execute ⟦ "skip" ⟧ = complete

(2) execute ⟦ x:Ident ":=" E:Expr ⟧ =
| | give the cell bound to x
| and
| | evaluate E
then
| store the given Value#2 in the given cell#1

(3) execute ⟦ C_1:Comm ";" C_2:Comm ⟧ = execute C_1 and then execute C_2

3.9.2.6 Lenguaje $\mathcal{L}_5$

(1) execute ⟦ "while" E:Expr "do" C:Comm ⟧ =
unfolding
| evaluate E then
| | check the given Boolean and then execute C
| | and then unfold
| or
| | check not the given Boolean and then complete

(2) execute ⟦ "if" E:Expr "then" C_1:Comm "else" C_2:Comm ⟧ =
evaluate E then
| | check the given boolean and then execute C_1
| or
| | check not the given boolean and then execute C_2

3.9.3 Valoración

- *No ambigüedad.* A pesar de que pueda parecer que las descripciones en semántica de acción son realizadas en lenguaje natural. En realidad, se utiliza un lenguaje formal con una sintaxis flexible que permite utilizar identificadores formados por varias palabras, por ejemplo *the given value* pero que son reconocidos por el analizador sintáctico sin que se produzcan ambigüedades.

- *Modularidad.* La modularidad es uno de los objetivos de diseño. La cual se alcanza por el hecho de que las acciones son elementos de primera clase. No obstante, resulta difícil combinar diferentes facetas y no queda claro cómo añadir nuevas facetas.

- *Reusabilidad.* A pesar de alcanzar una cierta modularidad, no es posible definir componentes semánticos reutilizables. Recientemente, [49] plantea un sistema que permite reusabilidad de componentes, aunque el sistema no ha sido implementado. El sistema de combinación de definiciones puede llevar a conflictos que no indica cómo resolver.

- *Demostración.* La demostración de propiedades de los programas es bastante difícil al basarse en semántica operacional. En [239] se plantea la relación entre la semántica de acción y la semántica monádica modular. Para

ello define la notación de acciones mediante transformadores monádicos en lugar de utilizar semántica operacional estructurada.

- *Prototipo.* Existen varias implementaciones de la semántica de acción, como el sistema Actress [30] y el sistema OASIS [184], que permite generar compiladores a partir de la especificaciones.

- *Legibilidad.* Otro de los objetivos de diseño es la legibilidad y se puede considerar que se alcanza a expensas de las dificultades que ello puede suponer para el desarrollo de herramientas que procesen descripciones en semántica de acción.

- *Flexibilidad.* La semántica de acción se adapta a lenguajes para los que existan facetas predefinidas. Sin embargo, si se plantea la especificación de lenguajes de diferentes paradigmas, el formalismo se resiente, ya que sería necesario definir nuevas facetas.

- *Experiencia.* Se han realizado especificaciones de varios lenguajes de alto nivel como Pascal, Standard ML o Java. Dichas especificaciones utilizan subconjuntos de los lenguajes y son realizadas a posteriori.

3.10 Semántica Monádica Modular

La semántica monádica modular surge a partir de la utilización de mónadas para describir una semántica denotacional modular. Las mónadas, tomadas de la teoría de la categoría y propuestas por Moggi [170] para la semántica de lenguajes, permiten separar valores de computaciones. Con dicha separación se favorece la modularidad, pudiendo especificarse características de los lenguajes de forma separada. Puesto que este tipo de especificación es el que se utilizará en el sistema de prototipado de lenguajes, el siguiente capítulo se dedica íntegramente a su descripción.

Capítulo 4

Semántica Monádica modular

.

The designer of a new [programming language] feature should concentrate on one feature at a time C. A. R. Hoare [89]

4.1 Introducción

4.1.1 Evolución

La *semántica monádica modular* fue propuesta por S. Liang, P. Hudak y M. P. Jones [147] con el fin de construir un sistema de desarrollo modular de especificaciones semánticas de lenguajes de programación. Se basa en la utilización de *mónadas* y utiliza el concepto de transformadores de mónadas.

Las *mónadas* permiten encapsular características impuras como estados, Entrada/Salida y no determinismo en un lenguaje puramente funcional. El concepto de mónada o *triple* fue originalmente propuesto en Teoría de Categorías [17]. Posteriormente, E. Moggi lo aplica a la modularización de la descripción semántica de lenguajes [170, 171]y P. Wadler populariza su utilización para la encapsulación de características imperativas en lenguajes puramente funcionales como *Haskell* [233, 234, 235]. Como aplicación práctica, construye un intérprete de un lenguaje de programación simple y va añadiéndole nuevas características de forma modular sin modificar las definiciones existentes. Su trabajo será una fuente de inspiración para la búsqueda de técnicas de construcción modular de intérpretes.

Una mónada captura una determinada noción de computación reflejando la estructura computacional del lenguaje que se está definiendo y separando la noción de computación del valor devuelto por una computación. El problema surge a la hora de componer dos mónadas. En [109] se muestra que en general no es posible obtener una mónada mediante composición de otras dos.

A principios de los años 90, se suceden varios intentos para lograr la combinación entre mónadas. En [122] se introduce el término *mónada combinada*

y se dan varias condiciones bajo las cuales ciertas mónadas pueden combinarse con otras mediante simple composición. G. Steele [217] logra construir un intérprete a partir de unidades independientes, que denomina *pseudomónadas*. Su propuesta no es capaz, sin embargo, de resolver varios problemas con el sistema de tipos de *Haskell*, por lo que crea un preprocesador que transforma las especificaciones en código *Haskell*.

Otros autores continúan la búsqueda de intérpretes modulares. En [36] se utiliza la *Semántica directa extendida* para construcción modular de especificaciones en semántica denotacional, permitiendo añadir nuevas características a un lenguaje sin modificar las anteriores así como la construcción modular de intérpretes. La implementación del sistema se realiza en *Scheme*. Espinosa [54] crea un sistema denominado *Semantic Lego* en el que vuelve a sacar a la luz los transformadores de mónadas ya propuestos por E. Moggi (él los denominaba morfismos de mónadas). Para la transformación entre mónadas, desarrolla las operaciones de *elevación* (*lifting*) así como unas estructuras que denomina *mónadas estratificados* que permiten mayor flexibilidad. Su desarrollo utiliza, sin embargo, el lenguaje *Scheme* que permite mayor flexibilidad al carecer de chequeo de tipos.

En [147], S. Liang, P. Hudak y M. P. Jones retoman la utilización de transformadores de mónadas y mediante la utilización de clases de constructores de tipos (desarrolladas por M. Jones para el sistema Gofer [106, 105]) crean un intérprete modular de cálculo lambda añadiendo estado, continuaciones, no-determinismo, funciones de orden superior, etc [146, 145].

En su trabajo, mencionan la posible relación entre la semántica monádica modular y la semántica de acción. Esta línea fue investigada por K. Wansbrough [239] construyendo un sistema que transforma descripciones en semántica de acción en descripciones de semántica monádica modular.

4.2 Descripción

En general, la descripción de un lenguaje de programación puede describirse mediante una función de la forma $\Sigma \rightarrow \mathsf{M}\,\mathsf{V}$ donde

- M representa una computación que devuelve un valor V. Dicha computación se obtendrá mediante composición de transformadores de mónadas a una mónada básica. Por tanto, $\mathsf{M} = (\mathcal{T}_1.\mathcal{T}_2 \ldots \mathcal{T}_n)\mathsf{M}'$ donde $\mathcal{T}_i$ es un transformador de mónadas que añade una determinada noción computacional y M' es una mónada básica.

- V representará el dominio de posibles valores. Dicho dominio podrá definirse a partir de la unión de conjuntos de valores más sencillos como enteros, booleanos, etc.

- Σ será la sintaxis abstracta del lenguaje.

La descipción semántica modular se realiza en un *metalenguaje*. Los metalenguajes utilizados pueden ser lenguajes de programación de propósito general como Scheme ([54]) o *Haskell* ([147]), o de dominio específico como ([37]). El metalenguaje debe tener capacidad para definir mónadas, transformadores de mónadas y funciones semánticas.

4.3 Bloques semánticos

4.3.1 Mónadas

Aunque la noción de mónada deriva de la Teoría de Categorías, en esta sección se utiliza una definición de mónada más afín a la programación funcional, tomada de [235]. En A.4 se incluye la definición utilizada en Teoría de Categoría y en [80] se comparan las diferentes definiciones.

Definición 4.1 (Mónada) *Una* mónada *consiste en una terna*

$$\langle \mathsf{M}, return_{\mathsf{M}}, \gg\!=_{\mathsf{M}} \rangle$$

donde M *es un constructor de tipos, mientras que* $return_{\mathsf{M}} : \alpha \to \mathsf{M}\,\alpha$ *y* $(\gg\!=_{\mathsf{M}}) : \mathsf{M}\,\alpha \to (\alpha \to \mathsf{M}\,\beta) \to \mathsf{M}\,\beta$ *son dos operaciones*[1] *que cumplen las siguientes propiedades:*

$$\begin{array}{ll} c \gg\!= return & \equiv c \\ (return\ a) \gg\!= c & \equiv c\ a \\ (c_1 \gg\!= \lambda v_1 \to c_2) \gg\!= \lambda v_2 \to c_3 & \equiv c_1 \gg\!= \lambda v_1 \to (c_2 \gg\!= \lambda v_2 \to c_3) \end{array}$$

Intuitivamente, las dos primeras propiedades indican que las computaciones triviales pueden suprimirse, la tercera propiedad captura la idea básica de secuencialidad.

Definición 4.2 (Función monádica) *Dada una mónada* M*, una* función monádica *es una función* $f : \alpha \to \mathsf{M}\beta$

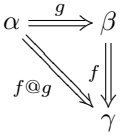
Una función monádica $f : \alpha \to \mathsf{M}\beta$ se representará con el siguiente diagrama

$$\alpha \stackrel{f}{\Longrightarrow} \beta$$

Definición 4.3 (Composición monádica) *Dadas dos funciones monádicas* $f : \beta \to \mathsf{M}\gamma$ *y* $g : \alpha \to \mathsf{M}\beta$*, la composición monádica de* f *y* g*,* $f @ g : \alpha \to \mathsf{M}\gamma$ *se define como:*

$$f \mathbin{@} g \;=\; \lambda x \to g\ x \gg\!= f$$

La composición monádica puede representarse como:



Las ecuaciones 4.1 pueden reescribirse utilizando la composición monádica

$$\begin{array}{ll} f \mathbin{@} return & \equiv f \\ return \mathbin{@} f & \equiv f \end{array}$$

[1] El subíndice M de $return_{\mathsf{M}}$ y $\gg\!=_{\mathsf{M}}$ no suele escribirse si se trabaja con la misma mónada

$$f \mathbin{@} (g \mathbin{@} h) \equiv (f \mathbin{@} g) \mathbin{@} h$$

La notación-*do* de *Haskell* permite simplificar las expresiones que utilizan mónadas.

$$\begin{array}{ll}
\textbf{do}\,\{\, m;\ e \,\} & \equiv m \gg\!= \lambda_ \rightarrow \textbf{do}\,\{\, e \,\} \\
\textbf{do}\,\{\, x \leftarrow m;\ e \,\} & \equiv m \gg\!= \lambda x \rightarrow \textbf{do}\,\{\, e \,\} \\
\textbf{do}\,\{\, \textbf{let}\ exp;\ e \,\} & \equiv \textbf{let}\ exp\ \textbf{in}\ \textbf{do}\,\{\, e \,\} \\
\textbf{do}\,\{\, e \,\} & \equiv e
\end{array}$$

La principal aportación de las mónadas es la separación entre computación y valor producido por dicha computación. Intuitivamente, un elemento de tipo $\mathsf{M}\,\alpha$ denota una computación M que devuelve un valor de tipo α.

Desde un punto de vista de ingeniería del software, una mónada M puede considerarse como un tipo abstracto de datos con dos operaciones, *return* que sirve para introducir valores dentro de una computación y $\gg\!=$ que enlaza dos computaciones.

La forma más común de declarar mónadas en *Haskell* consiste en utilizar una clase de constructores de tipos siguiendo [106, 105].

$$\begin{array}{l}
\textbf{class}\ \mathit{Monad}\ \mathsf{m} \\
\quad \textbf{where} \\
\quad\quad \mathit{return} : \alpha \rightarrow \mathsf{m}\,\alpha \\
\quad\quad (\gg\!=) : \mathsf{m}\,\alpha \rightarrow (\alpha \rightarrow \mathsf{m}\,\beta) \rightarrow \mathsf{m}\,\beta \\
\quad\quad (\gg) : \mathsf{m}\,\alpha \rightarrow \mathsf{m}\,\alpha \rightarrow \mathsf{m}\,\alpha \\
\quad\quad m_1 \gg m_2 = m_1 \gg\!= \lambda_ \rightarrow m_2
\end{array}$$

Mediante polimorfismo de primera clase es posible utilizar otra definición que permite utilizar mónadas como valores simples (véase B.8.2).

La utilización de mónadas permite capturar los diferentes conceptos computacionales. Para ello, se definen tipos de mónadas específicos que añaden nuevas operaciones a las operaciones primitivas. En las siguientes secciones se definen y justifican varios tipos específicos de mónadas.

4.3.1.1 Mónada Identidad

La mónada más sencilla es aquélla que no realiza ningún tipo de computación. Se denotará por IdM y puede definirse como

$$\mathsf{IdM}\,\alpha \triangleq \alpha$$

$$\begin{array}{l}
\textbf{instance}\ \mathit{Monad}\ \mathsf{IdM} \\
\quad \textbf{where} \\
\quad\quad \mathit{return}\ x = x \\
\quad\quad m \gg\!= f = f\ m
\end{array}$$

4.3.1.2 Computaciones parciales

Una computación parcial es una computación que puede producir un error. Con el fin de informar del error correspondiente, se crea un tipo especial de mónada. ErrM es una mónada con una operación

$$\begin{array}{l}\textbf{class}\ (\mathit{Monad}\ \mathsf{m})\ \Rightarrow\ \mathsf{ErrM}\ \mathsf{m}\\ \quad\textbf{where}\\ \quad\quad \mathit{err} : \mathit{String} \rightarrow \mathsf{m}\ \alpha\end{array}$$

La operación *err* satisface la siguiente propiedad

$$\mathit{err}\ x \gg\!= f\ \equiv\ \mathit{err}\ x$$

4.3.1.3 Acceso a un entorno

En ocasiones, el valor de una expresión depende del entorno en el que se evalúa. Para modelizar esta situación, $\mathsf{EnvM}_{\mathsf{Env}}$ modeliza una mónada que accede a un entorno.

$$\begin{array}{l}\textbf{class}\ (\mathit{Monad}\ \mathsf{m})\ \Rightarrow\ \mathsf{EnvM}_{\mathsf{Env}}\ \mathsf{m}\\ \quad\textbf{where}\\ \quad\quad \mathit{rdEnv}\ :\ \mathsf{m}\ \mathit{Env}\\ \quad\quad \mathit{inEnv}\ :\ \mathit{Env} \rightarrow \mathsf{m}\ \alpha \rightarrow \mathsf{m}\ \alpha\end{array}$$

Las operaciones de $\mathsf{EnvM}_{\mathsf{Env}}$ son

- *rdEnv* es una computación que devuelve el entorno actual.
- *inEnv* ρ c ejecuta la computación c en el entorno ρ.

La mónada de acceso al entorno debe cumplir los siguientes axiomas

$$\begin{array}{ll}(\mathit{inEnv}\ \rho)\ (\mathit{return}\ x) & \equiv \mathit{return}\ x\\ \mathit{inEnv}\ \rho\ (m \gg\!= \lambda x \rightarrow f\ x) & \equiv \mathit{inEnv}\ \rho\ m \gg\!= (\lambda x \rightarrow \mathit{inEnv}\ \rho\ (f\ a))\\ \mathit{inEnv}\ \rho\ \mathit{rdEnv} & \equiv \mathit{return}\ \rho\\ \mathit{inEnv}\ \rho'\ (\mathit{inEnv}\ \rho\ m) & \equiv \mathit{inEnv}\ \rho\ m\end{array}$$

Los axiomas anteriores reflejan los aspectos intuitivos de acceso a un entorno. El primer axioma indica que una computación trivial no depende del entorno, el segundo indica que el entorno no es modificado en una secuencia de computaciones, el tercero define el comportamiento básico de la operación de acceso al entorno y el último indica que una computación depende únicamente del entorno actual en el que se ejecuta.

4.3.1.4 Modificación de un estado

El modelo computacional imperativo se basa en la utilización de un estado global en el que se almacenan los valores de las variables. Este estado suele coincidir con la memoria del computador, aunque no es conveniente restringir posibilidades. De hecho, en muchas ocasiones, el estado contiene valores de registros del procesador, y cualquier otra información que se modifica a lo largo de la ejecución de un programa.

$\mathsf{StM}_{\mathsf{State}}$ es una mónada transformadora de un estado ς que incluye la operación:

$$\textbf{class}\ (\mathit{Monad}\ \mathsf{m})\ \Rightarrow\ \mathsf{StM}_{\mathsf{State}}\ \mathsf{m}$$

$$
\begin{array}{l}
\textbf{where} \\
\quad \begin{array}{rcl}
update & : & (State \rightarrow State) \rightarrow \mathsf{m}\ State \\
fetch & : & \mathsf{m}\ State \\
fetch & = & update\ (\lambda x \rightarrow x) \\
set & : & State \rightarrow \mathsf{m}\ State \\
set\ \varsigma & = & update\ (\lambda\_ \rightarrow \varsigma)
\end{array}
\end{array}
$$

El significado intuitivo de las operaciones de $\mathsf{StM}_{\mathsf{State}}$ es:

- *update f* aplica la función de actualización f al estado actual y devuelve el estado actualizado.
- *fetch* obtiene el estado actual.
- *set* ς cambia el estado actual por el nuevo estado ς.

Obsérvese que las operaciones *fetch* y *set* se definen a partir de *update*.
La mónada de acceso a un estado debe cumplir los siguientes axiomas

$$
\begin{array}{ll}
update\ f \gg\!= update\ f' & \equiv update\ (f\ .\ f') \\
set\ \varsigma \gg\!= \lambda\_ \rightarrow fetch & \equiv return\ \varsigma \\
fetch \gg\!= \lambda\_ \rightarrow m & \equiv m
\end{array}
$$

4.3.1.5 Backtracking

La posibilidad de realizar *backtracking* en una computación se explota fundamentalmente en los lenguajes de programación lógica. BackM representa un tipo de mónada con dos operaciones para la realización de *backtracking*.

$$
\begin{array}{l}
\textbf{class}\ (Monad\ \mathsf{m}) \Rightarrow \mathsf{BackM\ m} \\
\quad \textbf{where} \\
\quad\quad \begin{array}{rcl}
failure & : & \mathsf{m}\ \alpha \\
(\Cup) & : & \mathsf{m}\ \alpha \rightarrow \mathsf{m}\ \alpha \rightarrow \mathsf{m}\ \alpha
\end{array}
\end{array}
$$

failure representa una computación fallida, mientras que $m_1 \Cup m_2$ ejecuta m_1, si m_1 finaliza devolviendo un valor, devuelve dicho valor. En caso de que al ejecutar m_1 se produzca un fallo, ejecuta m_2.
Las operaciones deben cumplir las siguientes propiedades observacionales:

$$
\begin{array}{ll}
failure \Cup m & \equiv m \\
m \Cup failure & \equiv m \\
m \Cup (n \Cup o) & \equiv (m \Cup n) \Cup o \\
failure \gg\!= k & \equiv failure \\
(m \Cup n) \gg\!= k & \equiv (m \gg\!= k) \Cup (n \gg\!= k)
\end{array}
$$

4.3.1.6 Tratamiento de Excepciones

Con el fin de proporcionar un sistema de tratamiento de excepciones, se define la mónada $\mathsf{ExcM}_{\mathsf{Exc}}$ con las siguientes operaciones

$$\textbf{class}\ (Monad\ \mathsf{m}) \Rightarrow \mathsf{ExcM}_{\mathsf{Exc}}\ \mathsf{m}$$

```
where
  raise      : Exc → m α
  try        : m α → m (ϵ‖α)

  handle     : m α → (Exc → m α) → m α
  handle m f = do
                 v ← try m
                 case v of
                   L ϵ → f ϵ
                   R a → return a
```

- *raise* ϵ lanza una excepción ϵ

- *try* m intenta ejecutar una computación y devuelve el valor que devuelva dicha computación o una excepción en caso de que se produzca.

- *handle* $m\,f$ ejecuta la computación m devolviendo su resultado si ésta no lanza ninguna excepción. Si la computación m lanza una excepción, entonces ejecuta la función f pasándole como argumento dicha excepción.

Las operaciones *raise* y *handle* son similares a las operaciones *throw* y *catch* predefinidas en *Haskell* para el control de excepciones de Entrada/Salida.

Las operaciones deben cumplir las siguientes propiedades.

```
raise ϵ >>= f       ≡ raise ϵ
try (raise ϵ)       ≡ return (L ϵ)
try (try m >>= f)   ≡ try m >>= (λ x → try (f a))
```

4.3.1.7 Continuaciones

Las continuaciones pueden utilizarse para manejar el control del programa. Una continuación indica cómo debe continuar la ejecución. La mónada ContM contiene la operación *callcc* que permite acceder a la continuación actual.

```
class (Monad m) ⇒ ContM m
  where
    callcc : ((m α → m β) → m α) → m α
```

callcc f captura la continuación actual κ y llama a f pasándole como argumento dicha continuación.

En realidad, el tipo de *callcc* podría ser

```
callcc : ((α → m β) → m α) → m α
```

Sin embargo, el tipo propuesto facilita la construcción de intérpretes que admiten funciones sobre computaciones como valores de primera clase.

4.3.1.8 Entrada/Salida

Una de las razones principales de la popularidad de las mónadas ha sido su aplicación en la modelización de la Entrada/Salida en *Haskell*. Cuando se desean describir acciones semánticas de comunicación con el exterior es necesario recurrir a un tipo especial de mónada de Entrada/Salida. La mónada IOM contiene las operaciones necesarias para realizar Entrada/Salida

```
class (Monad m) ⇒ IOM m
  where
    put : Char → m ()
    get : m Char
    putStr : String → m ()
    getStr : m String
```

- *put c* imprime el carácter *c* en la salida estándar
- *get* es una computación de Entrada/Salida que devuelve el siguiente carácter de la entrada estándar
- *putStr str* imprime el mensaje *str* en la salida estándar
- *getStr* es una computación que devuelve la siguiente línea de la entrada estándar

Siguiendo [85], el tipo *IO* predefinido de *Haskell* puede declararse una instancia de esta mónada evitando la definición de un transformador de mónadas específico para Entrada/Salida y permitiendo reutilizar el tipo predefinido con todas las ventajas prácticas que ello supone.

4.3.1.9 Depuración

Una característica interesante a la hora de construir un intérprete de un lenguaje de programación es la posibilidad de mostrar información sobre el comportamiento de un programa durante su ejecución. Conceptualmente, esta información debe mostrarse por un canal diferente al de la salida estándar (aunque en muchas ocasiones, ambos canales coinciden).

La mónada DbgM contiene una operación:

```
class (Monad m) ⇒ DbgM m
  where
    infoDbg : String → m ()
```

infoDbg msg muestra la información de depuración *msg*.

Dependiendo de la instanciación particular de DbgM será posible interactuar con el usuario, interrumpiendo la ejecución y mostrando otra información de depuración adicional.

4.3.1.10 Reanudaciones

Para la descripción semántica de concurrencia, en 1976 G. Plotkin utiliza *powerdomains*, a partir de los cuales, se definen las reanudaciones.

Intuitivamente, una reanudación (*resumption*) permite interrumpir una ejecución que puede ser reanudada posteriormente. De esta forma, es posible describir computaciones que pueden suspenderse temporalmente para ejecutar otras computaciones y luego reanudarse, llegando a la descripción de sistemas concurrentes [209].

En [239] se define una mónada con reanudaciones.

$$\begin{array}{l}\textbf{class}\ (\mathit{Monad}\ \mathsf{m})\ \Rightarrow\ \mathsf{ResM}\ \mathsf{m}\\ \quad\textbf{where}\\ \quad\quad \mathit{pause}\ :\ \mathsf{m}\,\alpha\ \rightarrow\ \mathsf{m}\,\alpha\\ \quad\quad \mathit{indiv}\ :\ \mathsf{m}\,\alpha\ \rightarrow\ \mathsf{m}\,\alpha\end{array}$$

El significado intuitivo de las operaciones de ResM es:

- *pause m* añade un punto de parada a la computación m.
- *indiv m* elimina los puntos de parada de la computación, haciendo que se ejecuta de forma indivisible.

4.3.1.11 No determinismo

En [147] se define una mónada para modelizar computaciones no deterministas.

$$\begin{array}{l}\textbf{class}\ (\mathit{Monad}\ \mathsf{m})\ \Rightarrow\ \mathsf{NoDetM}\ \mathsf{m}\\ \quad\textbf{where}\\ \quad\quad \mathit{amb}\ :\ \mathsf{m}\,\alpha\ \rightarrow\ \mathsf{m}\,\alpha\ \rightarrow\ \mathsf{m}\,\alpha\end{array}$$

- *amb* m_1 m_2 ejecuta la computación m_1 o la computación m_2 de forma no determinista

4.3.1.12 Paralelismo

En [239] se define la mónada que ejecuta dos computaciones en paralelo y en [40] se describe una mónada con concurrencia.

$$\begin{array}{l}\textbf{class}\ (\mathit{Monad}\ \mathsf{m})\ \Rightarrow\ \mathsf{ParM}\ \mathsf{m}\\ \quad\textbf{where}\\ \quad\quad \mathit{par}\ :\ \mathsf{m}\,\alpha\ \rightarrow\ \mathsf{m}\,\alpha\ \rightarrow\ \mathsf{m}\,\alpha\end{array}$$

- *par* m_1 m_2 ejecuta en paralelo m_1 y m_2, devolviendo el resultado de la primera computación que finaliza.

4.3.2 Transformadores de mónadas

A partir del concepto de morfismo de mónadas introducido por E. Moggi [170], se realiza la definición de un transformador de mónadas que permitirá transformar una mónada en otra mónada añadiéndole características computacionales a la mónada original.

Definición 4.4 (Morfismo de mónadas) *Un* morfismo de mónadas f *entre dos mónadas* M *y* M' *es una función de tipo* $f : \mathsf{M}\,\alpha \to \mathsf{M}'\,\alpha$ *que cumple las siguientes propiedades:*

$$f\ .\ return_{\mathsf{M}} = return_{\mathsf{M}'}$$
$$f\ (m \gg\!=_{\mathsf{M}} k) = (f\ m) \gg\!=_{\mathsf{M}'} (f\ .\ k)$$

Definición 4.5 (Transformador de mónadas) *Un* transformador de mónadas *es un constructor de tipos* $\mathcal{T}$ *con una función de elevación asociada* $lift_{\mathcal{T}}$*, donde* $\mathcal{T}$ *convierte una mónada*

$$\langle \mathsf{M}, return_{\mathsf{M}}, \gg\!=_{\mathsf{M}} \rangle$$

en una nueva mónada

$$\langle \mathcal{T}\mathsf{M}, return_{\mathcal{T}\mathsf{M}}, \gg\!=_{\mathcal{T}\mathsf{M}} \rangle$$

Además, $lift_{\mathcal{T}}$ *es un morfismo de mónadas entre* M *y* $\mathcal{T}\ \mathsf{M}$

El hecho de que $lift_{\mathcal{T}}$ sea un morfismo de mónadas implica que la elevación (*lifting*) de computaciones triviales sigue siendo una computación trivial; y que la elevación de una secuencia de computaciones es equivalente a elevar cada computación por separado y luego combinarlas en la nueva mónada.

Los transformadores de mónadas pueden modelizarse mediante la siguiente clase de tipos

$$\begin{array}{l}\textbf{class}\ (Monad\ \mathsf{m}) \Rightarrow MonadT\ \mathcal{T}\ \mathsf{m} \\ \quad \textbf{where} \\ \quad\quad lift : \mathsf{m}\,\alpha \to \mathcal{T}\ \mathsf{m}\,\alpha\end{array}$$

Definición 4.6 (Composición de transformadores de mónadas) *Dados dos transformadores de mónadas* $\mathcal{T}_1$ *y* $\mathcal{T}_2$*, su composición será un constructor de tipos tal que*

$$\begin{array}{ll}(\mathcal{T}_1\ .\ \mathcal{T}_2)\ \mathsf{M}\,\alpha & \triangleq \mathcal{T}_1\ (\mathcal{T}_2\ \mathsf{M})\ \alpha \\ lift_{(\mathcal{T}_1.\mathcal{T}_2)} & = lift_{\mathcal{T}_1}\ .\ lift_{\mathcal{T}_2}\end{array}$$

Teorema 4.1 *El resultado de componer dos transformadores de mónadas es un transformador de mónadas*

Normalmente, se requiere observar el resultado de una ejecución de la mónada transformada desde la mónada original. Para ello, se define la clase

$$\begin{array}{l}\textbf{class}\ (MonadT\ \mathcal{T}\ \mathsf{m}) \Rightarrow MonadR\ \mathcal{T}\ \mathsf{m} \\ \quad \textbf{where} \\ \quad\quad run\ :\ \mathcal{T}\ \mathsf{m}\,\alpha \to \mathsf{m}\,\alpha\end{array}$$

La definición de transformadores de mónadas no es una tarea sencilla ya que es necesario verificar que la nueva mónada cumple las propiedades de las nuevas operaciones y sigue manteniendo las propiedades de las operaciones de la mónada anterior. Recientemente, R. Hinze [87] muestra cómo derivar un transformador de mónadas que añade *backtracking*, a partir de su especificación.

A continuación se describen varios transformadores de mónadas que se utilizarán en la especificación de lenguajes de los capítulos siguientes.

4.3.2.1 Transformador para ErrM

$\mathcal{T}_{Err}$ es un transformador que toma una mónada M y la convierte en una nueva mónada ErrM de acceso a un entorno (véase 4.3.1.3). Las definiciones correspondientes son:

$$Error\ \alpha\ \triangleq\ \alpha \parallel String$$

$$\mathcal{T}_{Err}\ \mathsf{M}\ \alpha\ \triangleq\ \mathsf{M}\ (Error\ \alpha)$$

$$\begin{array}{l}\textbf{instance}\ (Monad\ m)\ \Rightarrow\ Monad\ (\mathcal{T}_{Err}\ m)\\ \quad\textbf{where}\\ \quad\quad return\ x\ = return\ (L\ x)\\ \quad\quad x \gg\!= f\ \ = x \gg\!= \lambda y \rightarrow\ \textbf{case}\ y\ \textbf{of}\\ \quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad L\ v\ \rightarrow\ v\\ \quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad R\ e\ \rightarrow\ e\end{array}$$

$$\begin{array}{l}\textbf{instance}\ (Monad\ m,\ Monad\ (\mathcal{T}_{Err}\ m))\ \Rightarrow\ MonadT\ \mathcal{T}_{Err}\ m\\ \quad\textbf{where}\\ \quad\quad lift\ x\ = m \gg\!= \lambda x \rightarrow return\ (L\ x)\end{array}$$

$$\begin{array}{l}\textbf{instance}\ (Monad\ m)\ \Rightarrow\ ErrM\ (ErrT\ m)\\ \quad\textbf{where}\\ \quad\quad err\ msg\ =\ return\ (R\ msg)\end{array}$$

4.3.2.2 Transformador para $\mathsf{EnvM}_{\mathsf{Env}}$

$\mathcal{T}_{Env}$ es un transformador que toma un entorno *Env* y una mónada M y la convierte en una nueva mónada de acceso a un entorno $\mathsf{EnvM}_{\mathsf{Env}}$ (véase 4.3.1.3).

Las definiciones correspondientes son:

$$\mathcal{T}_{Env}\ \mathsf{M}\ \alpha\ \triangleq\ Env\ \rightarrow\ \mathsf{M}\ \alpha$$

$$\begin{array}{l}\textbf{instance}\ (Monad\ \mathsf{m})\ \Rightarrow\ Monad\ (\mathcal{T}_{Env}\ \mathsf{M})\\ \quad\textbf{where}\\ \quad\quad return\ x\ = \lambda\rho \rightarrow return\ x\\ \quad\quad x \gg\!= f\ \ = \lambda\rho \rightarrow (x\ \rho) \gg\!= (\lambda a \rightarrow f\ \ a\ \ \rho)\end{array}$$

$$\begin{array}{l}\textbf{instance}\ (Monad\ \mathsf{m},\ Monad\ (\mathcal{T}_{Env}\ \mathsf{m}))\ \Rightarrow\ MonadT\ \mathcal{T}_{Env}\ \mathsf{m}\\ \quad\textbf{where}\\ \quad\quad lift\ x\ = \lambda\rho \rightarrow x \gg\!= return\end{array}$$

$$\begin{array}{l}\textbf{instance}\ (Monad\ \mathsf{m})\ \Rightarrow\ \mathsf{EnvM}_{\mathsf{Env}}\ (\mathcal{T}_{Env}\ \mathsf{m})\\ \quad\textbf{where}\\ \quad\quad rdEnv\ \ \ \ \ \ = \lambda\rho \rightarrow return\ \rho\\ \quad\quad inEnv\ \rho\ x\ = \lambda\_ \rightarrow x\ \ \rho\end{array}$$

4.3.2.3 Transformador para $\mathsf{StM}_{\mathsf{State}}$

$\mathcal{T}_{State}$ es un transformador que toma un estado *State* y una mónada M y la convierte en una nueva mónada modificadora de estado $\mathsf{StM}_{\mathsf{State}}$ (véase 4.3.1.4).

Las definiciones correspondientes son:

$$\mathcal{T}_{State}\ \mathsf{m}\ \alpha\ \triangleq\ State\ \rightarrow\ \mathsf{m}\ (\alpha, State)$$

$$\begin{array}{l}\textbf{instance}\ (Monad\ \mathsf{m})\ \Rightarrow\ Monad\ (\mathcal{T}_{State}\ \mathsf{m})\\ \quad\textbf{where}\\ \quad\quad return\ x\ = \lambda\varsigma \rightarrow return\ (x, \varsigma)\\ \quad\quad x \gg\!= f\ \ = \lambda\varsigma \rightarrow (x\ \varsigma) \gg\!= (\lambda(v, \varsigma') \rightarrow f\ \ v\ \ \varsigma')\end{array}$$

$$\begin{array}{l}\textbf{instance}\ (Monad\ \mathsf{m},\ Monad\ (\mathcal{T}_{State}\ \mathsf{m}))\ \Rightarrow\ MonadT\ \mathcal{T}_{State}\ \mathsf{m}\\ \quad\textbf{where}\\ \quad\quad lift\ x\ = \lambda\varsigma \rightarrow x \gg\!= (\lambda x \rightarrow return(x, \varsigma))\end{array}$$

$$\begin{array}{l}\textbf{instance}\ (Monad\ \mathsf{m})\ \Rightarrow\ \mathsf{StM}_{\mathsf{State}}\ (\mathcal{T}_{State}\ \mathsf{m})\\ \quad\textbf{where}\\ \quad\quad update\ f\ = \lambda\varsigma \rightarrow return\ (\varsigma,\ f\ \varsigma)\end{array}$$

Toda mónada transformadora de estado puede considerarse, a la vez, lectora del entorno.

$$\begin{array}{l}\textbf{instance}\ (\mathsf{StM}_{\mathsf{State}}\ \mathsf{m})\ \Rightarrow\ \mathsf{EnvM}_{\mathsf{State}}\ \mathsf{m}\\ \quad\textbf{where}\\ \quad\quad inEnv\ \rho\ m\ =\ \textbf{do}\\ \quad\quad\quad\quad\quad\quad\quad\quad \rho'\ \leftarrow\ update\ (\lambda\_\ \rightarrow\ \rho)\\ \quad\quad\quad\quad\quad\quad\quad\quad val\ \leftarrow\ m\\ \quad\quad\quad\quad\quad\quad\quad\quad update\ (\lambda\_\ \rightarrow\ \rho')\\ \quad\quad\quad\quad\quad\quad\quad\quad return\ val\\ \quad\quad rdEnv\ \ \ \ \ \ =\ update\ id\end{array}$$

4.3.2.4 Transformador para $\mathsf{ExcM}_{\mathsf{Exc}}$

$\mathcal{T}_{Exc}$ es un transformador que toma un tipo de excepciones *Exc* y una mónada M y la convierte en una nueva mónada que admite tratamiento de excepciones $\mathsf{ExcM}_{\mathsf{Exc}}$ (véase 4.3.1.6).

Las definiciones correspondientes son:

$$\mathcal{T}_{Exc}\ \mathsf{m}\ \epsilon\ \alpha\ \triangleq\ \mathsf{m}(\epsilon \| \alpha)$$

$$\begin{array}{l}\textbf{instance}\ (Monad\ \mathsf{m})\ \Rightarrow\ Monad\ (\mathcal{T}_{Exc}\ \mathsf{m})\\ \quad\textbf{where}\\ \quad\quad return\ \ = return\ .\ R\\ \quad\quad x \gg\!= f\ =\ \textbf{do}\\ \quad\quad\quad\quad\quad\quad\quad\quad v\ \leftarrow\ m\\ \quad\quad\quad\quad\quad\quad\quad\quad \textbf{case}\ v\ \textbf{of}\\ \quad\quad\quad\quad\quad\quad\quad\quad\quad R\ x \rightarrow f\ x\\ \quad\quad\quad\quad\quad\quad\quad\quad\quad L\ \epsilon \rightarrow return\ (L\ \epsilon)\end{array}$$

$$\begin{array}{l}\textbf{instance}\ (\mathit{Monad}\ \mathsf{m},\ \mathit{Monad}\ (\mathcal{T}_{Exc}\ \mathsf{m}))\ \Rightarrow\ \mathit{MonadT}\ \mathcal{T}_{Exc}\ \mathsf{m}\\ \quad\textbf{where}\\ \quad\quad \mathit{lift}\ m\ = m \gg\!= (\mathit{return}\ .\ R)\end{array}$$

$$\begin{array}{l}\textbf{instance}\ (\mathit{Monad}\ \mathsf{m})\ \Rightarrow\ \mathsf{ExcM}_{\mathsf{Exc}}\ (\mathcal{T}_{Exc}\ \epsilon\ \mathsf{m})\\ \quad\textbf{where}\\ \quad\quad \mathit{raise}\ \epsilon\ \ \ \ \ \ = \mathit{return}\ .\ L\\ \quad\quad \mathit{handle}\ m\ h\ =\ \textbf{do}\\ \quad\quad\quad\quad\quad\quad v\ \leftarrow\ m\\ \quad\quad\quad\quad\quad\quad \textbf{case}\ v\ \textbf{of}\\ \quad\quad\quad\quad\quad\quad\quad L\ \epsilon\ \rightarrow\ h\ e\\ \quad\quad\quad\quad\quad\quad\quad R\ x\ \rightarrow\ \mathit{return}\ (R\ x)\end{array}$$

4.3.2.5 Transformador para ContM

$\mathcal{T}_{Cont}$ es un transformador que toma un tipo de respuestas ω y una mónada M y la convierte en una nueva mónada que admite continuaciones ContM (véase 4.3.1.7).

Las definiciones correspondientes son:

$$\mathcal{T}_{Cont}\ \mathsf{m}\ \omega\ \alpha\ \triangleq\ (\alpha \rightarrow \mathsf{m}\omega) \rightarrow \mathsf{m}\omega$$

$$\begin{array}{l}\textbf{instance}\ (\mathit{Monad}\ \mathsf{m})\ \Rightarrow\ \mathit{Monad}\ (\mathcal{T}_{Cont}\ \mathsf{m})\\ \quad\textbf{where}\\ \quad\quad \mathit{return}\ x\ = \lambda\kappa \rightarrow \kappa\ x\\ \quad\quad x \gg\!= f\ \ \ = \lambda\kappa \rightarrow x\ (\lambda v \rightarrow f\ \ v\ \ \kappa)\end{array}$$

$$\begin{array}{l}\textbf{instance}\ (\mathit{Monad}\ \mathsf{m},\ \mathit{Monad}\ (\mathcal{T}_{Cont}\ \mathsf{m}))\ \Rightarrow\ \mathit{MonadT}\ \mathcal{T}_{Cont}\ \mathsf{m}\\ \quad\textbf{where}\\ \quad\quad \mathit{lift}\ x\ = \lambda\kappa \rightarrow x \gg\!= \kappa\end{array}$$

$$\begin{array}{l}\textbf{instance}\ (\mathit{Monad}\ \mathsf{m})\ \Rightarrow\ \mathsf{ContM}\ (\mathcal{T}_{Cont}\ \omega\ \mathsf{m})\\ \quad\textbf{where}\\ \quad\quad \mathit{callcc}\ f\ = \lambda\kappa \rightarrow (f\ (\lambda a \rightarrow (\lambda\_ \rightarrow\ a\ \kappa))\ \kappa)\end{array}$$

La definición anterior indica que, al ejecutar $\mathit{callcc}\ f$ en una continuación κ, llama a la función f pasándole como argumento una función que, si se aplica, entonces deshecha la continuación existente en ese momento (denotada por _) y llama a la continuación κ.

4.3.2.6 Transformador para BackM

$\mathcal{T}_{Back}$ añade a una mónada capacidades de backtracking. La definición utilizada ha sido tomada de [87] donde se realiza una derivación de la misma a partir de la definición axiomática.

Las definiciones correspondientes son:

$$\mathcal{T}_{Back}\ \mathsf{m}\ \alpha\ \triangleq\ \forall\beta.((\alpha \rightarrow \mathsf{m}\ \beta \rightarrow \mathsf{m}\ \beta) \rightarrow \mathsf{m}\ \beta \rightarrow \mathsf{m}\ \beta)$$

```
instance (Monad m) ⇒ Monad (T_Back m)
  where
    return x  = λκ → κ x
    x >>= f   = λκ → x (λv → f v κ)

instance (Monad m, Monad (T_Back m)) ⇒ MonadT T_Cont m
  where
    lift m = λ κ f → do
                       x ← m
                       κ x f

instance (Monad m) ⇒ BackM (T_Back m)
  where
    failure  = λκ → (λ x → x)
    m ⋓ n    = λ κ f → m κ (n κ f)
```

4.4 Dominios extensibles

El conjunto de valores V se formará mediante uniones extensibles de subconjuntos de valores.

Idealmente, las uniones extensibles podrían definirse en un sistema de tipos con *Variantes extensibles* (*extensible variants*) . En [60] se define una ampliación del sistema de tipos de *Haskell* que incorpora tanto variantes como registros extensibles. Sin embargo, dicho sistema no ha sido implementado.

Para subsanar dicha limitación, en [147] se implementa un mecanismo de subtipación mediante clases de tipos con múltiples parámetros e instancias solapables.

La idea consiste en definir una clase de tipos *Subtype a b* que relaciona un subtipo a con un supertipo b :

```
class SubType α β
  where
    ↑ :: α → β
    ↓ :: β → α
```

Donde $\uparrow x$ convierte el valor de un subtipo en el supertipo, mientras que $\downarrow x$ convierte el valor del supertipo en el subtipo. En la práctica, esta función es parcial y sería más lógico que tuviese el tipo $\downarrow: \beta \rightarrow Maybe\ \alpha$. Se ha simplificado su tipo para facilitar la legibilidad de las especificaciones.

Para definir la unión entre dos tipos de valores se utiliza el tipo definido en la página 79.

Declarando las instancias correspondientes:

```
instance SubType α (α ‖ β)
  where
    ↑ x       = L x
    ↓ (L y)   = y

instance (SubType α β) ⇒ SubType α (χ ‖ β)
```

$$\begin{array}{l}\textbf{where}\\ \quad \uparrow \qquad = R\,.\,\uparrow\\ \quad \downarrow (R\,x) \;=\; \downarrow x\end{array}$$

Ejemplo 4.1 *Si se define el dominio de valores*

$$\mathit{Value} \triangleq \mathit{Int} \parallel \mathit{Bool} \parallel ()$$

entonces, $\uparrow 3 : \mathit{Value}$, $\uparrow \mathit{True} : \mathit{Value}$

En la definición anterior de *Value*, el tipo () sirve únicamente para señalar el final del dominio de valores. En el resto de esta presentación, la definición anterior se simplificará escribiendo únicamente

$$\mathit{Value} \triangleq \mathit{Int} \parallel \mathit{Bool}$$

4.5 Especificación del ejemplo

A continuación se especificarán los lenguajes de la sección 3.2 utilizando semántica monádica modular.

4.5.0.7 Lenguaje $\mathcal{L}_0$

El lenguaje $\mathcal{L}_0$ está formado por expresiones aritméticas simples que denotan un valor entero. El dominio de valores será, por tanto, el conjunto *Int*.

$$\mathit{Value} \triangleq \mathit{Int}$$

La estructura computacional puede definirse como una mónada básica, por ejemplo la mónada identidad.

$$\mathsf{Comp} \triangleq \mathsf{IdM}$$

La función de evaluación tendrá, el tipo $\mathcal{E} : \mathit{Expr} \rightarrow \mathsf{Id}\ \mathit{Int}$, es decir, toma una expresión y devuelve una computación básica que al ejecutarse devolverá un entero.

Las definiciones semánticas serán:

$$\begin{array}{lll}\mathcal{E}[\![\mathit{Const}\,n]\!] & = \mathit{return}\ n\\ \mathcal{E}[\![e_1 + e_2]\!] & = \textbf{do}\\ & \qquad v_1 \leftarrow \mathcal{E}[\![e_1]\!]\\ & \qquad v_2 \leftarrow \mathcal{E}[\![e_2]\!]\\ & \qquad \mathit{return}\ \uparrow (\downarrow v_1 + \downarrow v_2)\\ \mathcal{E}[\![e_1 - e_2]\!] & = \textbf{do}\\ & \qquad v_1 \leftarrow \mathcal{E}[\![e_1]\!]\\ & \qquad v_2 \leftarrow \mathcal{E}[\![e_2]\!]\\ & \qquad \mathit{return}\ \uparrow (\downarrow v_1 - \downarrow v_2)\end{array}$$

4.5.0.8 Lenguaje $\mathcal{L}_1$

En el lenguaje $\mathcal{L}_1$ se añaden productos y divisiones. A nivel semántico, la inclusión de divisiones puede ocasionar que algunas expresiones denoten valores indefinidos. Para afrontar dicha posibilidad, se utiliza el transformador de mónadas que añade errores (sección 4.3.2.1).

El único cambio a realizar a la estructura computacional será añadir dicho transformador de mónadas:

$$\mathsf{Comp} \triangleq \mathcal{T}_{Err}\ \mathsf{IdM}$$

y añadir las definiciones semánticas correspondientes:

$$\begin{aligned}
\mathcal{E}[\![e_1 * e_2]\!] = \ & \mathbf{do} \\
& v_1 \leftarrow \mathcal{E}[\![e_1]\!] \\
& v_2 \leftarrow \mathcal{E}[\![e_2]\!] \\
& \mathit{return}\ \uparrow (\downarrow v_1 * \downarrow v_2) \\
\mathcal{E}[\![e_1 / e_2]\!] = \ & \mathbf{do} \\
& v_1 \leftarrow \mathcal{E}[\![e_1]\!] \\
& v_2 \leftarrow \mathcal{E}[\![e_2]\!] \\
& \mathbf{if}\ \downarrow v_2 = 0\ \mathbf{then} \\
& \quad \mathit{err}\ \text{“division by zero”} \\
& \mathbf{else} \\
& \quad \mathit{return}\ \uparrow (\downarrow v_1 / \downarrow v_2)
\end{aligned}$$

Es importante observar que las definiciones de la sección anterior no deben ser modificadas. Supone una ventaja sobre la semántica denotacional, donde la modificación de características computacionales implica una modificación de las definiciones semánticas afectadas. Se alcanza de esta forma la *modularidad semántica* definida en 1.1.

4.5.0.9 Lenguaje $\mathcal{L}_2$

En $\mathcal{L}_2$ se añaden valores booleanos. La única modificación necesaria es modificar el dominio de valores para incluir booleanos.

$$\mathit{Value} \triangleq \mathit{Int} \parallel \mathit{Bool}$$

Las definiciones semánticas serán

$$\begin{aligned}
\mathcal{E}[\![e_1 = e_2]\!] = \ & \mathbf{do} \\
& v_1 \leftarrow \mathcal{E}[\![e_1]\!] \\
& v_2 \leftarrow \mathcal{E}[\![e_2]\!] \\
& \mathit{return}\ \uparrow (\downarrow v_1 = \downarrow v_2) \\
\mathcal{E}[\![e_1 < e_2]\!] = \ & \mathbf{do} \\
& v_1 \leftarrow \mathcal{E}[\![e_1]\!] \\
& v_2 \leftarrow \mathcal{E}[\![e_2]\!] \\
& \mathit{return}\ \uparrow (\downarrow v_1 < \downarrow v_2)
\end{aligned}$$

4.5.0.10 Lenguaje $\mathcal{L}_3$

En $\mathcal{L}_3$ se incluyen variables cuyo valor debe obtenerse de un entorno. Será necesario modificar la estructura computacional mediante un transformador de mónadas.

$$\mathsf{Comp} \triangleq (\mathcal{T}_{Env} \,.\, \mathcal{T}_{Err})\,\mathsf{Id}\mathsf{M}$$

$$\begin{aligned}\mathcal{E}[\![Var\ x]\!] = \mathbf{do}\\ \quad \rho \leftarrow rdEnv\\ \quad return(lkp\ \rho\ x)\end{aligned}$$

4.5.0.11 Lenguaje $\mathcal{L}_4$

En el lenguaje $\mathcal{L}_4$ aparecen los comandos imperativos. Para su modelización semántica se debe modificar de nuevo la estructura computacional, incorporando un transformador de mónadas que añada un estado actualizable.

$$\mathsf{Comp} \triangleq (\mathcal{T}_{State} \,.\, \mathcal{T}_{Err})\,\mathsf{Id}\mathsf{M}$$

En la definición anterior, no es necesario incluir el transformador que añade acceso al entorno, ya que toda mónada transformadora de estado es, a su vez, una mónada con acceso al entorno (véase 4.3.2.3)

Las definiciones semánticas serán

$$\mathcal{C}[\![skip]\!] = return\ ()$$

$$\begin{aligned}\mathcal{C}[\![c_1; c_2]\!] = \mathbf{do}\\ \quad \mathcal{C}[\![c_1]\!]\\ \quad \mathcal{C}[\![c_2]\!]\end{aligned}$$

$$\begin{aligned}\mathcal{C}[\![x := e]\!] = \mathbf{do}\\ \quad v \leftarrow \mathcal{E}[\![e]\!]\\ \quad \varsigma \leftarrow fetch\\ \quad set\ (upd\ \varsigma\ x\ v)\end{aligned}$$

4.5.0.12 Lenguaje $\mathcal{L}_5$

Las definiciones semánticas del lenguaje $\mathcal{L}_5$ serán

$$\begin{aligned}\mathcal{C}[\![\mathsf{if}\ e\ \mathsf{then}\ c_1\ \mathsf{else}\ c_2]\!] = \mathbf{do}\\ \quad v \leftarrow \mathcal{E}[\![e]\!]\\ \quad \mathbf{if}\ v\ \mathbf{then}\\ \quad\quad \mathcal{C}[\![c_1]\!]\\ \quad \mathbf{else}\\ \quad\quad \mathcal{C}[\![c_2]\!]\end{aligned}$$

$$\mathcal{C}[\![\mathsf{while}\ e\ \mathsf{do}\ c]\!] = loop$$

```
where
  loop = do
           v ← E[[e]]
           if v then
             do
               C[[c1]]
               loop
           else
             return()
```

4.6 Valoración

- *No ambigüedad.* Este formalismo no contiene ambigüedad en sus especificaciones.
- *Modularidad.* Se alcanza modularidad semántica, ya que es posible añadir nuevas características ortogonales al lenguaje que se está describiendo sin alterar las definiciones anteriores.
- *Reusabilidad.* Los intérpretes definidos en [145] tienen carácter monolítico. El objetivo de dicho sistema es resolver el problema de modularidad semántica, sin atacar el problema de reutilización de descripciones.
- *Demostración.* Este tipo de descripción combina los aspectos descriptivos de la semántica denotacional y la semántica axiomática. Es posible establecer axiomas sobre el comportamiento de las diferentes mónadas. Por ejemplo, en [146, 145] se muestra cómo se pueden aplicar transformaciones a las descripciones semánticas para demostrar propiedades de los programas y facilitar la creación de compiladores.
- *Prototipo.* Puesto que la implementación del sistema se realiza mediante un metalenguaje incrustado en el lenguaje *Haskell*, es posible obtener intérpretes de los lenguajes descritos de forma directa.
- *Legibilidad.* La legibilidad no es un objetivo del sistema, aunque la notación *do* y las facilidades sintácticas del propio lenguaje *Haskell* aumentan la legibilidad de las descripciones.
- *Flexibilidad.* El sistema puede aplicarse a la descripción semántica de lenguajes de diversos paradigmas.
- *Experiencia.* No se conocen aplicaciones a lenguajes reales. En [73, 74] se presenta la obtención de compiladores a partir de descripciones basadas en transformadores de mónadas, pero dicha técnica no se aplica a lenguajes reales.

La presente tesis intenta resolver los problemas de reusabilidad de la semántica monádica modular mediante la incorporación de conceptos de programación genérica. El siguiente capítulo presenta dichos conceptos que formarán la base del Sistema de Prototipado de Lenguajes que se desarrolla en el capítulo 6.

Capítulo 5

Programación Genérica

The construction principle involved is best called abstraction*; we concentrate on features common to many phenomena, and we abstract away features too far removed from the conceptual level at which we are working. Thereby we have a better chance of formulating concepts which are indeed useful at a later stage.* O. J. Dahl, C. A. R. Hoare [46]

5.1 Introducción

En este capítulo se presentan los conceptos de programación genérica necesarios para comprender el mecanismo utilizado para descomponer descripciones sintácticas. La programación genérica constituye por sí sola un campo de investigación reciente [14, 22], En este capítulo sólo se pretende introducir en los conceptos utilizados.

5.1.1 Proceso de Generalización/Especialización

En cualquier disciplina científica aparece continuamente lo que puede denominarse como ciclo de generalización/espacialización. Mediante la abstracción, un concepto que se repite en diversos contextos, se transforma en un concepto más general que podrá aplicarse posteriormente a dichos contextos o a nuevos contextos similares que puedan aparecer.

Este ciclo tiene especial importancia en la creación de *software*. Un error muy común en un programador novel consiste en escribir programas que dependen fuertemente del problema concreto que pretende resolver en cada momento. Por ejemplo, si se solicita a un programador principiante que construya una función para pintar dos diagonales cruzadas en una pantalla de 640×480, el programador suele escribir

$$
\begin{aligned}
\mathit{diagonals} \;=\; & \mathbf{do} \\
& \quad \mathit{line}\ (0,0)\ (640,480) \\
& \quad \mathit{line}\ (0,480)\ (640,0)
\end{aligned}
$$

El problema surge cuando se intenta adaptar el código especializado a otros entornos. En el ejemplo anterior, ante un cambio de resolución, la función

definida debería modificarse. El paso obvio es crear una función genérica que tenga como parámetros las dimensiones de la pantalla

$$\begin{aligned} &\mathit{diagonals}\ (dx, dy)\ =\ \mathbf{do} \\ &\qquad\qquad\qquad\quad \mathit{line}\ (0,0)\ (dx, dy) \\ &\qquad\qquad\qquad\quad \mathit{line}\ (0, dy)\ (dx, 0) \end{aligned}$$

La función genérica, se especializará posteriormente para el problema concreto mediante la llamada *diagonales* $(640, 480)$.

La programación genérica surge como una nueva disciplina encargada de buscar técnicas que permitan desarrollar programas o algoritmos generales que posteriormente puedan aplicarse a situaciones concretas.

En el ejemplo anterior, la función *diagonales* se generalizó mediante una nueva versión parametrizada por las dimensiones de la pantalla.

Haskell permite definir funciones parametrizadas por el tipo de datos sobre el que trabajan (estas funciones se denominan polimórficas). Por ejemplo, la función que calcula el número de elementos de una lista, puede definirse como

$$\begin{aligned} &\mathit{elemsList} && :\ \forall\alpha\ .\ [\alpha]\ \rightarrow\ \mathit{Int} \\ &\mathit{elemsList}\ [] && =0 \\ &\mathit{elemsList}\ (x : xs) && =1\ +\ \mathit{elemsList}\ xs \end{aligned}$$

Este tipo de funciones, pueden especializarse posteriormente para calcular el número de elementos de una lista de enteros, o de una lista de caracteres, o de cualquier otro tipo. Sin embargo, si se desea calcular el número de elementos de un árbol, debe definirse una nueva función

$$\begin{aligned} &\mathit{elemsTree} && :\ \forall\alpha\ .\ \mathit{Tree}\ \alpha \rightarrow \mathit{Int} \\ &\mathit{elemsTree}\ \mathit{Leaf} && =0 \\ &\mathit{elemsTree}\ (\mathit{Fork}\ x\ l\ r) && =1\ +\ \mathit{elemTree}\ l\ +\ \mathit{elemTree}\ r \end{aligned}$$

Un paso más en el proceso de generalización sería intentar construir una función *elems* que devuelva el número de elementos de una estructura recursiva cualquiera, bien sea una lista, un árbol, o cualquier otra. Dichas funciones se denominan *politípicas* [100] y pueden definirse en lenguajes como Polyp [101].

Recientemente, se ha propuesto una ampliación del lenguaje *Haskell* que permita este tipo de definiciones en [82, 87, 83] y existe una implementación limitada en la versión 5.00 del compilador GHC [1].

5.1.2 Evolución

Las raíces de la programación genérica se pueden encontrar en los trabajos sobre desarrollo constructivo de algoritmos de Bird y Meertens [157] que llegarían a desarrollar el formalismo Bird-Meertens (BMF) que más tarde se convierte en *Squiggol*.

La definición de tipos de datos recursivos mediante puntos fijos de functores fue propuesta [152]. En [158] se estudian las propiedades derivadas del estudio de álgebras iniciales y en [162, 161] se describen y aplican los catamorfismos para la derivación de compiladores a partir de la semántica algebraica.

En [22] se resumen los avances logrados en el campo de la construcción algebraica de programas y en [14] se introducen los conceptos básicos de *Programación genérica*. L. Duponcheel propone la utilización de catamorfismos para la

facilitar la reutilización de especificaciones semánticas en [50]. Su trabajo será la principal fuente de inspiración para esta tesis doctoral.

Los catamorfismos monádicos son estudiados en el ámbito de la Teoría de Categorías en [59, 91] y en [160] se utilizan catamorfismos monádicos para la estructuración de algoritmos en lenguajes funcionales.

En [132] se describe la aplicación de catamorfismos monádicos a la semántica monádica modular y en [133] se describe la construcción de un sistema de prototipado de lenguajes basado en dichas técnicas.

En la actualidad, la programación genérica se ha convertido en un campo de investigación como tal con varios *workshops* dedicados [13, 104]

5.2 Tipos de Datos Recursivos y Functores

La noción de functor deriva de la Teoría de Categorías, donde se define como un homomorfismo entre categorías. Para un programador funcional, un functor F puede considerarse como un constructor de tipos de datos que convierte valores de tipo α en valores de tipo $\mathsf{F}\alpha$. Un ejemplo característico es el constructor de tipos *List*[1] que transforma valores de tipo α en valores *List* α.

Definición 5.1 (Functor) *Un functor* F *es un constructor de tipos que asigna un tipo* $\mathsf{F}\ \alpha$ *a cada tipo* α*, junto con una operación*

$$map\ :\ (\alpha\ \rightarrow\ \beta)\ \rightarrow\ (\mathsf{F}\ \alpha\ \rightarrow\ \mathsf{F}\ \beta)$$

que cumple las propiedades

$$\begin{aligned} map\ id\ \ \ \ \ \ &= id \\ map\ (f\ .\ g)\ &= map\ f\ .\ map\ g \end{aligned}$$

En *Haskell*, el concepto de functor puede representarse mediante una clase de constructores de tipos[2]

```
class Functor f
  where
    map : (α → β) → (f α → f β)
```

Un functor F puede generar un tipo de datos recursivo que se define como el menor punto fijo de F.

Definición 5.2 (Punto fijo) *x es un punto fijo de F si x es una solución de la ecuación*

$$\mathsf{F}\ x \cong x$$

El menor punto fijo de F se denomina comúnmente $\mu\mathsf{F}$. En *Haskell* puede definirse como

$$\mu\mathsf{F}\ \triangleq\ In\ (\mathsf{F}\ (\mu\mathsf{F}))$$

[1] En *Haskell*, *List* α se denota como $[\alpha]$

[2] Esta clase está predefinida en *Haskell*, aunque en el proceso de estandarización para Haskell98 se decidió utilizar el nombre *fmap* en lugar de *map* que se reserva para el tipo listas

En la definición anterior se utiliza de forma explícita el constructor de tipos $In : \mathsf{F}\ (\mu\mathsf{F}) \to \mu\mathsf{F}$. En ocasiones se requiere el operador inverso $out : \mu\mathsf{F} \to \mathsf{F}\ (\mu\mathsf{F})$ definido como:

$$out\ (In\ x)\ =\ x$$

Ejemplo 5.1 *Considérese un tipo de datos recursivo de expresiones aritméticas simples*

$$Expr\ \triangleq\ Num\ Int\ \mid\ Expr\ +\ Expr$$

Dicho tipo de datos puede expresarse como el menor punto fijo del functor no recursivo E

$$\mathsf{E}\ x\ \triangleq\ Num\ Int\ \mid\ x\ +\ x$$

instance *Functor* E
  where
    $map\ f\ (Num\ n)\ \ = Num\ n$
    $map\ f\ (e_1\ +\ e_2)\ = f\ e_1\ +\ f\ e_2$

$$Expr\ \triangleq\ \mu\mathsf{E}$$

De esa forma, $In\ ((In\ (Num\ 3))\ +\ (In\ (Num\ 4))) : Expr$ *representaría la expresión* $3 + 4$.

Los tipos de datos parametrizados también pueden definirse como puntos fijos de functores. Para ello, se añade un parámetro extra al functor.

Ejemplo 5.2 *Las listas podrían definirse como*

$$\mathsf{L}\ \alpha\ x\ \triangleq\ Nil\ \mid\ Cons\ \alpha\ x$$

$$List\ \alpha\ \triangleq\ \mu(\mathsf{L}\ \alpha)$$

De esa forma, $In\ (Cons\ 3\ (In\ (Cons\ 5\ (In\ (Cons\ 7\ (In\ Nil))))))$: $List\ Int$ *representaría la lista* $[3, 5, 7]$.

Ejemplo 5.3 *Los árboles binarios podrían definirse de forma similar*

$$\mathsf{T}\ \alpha\ x\ \triangleq\ Leaf\ \mid\ Fork\ \alpha\ x\ x$$

instance *Functor* T
  where
    $map\ f\ Leaf\ \ \ \ \ \ \ \ = Leaf$
    $map\ f\ (Fork\ a\ l\ r)\ = Fork\ a\ (f\ l)\ (f\ r)$

$$Tree\ \triangleq\ \mu(\mathsf{T}\ \alpha)$$

5.3 Functores polinómicos

Los functores polinómicos se forman mediante functores primitivos (functor identidad y functor constante) y la aplicación reiterada de sumas, productos y composiciones sobre ellos. A continuación se definen dichas operaciones.

- El functor más sencillo sería el functor identidad que no modifica el valor original. Su definición podría ser[3].

  $Id\ x \triangleq x$

  instance *Functor Id*
  where
  $map\ f\ x = f\ x$

- El functor constante $\underline{\alpha}$ devuelve siempre un valor de tipo α.

  $\underline{\alpha}\ x \triangleq \alpha$

  instance *Functor* $\underline{\alpha}$
  where
  $map\ f\ a = a$

- El functor suma $\oplus$ equivale a la unión disjunta de dos conjuntos. Para su definición se utiliza el tipo[4].

  $\alpha \parallel \beta \triangleq L\,\alpha \mid R\,\beta$

  El functor suma puede definirse como

  $(\mathsf{F} \oplus \mathsf{G})\ x \triangleq \mathsf{F}\ x \parallel \mathsf{G}\ x$

  instance (*Functor* f, *Functor* g) $\Rightarrow$ *Functor* $(\mathsf{f} \oplus \mathsf{g})$
  where
  $map\ f\ (L\ x) = L\ (map\ f\ x)$
  $map\ f\ (R\ x) = R\ (map\ f\ x)$

- El functor producto $\otimes$ equivale al producto cartesiano de dos conjuntos. Para su definición se utiliza el tipo de datos tupla (α, β) predefinido de *Haskell*. Puede definirse como

  $(\mathsf{F} \otimes \mathsf{G})\ x \triangleq (\mathsf{F}\ x,\ \mathsf{G}\ x)$

  instance (*Functor* f, *Functor* g) $\Rightarrow$ *Functor* $(\mathsf{f} \otimes \mathsf{g})$
  where
  $map\ f\ (x, y) = (map\ f\ x,\ map\ f\ y)$

[3] Tanto en esta definición como en las siguientes, no se escribe el constructor de tipos para facilitar la legibilidad de los ejemplos. En realidad la definición debería ser **data** $Id\ x = Id\ x$ y luego habría que acarrear el constructor Id en todos los ejemplos

[4] El tipo $\alpha \parallel \beta$ equivale al tipo predefinido $Either\ \alpha\ \beta$

- La composición de dos functores F y G se denotará mediante su yuxtaposición $\mathsf{F}\,\mathsf{G}$ y se define como

  $$\mathsf{F}\,\mathsf{G}\,x \triangleq \mathsf{F}\,(\mathsf{G}\,x)$$

  ```
  instance (Functor f, Functor g) ⇒ Functor (f g)
    where
      map f x = map (λv → map f v) x
  ```

Definición 5.3 (Functor polinómico) *Los functores polinómicos se obtienen mediante sumas, productos y composiciones de functores primitivos. Deben seguir, por tanto, la siguiente gramática (donde* F *denota un functor):*

$$F ::= Id \mid \underline{\alpha} \mid \mathsf{F} \oplus \mathsf{F} \mid \mathsf{F} \otimes \mathsf{F} \mid \mathsf{F}\,\mathsf{F}$$

Ejemplo 5.4 *El functor* $\mathsf{L}\,\alpha$ *que permitía definir listas en el ejemplo 5.2 puede definirse como el siguiente functor polinómico*

$$\mathsf{L}\,\alpha \triangleq \underline{()} \oplus (\underline{\alpha} \otimes Id)$$

Ejemplo 5.5 *De la misma forma, el functor* $\mathsf{T}\,\alpha$ *que permite definir árboles binarios en el ejemplo 5.3 puede definirse como:*

$$\mathsf{T}\,\alpha \triangleq \underline{()} \oplus (\underline{\alpha} \otimes Id \otimes Id)$$

Ejemplo 5.6 *En esta tesis, los functores polinómicos se aplicarán fundamentalmente para extender un tipo de datos recursivo sin modificar las funciones que trabajan sobre él. A modo de ejemplo, supóngase que el tipo de datos Expr del ejemplo 5.1 se desea aumentar con variables. Para ello, sólo es necesario definir un nuevo functor*

$$\mathsf{V}\,x \triangleq Var\ String$$

A partir de dicho functor se puede definir el nuevo lenguaje $\mathcal{L}_{EV}$ *como el punto fijo de la suma de los functores* E *y* V

$$\mathcal{L}_{EV} \triangleq \mu(\mathsf{E} \oplus \mathsf{V})$$

5.4 Álgebras

Definición 5.4 (F-álgebra) *Dado un functor* F *y un determinado tipo* α*, una función* $\varphi_{\mathsf{F}} : \mathsf{F}\,\alpha \to \alpha$ *se denomina una* F*-álgebra.* α *se denomina portador o* carrier *de la* F*-álgebra.*

Ejemplo 5.7 *Sobre el functor* E *definido en el ejemplo 5.1 y el conjunto Int se define una* E*-álgebra*

$$\begin{aligned} \varphi_{\mathsf{E}} &: \mathsf{E}\ Int \to Int \\ \varphi_{\mathsf{E}}\,(Num\ n) &= n \\ \varphi_{\mathsf{E}}\,(e_1 + e_2) &= e_1 + e_2 \end{aligned}$$

En ocasiones, puede ser conveniente representar F-ÁLgebras mediante una clase de tipos con 2 parámetros[5]

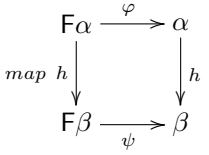
[5]Esta representación puede generar problemas de solapamiento de instancias cuando se desean definir varias F-álgebras distintas sobre el mismo portador

```
class Algebra F α
  where
    φ : F α → α
```

Definición 5.5 (Homomorfismo entre F-álgebras) *Un homomorfismo entre dos* F*-álgebras* $\varphi : \mathsf{F}\,\alpha \to \alpha$ *y* $\psi : \mathsf{F}\,\beta \to \beta$ *es una función* $h : \alpha \to \beta$ *que cumple la siguiente ecuación:*

$$h \cdot \varphi = \psi \cdot map\, h \tag{5.1}$$

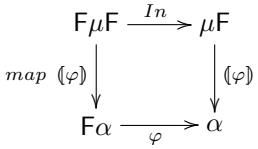
La ecuación anterior puede representarse mediante el diagrama



5.5 Catamorfismos o *folds*

A partir de un functor F se puede formar una nueva categoría cuyos objetos son F-álgebras y cuyos morfismos son homomorfismos entre F-álgebras. En dicha categoría, puede demostrarse que $In : \mathsf{F}(\mu\mathsf{F}) \to \mu\mathsf{F}$ es un objeto inicial. Se cumple, por tanto, que para cualquier F-álgebra $\varphi : \mathsf{F}\,\alpha \to \alpha$, existe un único homomorfismo $(\!|\varphi|\!) : \mu\mathsf{F} \to \alpha$ que cumple la ecuación 5.1.

$(\!|\varphi|\!)$ se denomina fold o *catamorfismo* y cumple el diagrama



A partir de dicho diagrama, puede obtenerse la definición:

```
(|_|)       : (Functor f) ⇒ (f α → α) → (μf → α)
(|φ|) . In  = φ . map (|φ|)
```

Ejemplo 5.8 *Para calcular el número de elementos de una lista (véase ejemplo 5.4) se define la* L*-álgebra*

```
φL           : L α Int → Int
φL (L _)     = 0
φL (R (a, l)) = 1 + l
```

La función elemsList : *List* $\alpha \to$ *Int se puede definir como*

```
elemsList : List α → Int
elemsList = (|φL|)
```

Ejemplo 5.9 *Para calcular el número de elementos de un árbol binario (véase ejemplo 5.5) se define la* T*-álgebra*

$$\begin{array}{ll} \varphi_{\mathsf{T}} & : \mathsf{T}\,\alpha\,\mathit{Int} \to \mathit{Int} \\ \varphi_{\mathsf{T}}\,(L\ \_) & = 0 \\ \varphi_{\mathsf{T}}\,(R\,(a,\,(l,r))) & = 1\ +\ l\ +\ r \end{array}$$

La función elemsTree que calcula el número de elementos del árbol se define de la misma forma que en el ejemplo anterior mediante un catamorfismo

$$\begin{array}{ll} \mathit{elemsTree} & : \ \mathit{Tree}\,\alpha \to \mathit{Int} \\ \mathit{elemsTree} & = (\!|\varphi_{\mathsf{T}}|\!) \end{array}$$

Obsérvese que $(\!|\_|\!)$ funciona para diversos tipos de datos (listas, árboles, etc.). Este tipo de operadores se denominan politípicos ya que no dependen del tipo de datos particular sobre el que se aplican.

Definición 5.6 ($\mathsf{F} \oplus \mathsf{G}$-álgebra) *Dada una* F*-álgebra* $\varphi : \mathsf{F}\,\alpha \to \alpha$ *y una* G*-álgebra* $\psi : \mathsf{G}\,\alpha \to \alpha$ *es posible definir una* $\mathsf{F} \oplus \mathsf{G}$*-álgebra mediante el operador* $\oplus$*:*

$$\begin{array}{l} \oplus\ : (\mathsf{F}\,\alpha \to \alpha) \to (\mathsf{G}\,\alpha \to \alpha) \to ((\mathsf{F} \oplus \mathsf{G})\,\alpha \to \alpha) \\ (\varphi_{\mathsf{F}}\ \oplus\ \varphi_{\mathsf{G}})\,(L\,x)\ =\ \varphi_{\mathsf{F}}\,x \\ (\varphi_{\mathsf{F}}\ \oplus\ \varphi_{\mathsf{G}})\,(R\,x)\ =\ \varphi_{\mathsf{G}}\,x \end{array}$$

Cuando el portador del álgebra consiste en una estructura computacional representada por una mónada, entonces, aplicando un catamorfismo sobre dicha álgebra, se obtiene directamente la función de interpretación.

Si la sintaxis del lenguaje se define como el punto fijo de la suma de varios functores, entonces pueden definirse las álgebras de cada functor por separado.

Ejemplo 5.10 *Supóngase que se desea construir un intérprete para el sencillo lenguaje formado por expresiones aritméticas y variables del ejemplo 5.6. Para modelizar la estructura computacional, se utilizará una mónada* $\mathsf{EnvM}_{\mathsf{Env}}$ *que accede a un entorno* $\rho : \mathit{Env}$*. Dicho entorno contiene una función*

$$\mathit{lkp}\ :\ \mathit{Env} \to \mathit{Var} \to \mathit{Int}$$

tal que $\mathit{lkp}\ \ \rho\,x$ *devuelve el valor de la variable* x *en el entorno* ρ*.*

El dominio de valores será el tipo Int de valores enteros. La estructura computacional será:

$$\mathsf{Comp}\ \triangleq\ (\mathcal{T}_{Env}\ \mathsf{IdM})$$

De esa forma, un valor del tipo $\mathsf{Comp}\ \mathit{Int}$ *indica una computación que puede acceder al entorno y devuelve un valor entero.*

Para definir la semántica de dicho lenguaje, se definen por separado las álgebras correspondientes al functor E *(ejemplo 5.1) y al functor* V *(ejemplo 5.6) tomando como portador* $\mathsf{Comp}\ \mathit{Int}$*.*

$$\varphi_{\mathsf{E}} \qquad : \ \mathsf{E}\,(\mathsf{Comp}\ \mathit{Int})\ \to\ \mathsf{Comp}\ \mathit{Int}$$

$$
\begin{array}{lll}
\varphi_{\mathsf{E}}\ (Num\ n) & = return\ n \\
\varphi_{\mathsf{E}}\ (m_1\ +\ m_2) & = \mathbf{do} \\
 & \qquad v_1 \leftarrow m_1 \\
 & \qquad v_2 \leftarrow m_2 \\
 & \qquad return\ (v_1\ +\ v_2)
\end{array}
$$

$$
\begin{array}{lll}
\varphi_{\mathsf{V}} & : \mathsf{V}\ (\mathsf{Comp}\ Int)\ \rightarrow\ \mathsf{Comp}\ Int \\
\varphi_{\mathsf{V}}\ (Var\ x) & = \mathbf{do} \\
 & \qquad \rho \leftarrow rdEnv \\
 & \qquad return\ (lookup\ \rho\ x)
\end{array}
$$

Aplicando un catamorfismo sobre dichas álgebras se obtiene directamente la función de interpretación del lenguaje

$$
\begin{array}{l}
\mathsf{Inter}_{\mathcal{L}_{EV}}\ :\ \mathcal{L}_{EV}\ \rightarrow\ \mathsf{Comp}\ Int \\
\mathsf{Inter}_{\mathcal{L}_{EV}}\ = (\!|\varphi_{\mathsf{E}} \oplus \varphi_{\mathsf{V}}|\!)
\end{array}
$$

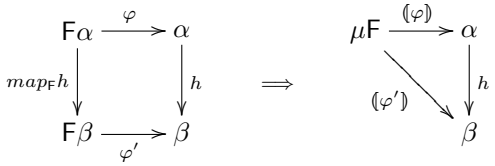
En [232] se demuestra que a partir del tipo de una función polimórfica es posible obtener el teorema libre (*free theorem*) que cumple dicha función.

En el caso de la función $(\!|\_|\!) : (\mathsf{F}\ \alpha \rightarrow \alpha) \rightarrow \mu\mathsf{F} \rightarrow \alpha$, el teorema libre se denomina ley de fusión [158, 159].

Teorema 5.1 (Ley de fusión) *Dadas dos* F*-álgebras* $\varphi : \mathsf{F}\ \alpha \rightarrow \alpha$ *y* $\varphi' : \mathsf{F}\ \beta \rightarrow \beta$*, para cualquier función* $h : \alpha \rightarrow \beta$ *se cumple que:*

$$h\ .\ \varphi\ =\ \varphi'\ .\ map\ h\ \Rightarrow\ h\ .\ (\!|\varphi|\!)\ =\ (\!|\varphi'|\!)$$

El siguiente diagrama representa la ley anterior



La ley de fusión facilita la transformación y optimización de programas basados en catamorfismos [210, 141], habiéndose desarrollado sistemas que realizan dichas transformaciones de forma automática [183].

Los catamorfismos capturan la idea intuitiva de generar un valor a partir de una estructura recursiva. De forma similar, se han desarrollado los *anamorfismos*, que permiten obtener una estructura recursiva a partir de un valor, los *hilomorfismos*, que transforman un valor en otro utilizando una estructura recursiva intermedia, etc. Las propiedades calculacionales de este tipo de operadores forman el núcleo de la algorítmica constructiva o programación genérica [14, 22, 58, 103].

5.6 Catamorfismos Monádicos

En [160] se introducen los catamorfismos monádicos como una técnica que permite combinar programas que realizan un recorrido recursivo de su entrada a la vez que generan efectos laterales en su salida. En este sentido la semántica de lenguajes de programación parece un marco natural para utilizar esta técnica ya que la entrada está formada por un árbol sintáctico que es una estructura recursiva y la salida se expresa como una mónada computacional sobre la que se generan efectos laterales.

La generalización de los catamorfismos a los catamorfismos monádicos fue estudiada en [59, 91] donde se estudian las condiciones requeridas para utilizar esta técnica para las diversas combinaciones de tipos de datos y mónadas. En [132] se particulariza su trabajo a la semántica monádica modular con el fin de observar las mejoras obtenidas de su aplicación.

Dada una mónada M sobre una categoría $\mathcal{C}$, se puede formar una nueva categoría $\mathcal{C}^{\mathsf{M}}$ cuyos objetos serán los objetos de $\mathcal{C}$ y cuyos morfismos serán las funciones monádicas.

En [187] se define la *extensión monádica de un functor* y se muestran las propiedades que satisface. Para su representación en *Haskell*, puede definirse una clase de tipos *MFunctor* (generalizando la clase *Functor* de la página 77)

```
class (Monad m, Functor f) ⇒ MFunctor f m where
  where
    mapm : (α → m β) → (f α → m (f β))
```

Ejemplo 5.11 *La extensión monádica del functor E de expresiones aritméticas (véase 5.1) puede declararse como:*

```
instance (Monad m) ⇒ MFunctor E m
  where
    mapm f (Num n)   = return (Num n)
    mapm f (e1 + e2) = do
                         v1 ← f e1
                         v2 ← f e2
                         return (v1 + v2)
```

Obsérvese que esta definición se adapta a cualquier mónada (no utiliza ninguna operación de un tipo de mónada particular) y que se especifica de forma explícita el orden de evaluación de los operandos (podría ser diferentes para tipos de mónadas específicos).

Ejemplo 5.12 *La extensión monádica del functor* V *puede declararse como*

```
instance (Monad m) ⇒ MFunctor V m
  where
    mapm f (Var n) = return (Var x)
```

Como puede comprobarse a partir de los ejemplos, las extensiones monádicas de los functores podrían generarse de forma automática.

Definición 5.7 (F-álgebra monádica) *Dado un functor* F*, una mónada* M *y un determinado tipo* α*, una función monádica* $\varpi_{\mathsf{F}} : \mathsf{F}\alpha \to \mathsf{M}\alpha$ *se denomina una* F-álgebra monádica*.*

De la misma forma que se definió una clase de tipos *Algebra* que representaba F-álgebras, es posible definir una clase de tipos *MAlgebra* que representa F-álgebras monádicas.

```
class MAlgebra m f α
  where
    ϖ : f α → m α
```

Definición 5.8 (Homomorfismos monádicos) *Un homomorfismo monádico entre dos* F*-álgebras monádicas* $\varpi : \mathsf{F}\,\alpha \to \mathsf{M}\,\alpha$ *y* $\varepsilon : \mathsf{F}\,\beta \to \mathsf{M}\,\beta$ *es una función* $h : \alpha \to \mathsf{M}\,\beta$ *que comple la siguiente ecuación*

$$h \mathbin{@} \varpi = \varepsilon \mathbin{@} \mathsf{map_m}\ h \tag{5.2}$$

La ecuación anterior puede representarse como

$$\begin{array}{ccc} \mathsf{F}\alpha & \xRightarrow{\varpi} & \alpha \\ {\scriptstyle \mathsf{map_m}\ h}\Big\Downarrow & & \Big\Downarrow{\scriptstyle h} \\ \mathsf{F}\beta & \xRightarrow[\varepsilon]{} & \beta \end{array}$$

A partir de un functor F y una mónada M se puede formar una nueva categoría cuyos objetos son F-álgebras monádicas y cuyos morfismos son homomorfismos monádicos entre ellas. En dicha categoría, el operador In_{m} definido como

```
Inm : (Monad m) ⇒ F μF → m μF
Inm = return . In
```

es un objeto inicial

Por tanto, para cualquier F-álgebra monádica ϖ existe un único homomorfismo monádico $(\!|\varpi|\!) : \mu\mathsf{F} \to \mathsf{m}\,\alpha$ que cumple la ecuación 5.2.

$(\!|\varpi|\!)$ se denomina *fold monádico* o *catamorfismo monádico* y cumple el diagrama

$$\begin{array}{ccc} \mathsf{F}\mu\mathsf{F} & \xRightarrow{In} & \mu\mathsf{F} \\ {\scriptstyle \mathsf{map_m}\ (\!|\varpi|\!)}\Big\Downarrow & & \Big\Downarrow{\scriptstyle (\!|\varpi|\!)} \\ \mathsf{F}\alpha & \xRightarrow[\varpi]{} & \alpha \end{array}$$

A partir de dicho diagrama, puede obtenerse la definición:

```
(|_|)          : (MFunctor f m) ⇒ (f α → m α) → (μf → m α)
(|ϖ|) @ Inm = ϖ @ mapm (|ϖ|)
```

En [59, 91] se estudian las propiedades necesarias para poder emplear leyes similares a la ley de fusión con catamorfismos monádicos.

Definición 5.9 ($\mathsf{F} \oplus \mathsf{G}$-álgebra monádica) *Dada una* F*-álgebra monádicas y una* G*-álgebra monádica, es posible construir una* $\mathsf{F} \oplus \mathsf{G}$*-álgebra monádica mediante el operador* $\oplus_{\mathsf{m}}$ *que se define como:*

$$
\begin{array}{ll}
\oplus_{\mathsf{m}} & : (\mathsf{F}\,\alpha \rightarrow \mathsf{M}\,\alpha) \rightarrow (\mathsf{G}\,\alpha \rightarrow \mathsf{M}\,\alpha) \rightarrow ((\mathsf{F} \oplus \mathsf{G})\,\alpha \rightarrow \mathsf{M}\,\alpha) \\
(\varpi_{\mathsf{F}} \oplus_{\mathsf{m}} \varpi_{\mathsf{G}})\,(L\,x) & = \varpi_{\mathsf{F}}\,x \\
(\varpi_{\mathsf{F}} \oplus_{\mathsf{m}} \varpi_{\mathsf{G}})\,(R\,x) & = \varpi_{\mathsf{G}}\,x
\end{array}
$$

Ejemplo 5.13 *El intérprete del ejemplo 5.10 se construyó mediante catamorfismos cuyas álgebras tomaban la estructura computacional como portadora. Si en lugar de definir álgebras, se definen álgebras monádicas, el intérprete resultante se definiría de la siguiente forma*

$$
\begin{array}{ll}
\varpi_{\mathsf{E}} & : \mathsf{E}\ \mathit{Int} \rightarrow \mathsf{Comp}\ \mathit{Int} \\
\varpi_{\mathsf{E}}\,(\mathit{Num}\ n) & = \mathit{return}\ n \\
\varpi_{\mathsf{E}}\,(v_1 + v_2) & = \mathbf{do} \\
 & \qquad \mathit{return}\,(v_1 + v_2)
\end{array}
$$

$$
\begin{array}{ll}
\varpi_{\mathsf{V}} & : \mathsf{V}\ \mathit{Int} \rightarrow \mathsf{Comp}\ \mathit{Int} \\
\varpi_{\mathsf{V}}\,(\mathit{Var}\ x) & = \mathbf{do} \\
 & \qquad \rho \leftarrow \mathit{rdEnv} \\
 & \qquad \mathit{return}\,(\mathit{lkp}\ \ \rho\,x)
\end{array}
$$

El intérprete se obtiene mediante un catamorfismo monádico:

$$
\begin{array}{ll}
\mathsf{Inter}'_{\mathcal{L}_{EV}} & : \mathcal{L}_{EV} \rightarrow \mathsf{Comp}\ \mathit{Int} \\
\mathsf{Inter}'_{\mathcal{L}_{EV}} & = (\!|[\varpi]|\!)
\end{array}
$$

La nueva versión del intérprete utilizando catamorfismos monádicos ha separado la especificación en dos fases:

- Evaluación recursiva de subcomponentes, la cual puede realizarse de forma automática.
- Especificación semántica de cada functor, que ya parte de los valores de los subcomponentes.

5.7 Combinación entre catamorfimos y catamorfismos monádicos

En ocasiones no es posible definir un catamorfimo monádico. Por ejemplo, en el caso de la sentencia condicional **if** e_1 **then** e_2 **else** e_3 no es posible aplicar un catamorfismo monádico ya que se evaluarían todas las subexpresiones independientemente del valor de la condición.

Se hace necesario, por tanto, establecer un mecanismo que permita combinar la semántica definida mediante F-álgebras que tomen $\mathsf{M}\,\mathsf{V}$ como portadora, con F-álgebras monádicas sobre una mónada M y una portadora V.

Para conseguir dicha combinación, se combinan las definiciones de catamorfismo (página 81) y catamorfismo monádico (página 85):

$$
\begin{array}{l}
(\!|\_,\_|\!) \;:\; (\mathsf{F}(\mathsf{m}\ v) \to \mathsf{m}\ v) \to (\mathsf{G}\ v \to \mathsf{m}\ v) \to (\mu(\mathsf{F} \oplus \mathsf{G}) \to \mathsf{m}\ v) \\
(\!|\varphi, \varpi|\!)\ (In\ x) \;=\; \mathbf{case}\ x\ \mathbf{of} \\
\qquad\qquad\qquad\qquad L\ v \;\to\; (\varphi\ .\ map\ (\!|\varphi, \varpi|\!))\ v \\
\qquad\qquad\qquad\qquad R\ v \;\to\; (\varpi\ @\ \mathsf{map_m}\ (\!|\varphi, \varpi|\!))\ v
\end{array}
$$

$$
\begin{array}{l}
(\![\_,\_]\!) \;:\; (\mathsf{F}\ v \to \mathsf{m}\ v) \to (\mathsf{G}\ (\mathsf{m}\ v) \to \mathsf{m}\ v) \to (\mu(\mathsf{F} \oplus \mathsf{G}) \to \mathsf{m}\ v) \\
(\![\varpi, \varphi]\!)\ (In\ x) \;=\; \mathbf{case}\ x\ \mathbf{of} \\
\qquad\qquad\qquad\qquad L\ v \;\to\; (\varpi\ @\ \mathsf{map_m}\ (\![\varpi, \varphi]\!))\ v \\
\qquad\qquad\qquad\qquad R\ v \;\to\; (\varphi\ .\ map\ (\![\varpi, \varphi]\!))\ v
\end{array}
$$

Ejemplo 5.14 *Sea* IF *el functor:*

$$\mathsf{IF}\ x \triangleq If\ x\ x\ x$$

sobre el cual se define una IF*-álgebra que toma como portadora la estructura computacional* Comp *Int del ejemplo 5.10.*

$$
\begin{array}{l}
\varphi_{\mathsf{IF}} \qquad\qquad\quad :\; \mathsf{IF}\ (\mathsf{Comp}\ Int) \;\to\; (\mathsf{Comp}\ Int) \\
\varphi_{\mathsf{IF}}\ (If\ e\ e_1\ e_2) \;=\; \mathbf{do} \\
\qquad\qquad\qquad\qquad b \;\leftarrow\; e \\
\qquad\qquad\qquad\qquad \mathbf{if}\ b \;==\; 0\ \mathbf{then} \\
\qquad\qquad\qquad\qquad\quad e_1 \\
\qquad\qquad\qquad\qquad \mathbf{else} \\
\qquad\qquad\qquad\qquad\quad e_2
\end{array}
$$

El posible definir un intérprete de un lenguaje formado por condiciones y números como:

$$\mathcal{L}_{IFN} \triangleq \mu(\mathsf{E} \oplus \mathsf{IF})$$

y un intérprete de dicho lenguaje se obtendría de la siguiente forma

$$
\begin{array}{l}
\mathsf{Inter}_{\mathcal{L}_{IFN}} \;:\; \mathcal{L}_{IFN} \;\to\; \mathsf{Comp}\ Int \\
\mathsf{Inter}_{\mathcal{L}_{IFN}} \;= (\![\varpi_{\mathsf{E}}, \varphi_{\mathsf{IF}}]\!)
\end{array}
$$

5.8 Tipos Mutuamente recursivos y catamorfismos

En las secciones anteriores se plantea la construcción de intérpretes de lenguajes cuya sintaxis se define como el punto fijo de un functor que captura el patrón recursivo.

En la práctica la sintaxis de un lenguaje suele descomponerse en varias categorías mutuamente recursivas. Por ejemplo, en un lenguaje imperativo tradicional, suelen definirse las categorías mutuamente recursivas *expresión* y *comando*.

Para afrontar el desarrollo modular de los functores de cada categoría, se generaliza el concepto de F-Álgebra para n tipos mutuamente recursivos siguiendo el trabajo de [58, 161, 99]. Para ello se trabajará con un álgebra multiclase (*many-sorted*).

Para facilitar la exposición, se utilizarán álgebras biclase, es decir, con $n = 2$. La generalización para n clases es inmediata.

A continuación se define el concepto de bifunctor, que será similar a un functor con 2 argumentos.

Definición 5.10 (bifunctor) *Un bifunctor* $\mathbb{F}$ *es un constructor de tipos que asigna un tipo* $\mathbb{F}\,\alpha\,\beta$ *a cada par de tipos* α *y* β*, junto con una operación*

$$bimap\ :\ (\alpha\ \to\ \gamma)\ \to\ (\beta\ \to\ \delta)\ \to\ (\mathbb{F}\,\alpha\,\beta\ \to\ \mathbb{F}\,\gamma\,\delta)$$

que cumple las propiedades

$$\begin{aligned} &bimap\ id\ id &&= id\\ &bimap\ (f\ .\ h)\ (g\ .\ i) &&= bimap\ f\ g\ .\ bimap\ h\ i \end{aligned}$$

El concepto de bifunctor se puede representar en *Haskell* con la siguiente clase de constructores de tipos

```
class BiFunctor F
  where
    bimap_F : (α → γ) → (β → δ) → (F α β → F γ δ)
```

La definición de dos tipos de datos mutuamente recursivos puede realizarse a partir del menor punto fijo de dos bifunctores. Los puntos fijos de dos bifunctores mutuamente recursivos se definen a partir de las ecuaciones

$$\begin{aligned} x\ &\cong\ \mathbb{F}\ x\ y\\ y\ &\cong\ \mathbb{G}\ x\ y \end{aligned}$$

Suponiendo que la solución de dichas ecuaciones es $x\ =\ \mu_1\mathbb{F}\mathbb{G}$ e $y\ =\ \mu_2\mathbb{F}\mathbb{G}$, se obtiene la siguiente definición en *Haskell*:

$$\begin{aligned} \mu_1\mathbb{F}\mathbb{G}\ &=\ In_1\ (\mathbb{F}\ (\mu_1\mathbb{F}\mathbb{G})\ (\mu_2\mathbb{F}\mathbb{G}))\\ \mu_2\mathbb{F}\mathbb{G}\ &=\ In_2\ (\mathbb{G}\ (\mu_1\mathbb{F}\mathbb{G})\ (\mu_2\mathbb{F}\mathbb{G})) \end{aligned}$$

Ejemplo 5.15 *Considérense los siguientes tipos de datos mutuamente recursivos que denotan expresiones aritméticas simples y comandos.*

$$\begin{aligned} Expr\ &\triangleq\ Num\ Int\ \mid\ Expr\ +\ Expr\\ Comm\ &\triangleq\ Comm\,;\ Comm\ \mid\ Name\ :=\ Expr \end{aligned}$$

Dichos tipos de datos pueden expresarse a partir de los bifunctores

$$\begin{aligned} \mathbb{E}\ e\ c\ &\triangleq\ Num\ Int\ \mid\ e\ +\ e\\ \mathbb{C}\ e\ c\ &\triangleq\ c\,;\ c\ \mid\ Name\ :=\ e \end{aligned}$$

```
instance BiFunctor E
  where
    bimap f g (Num n)   = Num n
    bimap f g (e1 + e2) = f e1 + f e2

instance BiFunctor C
  where
    bimap f g (c1 ; c2)   = g c1 ; g c2
    bimap f g (x := e)    = x := f e
```

$$\begin{aligned} Expr &\triangleq \mu_1 \mathbb{EC} \\ Comm &\triangleq \mu_2 \mathbb{EC} \end{aligned}$$

Pudiendo definirse los siguientes valores

$$\begin{aligned} v_1 &: \mathcal{L}_{Expr} \\ v_1 &= In_1\ (\ (In_1\ (Num\ 3))\ +\ (In_1\ (Num\ 4))) \end{aligned}$$

$$\begin{aligned} c_1 &: \mathcal{L}_{Comm} \\ c_1 &= In_2\ (\text{“}x''\ :=\ v_1) \end{aligned}$$

Definición 5.11 ($\mathbb{F}, \mathbb{G}$-biálgebra) *Dados dos bifunctores $\mathbb{F}$, $\mathbb{G}$ y dos tipos α, β, un par de funciones (φ, ψ) tal que:*

$$\begin{aligned} \varphi &: \mathbb{F}\ \alpha\ \beta\ \rightarrow\ \alpha \\ \psi &: \mathbb{G}\ \alpha\ \beta\ \rightarrow\ \beta \end{aligned}$$

se denomina $\mathbb{F}, \mathbb{G}$-biálgebra.
α, β se denominan portadores

Definición 5.12 (Homomorfismo entre biálgebras) *Dadas 2 biálgebras $(\varphi : \mathbb{F}\ \alpha\ \beta \rightarrow \alpha, \psi : \mathbb{G}\ \alpha\ \beta \rightarrow \beta)$ y $(\chi : \mathbb{F}\ \gamma\ \delta \rightarrow \gamma, \omega : \mathbb{G}\ \gamma\ \delta \rightarrow \delta)$ un homomorfismo entre ambas es un par de funciones $(f : \alpha \rightarrow \gamma, g : \beta \rightarrow \delta)$ que cumplen las siguientes ecuaciones:*

$$\begin{aligned} f\ \circ\ \varphi\ &= \chi\ \circ\ bimap_{\mathbb{F}}\ f\ g \\ g\ \circ\ \psi\ &= \omega\ \circ\ bimap_{\mathbb{G}}\ f\ g \end{aligned}$$

Las ecuaciones anteriores pueden representarse mediante los siguientes diagramas:

$$\begin{array}{ccc} \mathbb{F}\alpha\beta & \xrightarrow{\varphi} & \alpha \\ {\scriptstyle bimap_{\mathbb{F}}\ f\ g}\downarrow & & \downarrow{\scriptstyle f} \\ \mathbb{F}\gamma\delta & \xrightarrow[\chi]{} & \gamma \end{array}$$

$$\begin{array}{ccc} \mathbb{G}\alpha\beta & \xrightarrow{\psi} & \beta \\ {\scriptstyle bimap_{\mathbb{G}}\ f\ g}\downarrow & & \downarrow{\scriptstyle g} \\ \mathbb{G}\gamma\delta & \xrightarrow[\omega]{} & \delta \end{array}$$

A partir de dos bifunctores $\mathbb{F}, \mathbb{G}$ se forma una nueva categoría cuyos objetos son $\mathbb{F}, \mathbb{G}$-biálgebras y cuyos morfismos son homomorfismos entre $\mathbb{F}, \mathbb{G}$-biálgebras. En dicha categoría, puede demostrarse que

$$(In_1\ : \mathbb{F}\ (\mu_1 \mathbb{FG})\ (\mu_2 \mathbb{FG})\ \rightarrow\ \mu_1 \mathbb{FG},\ In_2\ : \mathbb{G}\ (\mu_1 \mathbb{FG})\ (\mu_2 \mathbb{FG})\ \rightarrow\ \mu_2 \mathbb{FG})$$

es un objeto inicial. De esa forma, para todo $\mathbb{F}, \mathbb{G}$-biálgebra $(\varphi : \mathbb{F}\ \alpha\ \beta \rightarrow \alpha$, $\psi : \mathbb{G}\alpha\beta \rightarrow \beta)$ existen dos únicas funciones $(f : \mu_1 \mathbb{FG}\ \rightarrow\ \alpha,\ g : \mu_2 \mathbb{FG}\ \rightarrow\ \beta)$ que

se denotarán como $(\!|\varphi,\psi|\!)_1$ y $(\!|\varphi,\psi|\!)_2$ respectivamente. Las funciones anteriores son una generalización para 2 argumentos de los catamorfismos definidos en la página 81 y cumplen los siguientes diagramas:

$$\begin{array}{ccc} \mathbb{F}(\mu_1\mathbb{F}\mathbb{G})(\mu_2\mathbb{F}\mathbb{G}) & \xrightarrow{In_1} & \mu_1\mathbb{F}\mathbb{G} \\ {\scriptstyle bimap_{\mathbb{F}}\ (\!|\varphi,\psi|\!)_1\ (\!|\varphi,\psi|\!)_2}\downarrow & & \downarrow{\scriptstyle (\!|\varphi,\psi|\!)_1} \\ \mathbb{F}\alpha\beta & \xrightarrow[\varphi]{} & \alpha \end{array}$$

$$\begin{array}{ccc} \mathbb{G}(\mu_1\mathbb{F}\mathbb{G})(\mu_2\mathbb{F}\mathbb{G}) & \xrightarrow{In_2} & \mu_2\mathbb{F}\mathbb{G} \\ {\scriptstyle bimap_{\mathbb{G}}\ (\!|\varphi,\psi|\!)_1\ (\!|\varphi,\psi|\!)_2}\downarrow & & \downarrow{\scriptstyle (\!|\varphi,\psi|\!)_2} \\ \mathbb{G}\alpha\beta & \xrightarrow[\psi]{} & \beta \end{array}$$

A partir de dichos diagramas se obtienen las definiciones de los bicatamorfismos:

$$\begin{array}{ll} (\!|\_,\_|\!)_1 & : (\mathbb{F}\,\alpha\,\beta \to \alpha) \to (\mathbb{G}\,\alpha\,\beta \to \beta) \to (\mu_1\mathbb{F}\mathbb{G} \to \alpha) \\ (\!|\varphi,\psi|\!)_1\ (In_1\ x) & = \varphi\ (bimap_{\mathbb{F}}\ (\!|\varphi,\psi|\!)_1\ (\!|\varphi,\psi|\!)_2\ x) \end{array}$$

$$\begin{array}{ll} (\!|\_,\_|\!)_2 & : (\mathbb{F}\,\alpha\,\beta \to \alpha) \to (\mathbb{G}\,\alpha\,\beta \to \beta) \to (\mu_2\mathbb{F}\mathbb{G} \to \beta) \\ (\!|\varphi,\psi|\!)_2\ (In_2\ x) & = \psi\ (bimap_{\mathbb{G}}\ (\!|\varphi,\psi|\!)_1\ (\!|\varphi,\psi|\!)_2\ x) \end{array}$$

Ejemplo 5.16 *Tomando los bifunctores $\mathbb{E}$ y $\mathbb{C}$ del ejemplo 5.15 es posible definir una $\mathbb{E},\mathbb{C}$-biálgebra que toma como portadores los tipos* $\mathsf{m}\ Int$ *y* $\mathsf{m}\ ()$ *para una mónada* m *que representa una estructura computacional. El tipo* $\mathsf{m}\ Int$ *indica que la semántica de las expresiones consiste en una computación que devuelve un entero, mientras que el tipo* $\mathsf{m}\ ()$ *indica que los comandos denotan computaciones que no devuelven ningún valor.*

$$\begin{array}{ll} \varphi_{\mathbb{E}} & : \mathbb{E}\ (\mathsf{m}\ Int)\ (\mathsf{m}\ ()) \to (\mathsf{m}\ Int) \\ \varphi_{\mathbb{E}}\ (Num\ n) & = return\ n \\ \varphi_{\mathbb{E}}\ (m_1 + m_2) & = \mathbf{do} \\ & \quad v_1 \leftarrow m_1 \\ & \quad v_2 \leftarrow m_2 \\ & \quad return\ (v_1 + v_2) \end{array}$$

$$\begin{array}{ll} \psi_{\mathbb{C}} & : \mathbb{C}\ (\mathsf{m}\ Int)\ (\mathsf{m}\ ()) \to (\mathsf{m}\ ()) \\ \psi_{\mathbb{C}}\ (m_1\ ;\ m_2) & = \mathbf{do}\ \{\ m_1\ ;\ m_2\ \} \\ \psi_{\mathbb{C}}\ (x := e) & = \mathbf{do} \\ & \quad v \leftarrow e \\ & \quad \varsigma \leftarrow fetch \\ & \quad set\ (upd\ \varsigma\ x\ v) \\ & \quad return\ () \end{array}$$

Mediante un catamorfismo sobre las biálgebras anteriores, se obtiene directamente el intérprete de comandos.

$$\begin{array}{ll} \mathsf{Inter}_{Comm} & : Comm \to \mathsf{m}\ () \\ \mathsf{Inter}_{Comm} & = (\!|\varphi_{\mathbb{E}},\varphi_{\mathbb{C}}|\!)_2 \end{array}$$

Con el fin de facilitar la extensibilidad de las definiciones semánticas, se pueden definir bifunctores polinómicos, en especial, la suma de bifunctores.

Definición 5.13 (Suma de Bifunctores) *Dados dos bifunctores* $\mathbb{F}$ *y* $\mathbb{G}$ *se define un bifunctor* $\mathbb{F} \boxplus \mathbb{G}$ *como:*

$$(\mathbb{F} \boxplus \mathbb{G})\ \alpha\ \beta \triangleq \mathbb{F}\ \alpha\ \beta \parallel \mathbb{G}\ \alpha\ \beta$$

$$\begin{array}{l}\textbf{instance}\ (\mathit{BiFunctor}\ \mathbb{F},\ \mathit{BiFunctor}\ \mathbb{G}) \Rightarrow \mathit{BiFunctor}\ (\mathbb{F} \boxplus \mathbb{G}) \\ \quad \textbf{where} \\ \quad\quad \mathit{bimap}\ f\ g\ (L\ x) = L\ (\mathit{bimap}\ f\ g\ x) \\ \quad\quad \mathit{bimap}\ f\ g\ (R\ x) = R\ (\mathit{bimap}\ f\ g\ x)\end{array}$$

Definición 5.14 (Operador $\boxplus_1$) *Dados dos bifunctores* $\mathbb{F}$ *y* $\mathbb{G}$ *y las funciones* $\phi_1 : \mathbb{F}\ \alpha\ \beta \to \alpha$ *y* $\phi_2 : \mathbb{G}\ \alpha\ \beta \to \alpha$ *se define el operador* $\boxplus_1 : (\mathbb{F} \boxplus \mathbb{G})\ \alpha\ \beta \to \alpha$ *como:*

$$\begin{array}{l}(\phi_1\ \boxplus_1\ \phi_2)\ (L\ x) = \phi_1\ x \\ (\phi_2\ \boxplus_1\ \phi_2)\ (R\ x) = \phi_2\ x\end{array}$$

Definición 5.15 (Operador $\boxplus_2$) *Dados dos bifunctores* $\mathbb{F}$ *y* $\mathbb{G}$ *y las funciones* $\psi_1 : \mathbb{F}\ \alpha\ \beta \to \beta$ *y* $\psi_2 : \mathbb{G}\ \alpha\ \beta \to \beta$ *se define el operador* $\boxplus_2 : (\mathbb{F} \boxplus \mathbb{G})\ \alpha\ \beta \to \beta$ *como:*

$$\begin{array}{l}(\psi_1\ \boxplus_2\ \psi_2)\ (L\ x) = \psi_1\ x \\ (\psi_2\ \boxplus_2\ \psi_2)\ (R\ x) = \psi_2\ x\end{array}$$

Ejemplo 5.17 *Es posible añadir un nuevo tipo de comando al lenguaje del ejemplo 5.16 sin modificar las definiciones realizadas. A modo de ejemplo, se añadirá la sentencia condicional if.*

$$\mathbb{I}\ e\ c\ c \triangleq \textbf{if}\ e\ c\ c$$

$$\begin{array}{l}\textbf{instance}\ \mathit{BiFunctor}\ \mathbb{I} \\ \quad \textbf{where} \\ \quad\quad \mathit{bimap}\ f\ g\ (\textbf{if}\ e\ c_1\ c_2) = \textbf{if}\ (f\ e)\ (g\ c_1)\ (g\ c_2)\end{array}$$

El nuevo lenguaje se define como:

$$\mathit{Comm}' \triangleq \mu_2\mathbb{E}(\mathbb{C} \boxplus \mathbb{I})$$

La semántica de la sentencia if puede definirse como:

$$\begin{array}{lll}\psi_{\mathbb{I}} & : & \mathbb{I}\ (\mathsf{m}\ \mathit{Int})\ (\mathsf{m}\ ()) \to (\mathsf{m}\ ()) \\ \psi_{\mathbb{I}}\ (\textbf{if}\ e\ m_1\ m_2) & = & \textbf{do} \\ & & \quad v \leftarrow e \\ & & \quad \textbf{if}\ v == 0\ \textbf{then} \\ & & \quad\quad m_1 \\ & & \quad \textbf{else} \\ & & \quad\quad m_2\end{array}$$

*De esa forma, el intérprete del lenguaje Imp*1 *sería:*

$$\mathit{Interp}_{\mathit{Comm}'} \triangleq (\!|\varphi_{\mathbb{E}},\ \psi_{\mathbb{C}} \boxplus_2 \psi_{\mathbb{I}}|\!)_2$$

5.9 Extensión de Functores a BiFunctores

La extensión de un functor a un bifunctor será requerida, por ejemplo, para reutilizar expresiones (definidas a partir de un functor unario) en un lenguaje imperativo que contiene expresiones y comandos.

La extensión prescinde de uno de los argumentos.

Definición 5.16 (Extensiones de functor a bifunctor) *Dado un functor* F *se definen los bifunctores* F_1^2 *y* F_2^2*) como:*

$$\mathsf{F}_1^2\ \alpha\ \beta\ \triangleq\ \mathsf{F}\ \alpha$$
$$\mathsf{F}_2^2\ \alpha\ \beta\ \triangleq\ \mathsf{F}\ \beta$$

instance (*Functor* f) $\Rightarrow$ *BiFunctor* f_1^2 *where*
  where
    bimap $f\ g\ x\ = f\ x$

instance (*Functor* f) $\Rightarrow$ *BiFunctor* f_2^2 *where*
  where
    bimap $f\ g\ x\ = g\ x$

Ejemplo 5.18 *A partir de las especificaciones de expresiones (ejemplo 5.6) y de comandos (ejemplo 5.17) se puede definir un lenguaje imperativo simple* $\mathcal{L}_{Imp}$

$$\mathcal{L}_{Imp}\ \triangleq\ \mu_2((\mathsf{E} \oplus \mathsf{V})_1^2\ (\mathbb{I} \boxplus \mathbb{C}))$$

Para extender una F-álgebra a una F_1^2-biálgebra se define el operador ϵ_1^2.

$$\epsilon_1^2\ :\ (\mathsf{F}\ \alpha \to \alpha) \to (\mathsf{F}_1^2\ \alpha\ \beta \to \alpha)$$
$$\epsilon_1^2\ \varphi\ x\ =\ \varphi\ x$$

De la misma forma, el operador ϵ_2^2 permite extender una F-álgebra a una F_2^2-biálgebra.

$$\epsilon_2^2\ :\ (\mathsf{F}\ \beta \to \beta) \to (\mathsf{F}_2^2\ \alpha\ \beta \to \beta)$$
$$\epsilon_2^2\ \varphi\ x\ =\ \varphi\ x$$

Ejemplo 5.19 *Es posible obtener de forma directa un intérprete para el lenguaje del ejemplo 5.18 mediante el siguiente bicatamorfismo*

$$\mathsf{Inter}_{\mathcal{L}_{Imp}}\ =\ (\!|\epsilon_1^2(\varphi_{\mathsf{E}} \oplus \varphi_{\mathsf{V}}), \varphi_{\mathbb{C}} \boxplus_2 \varphi_{\mathbb{I}}|\!)_2$$

5.10 Clases Genéricas

La versión 5.00 del compilador GHC de *Haskell* incorpora una implementación que permite definir clases genéricas siguiendo [83]. Dicha implementación tiene actualmente varias limitaciones: sólo se permiten clases de tipos regulares y no se permiten clases de constructores de tipos ni con múltiples parámetros. De esa forma, las definiciones que se incluyen a continuación no son admitidas actualmente, aunque se espera que lo sean en futuras versiones del compilador.

La clase genérica *Functor* podría definirse como:

$$
\begin{array}{l}
\textbf{class}\ \mathit{Functor}\ \mathsf{f} \\
\quad \textbf{where} \\
\quad\quad \begin{array}{ll}
\mathit{map} & :\ (\alpha \to \beta) \to (\mathsf{f}\ \alpha \to \mathsf{f}\ \beta) \\
\mathit{map}\ \{\!|\lambda\ \alpha.\alpha|\!\}\ h\ x & = h\ x \\
\mathit{map}\ \{\!|\lambda\ \alpha.\mathit{Int}|\!\}\ h\ x & = x \\
\mathit{map}\ \{\!|\lambda\ \alpha.f\ \alpha : + : g\ \alpha|\!\}\ h\ (\mathit{Inl}\ x) & = \mathit{Inl}\ (\mathit{map}\ h\ x) \\
\mathit{map}\ \{\!|\lambda\ \alpha.f\ \alpha : + : g\ \alpha|\!\}\ h\ (\mathit{Inr}\ y) & = \mathit{Inr}\ (\mathit{map}\ h\ y) \\
\mathit{map}\ \{\!|\lambda\ \alpha.f\ \alpha : * : g\ \alpha|\!\}\ h\ (x, y) & = (\mathit{map}\ h\ x : * : \mathit{map}\ h\ y)
\end{array}
\end{array}
$$

El parámetro marcado $\{\!|\_|\!\}$ indica un patrón de tipo. Existen diversos patrones, $\lambda\ \alpha.\alpha$ indica el patrón identidad, $\lambda\ \alpha.\mathit{Int}$, el patrón constante (en este caso un valor de tipo entero), $\lambda\ \alpha.\ f\ \alpha : + : g\ \alpha$ la suma disjunta que puede tomar el valor izquierdo (Inl) o derecho (Inr), y $\lambda\ \alpha.f\ \alpha : * : g\ \alpha$ el producto.

A partir de la definición anterior, cualquier functor regular puede declararse como una instancia de la clase *Functor* obteniéndose de forma automática la correspondiente función *map*. La derivación automática de estas funciones libera al implementador de tener que declarar dichas definiciones evitando una posible fuente de errores.

La clase *MFunctor* definida en 5.6 también podría definirse de forma genérica.

$$
\begin{array}{l}
\textbf{class}\ (\mathit{Monad}\ \mathsf{m}) \Rightarrow\ \mathit{MFunctor}\ \mathsf{m}\ \mathsf{f} \\
\quad \textbf{where} \\
\quad\quad \begin{array}{ll}
\mathsf{map_m} & :\ (\alpha \to \mathsf{m}\ \beta) \to (\mathsf{f}\ \alpha \to \mathsf{m}\ (\mathsf{f}\ \beta)) \\
\mathsf{map_m}\ \{\!|\lambda\ \alpha.\alpha|\!\}\ h\ x & = h\ x \\
\mathsf{map_m}\ \{\!|\lambda\ \alpha.\mathit{Int}|\!\}\ h\ x & = \mathit{return}\ x \\
\mathsf{map_m}\ \{\!|\lambda\ \alpha.f\ \alpha : + : g\ \alpha|\!\}\ h\ (\mathit{Inl}\ x) & = \mathsf{map_m}\ h\ x \gg\!= (\mathit{return}\ .\ \mathit{Inl}) \\
\mathsf{map_m}\ \{\!|\lambda\ \alpha.f\ \alpha : + : g\ \alpha|\!\}\ h\ (\mathit{Inr}\ y) & = \mathsf{map_m}\ h\ x \gg\!= (\mathit{return}\ .\ \mathit{Inr}) \\
\mathsf{map_m}\ \{\!|\lambda\ \alpha.f\ \alpha : * : g\ \alpha|\!\}\ h\ (x, y) & = \textbf{do} \\
 & \qquad v \leftarrow \mathsf{map_m}\ h\ x \\
 & \qquad w \leftarrow \mathsf{map_m}\ h\ y \\
 & \qquad \mathit{return}\ (v : * : w)
\end{array}
\end{array}
$$

Obsérvese que la definición anterior supone un orden explícito de evaluación de subcomponentes de izquierda a derecha. En ocasiones, este orden podría no ser deseable, siendo necesaria una redefinición.

Finalmente, la clase *BiFunctor* definida en 5.8 podría definirse como

$$
\textbf{class}\ \mathit{BiFunctor}\ \mathbb{F}
$$

where

$$
\begin{aligned}
&bimap\ :\ (\alpha \to \gamma) \to (\beta \to \delta) \to (\mathbb{F}\,\alpha\,\beta \to \mathbb{F}\,\gamma\,\delta)\\
&bimap\ \{\!|\lambda\alpha\beta.\alpha|\!\}\ h\ h'\ x\ =\ h\ x\\
&bimap\ \{\!|\lambda\alpha\beta.\beta|\!\}\ h\ h'\ x\ =\ h'\ x\\
&bimap\ \{\!|\lambda\alpha\beta.Int|\!\}\ h\ h'\ x\ =\ x\\
&bimap\ \{\!|\lambda\alpha\beta.f\ \alpha\ \beta : + : g\ \alpha\ \beta|\!\}\ h\ h'\ (Inl\ x)\ =\ Inl\ (bimap\ h\ h'\ x)\\
&bimap\ \{\!|\lambda\alpha\beta.f\ \alpha\ \beta : + : g\ \alpha\ \beta|\!\}\ h\ h'\ (Inr\ y)\ =\ Inr\ (bimap\ h\ h'\ y)\\
&bimap\ \{\!|\lambda\alpha\beta.f\ \alpha\ \beta : * : g\ \alpha\ \beta|\!\}\ h\ h'\ (x, y)\ =\ (bimap\ h\ h'\ x\ :*:\ bimap\ h\ h'\ y)
\end{aligned}
$$

Capítulo 6

Sistema de prototipado de Lenguajes

> *Language designers also have an obligation to provide languages that encourage good style, since we all know that style is strongly influenced by the language in which it is expressed.* Knuth [125]

6.1 Introducción

En este capítulo se describirá el sistema de prototipado de lenguajes que forma el núcleo de la presente tesis doctoral. El sistema se basa en la integración entre la semántica monádica modular, descrita en el capítulo 4, y los conceptos de catamorfismos y catamorfismos monádicos tomados de la programación genérica y descritos en el capítulo 5.

6.2 Arquitectura del sistema

En la figura 6.1 se presenta un esquema general del sistema de prototipado de lenguajes desarrollado.

El sistema consta fundamentalmente de los siguientes bloques:

- Una serie de descripciones de diferentes lenguajes de programación. El usuario puede añadir nuevos lenguajes incluyendo el analizador sintáctico, el sistema de impresión (*pretty printing*) y la especificación semántica.
- El *Marco interactivo* permite seleccionar e interpretar los diferentes lenguajes de programación definidos.
- Para la descripción de cada lenguaje, el usuario puede acceder a una serie de módulos con herramientas comunes (functores polinómicos, álgebras, catamorfismos y catamorfismos monádicos, etc.)
- Finalmente, los bloques semánticos permitirán definir la estructura computacional. Se incluye una librería de bloques semánticos predefinidos (con los transformadores de mónadas correspondientes), aunque el usuario puede definir sus propios bloques semánticos si lo requiere.

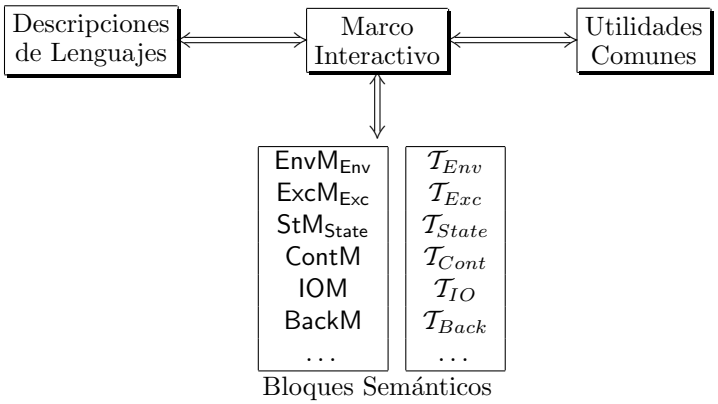


Figura 6.1: Sistema de Prototipado de Lenguajes

Para añadir la descripción semántica de un lenguaje particular se deben realizar las siguientes acciones:

- Definir la estructura computacional accediendo a los bloques semánticos y buscando el tipo espcífico de mónada. Este tipo de mónada se puede obtener mediante composición de varios transformadores de mónadas
- Describir el dominio de valores. Generalmente, se utilizarán uniones extensibles que permitirán añadir o eliminar fácilmente conjuntos de valores.
- Establecer la estructura sintáctica. Preferiblemente, se utilizarán puntos fijos de uniones de functores, que permitirán añadir nuevos componentes sintácticos posteriormente
- Definición de las funciones semánticas mediante álgebras ó álgebras monádicas.

A partir de las definiciones anteriores, se obtiene un intérprete de forma automática del lenguaje correspondiente que podrá ser ejecutado en el marco interactivo, y seguirá la especificación semántica utilizada.

En las siguientes secciones se detallan cada una de las partes anteriores.

6.3 Estructura computacional

La estructura computacional será una mónada M que podrá definirse como: $\mathsf{M} = (\mathcal{T}_1.\mathcal{T}_2 \ldots \mathcal{T}_n)\mathsf{M}'$ donde $\mathcal{T}_i$ es un transformador monádico que añade una determinada noción computacional y M' es una mónada básica.

El principal problema en la definición de la estructura computacional es la interacción entre transformadores de mónadas. En [145] se indican algunas interacciones entre el transformador que añade no determinismo y el que añade continuaciones.

6.4 Componentes Sintácticos

Para cada lenguaje que se quiera especificar, deberá definirse una sintaxis abstracta. Dicha sintaxis abstracta será un tipo de datos recursivo que se de-

finirá como el menor punto fijo de un functor no recursivo F que marcará la estructura del lenguaje.

Normalmente, este functor será la suma de n functores que representan las diferentes partes del lenguaje.

Ejemplo 6.1 *Supóngase que se va a definir un lenguaje de expresiones aritméticas y booleanas. En lugar de definir un tipo de datos recursivo monolítico, se definen functores independientes para las expresiones aritméticas y para las expresiones booleanas.*

$$\mathsf{A}\, x \triangleq \mathit{Num}\, \mathit{Int} \mid x + x \mid x - x \mid x \times x \mid x \div x$$

$$\mathsf{B}\, x \triangleq B\, \mathit{Bool} \mid x \wedge x \mid x \vee x$$

A partir de dichas definiciones, mediante el punto fijo de la unión de ambos functores, se obtiene el lenguaje $\mathcal{L}_{AB}$ deseado

$$\mathcal{L}_{AB} \triangleq \mu\,(A \oplus B)$$

Si se desean añadir comparaciones, por ejemplo, sólo es necesario declarar el functor correspondiente

$$\mathsf{C}\, x \triangleq x > x \mid x < x \mid x \leq x \mid x \geq x \mid x == x \mid x \neq x$$

y definir el nuevo lenguaje

$$\mathcal{L}_{ABC} \triangleq \mu\,(A \oplus B \oplus C)$$

6.5 Componentes Semánticos

Una vez que la sintaxis del lenguaje ha sido dividida en diferentes functores, para obtener la especificación semántica puede realizarse especificando la semántica de cada componente de forma independiente.

Para cada functor F_i, se definirá una F_i-Álgebra φ_{F_i} sobre la estructura computacionl. El intérprete se obtiene de forma automática mediante un catamorfismo sobre la suma de las F_i-álgebras definidas.

Ejemplo 6.2 *El intérprete del lenguaje definido en el ejemplo 6.1 se obtiene de forma automática mediante:*

$$\mathsf{Inter}_{\mathcal{L}_{ABC}} = (\!|\varphi_{\mathsf{A}} \oplus \varphi_{\mathsf{B}} \oplus \varphi_{\mathsf{C}}|\!)$$

En el caso de componentes sintácticos cuya semántica requiera la evaluación recursiva de subcomponentes puede avanzarse un paso más en la separación de componentes mediante la utilización de catamorfismos monádicos (véase sección 5.6). A la hora de definir las álgebras monádicas, se parte de valores ya evaluados en lugar de computaciones.

En caso de dividir el lenguaje en varias categorías sintácticas. Se definen varios n-functores y se aplica la teoría desarrollada en la sección 5.8 sobre n-catamorfismos.

6.6 Marco Interactivo

El sistema de prototipado de lenguajes consta de un subsistema que permite cargar e interpretar programas escritos en distintos lenguajes de programación de forma interactiva.

Para ello, es necesario definir un analizador sintáctico para cada lenguaje así como una función de impresión de las expresiones de cada lenguaje. Una vez definidos, el lenguaje puede integrarse en el marco de ejecución interactivo.

6.6.1 Combinadores de Analizadores Sintácticos

Aunque no forma parte del núcleo principal de este trabajo, la descripción de la estructura sintáctica y del correspondiente analizador sintáctico, es una parte fundamental de la especificación de cualquier lenguaje de programación. Entre las principales ventajas de los lenguajes funcionales se encuentra la posibilidad de utilizar combinadores o funciones de orden superior que pueden encapsular tareas complejas. Las librerías de combinadores para construir analizadores sintácticos recursivos descendentes permiten utilizar una notación cercana a la notación BNF. De esa forma, construir un analizador sintáctico es tan sencillo como desarrollar la gramática libre de contexto. Este tipo de librerías se conocen como combinadores de analizadores sintácticos (*parser combinators*) y su estudio reciente ha sido fuente de numerosas publicaciones [57, 201, 130] En [97, 98] se propone la utilización de mónadas para estructurar tales combinadores y recientemente, J. Hughes, ha propuesto una generalización de las mónadas para aumentar subsanar algunos problemas de eficiencia de tales librerías [94].

En el Sistema de Prototipado de Lenguajes se ha optado por utilizar la librería *Parsec* desarrollada por D. Leijen [143] inspirándose en los trabajos mencionados.

La librería define una mónada $Parser\ \alpha$ que encapsula analizadores sintácticos de valores de tipo α. Además de las operaciones *return* y ($\gg=$) propias de la mónada, define diversas operaciones de utilidad, entre las que se pueden destacar[1].

$parse : Parser\ \alpha \rightarrow String \rightarrow \alpha$	$parse\ p\ s$ ejecuta el analizador p sobre la entrada s devolviendo el valor analizado
$char : Char \rightarrow Parser\ ()$	$char\ c$ comprueba si a la entrada aparece c
$(\mid) : Parser\ \alpha \rightarrow Parser\ \alpha \rightarrow Parser\ \alpha$	$p_1 \mid p_2$ devuelve el resultado de p_1 salvo que se produzca un error, en cuyo caso devuelve el resultado de p_2
$reserved : String \rightarrow Parser\ ()$	$reserved\ s$ comprueba si a la entrada aparece la palabra reservada s
$sepBy : Parser\ \alpha \rightarrow Parser\ s \rightarrow Parser\ [\alpha]$	p ‘$sepBy$‘ s comprueba cero o más apariciones de p separadas por s, devuelve una lista de las apariciones encontradas

Ejemplo 6.3 *A continuación se presenta un analizador sintáctico de un lenguaje imperativo simple. La sintaxis abstracta viene dada por*

[1] El tipo de algunas funciones se ha simplificado en esta presentación

$$Comm \triangleq While\ Expr\ Comm \mid IF\ Expr\ Comm\ Comm \mid String := Expr \mid Seq\ Comm\ Comm \mid Skip$$

Suponiendo que ya se ha definido el analizador sintáctico de expresiones $expr : Parser\ Expr$. *El analizador de comandos sería*

```
comm : Parser Comm
comm = do
         cs ← simpleComm `sepBy` (char ';')
         return (foldr Seq Skip cs)


simpleComm : Parser Comm
simpleComm = while | if | assign

while = do
          reserved "while"
          e ← expr
          reserved "do"
          c ← comm
          return (While e c)
if = do
       reserved "if"
       e ← expr
       reserved "then"
       c1 ← comm
       reserved "else"
       c2 ← comm
       return (If e c1 c2)
assign = do
           x ← identifier
           reserved " :="
           e ← expr
           return (x := e)
```

6.6.2 Combinadores de Impresión

En [95], J. Hughes deriva una librería de combinadores de impresión. Los combinadores de impresión permiten imprimir la sintaxis abstracta del lenguaje de una forma *bonita* teniendo en cuenta el tamaño de la página y diversas consideraciones estéticas.

Dicha librería define un tipo abstracto *Doc* que representa documentos a imprimir e incluye, entre otros, los siguientes combinadores.

$render : Doc \rightarrow String$	$render\ d$ convierte un documento en una representación textual según los valores estéticos por defecto. Existe una función *fullRender* que permite modificar dichos valores
$text : String \rightarrow Doc$	$text\ s$ convierte la cadena de caracteres s en un documento
$(<+>) : Doc \rightarrow Doc \rightarrow Doc$	$d_1 <+> d_2$ junta horizontalmente los documentos d_1 y d_2 separándolos por un espacio
$nest : Int \rightarrow Doc \rightarrow Doc$	$nest\ n\ d$ aplica una sangría de n espacios al documento d si es necesario
$sep : [Doc] \rightarrow Doc$	$sep\ ds$ enlaza los documentos ds horizontal o verticalmente dependiendo de las consideraciones estéticas planteadas
$empty : Doc$	$empty$ representa un documento vacío

Ejemplo 6.4 *Una posible aplicación a la impresión del lenguaje imperativo de la sección anterior sería la siguiente*

$$
\begin{array}{l}
pComm \quad : \ Comm \ \rightarrow \ Doc \\
pComm\ Skip \quad = empty \\
pComm\ (Seq\ c_1\ c_2) \ = \\
\quad sep\ [pComm\ c_1 <+> text\text{“;”}, pComm\ c_2] \\
pComm\ (While\ e\ c) \ = \\
\quad sep\ [text\text{“while”} <+> pExpr\ e <+> text\text{“do”} \\
\qquad ,nest\ 3\ (pComm\ c)] \\
pComm\ (If\ e\ c_1\ c_2) \ = \\
\quad sep\ [text\text{“if”} <+> pExpr\ e \\
\qquad ,text\ \text{“then”} \\
\qquad ,nest\ 3\ (pComm\ c_1) \\
\qquad ,text\ \text{“else”} \\
\qquad ,nest\ 3\ (pComm\ c_2)] \\
pComm\ (x\ :=\ e) \quad = text\ x\ <+>\ text\ \text{“}:=\text{”}\ <+>\ pExpr\ e
\end{array}
$$

El combinador $pExpr : Doc \rightarrow Expr$ *se definiría de forma similar.*

6.6.3 Integración de Lenguajes

Con el fin de seleccionar el lenguaje que se desea interpretar en tiempo de ejecución, es necesario almacenar en una misma estructura los diferentes lenguajes para poder ejecutar la función de interpretación correspondiente. En la práctica, este diseño tiene un problema, los tipos de los lenguajes y de sus funciones de interpretación son diferentes y no puede utilizarse la misma estructura en *Haskell*.

Para salvar este escollo, se ha recurrido a la combinación de tipos existenciales con clases de tipos siguiendo el esquema propuesto por [140]. Dicho esquema se presenta brevemente en B.8.1.

A continuación se resumen las estructuras básicas requeridas. Se declara una clase de tipos *Syntax*

```
class Syntax s
  where
    exec : s → IO ()
```

y un tipo de datos $\mathcal{L}$ con un componente existencial

$$\mathcal{L} \triangleq \forall\, s\,.\,(Syntax\ s) \Rightarrow MkL\ s$$

y se declara la instancia general

```
instance Syntax L
  where
    exec (MkL s) = exec s
```

El sistema utiliza una única estructura en la que almacena elementos de tipo $\mathcal{L}$. Para añadir un nuevo lenguaje, sólo es necesario que sea una instancia de la clase *Syntax*. Por ejemplo, supóngase que se desea añadir el lenguaje $LABC$ definido anteiormente.

6.6.4 Marco común de ejecución

El Marco de Ejecución ofrece un sistema interactivo de interpretación de lenguajes. El sistema puede configurarse con una lista de lenguajes $L_s = [l_1, l_2, \ldots, l_n]$. En cada momento contiene un lenguaje de programación activo $l_i \in L_s$ y permite las siguientes opciones:

- Cargar un programa p_i en el lenguaje de programación activo l_i
- Ejecutar el programa p_i
- Seleccionar un nuevo lenguaje de programación activo $l_{i'} \in L_s$
- Interrumpir y depurar el programa que se está ejecutando en un momento dado
- Mostrar información del programa cargado p_i
- Mostrar información del lenguaje activo L_i

6.6.5 Utilidades comunes

Con el fin de facilitar las descripciones semánticas, se incluyen una serie de módulos comunes de utilidades. A continuación se describen algunas de dichas funciones que se utilizarán en los siguientes capítulos.

6.6.5.1 Gestión de Memoria Dinámica

El módulo *Heap* permite simular un sistema de gestión de memoria mediante montículo. El módulo define un tipo abstracto de datos *Heap* α que indica montículos direccionables.

Las direcciones serán valores de tipo *Loc*. Se incluyen las siguientes funciones:

$alloc_H : \alpha \to Heap\ \alpha \to (Loc, Heap\ \alpha)$	$alloc_H\ v\ h$ inserta v en h. Devuelve el nuevo montículo junto con la dirección del valor insertado
$lkp_H : Loc \to Heap\ \alpha \to \alpha$	$lkp_H\ l\ h$ devuelve el valor de la dirección l en h
$upd_H : Loc \to \alpha \to Heap\ \alpha \to Heap\ \alpha$	$upd_H\ l\ v\ h$ actualiza la dirección l de h con el valor v

6.6.5.2 Tabla de símbolos

El módulo *Table* permite definir tablas de símbolos. Define un tipo abstracto de datos $Table\ \alpha$. Los símbolos serán valores de tipo *Name*.

$lkp_T : Name \to Table\ \alpha \to \alpha$	$lkp_T\ n\ t$ = valor de n en t.
$upd_T : Name \to \alpha \to Table\ \alpha \to Table\ \alpha$	$upd_T\ n\ v\ t$ = tabla obtenida al insertar n con valor v en t
$(+\!\!+) : Table\ \alpha \to Table\ \alpha \to Table\ \alpha$	$t_1 +\!\!+ t_2$ = tabla resultante de unir t_1 y t_2

6.7 Especificación del ejemplo

En esta sección se realizará una especificación de los ejemplos de lenguajes definidos en 3.2.

6.7.0.3 Lenguaje $\mathcal{L}_0$

El lenguaje $\mathcal{L}_0$ está formado por expresiones aritméticas. La especificación semántica se realizará definiendo diferentes functores por cada concepto del lenguaje.

Se define el functor E para las expresiones aritméticas

$$\mathsf{E}\ x \triangleq Num\ Int \mid x + x \mid x - x$$

El dominio de valores seá el conjunto de enteros

$$Value \triangleq Int$$

y la estructura computacional se puede definirse como la mónada identidad.

$$\mathsf{Comp} \triangleq \mathsf{IdM}$$

Las definiciones semánticas serán

$$\begin{aligned}
\varphi_{\mathsf{E}} &: \mathsf{E}(\mathsf{Comp}\ Value) \to \mathsf{Comp}\ Value \\
\varphi_{\mathsf{E}}\ (Num\ n) &= return\ n \\
\varphi_{\mathsf{E}}\ (e_1 + e_2) &= \mathbf{do} \\
&\qquad v_1 \leftarrow e_1 \\
&\qquad v_2 \leftarrow e_2 \\
&\qquad return\ \uparrow (\downarrow v_1 + \downarrow v_2) \\
\varphi_{\mathsf{E}}\ (e_1 - e_2) &= \mathbf{do} \\
&\qquad v_1 \leftarrow e_1 \\
&\qquad v_2 \leftarrow e_2 \\
&\qquad return\ \uparrow (\downarrow v_1 - \downarrow v_2)
\end{aligned}$$

El lenguaje $\mathcal{L}_0$ puede definirse como el punto fijo del functor E

$$\mathcal{L}_0 \triangleq \mu\mathsf{E}$$

obteniendo directamente un prototipo para $\mathcal{L}_0$ mediante un catamorfismo sobre el álgebra φ_{E} definida anteriormente

$$\begin{aligned}&\mathsf{Inter}_{\mathcal{L}_0} \;:\; \mathcal{L}_0 \to \mathsf{Comp}\ \mathit{Value}\\ &\mathsf{Inter}_{\mathcal{L}_0} \;=\; (\!|\mathsf{E}|\!)\end{aligned}$$

6.7.0.4 Lenguaje $\mathcal{L}_1$

El lenguaje $\mathcal{L}_1$ añade multiplicaciones y divisiones a $\mathcal{L}_0$. Para su modelización, se define un nuevo functor que capture la sintaxis abstracta de ambos elementos.

$$\mathsf{F}\ x \triangleq x \times x \mid x \div x$$

La incorporación de divisiones supone la aparición de posibles errores. Para modelizar dichas situaciones se modifica la estructura computacional

$$\mathsf{Comp} \triangleq \mathcal{T}_{Err}\ \mathsf{IdM}$$

y se especifica la F-álgebra φ_{F} como

$$\begin{aligned}&\varphi_{\mathsf{F}} \;:\; \mathsf{F}(\mathsf{Comp}\ \mathit{Value}) \to \mathsf{Comp}\ \mathit{Value}\\ &\varphi_{\mathsf{F}}\ (e_1 \times e_2) = \mathbf{do}\\ &\qquad v_1 \leftarrow e_1\\ &\qquad v_2 \leftarrow e_2\\ &\qquad \mathit{return}\ \uparrow (\downarrow v_1 * \downarrow v_2)\\ &\varphi_{\mathsf{F}}\ (e_1 \div e_2) = \mathbf{do}\\ &\qquad v_1 \leftarrow e_1\\ &\qquad v_2 \leftarrow e_2\\ &\qquad \mathbf{if}\ \downarrow v_2 = 0\ \mathbf{then}\\ &\qquad\quad \mathit{err}\ \text{“division by zero”}\\ &\qquad \mathbf{else}\\ &\qquad\quad \mathit{return}\ \uparrow (\downarrow v_1 / \downarrow v_2)\end{aligned}$$

El lenguaje $\mathcal{L}_1$ se puede definir como el punto fijo de la unión de los dos functores.

$$\mathcal{L}_1 \triangleq \mu(\mathsf{E} \oplus \mathsf{F})$$

y el intérprete prototipo de $\mathcal{L}_1$ se obtiene directamente mediante

$$\begin{aligned}&\mathsf{Inter}_{\mathcal{L}_1} \;:\; \mathcal{L}_1 \to \mathsf{Comp}\ \mathit{Value}\\ &\mathsf{Inter}_{\mathcal{L}_1} \;=\; (\!|\varphi_{\mathsf{E}} \oplus \varphi_{\mathsf{F}}|\!)\end{aligned}$$

6.7.0.5 Lenguaje $\mathcal{L}_2$

En $\mathcal{L}_2$ se añaden expresiones booleanas. Para su modelización se define el functor B.

$$\mathsf{B}\ x \triangleq \mathsf{True} \mid \mathsf{False} \mid x == x \mid x < x$$

La incorporación de valores booleanos requiere modificar el dominio de valores

$$\mathit{Value} \triangleq \mathit{Int} \parallel \mathit{Bool}$$

y definir el B-álgebra correspondiente

$$\begin{array}{lll}
\varphi_{\mathsf{B}} & : & \mathsf{B}(\mathsf{Comp}\ \mathit{Value}) \to \mathsf{Comp}\ \mathit{Value} \\
\varphi_{\mathsf{B}}\ (\mathit{True}) & = & \mathit{return} \uparrow \mathit{True} \\
\varphi_{\mathsf{B}}\ (\mathit{False}) & = & \mathit{return} \uparrow \mathit{False} \\
\varphi_{\mathsf{B}}\ (e_1 == e_2) & = & \mathbf{do} \\
 & & \quad v_1 \leftarrow e_1 \\
 & & \quad v_2 \leftarrow e_2 \\
 & & \quad \mathit{return} \uparrow (\downarrow v_1 =\downarrow v_2) \\
\varphi_{\mathsf{B}}\ (e_1 < e_2) & = & \mathbf{do} \\
 & & \quad v_1 \leftarrow e_1 \\
 & & \quad v_2 \leftarrow e_2 \\
 & & \quad \mathit{return} \uparrow (\downarrow v_1 <\downarrow v_2)
\end{array}$$

El lenguaje $\mathcal{L}_2$ se define entonces como

$$\mathcal{L}_2 \triangleq \mu(\mathsf{E} \oplus \mathsf{F} \oplus \mathsf{B})$$

y el intérprete de $\mathcal{L}_2$ se obtiene como

$$\begin{array}{l}
\mathsf{Inter}_{\mathcal{L}_2} : \mathcal{L}_2 \to \mathsf{Comp}\ \mathit{Value} \\
\mathsf{Inter}_{\mathcal{L}_2} = (\!|\varphi_{\mathsf{E}} \oplus \varphi_{\mathsf{F}} \oplus \varphi_{\mathsf{B}}|\!)
\end{array}$$

6.7.0.6 Lenguaje $\mathcal{L}_3$

Para la definición del lenguaje $\mathcal{L}_3$ se añade el functor

$$\mathsf{V}\ x \triangleq \mathit{Var}\ \mathit{String}$$

El valor de una variables debe buscarse en el entorno. La modelización de este concepto semántico se realiza transformando la mónada anterior en una mónada lectora del entorno.

$$\mathsf{Comp} \triangleq (\mathcal{T}_{\mathit{Env}} \,.\, \mathcal{T}_{\mathit{Err}})\ \mathsf{IdM}$$

$$\begin{array}{lll}
\varphi_{\mathsf{V}} & : & \mathsf{V}(\mathsf{Comp}\ \mathit{Value}) \to \mathsf{Comp}\ \mathit{Value} \\
\varphi_{\mathsf{V}}\ (\mathit{Var}\ x) & = & \mathbf{do} \\
 & & \quad \rho \leftarrow \mathit{rdEnv} \\
 & & \quad \mathit{return}(\mathit{lkp}\ \rho\ x)
\end{array}$$

El lenguaje $\mathcal{L}_3$ se define como

$$\mathcal{L}_3 \triangleq \mu(\mathsf{E} \oplus \mathsf{F} \oplus \mathsf{B} \oplus \mathsf{V})$$

y su intérprete se obtiene de forma directa

$$\begin{aligned}&\mathsf{Inter}_{\mathcal{L}_3} \;:\; \mathcal{L}_3 \to \mathsf{Comp}\ \mathit{Value}\\ &\mathsf{Inter}_{\mathcal{L}_3} = (\!|\varphi_{\mathsf{E}} \oplus \varphi_{\mathsf{F}} \oplus \varphi_{\mathsf{B}} \oplus \varphi_{\mathsf{V}}|\!)\end{aligned}$$

6.7.0.7 Lenguaje $\mathcal{L}_4$

La incorporación de comandos supone utilizar dos categorías sintácticas. Se define el bifunctor $\mathbb{S}$ como

$$\mathbb{S}\ e\ c \triangleq \mathit{Skip} \mid c\,;\,c \mid \mathit{String} := e$$

A nivel semántico, la posibilidad de utilizar un estado modificable durante la ejecución, requiere la utilización del transformador de mónadas que añade estados actualizables.

$$\mathsf{Comp} \triangleq (\mathcal{T}_{\mathit{State}}\,.\,\mathcal{T}_{\mathit{Err}})\,\mathsf{IdM}$$

Se define la biálgebra

$$\begin{aligned}&\psi_{\mathbb{S}} && : \mathbb{S}\,(\mathsf{Comp}\ \mathit{Value})\,(\mathsf{Comp}\,())\ \to\ (\mathsf{Comp}\,())\\ &\psi_{\mathbb{S}}\,(\mathit{Skip}) && = \mathit{return}\,()\\ &\psi_{\mathbb{S}}\,(c_1\,;\,c_2) && = \mathbf{do}\,\{\,c_1\,;\,c_2\,\}\\ &\psi_{\mathbb{S}}\,(x := e) && = \mathbf{do}\\ &&&\qquad v \leftarrow e\\ &&&\qquad \varsigma \leftarrow \mathit{fetch}\\ &&&\qquad \mathit{set}\,(\mathit{upd}\ \varsigma\ x\ v)\\ &&&\qquad \mathit{return}\,()\end{aligned}$$

La definición del lenguaje $\mathcal{L}_4$ requiere un bifunctor que defina las expresiones. Aunque es posible definir dicho bifunctor, parece más interesante reutilizar las definiciones de las secciones anteriores. Mediante las definiciones de la sección 5.9, se extiende el functor patrón del lenguaje $\mathcal{L}_3$ a un bifunctor.

$$\mathbb{E} \triangleq (\mathsf{E} \oplus \mathsf{F} \oplus \mathsf{B} \oplus \mathsf{V})^2_1$$

con la correspondiente biálgebra

$$\begin{aligned}&\varphi_{\mathbb{E}} \;:\; \mathbb{E}\,(\mathsf{Comp}\ \mathit{Value})\,(\mathsf{Comp}\,()) \to \mathsf{Comp}\ \mathit{Value}\\ &\varphi_{\mathbb{E}} = \epsilon^2_1\,(\varphi_{\mathsf{E}} \oplus \varphi_{\mathsf{F}} \oplus \varphi_{\mathsf{B}} \oplus \varphi_{\mathsf{V}})\end{aligned}$$

con lo que el lenguaje $\mathcal{L}_4$ se define fácilmente como el punto fijo de los bifunctores $\mathbb{E}$ y $\mathbb{S}$

$$\mathcal{L}_4 \triangleq \mu_2\mathbb{E}\mathbb{S}$$

El intérprete de $\mathcal{L}_4$ será un bicatamorfismo (véase 5.8)

$$\begin{aligned}&\mathsf{Inter}_{\mathcal{L}_4} \;:\; \mathcal{L}_4 \to \mathsf{Comp}\,()\\ &\mathsf{Inter}_{\mathcal{L}_4} = (\!|\varphi_{\mathbb{E}}, \varphi_{\mathbb{S}}|\!)_2\end{aligned}$$

6.7.0.8 Lenguaje $\mathcal{L}_5$

En el lenguaje $\mathcal{L}_5$ se incorporan sentencias de control como el condicional y la repetición. Se utilizará el bifunctor $\mathbb{C}$

$$\mathbb{C}\ e\ c \triangleq \mathit{If}\ e\ c\ c \mid \mathit{While}\ e\ c$$

junto con la definición

$$\begin{array}{l}
\psi_{\mathbb{C}} \quad : \mathbb{C}\,(\mathsf{Comp}\ \mathit{Value})\,(\mathsf{Comp}\,())\ \rightarrow\ (\mathsf{Comp}\,()) \\
\psi_{\mathbb{C}}\,(\mathit{If}\ e\ c_1\ c_2)\ = \mathbf{do} \\
\qquad\qquad\qquad\qquad v\ \leftarrow\ e \\
\qquad\qquad\qquad\qquad \mathbf{if}\ v\ \mathbf{then} \\
\qquad\qquad\qquad\qquad\quad c_1 \\
\qquad\qquad\qquad\qquad \mathbf{else} \\
\qquad\qquad\qquad\qquad\quad c_2 \\
\psi_{\mathbb{C}}\,(\mathit{While}\ e\ c)\ = \mathit{loop} \\
\quad \mathbf{where} \\
\qquad \mathit{loop}\ =\ \mathbf{do} \\
\qquad\qquad\qquad v\ \leftarrow\ e \\
\qquad\qquad\qquad \mathbf{if}\ v\ \mathbf{then} \\
\qquad\qquad\qquad\quad \mathbf{do} \\
\qquad\qquad\qquad\qquad c \\
\qquad\qquad\qquad\qquad \mathit{loop} \\
\qquad\qquad\qquad \mathbf{else} \\
\qquad\qquad\qquad\quad \mathit{return}()
\end{array}$$

el lenguaje $\mathcal{L}_5$ puede definirse como

$$\mathcal{L}_5\ =\ \mu_2\mathbb{E}(\mathbb{S} \boxplus \mathbb{C})$$

y el intérprete de $\mathcal{L}_5$ será

$$\begin{array}{l}
\mathsf{Inter}_{\mathcal{L}_5}\ :\ \mathcal{L}_5 \rightarrow \mathsf{Comp}\,() \\
\mathsf{Inter}_{\mathcal{L}_5}\ = (\!|\varphi_{\mathbb{E}}, \varphi_{\mathbb{S}} \boxplus_2 \varphi_{\mathbb{C}}|\!)_2
\end{array}$$

6.8 Valoración

- *No ambigüedad.* Este formalismo no contiene ambigüedad en sus especificaciones.

- *Modularidad.* La modularidad semántica se alcanza mediante la utilización de mónadas y transformadores de mónadas. Recuérdese que este sistema se basa en la semántica monádica modular, en la cual existe modularidad semántica.

- *Reusabilidad.* Como puede comprobarse, es posible crear componentes semánticos reutilizables. De hecho, la reusabilidad de especificaciones es uno de los objetivos de diseño.

- *Demostración.* Las definiciones se basan en semántica denotacional y cada mónada lleva consigo una serie de axiomas ecuacionales que facilitarían el razonamiento, verificación y optimización de programas. Además, puesto que el sistema está empotrado en Haskell, se realiza una comprobación de tipos de las especificaciones.

- *Prototipo.* A partir de las especificaciones se obtienen prototipos de forma directa de todas aquellas especificaciones que tengan un tipo legal. Lo cual es comprobado por el sistema de tipos de Haskell.

- *Legibilidad.* Dado que el sistema se ha implementado como un metalenguaje empotrado en Haskell, la legibilidad de las especificaciones se ve perjudicada por la mezcla sintáctica entre elementos del lenguaje anfitrión y elementos del metalenguaje. A pesar de la flexibilidad sintáctica de Haskell, una implementación independiente permitiría mejorar la legibilidad de las especificaciones y facilitaría el trabajo del diseñador de lenguajes.

- *Flexibilidad.* El sistema puede adaptarse a la especificación de lenguajes de diferentes paradigmas. De hecho, en los siguientes capítulos se muestra la especificación de lenguajes prototípicos de los paradigmas lógico, funcional, imperativo y orientado a objetos.

- *Experiencia.* No se han realizado especificaciones de lenguajes *reales* hasta el momento.

Capítulo 7

Especificación Lenguaje Funcional

> *An important distinction is the one between indicating what behavior, step by step, you want the machine to perform, and merely indicating what outcome you want.* P. Landin [139]

En este capítulo se presenta la especificación de un lenguaje funcional con características imperativas. El lenguaje admite expresiones aritméticas y booleanas, variables, declaraciones locales recursivas, referencias y asignaciones, funciones de orden superior, Entrada/Salida, excepciones y continuaciones de primera clase.

No se incluye chequeo de tipos, ya que el objetivo es mostrar cómo se puede modelizar la semántica dinámica del lenguaje, no la semántica estática.

7.1 Conceptos de Programación funcional

7.1.1 Abstracción y aplicación

La programación funcional tiene su origen en el cálculo λ. desarrollado por Church en los años 30. Este cálculo se basa en la utilización de dos mecanismos básicos: la abstracción y la aplicación.

La *abstracción* es la posibilidad de factorizar patrones repetitivos. Por ejemplo, la expresión $(2+3)*(2+3)$ incluye una multiplicación de un valor por sí mismo. Mediante la notación lambda $\lambda x \rightarrow x*x$ se crea una abstracción que representa una función que toma un argumento y lo multiplica por sí mismo.

La *aplicación* de una abstracción a un valor consiste en substituir ese valor por el parámetro de la expresión lambda. Se denota habitualmente mediante un espacio en blanco. La realización de una aplicación también se denomina reducción β.

Ejemplo 7.1 *La aplicación* $(\lambda x \rightarrow x*x)(2+3)$ *es equivalente a la expresión* $(2+3)*(2+3)$

Los lenguajes funcionales admiten la posibilidad de definir *funciones de orden superior*, es decir funciones que pueden tener como argumentos otras funciones y devolver funciones como resultados.

Ejemplo 7.2 *La expresión*

$$\lambda f \ \rightarrow \ (\lambda x \ \rightarrow \ f \ (f \ x))$$

indica una función que toma como argumento una función f y devuelve una función como resultado, la cual toma un valor x y devuelve la aplicación reiterada de f a x.

Definición 7.1 (Transparencia referencial) *La* transparencia referencial *es una propiedad que se cumple cuando el valor de una expresión depende únicamente del valor de las subexpresiones que la componen.*

Los lenguajes funcionales que cumplen esta propiedad se denominan lenguajes puramente funcionales.

7.1.2 Mecanismos de evaluación

El *modelo computacional* de la programación funcional se basa en la evaluación de expresiones mediante reescritura aplicando reducciones simples. Existen tres modelos de evaluación.

- Por nombre. Se realiza la reducción β antes de evaluar la expresión. La expresión sólo se evalúa cuando se necesita su valor.

  Ejemplo 7.3 *Al aplicar evaluación por nombre a $(\lambda x \rightarrow x * x)(2+3)$ se obtiene la siguiente secuencia de reducciones:*

  $$(\lambda x \rightarrow x * x)(2+3) \rightsquigarrow (2+3) * (2+3) \rightsquigarrow 5 * (2+3) \rightsquigarrow 5 * 5 \rightsquigarrow 25$$

  Este tipo de evaluación puede suponer un paso extra al evaluar de forma repetida la expresión $2+3$.

- Por valor. A la hora de evaluar la aplicación de una función a una expresión, se evalúa la expresión y luego se realiza la reducción β.

  Ejemplo 7.4 *La secuencia de pasos para evaluar $(\lambda x \rightarrow x * x)(2+3)$ por valor sería:*

  $$(\lambda x \rightarrow x * x)(2+3) \rightsquigarrow (\lambda x \rightarrow x * x)(5) \rightsquigarrow 5 * 5 \rightsquigarrow 25$$

  Este método evalúa el argumento sin saber si necesita su valor o no.

  Ejemplo 7.5 *La expresión $(\lambda x \rightarrow 34)(1/0)$ devuelve un error de división por cero al evaluarse por valor. Sin embargo, si se evalúa por nombre, devuelve 34.*

- Por necesidad o *by need*. Se realiza antes la reducción β, pero la expresión del argumento se mantiene en un único lugar. De forma que, cuando se necesite su valor, se evalúa, pero si se vuelve a necesitar, se obtiene directamente.

Ejemplo 7.6 *La secuencia de pasos para evaluar* $(\lambda x \rightarrow x * x)(2+3)$ *con evaluación perezosa es:*

$$(\lambda x \rightarrow x * x)(2+3) \rightsquigarrow (. * .) \rightsquigarrow 5 * 5 \rightsquigarrow 25$$
$$\downarrow \quad \downarrow$$
$$(2+3)$$

La llamada por necesidad también se conoce como evaluación perezosa y fue propuesta por C. Wadrworth en 1971 como una combinación de las ventajas de la evaluación por valor, al no repetir la evaluación de los argumentos y de la evaluación por nombre, al no evaluar el argumento hasta que no lo necesita. En [153] se estudian las propiedades del cálculo λ asociado a este método.

Una caracteristica habitual de los lenguajes funcionales es la utilización de declaraciones locales que pueden ser recursivas. La expresión **let** $x = e_1$ **in** e_2 indica que todas las referencias a x en la expresión e_2 tienen el valor e_1.

Ejemplo 7.7 *La siguiente expresión calcula el factorial de 5 mediante la definición de una función* f *de forma recursiva. En la definición se utiliza también la expresión condicional* **if**.

```
let
  f = λn → if n == 0 then
             1
           else
             n * f (n − 1)
in
f 5
```

Para la resolución de declaraciones locales puede también aplicarse los métodos anteriores, es decir, evaluación por nombre, por valor y perezosa.

7.1.3 Otras características

7.1.3.1 Continuaciones de primera clase

Las continuaciones fueron introducidas en la sección 3.5.2.7 para indicar cómo se pueden modelizar estructuras de control en semántica denotacional. Los lenguajes *ML* y *Scheme* incluyen instrucciones que permiten acceder a la continuación actual, la cual puede almacenarse y ser recuperada posteriormente. Con esta técnica es posible codificar estructuras de control avanzadas [211].

La instrucción *callcc* f captura la continuación actual κ y llama a la función f pasándole como argumento dicha continuación. Si la función f realiza una llamada a la continuación κ entonces se volverá la ejecución al momento en que se hizo la llamada a *callcc*.

Ejemplo 7.8 *La definición*

$$\lambda\, x\; y \;\rightarrow\; x \;/\; callcc\;(\,\lambda\, k \;\rightarrow\; \textbf{if}\; y == 0\; \textbf{then}\; k\; 0\; \textbf{else}\; y\,)$$

implementa una función de 2 argumentos x *e* y *que devuelve* 0 *si* y *es cero o* x / y *en caso contrario.*

7.1.3.2 Asignaciones y Referencias

El lenguaje *ML* admite la declaración de variables *referencia* cuyo valor puede ser modificado durante la ejecución del programa. Se utilizan las siguientes expresiones:

- *ref* v reserva en la memoria una dirección, inserta en dicha dirección el valor inicial v y devuelve dicha dirección.
- $!\, v$ devuelve los contenidos de la dirección de la variable v.
- $e_1 := e_2$ actualiza la dirección que denota la expresión e_1 con el resultado de evaluar e_2.
- $e_1 \,;\, e_2$ ejecuta e_1 y después e_2, devuelve el valor de e_2.

Ejemplo 7.9 *Al incluir referencias y asignaciones, se pierde la transparencia referencial. La siguiente expresión devuelve* False.

$$
\begin{array}{l}
\textbf{let} \\
\quad b \;=\; \mathit{ref}\ 0 \\
\quad f \;=\; \lambda n \rightarrow b \;:=\; !b \;+\; 1;\; n \;+\; b \\
\textbf{in} \\
f\ (2+3) \;==\; f\ (3+2)
\end{array}
$$

7.1.3.3 Excepciones

Los lenguajes *Haskell* y *ML* contienen un mecanismo de control de excepciones basado en las siguientes expresiones.

- *raise* ϵ lanza la excepción ϵ.
- *handle* e h evalúa la expresión e. Si e se evalúa normalmente, devuelve su valor. Si al evaluar e se produce una excepción, entonces llama a la función manejadora h pasándole como argumento la excepción producida.

Ejemplo 7.10 *La expresión*

$$
\begin{array}{l}
\textbf{let} \\
\quad f \;=\; \lambda\, x \;\rightarrow\; \textbf{if}\ x \;==\; 0\ \textbf{then}\ \mathit{raise}\ \text{“error”} \\
\qquad\qquad\qquad\qquad\qquad\quad \textbf{else}\ x+3 \\
\textbf{in} \\
\mathit{handle}\ (f\ 0)\ (\lambda\, e \;\rightarrow\; 2)
\end{array}
$$

devuelve un 2.

7.2 Estructura sintáctica

Con el fin de simplificar la presentación, en la estructura sintáctica se utiliza una categoría de expresiones únicamente. Dicha categoría se subdividirá en componentes sintácticos que permitirán una descripción semántica independiente.

Los componentes sintácticos serán [1]:

[1] Las 3 primeras componentes son las mismas que las descritas en la página 97

- Expresiones aritméticas.

  $\mathsf{Num}\, x \triangleq \mathit{Num}\ \mathit{Int} \mid x + x \mid x - x \mid x \times x \mid x \div x$

- Expresiones booleanas

  $\mathsf{Bool}\, x \triangleq \mathit{B}\ \mathit{Bool} \mid x \wedge x \mid x \vee x$

- Comparaciones

  $\mathsf{Cmp}\, x \triangleq x > x \mid x < x \mid x \leq x \mid x \geq x \mid x == x \mid x \neq x$

- Variables

  $\mathsf{Var}\, x \triangleq \mathit{V}\ \mathit{String}$

- Abstracciones Funcionales

  $\mathsf{Func}\, x \triangleq \lambda_N\ \mathit{String}\ x \mid \lambda_V\ \mathit{String}\ x \mid \lambda_L\ \mathit{String}\ x \mid x\,@\,x$

  $\lambda_X\ n\ e$ indica abstracción lambda (por ejemplo, $\lambda n \rightarrow n + 3$). Se utilizan tres tipos diferentes de abstracciones lambda. Por nombre (λ_N), por valor (λ_V) o con evaluación perezosa (λ_L). $e_1\,@\,e_2$ indica la aplicación de la expresión e_1 sobre la expresión e_2.

- Declaraciones locales

  $\mathsf{Dec}\, x \triangleq \mathit{Let_N}\ \mathit{String}\ x\ x \mid \mathit{Let_V}\ \mathit{String}\ x\ x \mid \mathit{Let_L}\ \mathit{String}\ x\ x$

  $\mathit{Let_X}\ v\ e_1\ e_2$ evalúa la expresión e_2 asignando a x el valor de e_1. Dicha asignación se puede realizar de tres formas, por nombre ($\mathit{Let_N}$), por valor ($\mathit{Let_V}$) o con evaluación perezosa ($\mathit{Let_L}$).

- Referencias y asignaciones destructivas

  $\mathsf{Ref}\, x \triangleq \mathit{ref}\ x \mid !\, x \mid x := x \mid x\,;\,x$

  Este bloque contiene un conjunto de instrucciones que permiten utilizar referencias a posiciones de memoria, asignaciones destructivas y secuencialidad.

  ref e crea una nueva posición de memoria en el valor de la expresión e. $!\,x$ obtiene el contenido de la posición de memoria del valor de e. $e_1 := e_2$ asigna a la posición de memoria del valor de e_1, el valor de e_2. Finalmente, $e_1\,;\,e_2$ evalúa la expresión e_2 después de evaluar la expresión e_1.

- Entrada/Salida

  $\mathsf{IO}\, x \triangleq \mathit{read}\ x \mid \mathit{write}\ x$

  read e lee un valor de la entrada estándar y asigna dicho valor al valor referenciado por e. *write* e escribe por la salida estándar el valor de e.

- Excepciones

  $\mathsf{Exc}\, x \triangleq \mathit{raise}\ \mathit{Exc} \mid \mathit{handle}\ x\ x$

- Continuaciones de primera clase (*callcc*)

  $\mathsf{CC}\, x \triangleq \mathit{Callcc}$

  La función *Callcc* ha sido explicada en la sección 3.5.2.7.

El lenguaje funcional se puede definir como el punto fijo de la suma de los functores anteriores

$$\mathcal{L}_{Fun} = \mathsf{Num} \oplus \mathsf{Bool} \oplus \mathsf{Cmp} \oplus \mathsf{Var} \oplus \mathsf{Func} \oplus \mathsf{Dec} \oplus \mathsf{Ref} \oplus \mathsf{IO} \oplus \mathsf{Exc} \oplus \mathsf{CC}$$

7.3 Dominio de valores

En el lenguaje a especificar Se utilizan dos únicos tipos de valores primitivos, los enteros (*Int*) y los booleanos (*Bool*). Además, como el lenguaje es funcional, se añade el tipo compuesto de funciones. De esa forma, el dominio de valores podrá modelizarse como:

$$\mathit{Value} \triangleq \mathit{Int} \| \mathit{Bool} \| \mathit{Function} \| \mathit{Loc} \| \mathit{Exc}$$

El dominio de funciones se definirá como

$$\mathit{Function} \triangleq \mathsf{Comp}\ \mathit{Value} \rightarrow \mathsf{Comp}\ \mathit{Value}$$

Aunque podría utilizarse el tipo $\mathit{Value} \rightarrow \mathit{Value}$ en un lenguaje puramente funcional, la modelización de funciones como valores que toman computaciones y producen computaciones, facilita la especificación de diferentes mecanismos de evaluación.

7.4 Estructura computacional

La modelización de la estructura computacional se realizará mediante una mónada. Dicha mónada debe dar soporte a las operaciones requeridas por los diferentes componentes sintácticos especificados. A continuación se presentan los requisitos que debe soportar:

- Acceso a un entorno para consultar el valor de las variables
- Modificación de un estado global en el que se simulará una memoria principal, con el fin de modelizar referencias y asignaciones
- Computaciones parciales para soportar errores. Por ejemplo, divisiones entre cero, variables no inicializadas, etc.
- Continuaciones, para dar soporte a la función *callcc*.
- Entrada/Salida

La mónada resultante se obtendrá mediante la aplicación reiterada de los transformadores de mónadas correspondientes a una mónada base. Se utilizará como mónada base la mónada IO predefinida que permitirá ejecutar acciones de Entrada/Salida de forma directa. Por tanto, las computaciones se definen a partir de la mónada

$$\mathsf{Comp} \triangleq (\mathcal{T}_{Err} \,.\, \mathcal{T}_{State} \,.\, \mathcal{T}_{Env} \,.\, \mathcal{T}_{Cont})\,\mathsf{IO}$$

Un valor de tipo Comp *Value* indicará una computación Comp que devuelve un valor de tipo *Value*.

Es necesario distinguir entre el entorno y el estado. El entorno no es modificable durante la ejecución e indica qué variables están activas. A cada variable activa le asigna una posición en el estado. El estado asigna a cada posición un valor.

Ejemplo 7.11 *En la figura 7.1 se presenta una posible situación en la que la el valor de x es* 45*. Para obtener dicho valor, el entorno indica que el valor de x está en la posición* 3 *del estado y en dicha posición se almacena el valor* 45*.*

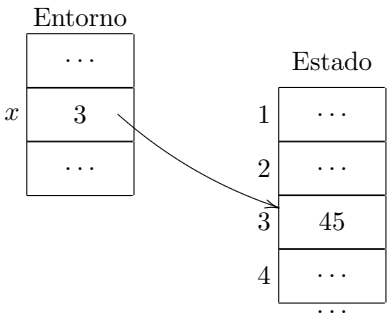


Figura 7.1: Indirección

Una asignación modifica el estado, pero no el entorno. Por ejemplo, la sentencia

$$x := !x + 1$$

modificaría el valor de la dirección 3 *del estado, que pasaría a ser* 46*, dejando el entorno sin modificar.*

Para facilitar la implementación de evaluación perezosa, se almacenarán computaciones tanto en el entorno como en el estado. El tipo resultante será:

$$\begin{aligned} \mathit{State} &\triangleq \mathit{Heap}\ (\mathsf{Comp}\ \mathit{Value}) \\ \mathit{Env} &\triangleq \mathit{Table}\ (\mathsf{Comp}\ \mathit{Value}) \end{aligned}$$

7.4.1 Especificación Semántica

Las funciones semánticas pueden especificarse de forma independiente para cada uno de los componentes sintácticos descritos en la sección 7.2. En el caso de componentes sintácticos que requieran evaluación recursiva de subcomponentes, se utilizan álgebras monádicas, en el resto, se utilizan álgebras.

7.4.2 Evaluación y comprobación de tipo

- *evalWith* $\odot$ v_1 v_2 aplica el operador $\odot$ a los operandos después de convertirlos al subtipo adecuado. Tras la aplicación, inyecta el valor en el subtipo respuesta.

$$\begin{aligned} &\mathit{evalWith} && : (\ \mathit{SubType}\ \alpha\ \beta \\ &&& ,\ \ \mathit{SubType}\ \gamma\ \beta \\ &&& ,\ \ \mathit{Monad}\ \mathsf{m})\ \Rightarrow\ (\alpha \to \alpha \to \gamma) \to \beta \to \beta \to \mathsf{m}\ \beta \\ &\mathit{evalWith}\ \odot\ v_1\ v_2 &&= \mathit{return}(\uparrow (\downarrow v_1 \odot \downarrow v_2)) \end{aligned}$$

7.4.3 Funciones de Gestión de Memoria

El lenguaje requiere la utilización de una memoria dinámica en la que se almacenarán resultados parciales. Como se ha indicado, la memoria tendrá el tipo *Heap* (Comp *Value*), es decir, la memoria almacenará computaciones que devuelven un valor, en lugar de valores. Se definen las siguientes funciones auxiliares que utilizan las funciones definidas en 6.6.5.1.

- *alloc* : Comp *Value* $\to$ Comp *Loc*, *alloc m* crea una nueva dirección de memoria y asigna a dicha dirección la computación m. Devuelve la dirección recién asignada.

$$\begin{aligned} &\mathit{alloc}\ m\ =\ \mathbf{do} \\ &\qquad h\ \leftarrow\ \mathit{fetch} \\ &\qquad \mathbf{let}\ (l, h')\ =\ \mathit{alloc}_H\ m\ h \\ &\qquad \mathit{set}\ h' \\ &\qquad \mathit{return}\ l \end{aligned}$$

- *lkpLoc* : *Loc* $\to$ Comp *Value*, *lkpLoc l* devuelve la computación a la que apunta l.

$$\begin{aligned} &\mathit{lkpLoc}\ l\ =\ \mathbf{do} \\ &\qquad h\ \leftarrow\ \mathit{fetch} \\ &\qquad \mathit{lkp}_H\ l\ h \end{aligned}$$

- *updLoc* : *Loc* $\to$ Comp *Value* $\to$ Comp *Value*, *updLoc loc m* actualiza la dirección *loc* con la computación m

$$\mathit{updLoc}\ l\ m\ =\ \mathit{update}\ (\mathit{upd}_H\ l\ m)$$

7.4.4 Álgebras y Álgebras monádicas

La especificación semántica propiamente dicha consistirá en definir álgebras o álgebras monádicas, que tomen la estructura computacional como portadora.

- Expresiones aritméticas

$$\varpi_{\mathsf{Num}} \quad : \mathsf{Num}\ \mathit{Value} \to \mathsf{Comp}\ \mathit{Value}$$

$$
\begin{aligned}
\varpi_{\mathsf{Num}}\ [\![Num\ n]\!] &= return\ (inj\ n)\\
\varpi_{\mathsf{Num}}\ [\![x\ +\ y]\!] &= evalWith\ (+)\ x\ y\\
\varpi_{\mathsf{Num}}\ [\![x\ -\ y]\!] &= evalWith\ (-)\ x\ y\\
\varpi_{\mathsf{Num}}\ [\![x\ \times\ y]\!] &= evalWith\ (\times)\ x\ y\\
\varpi_{\mathsf{Num}}\ [\![x\ \div\ y]\!] &= evalWith\ (\div)\ x\ y
\end{aligned}
$$

- Expresiones booleanas

$$
\begin{aligned}
\varpi_{\mathsf{Bool}} &: \mathsf{Bool}\ Value \rightarrow \mathsf{Comp}\ Value\\
\varpi_{\mathsf{Bool}}\ [\![B\ b]\!] &= return\ (inj\ b)\\
\varpi_{\mathsf{Bool}}\ [\![x\ \wedge\ y]\!] &= evalWith\ (\wedge)\ x\ y\\
\varpi_{\mathsf{Bool}}\ [\![x\ \vee\ y]\!] &= evalWith\ (\vee)\ x\ y
\end{aligned}
$$

- Comparaciones

$$
\begin{aligned}
\varpi_{\mathsf{Cmp}} &: \mathsf{Cmp}\ Value \rightarrow \mathsf{Comp}\ Value\\
\varpi_{\mathsf{Cmp}}\ [\![x\ >\ y]\!] &= evalWith\ (>)\ x\ y\\
\varpi_{\mathsf{Cmp}}\ [\![x\ <\ y]\!] &= evalWith\ (<)\ x\ y\\
\varpi_{\mathsf{Cmp}}\ [\![x\ \geq\ y]\!] &= evalWith\ (\geq)\ x\ y\\
\varpi_{\mathsf{Cmp}}\ [\![x\ \leq\ y]\!] &= evalWith\ (\leq)\ x\ y\\
\varpi_{\mathsf{Cmp}}\ [\![x\ ==\ y]\!] &= evalWith\ (==)\ x\ y\\
\varpi_{\mathsf{Cmp}}\ [\![x\ \neq\ y]\!] &= evalWith\ (\neq)\ x\ y
\end{aligned}
$$

- Variables

$$
\begin{aligned}
&\varpi_{\mathsf{Var}} : \mathsf{Cmp}\ Value \rightarrow \mathsf{Comp}\ Value\\
&\varpi_{\mathsf{Var}}\ [\![V\ x]\!] = \mathbf{do}\\
&\qquad\qquad \rho \leftarrow rdEnv\\
&\qquad\qquad lkp_T\ x\ \rho
\end{aligned}
$$

- Referencias y asignaciones destructivas. Para modelizar este bloque se utilizarán las funciones auxiliares *alloc* y *lkpLoc* y *updLoc* definidas en 7.4.3.

$$
\begin{aligned}
&\varphi_{\mathsf{Ref}} : \mathsf{Ref}\ (\mathsf{Comp}\ Value) \rightarrow \mathsf{Comp}\ Value\\
&\varphi_{\mathsf{Ref}}\ [\![ref\ e]\!] = \mathbf{do}\\
&\qquad\qquad v \leftarrow e\\
&\qquad\qquad loc \leftarrow alloc\ (return\ v)\\
&\qquad\qquad return\ loc
\end{aligned}
$$

$$
\begin{aligned}
&\varphi_{\mathsf{Ref}}\ [\![!\ e]\!] = \mathbf{do}\\
&\qquad\qquad v_{loc} \leftarrow e\\
&\qquad\qquad lkpLoc\ (\downarrow v_{loc})
\end{aligned}
$$

Para modelizar la asignación destructiva $e_1 := e_2$, se evalúa e_1 para obtener un valor $v_l oc$, se comprueba que $v_l oc$ es una dirección loc, se evalúa e_2 y se asigna a la dirección loc el valor obtenido.

$$\begin{aligned}
&\varphi_{\mathsf{Ref}}\ [\![e_1 := e_2]\!] = \mathbf{do}\\
&\qquad\qquad v_{loc} \leftarrow e_1\\
&\qquad\qquad v \leftarrow e_2\\
&\qquad\qquad \mathit{updLoc}\ (\downarrow v_{loc})\ (\mathit{return}\ v)\\
&\qquad\qquad \mathit{return}\ v
\end{aligned}$$

La ejecución secuencial consiste en enlazar dos evaluaciones.

$$\varphi_{\mathsf{Ref}}\ [\![e_1\ ;\ e_2]\!] = e_1 \gg e_2$$

- Abstracciones funcionales

$$\begin{aligned}
&\varphi_{\mathsf{Fun}} : \mathsf{Fun}\ (\mathsf{Comp}\ \mathit{Value}) \rightarrow \mathsf{Comp}\ \mathit{Value}\\
&\varphi_{\mathsf{Fun}}\ [\![\lambda_V\ x\ e]\!] = \mathbf{do}\\
&\qquad\qquad \rho \leftarrow \mathit{rdEnv}\\
&\qquad\qquad \mathit{return}\ (\uparrow\ (\lambda m \rightarrow \mathbf{do}\\
&\qquad\qquad\qquad\qquad\qquad v \leftarrow m\\
&\qquad\qquad\qquad\qquad\qquad \mathit{inEnv}\ (\mathit{upd}_T\ x\ (\mathit{return}\ v)\ \rho)\ e\\
&\qquad\qquad))\\
&\varphi_{\mathsf{Fun}}\ [\![\lambda_N\ x\ e]\!] = \mathbf{do}\\
&\qquad\qquad \rho \leftarrow \mathit{rdEnv}\\
&\qquad\qquad \mathit{return}\ (\uparrow\ (\lambda m \rightarrow \mathit{inEnv}\ (\mathit{upd}_T\ x\ m\ \rho)\ e))
\end{aligned}$$

El problema de la evaluación por nombre es que en ocasiones puede ocasionar una evaluación múltiple de la misma expresión. Para evitarlo, la evaluación perezosa (λ_L) crea una computación (denominada *thunk*) que sólo se evalúa si se necesita y que, si se intenta evaluar posteriormente, devuelve el valor previamente obtenido sin volver a evaluar.

$$\begin{aligned}
&\varphi_{\mathsf{Fun}}\ [\![\lambda_L\ x\ e]\!] = \mathbf{do}\\
&\qquad\qquad \rho \leftarrow \mathit{rdEnv}\\
&\qquad\qquad \mathit{return}\ (\uparrow\ (\lambda m \rightarrow \mathbf{do}\\
&\qquad\qquad\qquad\qquad\qquad \mathit{loc} \leftarrow \mathit{alloc}\ m\\
&\qquad\qquad\qquad\qquad\qquad \mathit{updLoc}\ \mathit{loc}\ (\mathit{mkThunk}\ \mathit{loc}\ m)\\
&\qquad\qquad\qquad\qquad\qquad \mathit{inEnv}\ (\mathit{upd}_T\ x\ (\mathit{lkpLoc}\ \mathit{loc})\ \rho)\ e\\
&\qquad\qquad))
\end{aligned}$$

La función $\mathit{mkThunk}\ l\ m$ almacena una computación en la dirección l. La primera vez que se intente evaluar dicha computación, evaluará la computación m y modifica la dirección l para que en los sucesivos intentos de evaluación, se devuelva el valor directamente.

$$\begin{aligned}
&\mathit{mkThunk}\ \mathit{loc}\ m = \mathbf{do}\\
&\qquad\qquad v \leftarrow m\\
&\qquad\qquad \mathit{updLoc}\ \mathit{loc}\ (\mathit{return}\ v)\\
&\qquad\qquad \mathit{return}\ v
\end{aligned}$$

Para la aplicación $e_1\ @\ e_2$ se evalúa e_1 y se comprueba que el valor obtenido es una función f que toma como argumento una computación y devuelve

otra computación. El resultado consistirá en evalúar f pasándole como argumento la computación de e_2 en el entorno actual.

$$\begin{aligned}\varphi_{\mathsf{Fun}}\,[\![e_1 \mathbin{@} e_2]\!] \;=\; &\mathbf{do}\\ &\quad v_f \leftarrow e_1\\ &\quad \rho \leftarrow \mathit{rdEnv}\\ &\quad (\downarrow v_f)(\mathit{inEnv}\ \rho\ e_2)\end{aligned}$$

- Declaraciones locales.

  Para la especificación semántica de los bloques con declaraciones locales de la forma $\mathit{Let}_X\ v\ e_1\ e_2$ se deben tener en cuenta los diferentes tipos de evaluación.

  En evaluación por valor (Let_V), se evalúa en primer lugar e_1 para obtener un valor v_1 y el resultado final consiste en evaluar e_2 en un entorno que asigna a v el valor v_1.

$$\begin{aligned}&\varphi_{\mathsf{Dec}} \;:\; \mathsf{Dec}\,(\mathsf{Comp}\ \mathit{Value}) \to \mathsf{Comp}\ \mathit{Value}\\ &\varphi_{\mathsf{Dec}}\,[\![\mathit{Let}_V\ x\ e_1\ e_2]\!] \;=\; \mathbf{do}\\ &\qquad (\mathit{loc}, \rho) \leftarrow \mathit{prepareDecl}\ e_1\ x\\ &\qquad v \leftarrow \mathit{inEnv}\ \rho\ e_1\\ &\qquad \mathit{updLoc}\ \mathit{loc}\ (\mathit{return}\ v)\\ &\qquad \mathit{inEnv}\ \rho\ e_2\end{aligned}$$

  En evaluación por nombre (Let_N), se evalúa e_2 en un entorno que asigna a la variable v la expresión sin evaluar e_1. Cada vez que se necesite el valor de v se evaluará dcha expresión.

$$\begin{aligned}\varphi_{\mathsf{Dec}}\,[\![\mathit{Let}_N\ x\ e_1\ e_2]\!] \;=\; &\mathbf{do}\\ &\quad (\mathit{loc}, \rho) \leftarrow \mathit{prepareDecl}\ e_1\ x\\ &\quad \mathit{updLoc}\ \mathit{loc}\ (\mathit{inEnv}\ \rho\ e_1)\\ &\quad \mathit{inEnv}\ \rho\ e_2\end{aligned}$$

  La declaración local con evaluación perezosa evaluará e_2 en un entorno en el que la variable v está asignada a la *thunk* creada para evaluar e_1.

$$\begin{aligned}\varphi_{\mathsf{Dec}}\,[\![\mathit{Let}_L\ x\ e_1\ e_2]\!] \;=\; &\mathbf{do}\\ &\quad (\mathit{loc}, \rho) \leftarrow \mathit{prepareDecl}\ e_1\ x\\ &\quad \mathit{updLoc}\ (\mathit{mkThunk}\ \mathit{loc}\ (\mathit{inEnv}\ \rho\ e_1))\\ &\quad \mathit{inEnv}\ \rho\ e_2\end{aligned}$$

  La función auxiliar *prepareDecl* m x reserva una posición de memoria para la computación m y asigna a la variable x la computación que al evaluarse busca en memoria dicha computación. Devuelve la posición de memoria y la nueva tabla.

$$\begin{aligned}\mathit{prepareDecl}\ m\ x \;=\; &\mathbf{do}\\ &\quad \mathit{loc} \leftarrow \mathit{alloc}\ m\\ &\quad \rho \leftarrow \mathit{rdEnv}\\ &\quad \mathit{return}\ (\mathit{loc},\ \mathit{upd}_T\ x\ (\mathit{lkpLoc}\ \mathit{loc})\ \rho)\end{aligned}$$

- Entrada/Salida

$$
\begin{aligned}
&\varphi_{\mathsf{IO}} \quad : \ \mathsf{IO}\,(\mathsf{Comp}\ \mathit{Value}) \to \mathsf{Comp}\ \mathit{Value}\\
&\varphi_{\mathsf{IO}}\ [\![\mathit{read}\ e]\!] \ = \ \mathbf{do}\\
&\qquad\qquad v_{loc} \leftarrow e\\
&\qquad\qquad \mathit{loc} \leftarrow \mathit{checkType}\ v_{loc}\\
&\qquad\qquad \mathit{putStr}\ \text{“Value?”}\\
&\qquad\qquad s \leftarrow \mathit{getLine}\\
&\qquad\qquad \mathit{upd}\ (\mathit{str2v}\ s) \quad \text{— } \mathit{str2v} \text{ convierte de } \mathit{String} \text{ en } \mathit{Value}
\end{aligned}
$$

$$
\begin{aligned}
&\varphi_{\mathsf{IO}}\ [\![\mathit{write}\ e]\!] \ = \ \mathbf{do}\\
&\qquad\qquad v \leftarrow e\\
&\qquad\qquad \mathit{putStr}\ (\mathit{v2str}\ v) \quad \text{— } \mathit{v2str} \text{ convierte de } \mathit{Value} \text{ en } \mathit{String}
\end{aligned}
$$

- Excepciones

$$
\begin{aligned}
&\varphi_{\mathsf{Exc}} \quad : \ \mathsf{Exc}\,(\mathsf{Comp}\ \mathit{Value}) \to \mathsf{Comp}\ \mathit{Value}\\
&\varphi_{\mathsf{Exc}}\ [\![\mathit{raise}\ \epsilon]\!] \ = \mathit{raise}\ \epsilon\\
&\varphi_{\mathsf{Exc}}\ [\![\mathit{handle}\ e_1\ e_2]\!] \ = \ \mathbf{do}\\
&\qquad\qquad\qquad v_f \leftarrow e_2\\
&\qquad\qquad\qquad \mathit{handle}\ e_1\ (\lambda\epsilon \to f\ (\mathit{return}(\uparrow \epsilon)))
\end{aligned}
$$

- Continuaciones

$$
\begin{aligned}
&\varphi_{\mathsf{CC}} \quad : \ \mathsf{CC}\,(\mathsf{Comp}\ \mathit{Value}) \to \mathsf{Comp}\ \mathit{Value}\\
&\varphi_{\mathsf{CC}}\ [\![\mathit{Callcc}]\!] \ = \mathit{return}\ (\uparrow\ \mathit{mkFun})\\
&\quad \mathbf{where}\\
&\qquad \mathit{mkFun}\ m \ = \ \mathbf{do}\\
&\qquad\qquad\qquad v_f \leftarrow m\\
&\qquad\qquad\qquad \mathit{callcc}\ (\lambda\kappa \to\!\downarrow v_f\ (\mathit{return}\ (\uparrow \kappa)))
\end{aligned}
$$

Una vez definidas las álgebras y las álgebras monádicas, es posible obtener el intérprete mediante una combinación entre catamorfismos y catamorfismos monádicos (véase sección 5.7).

$$
\begin{aligned}
&\varpi_{\mathcal{L}_{\mathsf{Fun}}} \ : \ (\mathsf{Num} \oplus \mathsf{Bool} \oplus \mathsf{Cmp} \oplus \mathsf{Var})\ \mathit{Value} \ \to \ \mathsf{Comp}\ \mathit{Value}\\
&\varpi_{\mathcal{L}_{\mathsf{Fun}}} \ = \ \varpi_{\mathsf{Num}} \oplus_{\mathsf{m}} \varpi_{\mathsf{Bool}} \oplus_{\mathsf{m}} \varpi_{\mathsf{Cmp}} \oplus_{\mathsf{m}} \varpi_{\mathsf{Var}}
\end{aligned}
$$

$$
\begin{aligned}
&\varphi_{\mathcal{L}_{\mathsf{Fun}}} \ : \ (\mathsf{Ref} \oplus \mathsf{Fun} \oplus \mathsf{Dec} \oplus \mathsf{IO} \oplus \mathsf{Exc} \oplus \mathsf{CC})(\mathsf{Comp}\ \mathit{Value}) \to \ \mathsf{Comp}\ \mathit{Value}\\
&\varphi_{\mathcal{L}_{\mathsf{Fun}}} \ = \ \varphi_{\mathsf{Ref}} \oplus \varphi_{\mathsf{Fun}} \oplus \varphi_{\mathsf{Dec}} \oplus \varphi_{\mathsf{IO}} \oplus \varphi_{\mathsf{Exc}} \oplus \varphi_{\mathsf{CC}}
\end{aligned}
$$

$$
\begin{aligned}
&\mathsf{Inter}_{\mathcal{L}_{Fun}} \ : \ \mathcal{L}_{Fun} \to \mathsf{Comp}\ \mathit{Value}\\
&\mathsf{Inter}_{\mathcal{L}_{Fun}} \ = \ (\!|[\varpi_{\mathcal{L}_{\mathsf{Fun}}}, \varphi_{\mathcal{L}_{\mathsf{Fun}}}]\!|)
\end{aligned}
$$

Capítulo 8

Especificación Lenguaje Orientado a Objetos

> *Everithing we can describe can be represented by the recursive composition of a single kind of behavioral building block that hides its combination of state and process inside itself and can be dealt with only through the exchange of messages.* A. C. Kay [120]

En este capítulo se especifica la semántica dinámica de un ejemplo de lenguaje Orientado a Objetos. Se identifican los conceptos más característicos de los lenguajes orientados a objetos, en especial, el concepto de herencia, y se plantea una codificación que soporte herencia con enlace estático y con enlace dinámico. Finalmente, al lenguaje se le añaden expresiones aritméticas simples reutilizando las definiciones del capítulo anterior.

8.1 Conceptos de Programación Orientada a Objetos

Los lenguajes de programación Orientada a Objetos [206] están teniendo un enorme éxito en el desarrollo del software, debido principalmente al esfuerzo empleado en proporcionar capacidades de reutilización de software [164].

Simula [47] puede considerarse el primer lenguaje Orientado a Objetos, creado en los años 60 con el propósito de describir y programar simulaciones computacionales. Smalltalk [64] se desarrolla posteriormente como un sistema uniforme Orientado a Objetos. Aunque originalmente interpretado y poco eficiente, Smalltalk incorporaba un completo entorno gráfico de desarrollo y formó una importante base conceptual para la ingeniería del software Orientado a Objetos.

En 1985, B. Stroustrup desarrolla C++ una extensión orientada a objetos del lenguaje C [223]. El lenguaje C++ supone una solución de compromiso que pretende disponer de la eficiencia de un lenguaje imperativo de bajo nivel a la vez que las capacidades de reutilización de los lenguajes Orientados a Objetos. La solución híbrida proporcionada por C++ tendrá una enorme aceptación industrial a pesar de la dificultad inherente de tales integraciones y de la tendencia del lenguaje a incorporar extensiones difíciles de digerir desde un punto de vista semántico.

Además de C++, se han propuesto varios lenguajes híbridos que añaden una capa de orientación a objetos a lenguajes existentes como CLOS [24], Modula-3 [180], Prolog++ [175], Objective-CAML [4], etc.

A principios de los años 90, Java [113] fue diseñado como una simplificación del lenguaje C++ que añadía concurrencia y gestión automática de memoria. La implementación del lenguaje, basada en una máquina abstracta facilitó su portabilidad a múltiples plataformas, incluidos los visualizadores de páginas Web, lo cual ha servido para que el número de usuarios del lenguaje haya crecido espectacularmente.

Aunque no existe un definición única de los requisitos que debe cumplir un lenguaje de programación para considerarse Orientado a Objetos, es posible identificar varias características comunes.

8.1.1 Objetos y Clases

Un objeto *encapsula* unos recursos o variables locales y proporciona operaciones de manipulación de dichos recursos. Esta encapsulación da lugar al concepto de *abstracción de datos* ya que el recurso encapsulado no puede ser manipulado directamente, sólo a través de las operaciones proporcionadas por el objeto. El conjunto de recursos encapsulados se denomina *estado de un objeto*, mientras que las operaciones exportadas forman el conjunto de *métodos de un objeto*. La expresión $o \mapsto m$ denotará la selección del método m del objeto o y también puede leerse como el resultado de enviar el mensaje m al objeto o.

En la definición de un objeto existe la posibilidad de referirse a sí mismo mediante la variable *self*[1]. Desde un punto de vista teórico, esta capacidad de autoreferencia requiere una modelización mediante recursividad o puntos fijos.

Los lenguajes orientados a objetos más populares utilizan el concepto de *clase* como una descripción de la estructura y comportamiento de objetos[2].

Ejemplo 8.1 *El siguiente ejemplo define una clase Point que contiene dos variables locales x e y, ambas inicializadas a 5, junto con las operaciones de acceso dist que calcula la distancia del punto al origen, move que mueve el objeto a una posición dada y closer que chequea si el punto actual está más cerca del origen que otro punto que se le pasa como parámetro.*

class *Point* **where**
locals $x = 5$
$y = 5$
methods *dist* $= \lambda() \rightarrow sqrt(x^2 + y^2)$
move $= \lambda(d_x, d_y) \rightarrow x := x + d_x;\ y := y + d_y;$
closer $= \lambda p \rightarrow self \mapsto dist\ <\ p \mapsto dist$

Una clase puede considerarse como un patrón que permite crear objetos denominados instancias de dicha clase. Todos los objetos generados a partir de una clase comparten los mismos métodos, aunque pueden contener valores diferentes de las variables locales.

[1] En C++ y Java se utiliza la variable *this*

[2] Es conveniente distinguir entre la clase de un objeto, que define la estructura y el comportamiento, del tipo, que define únicamente la estructura.

Dada una clase $\mathbb{C}$, el operador *new* $\mathbb{C}$ devuelve un nuevo objeto de dicha clase (normalmente denominado *instancia* de $\mathbb{C}$).

El concepto de clase no es estrictamente necesario en un lenguaje Orientado a Objetos. Existen lenguajes en los que una clase se representa como un objeto que permite generar otros objetos (este tipo de objetos se denominan normalmente *prototipos* [28] y se utilizan en lenguajes como Self [228]). En [6], los lenguajes que utilizan la noción de clases se denominan *lenguajes basados en clases* para distinguirlos de los *lenguajes basados en objetos* o *lenguajes basados en prototipos* que no requieren un concepto de clase independiente. A nivel conceptual los lenguajes basados únicamente en objetos son más sencillos de modelizar ya que sólo se necesita modelizar el concepto de objeto. Sin embargo, dada la mayor popularidad de los lenguajes basados en clases, en el resto de esta sección se utilizarán este tipo de lenguajes.

8.1.2 Herencia

Una piedra angular de los lenguajes Orientados a Objetos es la noción de *herencia* y *subclase*. Como cualquier clase, una subclase describe el comportamiento de un conjunto de objetos. Sin embargo, lo hace de forma incremental, mediante extensiones y cambios de una *superclase* o *clase base*. Las variables locales de la superclase son heredadas de forma implícita, aunque la subclase puede definir nuevas variables locales. Los métodos de la superclase pueden ser heredados por defecto o redefinidos por métodos de tipo similar en la subclase[3].

Ejemplo 8.2 *En el siguiente ejemplo se declara Circle como una subclase de Point añadiendo una variable local radius con valor inicial 2 y modificando el método que calcula la distancia al origen.*

class *Circle* **extends** *Point* **where**
  locals $radius = 2$
  methods $dist = \lambda() \rightarrow max(super \mapsto dist - radius, 0)$

Sin las subclases, la aparición de *self* en la declaración de una clase se refiere siempre a un objeto de dicha clase. Con subclases, esta afirmación no se cumple. En un método m heredado por una subclase $\mathbb{D}$ de $\mathbb{C}$, *self* se refiere a un objeto de la subclase $\mathbb{D}$, no a un objeto de la clase original $\mathbb{C}$. Para poder hacer referencia a los métodos de la clase original se utiliza *super*.

La *herencia múltiple* se obtiene cuando una subclase hereda de más de una clase. Un problema inmediato que surge al heredar de varias clases es decidir qué política seguir en caso de conflictos y duplicaciones entre los métodos y variables locales de las superclases. Aunque la solución más simple es identificar y prohibir dichos conflictos, existen diversas soluciones más elegantes (véase [32, 8]).

Un concepto relacionado con la utilización de subclases, es el de enlace dinámico. Si $\mathbb{D}$ es una subclase de $\mathbb{C}$ que redefine el método m y se tiene un objeto o de la clase $\mathbb{D}$, con *enlace dinámico*, la expresión $o.m$ selecciona el método redefinido en la subclase $\mathbb{D}$, mientras que con enlace estático se utiliza el método de la clase base.

[3]En Simula y C++ sólo es posible redefinir los métodos marcados como virtuales en la superclase

Ejemplo 8.3 *Considérese las siguiente definición:*

$$near : Point \rightarrow Bool$$
$$near\ p = p \mapsto dist < 6$$

La función near toma como parámetro un objeto de la clase Point y chequea si su distancia al origen es menor que 5.

Dado que todos los objetos de la clase Circle pertenecen a su vez a la clase Point, es posible pasarle tanto un objeto Point como un objeto Circle.

Supóngase que se declara la siguiente expresión:

$$\begin{array}{l}\textbf{let}\ p = new\ Point \\ \quad\ q = new\ Circle \\ \qquad \textbf{in}\ near\ p == near\ q\end{array}$$

Al calcular near q se debe buscar el método dist de q que pertenece a la clase Circle. Existen dos posibilidaddes:

- *Con enlace estático se utiliza el método dist de la clase Point*
- *Con enlace dinámico se utiliza el método dist de la clase Circle*

Con enlace dinámico, a la hora de implementar la función near no es posible conocer qué método dist se va a utilizar. Esta información sólo se conoce en tiempo de ejecución ya que depende del tipo de punto que se esté usando.

El enlace dinámico ofrece un importante mecanismo de abstracción: cada objeto sabe cómo debe comportarse de forma autónoma, liberando al programador de preocuparse de examinar qué objeto está utilizando y decidir qué operación aplicar.

Como se ha indicado, la setencia $o.m$ puede tomar distintas formas, dependiendo del valor concreto del objeto o. Este tipo polimorfismo se conoce como *polimorfismo de inclusión*[4]

8.2 Especificación de Lenguaje Orientado a Objetos

Se han propuesto varias técnicas para formalizar la semántica denotacional de los lenguajes orientados a objetos. En [32] se utiliza por primera vez un tipo recursivo para modelizar un objeto con auto-referencia. Dicha técnica se empleará posteriormente en [199, 119, 45, 198]. En [35] se propone la relación de subtipos. A mediados de los años 90 se investigan los fundamentos de sistemas de tipos estáticos con subtipos. Se han propuesto varias alternativas, tipos existenciales [193] y cálculos de objetos primitivos [7, 6] que serán comparadas en [31].

También existen especificaciones de lenguajes orientados a objetos en otros formalismos, por ejemplo, en [79] se utiliza semántica operacional, en [243] se

[4]En algunos contextos se denomina simplemente polimorfismo, sin embargo, se considera más conveniente distinguirlo del polimorfismo paramétrico (o genericidad) y del polimorfismo ad-hoc (o sobrecarga) [35]

utiliza semántica de acción para modelizar un subconjunto del lenguaje Java, denominado JOOS [78], en [215] se especifica el lenguaje Java mediante *máquinas de estado abstractas* y en [240] se comparan diversos métodos de especificación de Java.

En esta sección se plantea una especificación utilizando semántica monádica modular. Puesto que este formalismo tiene sus raíces en la semántica denotacional, se ha tomado como modelo la codificación de objetos como puntos fijos definida en [199].

8.2.1 Estructura sintáctica

En los lenguajes basados en objetos, es posible crear un objeto directamente sin utilizar una clase. Un objeto se forma a partir de una lista de variables locales (cada una de las cuales inicializada a una expresión), junto con una lista de etiquetas asociadas a expresiones que denotan los métodos correspondientes.

La sintaxis que se utilizará para denotar un objeto será la siguiente:

$$
\begin{array}{ll}
\textbf{object} & \\
\textbf{locals} & v_1 \;=\; e \\
 & \quad \ldots \\
 & v_i \;=\; e \\
\textbf{methods} & m_1 \;=\; e \\
 & \quad \ldots \\
 & m_j \;=\; e
\end{array}
$$

donde e serán expresiones, v_i nombres de variables locales, y m_i etiquetas identificativas de los métodos del objeto.

El functor Obj se utilizará para representar objetos.

$$\mathsf{Obj}\ e \triangleq \mathit{Obj}\ [(\mathit{Name}, e)]\ [(\mathit{Label}, e)]$$

Se utilizarán también expresiones que permitan seleccionar un método m de un objeto o (o enviar el mensaje m al objeto o).

$$\mathsf{Send}\ e \triangleq e \mapsto \mathit{Label}$$

Ejemplo 8.4 *En la siguiente expresión se declara un objeto con una variable local de valor inicial 0, y los métodos set que modifica dicho valor, get que devuelve el valor de la variable local, y eq que comprueba si la variable local es mayor que la de otro objeto (obsérvese la utilización de la pseudo-variable self). La expresión devuelve 2 después de modificar el contenido de la celda.*

$$
\begin{array}{lll}
\textbf{let}\ p \;=\; \textbf{object} & & \\
\quad\quad \textbf{locals} & x \;=\; 0 & \\
\quad\quad \textbf{methods} & \mathit{get} \;=\; x & \\
 & \mathit{set} \;=\; \lambda n \rightarrow x \;:=\; n & \\
 & \mathit{eq} \;=\; \lambda p \rightarrow (\mathit{self} \mapsto \mathit{get} \;==\; p \mapsto \mathit{get}) & \\
\quad \textbf{in}\ p \mapsto \mathit{set}\ 2\,; & & \\
\quad\quad p \mapsto \mathit{get} & &
\end{array}
$$

Sintácticamente, las clases se representarán de forma similar a los objetos añadiendo la palabra **class** a la definición:

$$
\begin{array}{ll}
\textbf{class} & \\
\textbf{locals} & v_1 \;=\; e \\
 & \quad \ldots \\
 & v_i \;=\; e \\
\textbf{methods} & m_1 \;=\; e \\
 & \quad \ldots \\
 & m_j \;=\; e
\end{array}
$$

El functor Class representa la estructura anterior

$$\mathsf{Class}\; e \;=\; \mathit{Class}\; [(\mathit{Name}, e)]\; [(\mathit{Label}, e)]$$

Aplicando el operador *new* a una clase $\mathbb{C}$ se obtendrá un objeto de $\mathbb{C}$ (denominado *instancia* de dicha clase). La sintaxis será:

$$\mathsf{New}\; e \;=\; \mathit{new}\; e$$

Ejemplo 8.5 *La siguiente expresión es equivalente al ejemplo 8.4, salvo que en esta ocasión se declara una clase Cell y la correspondiente instancia.*

$$
\begin{array}{lll}
\textbf{let}\; \mathit{Cell} \;= & \textbf{class} & \\
 & \textbf{locals} & x \;=\; 0 \\
 & \textbf{methods} & \mathit{get} \;=\; x \\
 & & \mathit{set} \;=\; \lambda n \rightarrow x \;:=\; n \\
 & & \mathit{eq} \;=\; \lambda p \rightarrow (\mathit{self} \mapsto \mathit{get} \;==\; p \mapsto \mathit{get}) \\
 & \textbf{end} & \\
p \;= & \mathit{new}\; \mathit{Cell} & \\
 & \textbf{in} & p \mapsto \mathit{set}\; 2\; ; \\
 & & p \mapsto \mathit{get}
\end{array}
$$

Semánticamente, las clases no añaden nada nuevo. Sin embargo, desde un punto de vista de ingeniería del software, la utilización de clases tiene ciertas ventajas ya que no existen objetos que no pertenezcan a ninguna clase.

Una subclase se definirá a partir de una clase añadiendo nuevas variables locales y añadiendo o redefiniendo métodos de la clase base.

Sintácticamente, se representará de la siguiente forma

$$
\begin{array}{ll}
\textbf{subclass} & \\
\textbf{extends} & e \\
\textbf{locals} & v_1 \;=\; e \\
 & \ldots \\
 & v_i \;=\; e \\
\textbf{methods} & m_1 \;=\; e \\
 & \ldots \\
 & m_j \;=\; e
\end{array}
$$

Ejemplo 8.6 *La siguiente expresión declara una subclase de la clase Cell del ejemplo 8.5. Se añade una variable local cont que cuenta el número de veces que se llama al método set y se redefine dicho método. Obsérvese que en la redefinición se utiliza la pseudo-variable super para localizar el método set de la clase base.*

$$
\begin{array}{lrl}
\textbf{class} & & \\
\textbf{extends} & \mathit{Cell} & \\
\textbf{locals} & \mathit{cont} & = 0 \\
\textbf{methods} & \mathit{set} & = \lambda n \rightarrow (\mathit{super} \mapsto \mathit{set}; \mathit{cont} := \mathit{cont} + 1) \\
 & \mathit{getCont} & = \mathit{cont} \\
\textbf{end} & &
\end{array}
$$

La sintaxis de las subclases puede representarse mediante el functor SubClass

$$\mathsf{SubClass}\ e = \mathit{SubClass}\ e\ [(\mathit{Name}, e)]\ [(\mathit{Label}, e)]$$

El lenguaje Orientado a Objetos se formará como el punto fijo de la suma de los functores definidos en esta sección. Además, se utilizará el functor E definido en la sección 5.1 que permite incorporar expresiones aritméticas.

$$\mathcal{L}_{OO} \triangleq \mu(\mathsf{Obj} \oplus \mathsf{Send} \oplus \mathsf{Class} \oplus \mathsf{New} \oplus \mathsf{SubClass} \oplus \mathsf{E})$$

8.2.2 Dominio de valores

Un objeto se representará como un registro cuyos campos pueden ser computaciones. De forma abstracta, un registro puede considerarse una función que toma una etiqueta y devuelve un valor.

$$\mathit{Record}\ v \triangleq \mathit{Label} \rightarrow v$$

Se utilizarán las siguientes funciones auxiliares sobre valores de tipo *Record*

$$
\begin{array}{lll}
\mathit{emptyRec} : \mathit{Record}\ v & & \text{— Registro vacío} \\
(\mapsto) & : \mathit{Record}\ v \rightarrow \mathit{Label} \rightarrow v & \text{— Selecciona un campo} \\
 & & \text{— de un registro} \\
(\uplus) & : \mathit{Record}\ v \rightarrow [(\mathit{Label}, v)] \rightarrow \mathit{Record}\ v & \text{— Extiende un registro} \\
 & & \text{— añadiendo nuevos campos}
\end{array}
$$

Un objeto será un valor del siguiente tipo:

$$\mathit{Object} \triangleq \mathit{Record}\ (\mathsf{Comp}\ \mathit{Value})$$

Como se ha indicado, la modelización semántica de un objeto requiere la auto-referencia. Para resolverla, se utilizará el punto fijo de una función generadora. La función generadora tomará un objeto y devuelve un nuevo objeto.

$$\mathit{Generator} \triangleq \mathit{Object} \rightarrow \mathit{Object}$$

Una clase en un mecanismo que permite generar objetos. Se representará, por tanto, como una computación que devuelve un objeto.

$$Class \triangleq \mathsf{Comp}\ Object$$

En el caso de las subclases, la modelización dependerá de la utilización de enlace estático o enlace dinámico. En caso de enlace estático, una subclase se representará de la misma forma que una clase. En caso de enlace dinámico, se utiliza la codificación empleada en [199] en la que se representa una clase como una computación que devuelve un entorno y un generador de objetos.

$$SubClass \triangleq \mathsf{Comp}\,(Env,\ Generator)$$

Una vez definidos los dominios anteriores, el dominio de valores será la unión disjunta de dichos dominios junto con los dominios básicos *Int*, *Bool*. Se añade el dominio *Function* que, como se ha indicado en 7.3 se define como una función que toma una computación y devuelve una computación.

$$Value \triangleq Int \parallel Bool \parallel Loc \parallel Function \parallel Object \parallel Class \parallel SubClass$$

8.2.3 Estructura computacional

La estructura computacional de un lenguaje Orientado a Objetos es similar a la de un lenguaje imperativo. En este caso, se utiliza:

$$\mathsf{Comp} \triangleq (\mathcal{T}_{Err} \,.\, \mathcal{T}_{State} \,.\, \mathcal{T}_{Env})\,\mathsf{IO}$$

Es decir, se transforma una mónada de entrada/salida, añadiendo la posibilidad de modificar un estado, de acceder a un entorno y de computaciones parciales.

8.2.4 Especificación Semántica

Se proporcionan dos modelos de herencia, herencia con enlace estático (tipo C++ sin clases virtuales) y herencia con enlace dinámico (tipo Smalltalk y C++ con clases virtuales)

8.2.4.1 Funciones auxiliares

- $\mu : (\alpha \to \alpha) \to \alpha$

  $\mu\, f$ calcula el punto fijo de f. Obsérvese que la definición depende de que el lenguaje tenga una semántica no estricta. En caso contrario, generaría una computación divergente.

  $$\mu\, f \ = \ f\ (\mu\, f)$$

- $updLocals : Env \to [(Name, \mathsf{Comp}\ Value)] \to \mathsf{Comp}\ Env$

  $updLocals\ \rho\ ls$ devuelve el entorno resultante de actualizar ρ con las declaraciones ls

  $$\begin{array}{lll} updLocals\ \rho\ [] & = return\ \rho \\ updLocals\ \rho\ ((x,m) : xs) & = \mathbf{do} \\ & \qquad loc \leftarrow alloc\ m \\ & \qquad updLocals\ (upd_T\ \rho\ x\ (return \uparrow loc))\ xs \end{array}$$

- $mkGen : Object \rightarrow Env \rightarrow [(Label, \mathsf{Result})] \rightarrow Generator$

  $mkGen\ \tau\ \rho\ ms$ crea un generador a partir del objeto τ añadiendo una modificación de los métodos ms. La modificación consiste en ejecutarlos en el entorno resultante de añadir el objeto *self* a ρ.

$$
\begin{array}{l}
mkGen\ \tau\ \rho\ ms\ = \lambda self \rightarrow \tau\ \uplus\ [(l, modify\ m) | (l, m) \leftarrow ms] \\
\quad \textbf{where} \\
\quad\quad modify\ m\ =\ inEnv\ (upd_T\ \text{“}\mathsf{self}''\ (return\ \uparrow self)\ \rho)\ m
\end{array}
$$

- $mkGenSub : Env \rightarrow [(Label, \mathsf{Result})] \rightarrow Generator \rightarrow Generator$

  $mkGenSub\ \rho\ ms\ gen$ crea un nuevo generador a partir del generador de la clase base *gen* añadiendo una modificación de los métodos de la subclase ms. La modificación consiste en ejecutarlos en el entorno resultante de añadir al entorno las referencias al objeto *self* del generador actual y al objeto *super* obtenido mediante *gen self*.

$$
\begin{array}{l}
mkGenSub\ \rho\ ms\ gen\ = \lambda self \rightarrow gen\ self\ \uplus\ [(l, modify\ m) | (l, m) \leftarrow ms] \\
\quad \textbf{where} \\
\quad\quad modify\ m\ =\ \textbf{do} \\
\quad\quad\quad\quad\quad\quad \rho' \leftarrow rdEnv \\
\quad\quad\quad\quad\quad\quad inEnv\ (mkEnv\ \rho')\ m \\
\quad\quad mkEnv\ \rho'\ = (upd_T\ \text{“}\mathsf{self}''\ (return\ \uparrow self)\ . \\
\quad\quad\quad\quad\quad\quad upd_T\ \text{“}\mathsf{super}''\ (return\ \uparrow (g\ self)))\ (\rho +\!\!+ \rho')
\end{array}
$$

- $close : SubClass \rightarrow Class$

  $close\ m_{sub}$ obtiene una clase a partir de una subclase m_{sub}. Para ello, genera una computación que obtiene el generador de la subclase a partir de la subclase y aplica el punto fijo a dicho generador.

$$
\begin{array}{l}
close\ m_{sub}\ =\ \textbf{do} \\
\quad\quad\quad\quad (\rho, gen)\ \leftarrow\ m_{sub} \\
\quad\quad\quad\quad return\ (\mu\ gen)
\end{array}
$$

8.2.4.2 Declaración de objetos

El significado de una declaración de objeto se obtiene tomando el entorno actual ρ, añadiendo los valores de las variables locales a ρ, reservando una posición de memoria para el objeto obtenido como un punto fijo de la función generadora y devolviendo dicha posición.

$$
\begin{array}{l}
\varphi_{\mathsf{Obj}}\ (Obj\ ls\ ms)\ = \\
\textbf{do} \\
\quad \rho \leftarrow rdEnv \\
\quad \rho' \leftarrow updLocals\ \rho\ ls \\
\quad loc \leftarrow alloc\ (return\ \uparrow\ (\mu\ (mkGen\ emptyRec\ \rho'\ ms))) \\
\quad return\ \uparrow loc
\end{array}
$$

8.2.4.3 Envío de mensajes

$$\begin{aligned}\varphi_{\mathsf{Send}}\,(e \mapsto m) \;=\; & \mathbf{do}\\ & \quad v \leftarrow e\\ & \quad (\downarrow v) \mapsto m\end{aligned}$$

8.2.4.4 Herencia con enlace estático

A continuación se especifica la semánica de las clases y subclases tomando el modelo de herencia con enlace estático.

- Declaración de clases

$$\begin{aligned}&\varphi_{\mathsf{Class}}\,(\mathbf{class\ locals}\ ls\ \mathbf{methods}\ ms)\\ &\quad = \mathit{return} \uparrow (\mathit{mkObj}\ ls\ ms)\\ &\\ &\mathbf{where}\\ &\quad \mathit{mkObj}\ ls\ ms \;=\; \mathbf{do}\\ &\qquad\qquad \rho \leftarrow \mathit{rdEnv}\\ &\qquad\qquad \rho' \leftarrow \mathit{updLocals}\ \rho\ ls\\ &\qquad\qquad \mathit{return}\ (\mu\,(\mathit{mkGen}\ \mathit{emptyRec}\ \rho'\ ms))\end{aligned}$$

- Declaración de subclases

$$\begin{aligned}&\varphi_{\mathsf{SubClass}}\,(\mathbf{subclass}\ e\ \mathbf{locals}\ ls\ \mathbf{methods}\ ms)\\ &\quad = \mathbf{do}\\ &\qquad\quad \mathit{sup} \leftarrow e\\ &\qquad\quad \mathit{return} \uparrow (\mathit{mkObj}\ \mathit{sup}\ ls\ ms)\\ &\mathbf{where}\\ &\quad \mathit{mkObj}\ \mathit{sup}\ ls\ ms \;=\; \mathbf{do}\\ &\qquad\qquad\qquad \tau \leftarrow\!\downarrow \mathit{sup}\\ &\qquad\qquad\qquad \rho \leftarrow \mathit{rdEnv}\\ &\qquad\qquad\qquad \rho' \leftarrow \mathit{updLocals}\ \rho\ ls\\ &\qquad\qquad\qquad \mathit{return}\ (\mu\,(\mathit{mkGen}\ \tau\ \rho'\ ms))\end{aligned}$$

- Creación de instancias

$$\begin{aligned}&\varphi_{\mathsf{New}}\,(\mathit{new}\ e) \;=\\ &\quad \mathbf{do}\\ &\qquad v_{cls} \leftarrow e\\ &\qquad \tau \leftarrow\!\downarrow v_{cls}\\ &\qquad \mathit{loc} \leftarrow \mathit{alloc}\,(\mathit{return} \uparrow \tau)\\ &\qquad \mathit{return} \uparrow \mathit{loc}\end{aligned}$$

8.2.4.5 Herencia con enlace dinámico

Con enlace estático, el punto fijo se realizaba en la definición de la clase y de la subclase. El problema es que, en ese momento, la variable *self* se refiere a la clase particular que se está definiendo. Para modelizar la herencia con enlace dinámico, es necesario retardar el cálculo del punto fijo hasta la creación del objeto, para que la variable *self* pueda tomar la clase particular de dicho objeto.

- Declaración de clases

$$
\begin{array}{l}
\varphi_{\mathsf{Class}}\ (\mathbf{class\ locals}\ ls\ \mathbf{methods}\ ms) \\
\quad =\ return\ \uparrow (mkCls\ ls\ ms) \\
\\
\quad \mathbf{where} \\
\quad\ mkCls\ ls\ ms\ =\ \mathbf{do} \\
\qquad\qquad\qquad \rho\ \leftarrow\ rdEnv \\
\qquad\qquad\qquad \rho'\ \leftarrow\ updLocals\ \rho\ ls \\
\qquad\qquad\qquad return\ (\rho',\ mkGen\ emptyRec\ \rho'\ ms)
\end{array}
$$

- Declaración de subclases

$$
\begin{array}{l}
\varphi_{\mathsf{SubClass}}\ (\mathbf{subclass}\ e\ \mathbf{locals}\ ls\ \mathbf{methods}\ ms) \\
\quad\ =\ \mathbf{do} \\
\qquad\ sup\ \leftarrow\ e \\
\qquad\ return\ \uparrow (mkSCls\ sup\ ls\ ms) \\
\quad \mathbf{where} \\
\quad\ mkSCls\ sup\ ls\ ms\ =\ \mathbf{do} \\
\qquad\qquad\qquad (\rho, gen)\ \leftarrow \downarrow sup \\
\qquad\qquad\qquad \rho'\ \leftarrow\ updLocals\ \rho\ ls \\
\qquad\qquad\qquad return\ (\rho',\ mkGenSub\ \rho'\ ms\ gen)
\end{array}
$$

- Creación de instancias

$$
\begin{array}{l}
\varphi_{\mathsf{New}}\ (new\ e)\ = \\
\quad \mathbf{do} \\
\qquad v_{scls}\ \leftarrow\ e \\
\qquad \tau\ \leftarrow\ close\ (\downarrow v_{scls}) \\
\qquad loc\ \leftarrow\ alloc\ (return\ \uparrow \tau) \\
\qquad return\ \uparrow loc
\end{array}
$$

A partir de las álgebras, es posible definir el intérprete mediante el siguiente catamorfismo

$$
\begin{array}{l}
\mathsf{Inter}_{\mathcal{L}_{OO}}\ :\ \mathcal{L}_{OO}\ \rightarrow\ \mathsf{Comp}\ Value \\
\mathsf{Inter}_{\mathcal{L}_{OO}}\ =\ (\!|\varphi_{\mathsf{Obj}} \oplus \varphi_{\mathsf{Send}} \oplus \varphi_{\mathsf{Class}} \oplus \varphi_{\mathsf{New}} \oplus \varphi_{\mathsf{SubClass}} \oplus \varphi_{\mathsf{E}}|\!)
\end{array}
$$

Capítulo 9

Especificación Lenguaje de Programación Lógica

The meaning of programs expressed in logic, on the other hand, can be defined in machine independent, human-oriented terms. As a consequence, logic programs are easier to construct, easier to understand, easier to improve, and easier to adapt to other purposes. R. Kowalski [128]

9.1 Conceptos de Programación Lógica

A principios de los años 70 R. Kowalski propone la utilización de técnicas de demostración automática de teoremas para la construcción de programas. Propone la fórmula

$$Algoritmo = Lógica + Control$$

Indicando que cualquier algoritmo para resolver un problema puede descomponerse en una descripción de *qué* es lo que se va a resolver (lógica), junto con una descripción de *cómo* resolverlo (control).

Mientras que en la programación imperativa tradicional, el programador debe especificar ambas componentes, en *programación lógica* [128, 150, 10] se pretende liberar al programador de especificar la parte de control, dejando al sistema que la resuelva de forma automática.

De esta forma, los programas serán más fáciles de desarrollar ya que el programador se encarga de menos tareas. Además el sistema puede optimizar de forma independiente la parte de control utilizando, por ejemplo, arquitecturas paralelas. La principal desventaja es que, en muchas ocasiones, el sistema no es capaz de superar al programador humano a la hora de optimizar la parte de control por lo que estos sistemas adolecen de una cierta ineficiencia.

La principal aplicación de este tipo de lenguajes ha sido en el campo de la *Inteligencia Artificial*, en el que habitualmente se conoce qué se quiere resolver pero no cómo resolverlo.

En 1972, A. Colmerauer, basándose en las ideas de R. Kowalski, crea el primer intérprete del lenguaje Prolog con el fin de facilitar las tareas de reconocimiento del lenguaje natural en la Universidad de Marsella.

Posteriormente, a principios de los 80, D. Warren desarrolla en la Universidad de Edimburgo el primer compilador utilizando una máquina abstracta. La eficiencia adquirida por la implementación de Edimburgo logra que la comunidad científica preste mayor atención a la programación lógica.

El proyecto japonés que pretendía desarrollar ordenadores de quinta generación basados en inferencias lógicas logra un crecimiento del interés industrial por este tipo de sistemas, levantando fuertes expectativas sobre las aplicaciones de la programación lógica para desarrollo de sistemas inteligentes.

A mediados de los años 90 se desarrolla el primer estándar del lenguaje Prolog según la norma ISO [48, 131].

Mientras que en programación funcional se ha prestado gran atención al desarrollo de nuevos lenguajes, existiendo cierta competencia entre diferentes lenguajes, Prolog ha tenido una posición dominante en programación lógica. La capacidad de elección en este paradigma se centra más en las diferentes implementaciones (muchas de ellas comerciales) que en los diferentes lenguajes. Las nuevas líneas de investigación en programación lógica pueden clasificarse en:

- Programación lógica paralela (Parlog [43]
- La *Programación lógica con restricciones* o *Constraint Logic Programming*
- Nuevos Lenguajes declarativos como Escher [53], Goedel [81], Mercury [212], λProlog [179]), etc.

El Prolog ha sido objeto de varios intentos de descripción semántica. En [181] se presenta una semántica denotacional, en [26] se presenta la semántica operacional. En [86] se describe una semántica axiomática con lógica ecuacional que servirá de base para la derivación de una mónada con backtracking en [87]. Dicha técnica se emplea en [42] para empotrar variables lógicas en Haskell y ha servido de inspiración para la codificación de expresiones Prolog de esta sección.

A continuación se realiza una breve introducción al paradigma de la programación lógica.

9.1.1 Algoritmo de unificación

Definición 9.1 (Término) *Un término es una constante, una variable o un término compuesto de la forma $f(t_1, t_2, \ldots, t_n)$ donde f es un identificador (llamado functor), t_i son términos y n es la aridad.*

En Prolog, las variables se distinguen de las constantes porque las primeras comienzan por mayúscula.

Ejemplo 9.1 *a es una constante y a la vez un término. De la misma forma, x es una variable y a la vez un término. Mientras que $p(a, x)$ es un término compuesto cuyo functor es p y cuya aridad es 2*

Definición 9.2 (Substitución) *Una substitución σ es un conjunto de la forma $\{v_1/t_1, \ldots, v_n/t_n\}$ donde v_i son variables distintas entre sí, t_i son términos y $v_i \neq t_i$.*

Ejemplo 9.2 *$\{x/f(y), y/b\}$ es una substitución.*

Definición 9.3 (Instancia) *Dado un término t y una substitución σ, la* instancia *de t según σ se denota como $\sigma(t)$ y se define como la substitución reiterada de todas las variables de t por los términos asociados en la substitución σ*

Ejemplo 9.3 *Sea $t = p(g(x, y), g(a, x))$ y $\sigma = \{x/f(y), y/b\}$, entonces $\sigma(t) = p(g(f(b), b), g(a, f(b)))$*

Definición 9.4 (Substitución vacía) *La substitución vacía está formada por el conjunto vacío y se denota como $\{\}$.*

Definición 9.5 (Actualización) *Dada una substitución $\sigma = \{v_1/t_1, \ldots, v_n/t_n\}$, una variable v y un término t, la actualización de v por t en σ, se denota como $\sigma \rhd \{v/t\}$, y es la substitución resultante de añadir el par $\{v/t\}$ a σ si $v \notin \{v_1, \ldots, v_n\}$ o de cambiar t_i por t en σ en caso contrario.*

Definición 9.6 (Composición de substituciones) *La composición de dos substituciones σ_1 y σ_2 se denota como $\sigma_1 \circ \sigma_2$ y se define como la substitución que cumple que $\sigma_1 \circ \sigma_2(t) = \sigma_1(\sigma_2(t))$.*

Definición 9.7 (Unificador) *Se dice que una substitución ω es un* unificador *de dos términos t_1 y t_2 si $\omega(t_1) = \omega(t_2)$.*

Ejemplo 9.4 *La substitución $\{x/f(a), v/a, y/b, w/b\}$ es un unificador de los términos $t_1 = p(x, g(y, a))$ y $t_2 = p(f(v), g(w, v))$.*

Definición 9.8 (Unificador más general) *Un unificador ω es el unificador más general, si para cualquier otro unificador ω' existe una substitución σ tal que $\omega' = \omega \circ \sigma$.*

Ejemplo 9.5 *$\omega = \{x/f(a), v/a, y/w$ es un unificador más general de los términos del ejemplo 9.4.*

En 1930, Herbrand presentó un algoritmo de unificación que sería rescatado posteriormente por Robinson en 1965 como una pieza fundamental del algoritmo de resolución. El algoritmo de unificación permite calcular el unificador más general de dos términos, si existe, o detecta que no son unificables. Para un tratamiento en mayor profundidad del algoritmo de unificación puede consultarse [124].

La versión del algoritmo que a continuación se presenta ha sido tomada de [102], donde se desarrolla un algoritmo de unificación politípico. La genericidad de este algoritmo se obtiene mediante la definición de clases de tipos y la declaración de instancias. En la versión que se expone se omiten dichas declaraciones.

```
unify      :  t → t → Maybe (Subst t)
unify t1 t2  = unify_S t1 t2 {}
```

$$
\begin{array}{l}
unify_S \quad : t \to t \to Subst\ t \to Maybe\ (Subst\ t) \\
unify_S\ t_1\ t_2\ \sigma \mid isVar\ t_1 \wedge isVar\ t_2 \wedge t_1 == t_2 = Just\ \sigma \\
unify_S\ t_1\ t_2\ \sigma \mid isVar\ t_1 = bind\ t_1\ t_2\ \sigma \\
unify_S\ t_1\ t_2\ \sigma \mid isVar\ t_2 = bind\ t_2\ t_1\ \sigma \\
unify_S\ t_1\ t_2\ \sigma \mid \textbf{otherwise} = \textbf{if}\ topEq\ t_1\ t_2\ \textbf{then} \\
\qquad\qquad\qquad\qquad\qquad\qquad uniTs\ t_1\ t_2\ \sigma \\
\qquad\qquad\qquad\qquad\qquad\ \textbf{else} \\
\qquad\qquad\qquad\qquad\qquad\qquad Nothing
\end{array}
$$

$$
\begin{array}{l}
uniTs : t \to t \to Subst\ t \to Maybe\ t \\
uniTs\ t_1\ t_2\ \sigma = foldr\ uni\ \sigma\ (zip\ (args\ t_1)\ (args\ t_2)) \\
\quad \textbf{where} \\
\qquad uni\ (a_1, a_2)\ r = \textbf{case}\ r\ \textbf{of} \\
\qquad\qquad\qquad\qquad\qquad Just\ \sigma' \to unify_S\ a_1\ a_2\ \sigma' \\
\qquad\qquad\qquad\qquad\qquad Nothing \to Nothing
\end{array}
$$

Se han utilizado las funciones auxiliares

$$
\begin{array}{ll}
args : t \to [t] & \text{— Devuelve lista de argumentos de un término} \\
topEq : t \to t \to Bool & \text{— Comprueba que dos términos son semejantes}
\end{array}
$$

y el tipo de datos predefinido *Maybe*

$$Maybe\ \alpha \triangleq Just\ \alpha \mid Nothing$$

La función $bind\ v\ t\ \sigma$ intenta asignar el término t a la variable v en la substitución σ. Si no puede, devuelve *Nothing*.

$$
\begin{array}{l}
bind \quad : Name \to t \to Subst\ t \to Maybe\ (Subst\ t) \\
bind\ v\ t\ \sigma = \textbf{if}\ \boxed{x \in \sigma(t)}\ \textbf{then} \\
\qquad\qquad\qquad\quad Nothing \\
\qquad\qquad\quad \textbf{else} \\
\qquad\qquad\qquad \textbf{case}\ lkp_S\ \sigma\ v\ \textbf{of} \\
\qquad\qquad\qquad\quad Nothing \to Nothing \\
\qquad\qquad\qquad\quad Just\ \sigma' \to Just\ (\sigma \rhd \{v/t\})
\end{array}
$$

donde $lkp_S : Subst\ t \to Name \to t$ es una función que busca el valor de una variable en una substitución.

En el algoritmo anterior, la expresión $x \in \sigma(t)$ encerrada en una caja se denomina *chequeo de ocurrencias* y en los sistemas Prolog convencionales se omite debido a la ineficiencia que acarrea.

9.1.2 Algoritmo de resolución

Definición 9.9 (Átomo) *Si p es un identificador y $t_1, \ldots, t_n$ son términos, entonces $p(t_1, \ldots, t_n)$ es un* átomo.

Definición 9.10 (Objetivo) *Un* objetivo *es una secuencia finita de átomos. Se representa normalmente como $? - A_1, \ldots, A_n$.*

Definición 9.11 (Cláusula) *Una* cláusula *es una construcción de la forma* $A \leftarrow B$ *donde* A *es un átomo y* B *es un objetivo.* A *se denomina habitualmente* cabeza de la cláusula *y* B cuerpo de la cláusula. *Si el objetivo no tiene átomos, entonces la cláusula se denomina* hecho *y se suprime la flecha. Si el objetivo tiene átomos, entonces la cláusula se denomina* regla.

Definición 9.12 (Procedimiento Prolog) *Un* procedimiento Prolog *es un conjunto de cláusulas cuya cabeza está formada por un átomo con el mismo functor y aridad.*

Ejemplo 9.6 *A continuación se presenta un procedimiento Prolog formado por un hecho y una regla.*

$p(a, b)$
$p(b, x) \leftarrow p(a, x)$

Definición 9.13 (Programa Prolog) *Un programa Prolog es una secuencia finita de procedimientos Prolog.*

Es conveniente observar que las variables de los objetivos están cuantificadas existencialmente, mientras que las variables de las cláusulas están cuantificadas universalmente.

Ejemplo 9.7 *Si se desea formalizar el siguiente razonamiento "Juan ama a María, María ama a los que ama Juan. Por tanto alguien se ama a sí mismo" en lógica de predicados, podría obtenerse :*

$$\frac{\begin{array}{c} ama(juan, maria) \\ \forall x(ama(juan, x) \rightarrow ama(maria, x)) \end{array}}{\exists x(ama(x, x))}$$

En programación lógica se escribiría

$ama(juan, maria)$
$ama(maria, x) \leftarrow ama(juan, x)$
$?\ ama(x, x)$

De manera informal, el proceso computacional seguido en programación lógica puede resumirse de la siguiente forma: Un programa P puede considerarse como un conjunto de axiomas y objetivo Q es una petición para encontrar una substitución σ tal que $\sigma(Q)$ sea una consecuencia de P.

Tal computación se construye mediante el denominado *algoritmo de resolución.*

Definición 9.14 (Algoritmo de resolución) *El algoritmo de resolución se basa en una secuencia de pasos básicos. Cada paso consiste en la selección de un átomo* A_i *del objetivo y en una cláusula* $A \leftarrow B$ *del programa. Si* A_i *unifica con* A *entonces el siguiente objetivo se obtiene substituyendo* A_i *por* B *y aplicando al resultado el unificador más general de* A_i *y* A.

La computación termina con éxito cuando se alcanza el objetivo vacío. La substitución de respuesta será la composición de todos los unificadores utilizados.

Ejemplo 9.8 *Al aplicar el algoritmo de resolución al ejemplo 9.7 se obtiene la siguiente secuencia*

$$\exists x\ ama(x,x)$$
$$\downarrow \{x/maria\}$$
$$ama(juan, maria)$$
$$\downarrow \{\}$$
$$\square$$

Obsérvese que el algoritmo descrito no es determinista. En cada paso existen varias indeterminaciones:

- En caso de que el objetivo actual tenga varios átomos, ¿qué átomo se escoge?
- Si hay varias cláusulas del programa cuya cabeza unifique con el átomo seleccionado, ¿qué cláusula se escoge?

La política que resuelve la primera pregunta será la *regla de computación*, mientras que la *regla de búsqueda* será la que dictamine la solución de la segunda pregunta.

Definición 9.15 (Regla de computación) *La* regla de computación *indica cuál de los átomos se selecciona en el objetivo.*

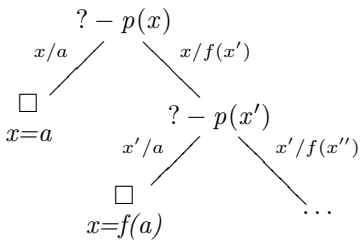
Definición 9.16 (Regla de búsqueda) *La* regla de búsqueda *indica cuál de las cláusulas del programa se utiliza.*

Definición 9.17 (Árbol de resolución) *Dado un programa, un objetivo y una regla de computación, el* árbol de resolución *se forma construyendo todos los caminos de resolución que se pueden formar a partir de dicho objetivo.*

Ejemplo 9.9 *A partir del programa Prolog*

$p(a).$
$p(f(x)) \leftarrow\ p(x)$

y el objetivo $? - p(x)$*, se desarrolla el siguiente árbol de resolución*



A la hora de buscar el objetivo vacío se pueden utilizar dos estrategias fundamentales:

- *Primero en profundidad*, intenta buscar el objetivo vacío a partir de una rama hasta que no pueda avanzar. En cuyo caso deshace parte del camino realizado y lo intenta por otra rama. Esta vuelta hacia atrás se conoce como *backtracking*.
- *Primero en anchura* va recorriendo el árbol por niveles.

De forma resumida, el lenguaje Prolog es una implementación del algoritmo de resolución que ha tomado las siguientes decisiones de diseño:

- La regla de computación selecciona el primer átomo del objetivo.
- La regla de búsqueda selecciona la primera cláusula tomadas según el orden en que aparecen en el programa.
- La estrategia de recorrido del árbol es primero en profundidad.

9.2 Lenguaje Programación Lógica Pura

En esta sección se realizará una especificación del núcleo puramente declarativo de un lenguaje de programación lógica.

9.2.1 Estructura sintáctica

Para la definición de términos, Se utilizará el punto fijo del siguiente functor:

$$\begin{array}{lll} \mathsf{T}\ t \triangleq & C\ Name & \text{— Constantes} \\ \mid & V\ Name & \text{— Variables} \\ \mid & F\ Name\ [t] & \text{— Términos compuestos} \end{array}$$

$$Term \triangleq \mu\mathsf{T}$$

En Haskell, una substitución puede definirse como una función

$$Subst\ t \triangleq Name \to t$$

Sintácticamente, las cláusulas (hechos o reglas) se representarán como las declaraciones locales de un lenguaje funcional, dejando el objetivo como una expresión a ejecutar.
Se utilizará el siguiente functor:

$$\begin{array}{lll} \mathsf{P}\ e \triangleq & Def\ Name\ [Name]\ e\ e & \text{— Definiciones} \\ \mid & e \wedge e & \text{— Conjunción} \\ \mid & e \vee e & \text{— Disyunción} \\ \mid & \exists(Name \to e) & \text{— Variables libres} \\ \mid & call\ Name\ [Term] & \text{— Llamada a predicado} \\ \mid & Term \stackrel{\circ}{=} Term & \text{— Unificación} \\ \mid & ?\ Name\ (Name \to e) & \text{— Objetivo} \end{array}$$

El lenguaje Prolog se formará a partir del punto fijo de P

$$Prolog \triangleq \mu\mathsf{P}$$

Ejemplo 9.10 *El programa Prolog*

$p(a).$
$p(f(x)) :- p(x).$

con el objetivo $? \; p(x).$ *se representaría de la siguiente forma:*

$$\begin{array}{l} \mathit{Def}\ p\ [v_1]\ (v_1 \stackrel{\circ}{=} a\ \vee\ (\exists(\lambda x \rightarrow v_1 \stackrel{\circ}{=} f(x)\ \wedge\ \mathit{call}\ p\ [x]))) \\ \qquad\qquad (?\ x\ (\lambda x \rightarrow \mathit{call}\ p\ [x])) \end{array}$$

9.2.2 Estructura computacional

La estructura computacional requiere las siguientes condiciones:

- Backtracking.
- Acceso al entorno en el que se localizarán las definiciones de predicados y la substitución actual.
- Modificación de un estado que proporcione nuevos nombres de variables. En este estado se puede almacenar simplemente un contador del número de variable generado hasta el momento. Cuando se vaya a asignar un nombre nuevo a una variable libre se incrementará este contador.
- Gestión de situaciones excepcionales.

La mónada resultante será la siguiente:

$$\mathsf{Comp} \triangleq (\mathcal{T}_{Back} \,.\, \mathcal{T}_{Env} \,.\, \mathcal{T}_{State} \,.\, \mathcal{T}_{Err})\ IO$$

El entorno estará formado por la substitución actual y la base de datos, la cual asocia a cada par (nombre de predicado,aridad) una lista de nombres de argumentos, junto con la computación correspondiente.

$$\begin{array}{lcl} \mathit{Database} & \triangleq & (\mathit{Name}, \mathit{Int}) \rightarrow ([\mathit{Name}], \mathsf{Comp}\ \mathit{Subst}) \\ \mathit{Env} & \triangleq & (\mathit{Database}, \mathit{Subst}\ \mathit{Term}) \end{array}$$

El estado global consiste en un número entero a partir del cual se puede generar un nombre de variable libre.

$$\mathit{State} \triangleq \mathit{Int}$$

Los valores que devolverá el sistema Prolog serán substituciones.

$$\mathit{Value} \triangleq \mathit{Subst}\ \mathit{Term}$$

9.2.3 Especificación Semántica

$$
\begin{array}{l}
\varphi_{\mathsf{P}}\ (\mathit{Def}\ p\ \mathit{vs}\ e_1\ e_2)\ = \\
\quad \textbf{do} \\
\qquad (\rho, \sigma)\ \leftarrow\ \mathit{rdEnv} \\
\qquad \mathit{inEnv}\ (\mathit{upd}_T\ (p, \mathit{length}\ \mathit{vs})\ (\mathit{vs}, e_1)\ \rho, \sigma)\ e_2
\end{array}
$$

$$
\begin{array}{l}
\varphi_{\mathsf{P}}\ (e_1\ \wedge\ e_2)\ = \\
\quad \textbf{do} \\
\qquad (\rho, \sigma)\ \leftarrow\ \mathit{rdEnv} \\
\qquad \sigma'\ \leftarrow\ e_1 \\
\qquad \mathit{inEnv}\ (\rho, \sigma')\ e_2
\end{array}
$$

$$
\begin{array}{l}
\varphi_{\mathsf{P}}\ (e_1\ \vee\ e_2)\ = \\
\quad \textbf{do} \\
\qquad (\rho, \sigma)\ \leftarrow\ \mathit{rdEnv} \\
\qquad \mathit{inEnv}\ (\rho, \sigma)\ e_1\ \Cup\ \mathit{inEnv}\ (\rho, \sigma)\ e_2
\end{array}
$$

$$
\begin{array}{l}
\varphi_{\mathsf{P}}\ (\exists\, f)\ = \\
\quad \textbf{do} \\
\qquad n\ \leftarrow\ \mathit{update}\ (+1) \\
\qquad f\ (\mathit{mkFree}\ n)
\end{array}
$$

$$
\begin{array}{l}
\varphi_{\mathsf{P}}\ (\mathit{call}\ p\ \mathit{ts})\ = \\
\quad \textbf{do} \\
\qquad (\rho, \sigma)\ \leftarrow\ \mathit{rdEnv} \\
\qquad (\mathit{vs}, m)\ \leftarrow\ \mathit{lkp}_T\ (p, \mathit{length}\ \mathit{ts})\ \rho \\
\qquad \sigma'\ \leftarrow\ \mathit{addBinds}\ \mathit{vs}\ \mathit{ts}\ \sigma \\
\qquad \mathit{inEnv}\ (\rho, \sigma')\ m
\end{array}
$$

$$
\begin{array}{l}
\varphi_{\mathsf{P}}\ (t_1 \stackrel{\circ}{=} t_2)\ = \\
\quad \textbf{do} \\
\qquad (\rho, \sigma)\ \leftarrow\ \mathit{rdEnv} \\
\qquad \textbf{case}\ \mathit{unify}_S\ t_1\ t_2\ \sigma\ \textbf{of} \\
\qquad\quad \mathit{Nothing}\ \rightarrow \mathit{failure} \\
\qquad\quad \mathit{Just}\ \sigma'\ \ \rightarrow \mathit{return}\ \sigma'
\end{array}
$$

$$
\begin{array}{l}
\varphi_{\mathsf{P}}\ (?\ x\ f)\ = \\
\quad \textbf{do} \\
\qquad n\ \leftarrow\ \mathit{update}\ (+1) \\
\qquad \textbf{let}\ v\ =\ \mathit{mkFree}\ n \\
\qquad \sigma\ \leftarrow\ f\ v \\
\qquad \mathit{putAnswer}\ x\ (\sigma(v))
\end{array}
$$

Se han usado las siguientes funciones auxiliares:

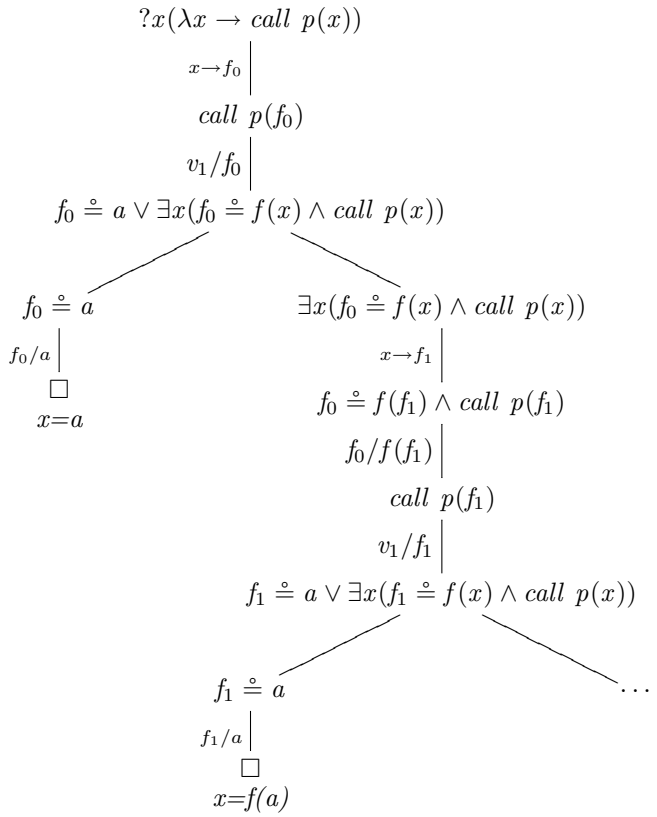
- $\mathit{addBinds}\ : [\mathit{Name}] \rightarrow [\mathit{Term}] \rightarrow \mathit{Subst} \rightarrow \mathit{Subst}$ crea una nueva substitución asignando a las variables los términos correspondientes.

- *mkFree* : *Int* → *Name* crea un nuevo nombre de variable a partir de un número.
- *putAnswer* : *Name* → *Term* → Comp *Value* imprime en pantalla la substitución de respuesta obtenida.

El intérprete del lenguaje Prolog se obtiene mediante un catamorfismo

$$\begin{aligned}&\mathsf{Inter}_{Prolog} \;:\; Prolog \to \mathsf{Comp}\,(Subst\ Term)\\&\mathsf{Inter}_{Prolog} \;= (\!|\varphi_{\mathsf{P}}|\!)\end{aligned}$$

Ejemplo 9.11 *A partir de la especificación semántica planteada, el árbol de resolución del ejemplo 9.10 podría representarse como:*



9.3 Añadiendo capacidades aritméticas

Con el fin de realizar operaciones aritméticas de forma eficiente, el lenguaje Prolog contiene varios predicados extra-lógicos, como el predicado (*is*) que permiten evaluar de forma aritmética sus argumentos. La mayoría de las especificaciones del lenguaje Prolog, como [26, 181, 213, 214], suelen evitar los predicados aritméticos ya que su comprensión puede interferir con las peculiaridades computacionales propias del lenguaje.

Con el sistema utilizado en esta tesis, la incorporación de este tipo de características se realiza de forma directa reutilizando bloques semánticos previamente definidos. En particular, se combinará el bloque semántico de términos aritméticos definido en el ejemplo 5.10 con el núcleo del lenguaje Prolog definido en la sección anterior.

Puesto que se van a combinar dos categorías sintácticas diferentes, la estructura sintáctica del predicado (*is*) se define mediante el siguiente bifunctor.

$$\mathbb{I}\ g\ e\ \triangleq\ \mathit{Term}\ \mathbf{is}\ e$$

cuya especificación semántica será

$$\begin{array}{ll} \varphi_{\mathbb{I}} & :\ \mathbb{I}\,(\mathsf{Comp}\ \mathit{Value})\,(\mathsf{Comp}\ \mathit{Int}) \rightarrow \mathsf{Comp}\ \mathit{Value} \\ \varphi_{\mathbb{I}}\,(t\ \mathbf{is}\ e) & = v \leftarrow e \\ & \quad (\rho, \sigma) \leftarrow \mathit{rdEnv} \\ & \quad \mathit{unify}_S\ t\ (\mathit{cnv}\ v)\ \sigma \end{array}$$

donde $\mathit{cnv} : \mathit{Int} \rightarrow \mathit{Term}$ convierte un número entero en un término constante.

La estructura sintáctica del lenguaje de Programación lógica con el predicado aritmético puede definirse como

$$\mathit{Prolog}^{+}\ \triangleq\ \mu_1\,(\mathsf{P}_1^2\ \boxplus_1\ \mathbb{I})\,\mathsf{E}_2^2$$

y el correspondiente intérprete se obtiene de forma automática mediante un bicatamorfismo

$$\begin{array}{ll} \mathsf{Inter}_{\mathit{Prolog}^{+}} & :\ \mathit{Prolog}^{+} \rightarrow \mathsf{Comp}\ \mathit{Subst} \\ \mathsf{Inter}_{\mathit{Prolog}^{+}} & = (\!|\epsilon_1^2\varphi_{\mathsf{P}} \boxplus_1 \varphi_{\mathbb{I}}, \epsilon_2^2\varphi_{\mathsf{E}}|\!)_1 \end{array}$$

Como puede observarse, la inclusión de capacidades aritméticas al lenguaje Prolog puro no ha supuesto ninguna modificación a las especificaciones anteriores.

Capítulo 10

Conclusiones

One aspect of doing science consists in choosing the dividing line between the relevant and the irrelevant. E. W. Dijkstra [51]

10.1 Comparación de Técnicas de Especificación Semántica

En la tabla 10.1 se presenta una valoración comparada de las diferentes técnicas de especificación semántica. La comparación entre estas características es cualitativa, no cuantitativa. Por ese motivo, se ha intentado huir de las valoraciones numéricas que podrían dar lugar a juicios equívocos. Se utiliza el siguiente esquema: si el formalismo X admite la característica Y, entonces en la celda (X, Y) aparece el símbolo ●. Si admite la característica de forma parcial, se utiliza el símbolo ◗. Finalmente, si no admite la característica, no se pone nada. La tabla también incluye el número de página en el que se ha realizado la valoración de cada técnica particular.

- No Ambigüedad (NAM). Evidentemente, el lenguaje natural es ambigüo y poco riguroso. En ese sentido, todos los formalismos resuelven el problema de la ambigüedad mediante especificaciones de raíz matemática. En el

	Pág.	NAM	MOD	REU	DEM	PRO	LEG	FLE	EXP
Leng. Natural	27		●	●			◗	●	●
Sem. Operacional Est.	31	●			◗	◗	◗	●	●
Sem. Natural	33	●			◗	●	◗	●	●
Sem. Denotacional	37	●			●	◗		●	◗
Sem. Axiomática	39	◗			●		◗		●
Sem. Algebraica	45	●		◗	●	●	◗	◗	◗
Máq. Estado Abstracto	49	●	●	◗	◗	●		◗	●
Sem. de Acción	54	●	●	◗	◗	●	●	◗	◗
Sem. Monádica Modular	74	●	●		●	◗		●	
Sist. Prot. Lenguajes	106	●	●	●	●	●		●	

Tabla 10.1: Comparación entre técnicas de especificación semántica

caso de la semántica axiomática pueden surgir problemas de ambigüedad debido a la no especificación de ciertas características.

- Modularidad semántica (MOD). La modularidad semántica es alcanzada en el lenguaje natural debido a la falta de rigurosidad, que permite realizar descripciones ligeramente vagas, que se adaptan a la introducción de nuevas características semánticas independientes. Los formalismos tradicionales no resuelven este problema y quizás sea ésa una de las razones de su escasa utilización práctica [242]. Las máquinas de estado abstracto resuelven el problema mediante la especialización de la abstracción del estado, la semántica de acción lo resuelve mediante la especialización de facetas y la semántica monádica modular junto con el sistema de prototipado de lenguajes, lo resuelven mediante transformadores de mónadas.

- Componentes semánticos reutilizables (REU). El lenguaje natural, por la misma razón anterior, admite la reutilización de las especificaciones, las cuales siguen la modelización conceptual de las características del lenguaje. Los formalismos tradicionales no atacan este problema y por tanto, tampoco lo resuelven. La semántica algebraica contiene potentes mecanismos de reutilización, aunque las especificaciones realizadas en este formalismo son monolíticas [63]. En la semántica de acción se han planteado soluciones al problema, aunque no han sido implementadas [49] y parece que su implementación traería problemas de inconsistencias. El sistema de prototipado de lenguajes desarrollado en esta tesis resuelve el problema, admitiendo la definición y reutilización de componentes semánticos que pueden incorporarse a un lenguaje de forma independiente. El sistema no produce inconsistencias al separar en categorías sintácticas diferentes las entidades que se van incorporando.

- Demostración de propiedades (DEM). El lenguaje natural no admite la demostración de propiedades de los lenguajes especificados. Las otras técnicas admiten la demostración de propiedades con mayor medida cuanto menos operacionales sean.

  La semántica axiomática y algebraica facilitan el razonamiento formal al utilizar axiomas independientes de una implementación particular. De la misma forma, la semántica denotacional describe el lenguaje mediante conceptos matemáticos abstractos con los que se pueden realizar demostraciones. La semántica monádica modular y el Sistema de Prototipado de Lenguajes tienen su base en la combinación entre semántica denotacional y semántica algebraica, ya que los conceptos del lenguaje se definen a partir de mónadas, los cuales tienen un comportamiento definido a partir de axiomas.

- Prototipos (PRO). Los sistemas operacionales facilitan la creación de prototipos. De hecho, todos los sistemas que tienen una base operacional (semántica operacional estructurada, semántica natural, máquinas de estado abstracto y semántica de acción) permiten el desarrollo de prototipos. La semántica axiomática tradicional no permite esta posibilidad, ya que la definición de los axiomas se realiza precisamente teniendo en cuenta la independencia de la implementación. A diferencia de ella, la semántica

algebraica, al basarse el lógica ecuacional, permite desarrollar prototipos mediante técnicas de reescritura. Por la misma razón, la semántica monádica modular y el sistema de prototipado de lenguajes permiten obtener prototipos de forma directa.

- Legibilidad (LEG). El único sistema que ha prestado gran importancia a la legibilidad evitando además los problemas de ambigüedad del lenguaje natural, es la semántica de acción. Podría ser interesante incorporar las facilidades sintácticas de la semántica de acción en el sistema de prototipado de lenguajes mediante la definición de un metalenguaje de dominio específico.

- Flexibilidad (FLE). Las técnicas operacionales y denotacionales se adaptan a la definición de lenguajes de diferentes paradigmas. Sin embargo, es más difícil desarrollar una semántica axiomática para nuevos lenguajes y paradigmas, lo cual requiere cierta experiencia.

  Como ya se ha indicado, el sistema de prototipado de lenguajes, al basarse en la semántica denotacional, admite la descripción de lenguajes de diversos paradigmas.

- Experiencia (EXP). Las técnicas tradicionales, especialmente, las técnicas de base operacional, han sido aplicadas a la descripción de lenguajes *reales*. En teoría, el sistema de prototipado de lenguajes desarrollado en esta tesis podría utilizarse para la especificación de lenguajes de programación prácticos.

10.2 Discusión General

En esta tesis doctoral se ha desarrollado un estudio comparativo de las principales técnicas de descripción semántica de lenguajes. Entre los puntos débiles de las técnicas existentes, la falta de modularidad semántica y la imposibilidad de creación de componentes semánticos reutilizables suponen una limitación importante para su aplicación generalizada.

Para evitar tal limitación, se ha propuesto una nueva técnica basada en la combinación de semántica monádica modular y conceptos de programación genérica. De esa forma, la semántica monádica modular soluciona el problema de la modularidad semántica, y los elementos genéricos introducidos facilitan la creación de especificaciones semánticas reutilizables.

Para demostrar la aplicabilidad práctica de esta técnica, se ha implementado un Sistema de Prototipado de Lenguajes como un metalenguaje de dominio específico empotrado dentro del lenguaje *Haskell*. El empotramiento de lenguajes de dominio específico en lenguajes de propósito general como *Haskell* se describe en [118, 41, 92].

Las principales ventajas de esta técnica son:

- Mayor facilidad de desarrollo, ya que no es necesario construir una nueva sintaxis ni semántica de un lenguaje de dominio específico. Además, la funcionalidad requerida puede ir probándose a medida que se desarrolla el sistema.

- Permite la reutilización de librerías y otras utilidades propias del lenguaje *Haskell*.
- Se apoya en el sistema de tipos del lenguaje anfitrión. En este caso, el potente sistema de inferencia y comprobación estática de tipos de *Haskell*.
- Acceso directo a las facilidades de los sistemas *Haskell* (depuradores, entornos de desarrollo, editores especializados, etc.)

Sin embargo, esta técnica también tiene una serie de desventajas, como son:

- Mezcla de mensajes de error. Cuando se produce un error, no es posible distinguir los mensajes del propio *Haskell* con los mensajes del sistema de prototipado de lenguajes.
- Dificultades para la depuración. No es posible controlar la ejecución del sistema.
- Limitaciones del sistema de tipos del lenguaje anfitrión. Los sistemas de tipos existentes en la actualidad limitan la expresividad de los programadores. En el caso del lenguaje Haskell, a pesar de la potencia de su sistema de tipos, existen ciertas construcciones que no han podido definirse debido a ciertas limitaciones (véase B.9).
- Dependencia del lenguaje anfitrión. Puesto que el metalenguaje está empotrado, no es posible obtener prototipos ejecutables en lenguajes distintos de Haskell.

Con el fin de estudiar su capacidad expresiva, el Sistema de prototipado de lenguajes se ha aplicado a la especificación de un lenguaje funcional con características imperativas y diferentes mecanismos de evaluación, un lenguaje de programación orientada a objetos con herencia estática y dinámica, y un lenguaje de programación lógica. En todos los casos se han reutilizado bloques comunes, como las expresiones aritméticas.

La especificación de tales lenguajes permite la reutilización de componentes previamente definidos. Además, el desarrollo mediante mónadas, facilita una aproximación incremental a la descripción de la estructura computacional.

10.3 Principales aportaciones

En la presente tesis doctoral, se han realizado las siguientes aportaciones[1].

- Estudio comparativo de las principales técnicas de especificación de lenguajes de programación: lenguaje natural, semántica operacional estructurada, semántica natural, denotacional, algebraica, de acción, monádica y máquinas de estado abstracto.

  Para realizar la comparación, se ha especificado un mismo conjunto de lenguajes en cada una de las técnicas y se han tomado en cuenta las características de no ambigüedad, modularidad semántica, reusabilidad, demostrabilidad, prototipado, legibilidad, flexibilidad y experiencia práctica.

[1] El autor de esta tesis considera que todas las aportaciones que se enumeran en este apartado son originales. No se tiene conocimiento, al menos, de que hayan sido realizadas previamente

- Desarrollo de una nueva técnica de especificación mediante la integración de la semántica monádica modular y los conceptos de programación genérica. Tal integración había sido propuesta en [50] y ha desarrollada en esta tesis doctoral.

- Aplicación por primera vez de los catamorfismos monádicos a la especificación semántica de lenguajes.

- Desarrollo teórico y aplicación de la especificación modular de lenguajes con n-categorías sintácticas mutuamente recursivas.

- Descripción e implementación de un marco interactivo de especificación y ejecución de prototipos de lenguajes de programación basado en la semántica monádica modular.

- Aplicación de la semántica monádica modular a la especificación de un lenguaje de programación Orientado a Objetos, a la especificación de un lenguaje de Programación Lógica y a la especificación de un lenguaje imperativo.

10.4 Publicaciones derivadas

En las siguientes publicaciones se han presentado algunos de los resultados de la presente tesis doctoral:

- J. E. Labra, *An Implementation of Modular Monadic Semantics using Folds and Monadic Folds*, Workshop on Research Themes on Functional Programming, Third International Summer School on Advanced Functional Programming, 1998, Braga – Portugal

- J. E. Labra, J. M. Cueva, C. Luengo, *Language Prototyping using Modular Monadic Semantics*, 3rd Latin-American Conference on Functional Programming, Recife – Brasil, 1999

- J. E. Labra, J. M. Cueva, M. C. Luengo, *Modular Development of Interpreters from Semantic Building Blocks*, 12th Nordic Workshop on Programming Theory, Bergen – Noruega, 2000

- J. E. Labra, J. M. Cueva, C. Luengo, M. González, *A Language Prototyping Tool based on Semantic Building Blocks*, Formal Methods and Tools for Computer Science, EUROCAST-2001, Workshop on Functional Programming and λ-Calculus, Las Palmas de Gran Canaria, España, 2001, aceptado para posterior publicación en *Lecture Notes in Computer Science*, Springer-Verlag

- J. E. Labra, J. M. Cueva, M. C. Luengo, A. Cernuda, *Modular Development of Interpreters from Semantic Building Blocks*, Nordic Journal of Computing (pendiente de publicación).

- J. E. Labra, J. M. Cueva, C. Luengo, A. Cernuda *LPS: A Language Prototyping System Using Modular Monadic Semantics* Workshop on Language Descriptions, Tools and Applications Génova – Italia, 2001, Satellite

events for ETAPS 2001, *Electronic Notes in Theoretical Computer Science*, vol. 44, Elsevier Science. Seleccionado para publicación en un volumen especial de *Science of Computer Programming*.

- J. E. Labra, J. M. Cueva, C. Luengo, A. Cernuda, *Specification of Logic Programming Languages from Reusable Semantic Building Blocks*, International Workshop on Functional and (Constraint) Logic Programming, Kiel, Alemania, Septiembre 2001

El autor ha participado en otras publicaciones relacionadas con el contenido de esta tesis, aunque no se pueden considerar fruto directo de la misma [131, 137, 19, 18, 151, 38, 65].

10.5 Nuevas Líneas de Investigación

Se espera que el presente trabajo sirva de base a la creación de nuevas líneas de investigación en el futuro. A continuación se clasifican las principales líneas que se pueden considerar.

- Se han encontrado numerosas lagunas teóricas en las que se podrían encontrar ejercicios intelectuales interesantes. Caben destacar:
  - Axiomatización de las diferentes variedades de mónadas. Actualmente, los diferentes tipos de mónadas se definen mediante una serie de operaciones con una signatura particular. Se conocen las definiciones axiomáticas de las mónadas con acceso al entorno, de transformación del estado, de control de excepciones, gestíón Entrada/Salida y *Backtracking*. Existen, sin embargo, mónadas con continuaciones de primera clase, reanudaciones, no determinismo, etc. cuya axiomatización facilitaría la derivación de transformadores de forma similar a la realizada por R. Hinze en [87].
  - Taxonomía de estructuras computacionales. A partir de la axiomatización anterior, la realización de una clasificación sistemática de las diferentes variedades de mónadas y su interrelación permitiría una mayor comprensión de los conceptos fundamentales de los lenguajes de programación.
  - Combinación sistemática entre transformadores de mónadas. Actualmente, la creación de una variedad computacional y la correspondiente transformación entre diferentes mónadas se realiza de forma manual, sujeta a numerosos errores.
  - Generalización de la teoría desarrollada para n categorías sintácticas mutuamente recursivas. En 5.8 se presentó la teoría de n-catamorfismos para $n = 2$. La generalización para valores arbitrarios de n es inmediata desde el punto de vista teórico. No obstante, la implementación para valores arbitrarios se resiente y podría requerir sistemas de tipos más flexibles.
  - Relación entre n-catamorfismos y n-catamorfismos monádicos. En 5.6 se presentó la integración entre catamorfismos y mónadas, resultando los catamorfismos monádicos. En la práctica, los catamorfismos monádicos facilitan la especificación, pero no pueden aplicarse

de forma generalizada. De esa forma, en 5.7 se presentó la combinación entre catamorfismos y catamorfismos monádicos, lo cual permite mezclar las diferentes especificaciones.

Al desarrollar n-catamorfismos, surgen de forma natural los n-catamorfismos monádicos. Aunque no se han investigado, no parece que su modelización teórica pueda representar grandes complejidades.

- Lenguajes multi-paradigma. La técnica de especificación semántica admite el desarrollo incremental de lenguajes facilitando la incorporación de caracteríscas. Por ejemplo, en la sección 9.3 se combinó la especificación del núcleo de un lenguaje de programación lógica con un lenguaje aritmético. Siguiendo esa línea sería posible combinar otras características como funciones de orden superior, diferentes mecanismos de evaluación, objetos, referencias, restricciones, etc. El desarrollo de lenguajes híbridos y multi-paradigma podría verse beneficiado por una técnica modulare como la planteada en esta tesis.
- Modelización de Objetos. La presentación semántica del lenguaje orientado a objetos de la sección 8 se realiza mediante puntos fijos y mónadas. Sin embargo, en la actualidad, la aplicación de coálgebras a la modelización de objetos está dando sus frutos [202, 196]. Una cuestión intrigante es la posible combinación entre coálgebras y semántica monádica modular.
- Profundizando en las posibilidades conceptuales de la teoría de la categoría, aparecen de forma natural las *comónadas*. Un algoritmo basado en una mónada interactúa con el entorno gestionando los efectos que produce. Sin embargo, un algoritmo basado en una comónada, se organiza mediante la reacción a los efectos producidos por el exterior. En [121, 188] se muestran varios ejemplos de programas que utilizan comónadas. La aplicación de comónadas a la modelización de sistemas reactivos podría suponer resultados interesantes [16]
- Otra posible aplicación sería la comparación sistemática entre semántica operacional (modelizada mediante anamorfismos) y semántica denotacional (modelizada mediante catamorfismos). Algunos pasos en esta dirección han sido descritos en [96]
- En [94] se presenta una generalización del concepto de mónada denominada flecha o *arrow*. Esta generalización, así como el desarrollo de transformadores de flechas, podría aportar una mayor flexibilidad al sistema.

- Aunque es posible implementar mónadas con reanudaciones, no determinismo y paralelismo, la especificación de lenguajes con tales características no se ha intentado. En principio, la tarea no parece excesivamente complicada, siempre que se desarrolle en varias fases identificando los tipos de mónadas y transformadores de mónadas adecuados. Faltaría por comprobar si al añadir tales características computacionales, se sigue manteniendo la reusabilidad de las especificiones semánticas.

- La especificación de nuevos lenguajes de programación, tanto *lenguajes reales* ya definidos como nuevos lenguajes en fase de diseño, es una tarea necesaria para comprobar la aplicabilidad de la técnica propuesta. En

este sentido, el autor de esta tesis está actualmente desarrollando especificaciones de nuevos lenguajes de programación en Internet utilizando los combinadores de servicios propuestos en [34].

- Respecto a la implementación del Sistema de Prototipado de Lenguajes, se pueden considerar las siguientes actuaciones.
  - Desempotrar la implementación mediante el diseño de un meta-lenguaje de dominio específico. En este sentido, E. Moggi [37, 169] ha realizado varias propuestas de metalenguajes en un ámbito más teórico. La implementación práctica de dichas propuestas parece una línea de trabajo accesible.
  - Mejora de la implementación actual. Las implementaciones del lenguaje *Haskell* están incorporando nuevas características como el polimorfismo de primera clase, los registros y las variantes extensibles, los parámetros implícitos, genericidad, etc. (véase B.8). Estas características no están estandarizadas y no todas están implementadas, por lo que es difícil realizar una valoración a priori de las ventajas de su aplicación. No obstante, la propia evolución del lenguaje *Haskell* determinará mejoras que beneficiarán la actual implementación del Sistema de Prototipado de Lenguajes.
  - Uno de los objetivos originales de la presente tesis doctoral era diseñar un Sistema de Prototipado de Lenguajes que permitiese la modificación del lenguaje que se está especificando de forma interactiva, permitiendo transformar el intérprete del prototipo sin tener que detener la ejecución del sistema. En la actualidad, ésto no es posible debido a que cada especificación de un lenguaje tiene un tipo diferente que es chequeado en base al sistema estático de tipos de *Haskell*.

    La modificación en tiempo de ejecución de los lenguajes a interpretar requeriría alguna forma de sistema dinámico de tipos que va contra la política del lenguaje *Haskell*. No obstante, si el tipo del lenguaje no se altera al modificar su semántica, es posible realizar estas modificaciones de forma interactiva. Ésto se ha conseguido en la implementación que utiliza registros extensibles, polimorfismo de primera clase y tipos existenciales. Los beneficios de tal posibilidad son escasos, ya que, en general, un cambio en la semántica del lenguaje, supone un cambio en la estructura computacional, que implica a su vez un cambio en el tipo de valor que se está ejecutando.

- Las limitaciones de la implementación incrustada del Sistema de prototipado de lenguajes se deben principalemente a las limitaciones del sistema de tipos de *Haskell*. La creación de un metalenguaje de dominio específico independiente podría resolver algunos de estos problemas ofreciendo además varias ventajas:
  - Independencia de *Haskell*. Sería posible obtener prototipos ejecutables en diferentes lenguajes. Actualmente, sólo se obtienen intérpretes en *Haskell*.
  - Demostración de propiedades. Si en lugar de generar un intérprete, se genera código que pueda introducirse en un sistema de demostración

automática de teoremas, sería posible demostrar propiedades de los programas escritos en los lenguajes especificados.

- Con el sistema de prototipado de lenguajes desarrollado en esta tesis es posible obtener intérpretes de los lenguajes especificados de forma automática. El siguiente paso sería automatizar la obtención del compilador correspondiente. En teoría, esta posibilidad puede alcanzarse mediante evaluación parcial [110], aunque los sistemas actuales tienen bastantes limitaciones. Otra posibilidad, investigada en [74], es la utilización de dos mónadas, una estática y otra dinámica. De esa forma, se identifican los tiempos de enlace a la vez que se mantiene la modularidad semántica al utilizar transformadores de mónadas.

- Una posibilidad interesante sería la creación de un entorno integral de desarrollo de lenguajes de programación siguiendo [76, 75]. En la actualidad existen varios sistemas, como ASF+SDF [230] o SmartTools [11]. Dicho entorno puede realizarse de forma independiente, añadiendo un interfaz gráfico de usuario al sistema desarrollado, o mediante la integración en otros entornos.

- Finalmente, y a más largo plazo, sería interesante relacionar las técnicas empleadas para el tratamiento de lenguaje natural con las técnicas utilizadas en lenguajes de programación. Al fin y al cabo, los lenguajes de programación se deben acercar a un modelo ideal que incorpore la flexibilidad del lenguaje natural junto con la rigurosidad de un lenguaje formal. En este sentido, los trabajos de Montague [172, 173] y las técnicas algebraicas de especificación semántica del lenguaje natural [149, 77] podrían aportar un campo interesante de investigación y desarrollo de nuevos lenguajes de programación.

Anexo A

Breve introducción a Teoría de Categorías

La Teoría de Categorías ofrece un potente mecanismo de abstracción que permite aislar conceptos básicos mediante un lenguaje común.

En este apéndice se define la noción de categoría y otras definiciones básicas que se han utilizado durante la tesis. Con esta breve introducción se pretende ofrecer una visión básica del campo. Existen diversos libros de texto sobre el tema, entre los que se pueden destacar [204, 80, 142]

A.1 Categorías

Definición A.1 (Categoría) *Una* categoría $\mathcal{C}$ *está formada por 6 componentes:*

- *Una colección de objetos* $Obj(\mathcal{C})$
- *Una colección de morfismos* $f : Obj(\mathcal{C}) \rightarrow Obj(\mathcal{C})$ *denominada* $Mor(\mathcal{C})$
- *Para cada morfismo* $f : \alpha \rightarrow \beta$*, dos funciones dom y cod que devuelven el dominio y el codominio. Es decir,* $dom(f) = \alpha$ *y* $cod(f) = \beta$
- *Un operador binario . de composición de morfismos. Si* $f : \alpha \rightarrow \beta$ *y* $g : \beta \rightarrow \gamma$*, entonces* $g.f : \alpha \rightarrow \gamma$
- *Para cada elemento* $\alpha \in Obj(\mathcal{C})$ *una función* $id_\alpha : \alpha \rightarrow \alpha$

Además, se cumplen las siguientes ecuaciones:

$$(f.g).h = f.(g.h) \tag{A.1}$$

$$id_\beta.f = f.id_\alpha = f \tag{A.2}$$

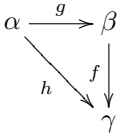
Ejemplo A.1 *El concepto de categoría puede encontrarse en numerosas estructuras matemáticas. Un ejemplo bastante obvio sería un grafo dirigido en el que los objetos serían los nodos, y los morfismos serían las aristas. En esta tesis se tomará como modelo la categoría Set en la que los objetos son conjuntos, los morfismos son funciones totales entre conjuntos y cuyas operaciones de identidad y composición son las equivalentes entre funciones.*

Definición A.2 (Objeto Inicial) *Un objeto α de una categoría $\mathcal{C}$ es* inicial *si para cualquier objeto B existe un único morfismo $f : \alpha \to \beta$. Dicho morfismo suele representarse como $(\!|\beta|\!)$*

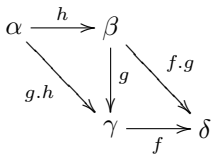
Definición A.3 (Objeto Terminal) *Un objeto α de una categoría $\mathcal{C}$ es* final *si para cualquier objeto β existe un único morfismo $f : \beta \to \alpha$.*

Definición A.4 (Isomorfismo) *Dos objetos α y β de una categoría $\mathcal{C}$ son isomórficos si existen dos morfismos $f : \alpha \to \beta$ y $g : \beta \to \alpha$ tales que $f.g = id_\beta$ y $g.f = id_\alpha$. En ese caso, f y g se denominan isomorfismos.*

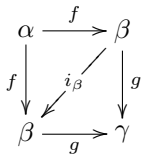
Las propiedades de las categorías se presentan habitualmente mediante *diagramas conmutables*. Un diagrama es un grafo cuyos nodos son objetos y cuyas aristas son morfismos. Un diagrama es conmutable si para cualquier par de caminos entre dos nodos, la composición de morfismos a través del primer camino es igual a la composición de morfismos a través del segundo camino. Por ejemplo, el siguiente diagrama serviría para indicar que $g.f = h$.



De la misma forma, mediante diagramas commutables, puede representarse el axioma A.1



y el axioma A.2



A.2 Functores

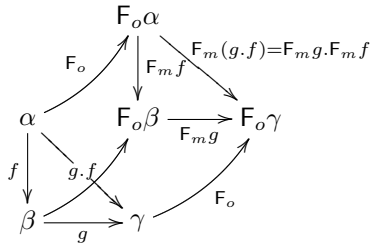
Un functor transforma una categoría en otra manteniendo la estructura de la categoría original.

Definición A.5 (Functor) *Un* functor $\mathsf{F} : \mathcal{A} \to \mathcal{B}$ *de la categoría $\mathcal{A}$ a la categoría $\mathcal{B}$ consiste en un par de funciones $F_o : Obj(\mathcal{A}) \to Obj(\mathcal{B})$ y $F_m : Mor(\mathcal{A}) \to Mor(\mathcal{B})$ que satisfacen las ecuaciones:*

$$F_m(id_\alpha) = id_{F_o(\alpha)} \tag{A.3}$$

$$F_m(g.f) = F_m(g).F_m(f) \tag{A.4}$$

Las propiedad A.4 pueden representarse mediante el siguiente diagrama



Habitualmente se denota mediante F tanto F_o como F_m.

Definición A.6 (endofunctor) *Un* endofunctor *sobre una categoría $\mathcal{C}$ es un functor $\mathsf{F} : \mathcal{C} \to \mathcal{C}$*

Definición A.7 (composición de functores) *Si $\mathsf{F} : \mathcal{C} \to \mathcal{D}$ y $\mathsf{G} : \mathcal{D} \to \mathcal{E}$ son dos functores, su composición es un functor $\mathsf{G}.\mathsf{F} : \mathcal{C} \to \mathcal{E}$ que se define tomando $(\mathsf{G}.\mathsf{F})(A) = \mathsf{G}(\mathsf{F}(A))$ y $(\mathsf{G}.\mathsf{F})(f) = \mathsf{G}(\mathsf{F}(f))$*

Definición A.8 (functor identidad) *Para toda categoría $\mathcal{C}$, se define el* functor identidad *$id_{\mathcal{C}} : \mathcal{C} \to \mathcal{C}$ como $id_{\mathcal{C}}(\alpha) = \alpha$ y $id_{\mathcal{C}}(f) = f$*

Si $\mathsf{F} : \mathcal{C} \to \mathcal{C}$ es un endofunctor y n es un número natural positivo, $\mathsf{F}^n : \mathcal{C} \to \mathcal{C}$ se utiliza para representar la composición de Fconsigo mismo n veces. Esta notación puede extenderse de forma que $\mathsf{F}^0 = id_{\mathcal{C}}$.

Teorema A.1 *Los functores preservan los isomorfismos*

Teorema A.2 *Los functores identidad son identidades para la composición de functores, es decir, si $\mathsf{F} : \mathcal{C} \to \mathcal{D}$ es un functor, entonces $\mathsf{F}.id_{\mathcal{C}} = id_{\mathcal{D}}.\mathsf{F} = \mathsf{F}$*

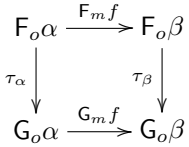
A.3 Transformaciones naturales

Definición A.9 (Transformación natural) *Dados dos functores $\mathsf{F}, \mathsf{G} : \mathcal{C} \to \mathcal{D}$, una* transformación natural *$\tau : \mathsf{F} \to \mathsf{G}$ es una transformación de un functor en otro que mantiene la estructura del functor original.*

Para cualquier morfismo $f : \alpha \to \beta$ la transformación natural τ se define asignando a cada $A \in Obj(\mathcal{C})$ un morfismo $\tau_\alpha : F_o(\alpha) \to F_o(\beta)$ tal que:

$$G_m(f).\tau_\alpha = \tau_\beta.F_m(f) \tag{A.5}$$

La propiedad anterior de las transformaciones naturales puede representarse como



A.4 Mónadas

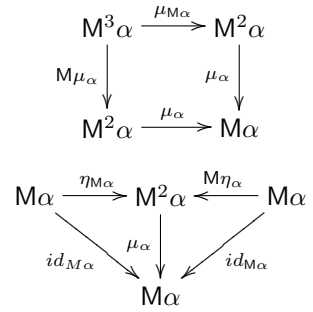
El concepto de mónada fue originalmente propuesto en el contexto de Teoría de Categorías con el nombre de *triplet*. Posteriormente, se popularizó el nombre de mónada con la siguiente definición, tomada de [171, 21].

Definición A.10 (Mónada) *Una mónada sobre una categoría $\mathcal{C}$ es una terna $\langle \mathsf{M}, \eta, \mu \rangle$ donde M es un endofunctor sobre $\mathcal{C}$, $\eta : \alpha \to \mathsf{M}\,\alpha$ y $\mu : \mathsf{M}^2\,\alpha \to \mathsf{M}\,\alpha$ son dos transformaciones naturales y se cumplen las siguientes propiedades.*

$$\mu_\alpha . \mu_{\mathsf{M}\alpha} = \mu_\alpha . \mathsf{M}\mu_\alpha \tag{A.6}$$

$$\mu_\alpha . \eta_{\mathsf{M}\alpha} = \mu_\alpha . \mathsf{M}\eta_\alpha = id_{\mathsf{M}\alpha} \tag{A.7}$$

Dichas propiedades pueden representarse mediante los siguientes diagramas.

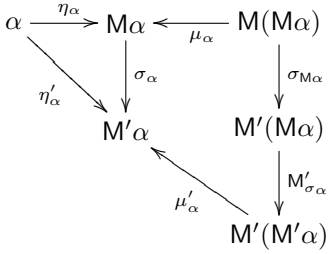


La definición de mónadas anterior es equivalente a la definición de la sección 4.1. La operación $\eta : \alpha \to \mathsf{M}\,\alpha$ equivale a la operación *return*, mientras que la operación $\mu : \mathsf{M}\,(\mathsf{M}\,\alpha) \to \mathsf{M}\,\alpha$ puede definirse como la operación *join* de la siguiente forma

$$\begin{aligned} &\mathit{join} \;:\; (\mathit{Monad}\;\mathsf{m}) \;\Rightarrow\; \mathsf{m}\,(\mathsf{m}\,\alpha) \;\to\; \mathsf{m}\,\alpha \\ &\mathit{join}\;z \;=\; z \gg\!= (\lambda\, m \;\to\; m) \end{aligned}$$

En el contexto de Teoría de Categorías, se puede realizar la siguiente definición de morfismo monádico.

Definición A.11 (Morfismo monádico) *Dadas dos mónadas $\langle \mathsf{M}, \eta, \mu \rangle$ y $\langle \mathsf{M}', \eta', \mu' \rangle$ sobre $\mathcal{C}$, un morfismo monádico es una transformación natural $\sigma : \mathsf{M}\,\alpha \to \mathsf{M}'\,\alpha$ que hace que el siguiente diagrama sea commutable.*



A partir de un morfismo monádico, se puede definir un transformador de mónadas (véase sección 4.3.2).

Anexo B

Introducción al lenguaje *Haskell*

B.1 Introducción

Haskell[1] surgió a partir de los esfuerzos de unificar los diversos lenguajes de programación funcional existentes a finales de los años 80. Para ello se creó un comité internacional que desarrollaría el primer informe sobre el lenguaje en 1988 [93].

Desde su definición original, *Haskell* era un lenguaje puramente funcional con semántica no estricta (evaluación perezosa) y un sistema de inferencia de tipos. También incorporaba ciertas facilidades notacionales, como las definiciones de funciones mediante encaje de patrones, las listas por comprehensión, etc. La principal aportación original del lenguaje fue la resolución sistemática de la sobrecarga mediante clases de tipos (véase B.6). A partir de la versión 1.3, el sistema de Entrada/Salida de Haskell se basa en la utilización de una mónada *IO* y se añaden las clases de constructores de tipos. En el año 1998, se decide afrontar un proceso de estandarización que permita congelar las características más importantes, surgiendo la versión *Haskell 98* descrita en [112].

En B.8 se incluyen algunas extensiones incluidas en las implementaciones pero que no forman del estándard del lenguaje y que se han utilizado en el desarrollo del sistema. En B.9 se incluyen extensiones propuestas pero no implementadas que podrían ser de utilidad para mejorar la presente implementación.

A continuación se presentan las características más llamativas del lenguaje de forma breve. Se ha intentado que la presentación sea autocontenida, aunque el objetivo principal es mostrar las características utilizadas en la tesis. Para obtener información más detallada puede consultarse [203, 23] o la propia página Web dedicada al lenguaje [2].

Se emplea una notación ligeramente diferente del código Haskell puro. En D.2 se describen las libertades notacionales que se han tomado.

[1] El nombre *Haskell* deriva de Haskell B. Curry, uno de los principales desarrolladores de la lógica de combinadores

B.2 Definición de funciones

B.2.1 Notación *lambda*

La expresión $f = \lambda x \rightarrow x + 3$ denota una función que, cuando se le pasa un valor x, devuelve $x + 3$

B.2.2 Declaraciones locales

Mediante *where* es posible introducir declaraciones locales en la definición de una función. Por ejemplo

```
test a b = add * subs
  where
    add = a + b
    subs = a - b
```

El ejemplo anterior también puede escribirse mediante las expresiones *let*

```
test' a b = let
              add  = a + b
              subs = a - b
            in
            add * subs
```

B.2.3 Encaje de patrones

En *Haskell*, las funciones pueden definirse mediante varias ecuaciones. Nótese la notación predefinida para listas en *Haskell*.

```
length          : [α] → Int      — [α] denota la lista de valores de tipo α
length []       = 0              — [] representa la lista vacía
length (x : xs) = 1 + length xs  — : añade un elemento a una lista
```

El sistema intentará encajar la expresión a evaluar con las ecuaciones siguiendo el orden en que fueron definidas y devolviendo la expresión correspondiente a la primera que encaje.

B.2.4 Expresiones *case*

En realidad, las expresiones definidas mediante encaje de patrones pueden escribirse mediante expresiones *case*. Así la función *length* de la sección anterior podría escribirse como:

```
length ls = case ls of
              [] → 0
              (x : xs) → 1 + length xs
```

B.3 Funciones de Orden Superior

Una de las características sobresalientes de los lenguajes funcionales consiste en poder utilizar funciones como un valor cualquiera. De esta forma, las funciones pueden formar parte de estructuras de datos, pueden ser argumentos de otras funciones o pueden ser el resultado de una función. Cuando un lenguaje permite este tipo de definiciones, se dice que el lenguaje incluye *funciones de orden superior* o de primera clase.

La *currificación* (*currying*) es una técnica muy utilizada en *Haskell* que permite simular funciones de varios argumentos mediante funciones de un solo argumento de orden superior. Así, la función:

$$\begin{aligned} &f \ : \ (Int,\ Int) \ \rightarrow \ Int \\ &f \ = \lambda(x, y) \rightarrow 2 * x + y \end{aligned}$$

Puede escribirse como [2] :

$$\begin{aligned} &f \ : \ Int \ \rightarrow \ (Int \ \rightarrow \ Int) \\ &f \ = \lambda x \rightarrow \lambda y \rightarrow 2 * x + y \end{aligned}$$

La siguiente función *map* aplica a todos los elementos de una lista una función f que se le pasa como argumento.

$$\begin{aligned} &map && : \ (\alpha \rightarrow \beta) \rightarrow [\alpha] \rightarrow [\beta] \\ &map\ f\ [] && = [] \\ &map\ f\ (x : xs) && = f\ x \ : \ map\ f\ xs \end{aligned}$$

Una función de orden superior de gran utilidad para el tratamiento de listas es la función *foldr* que puede definirse como

$$\begin{aligned} &foldr && : \ (\alpha \rightarrow \beta \rightarrow \beta) \rightarrow \beta \rightarrow [\alpha] \rightarrow \beta \\ &foldr\ \oplus\ e\ [] && = e \\ &foldr\ \oplus\ e\ (x : xs) && = x\ \oplus\ (foldr\ \oplus\ e\ xs\ ) \end{aligned}$$

A partir de dicha función pueden expresarse gran cantidad de funciones, por ejemplo, la función *length* podría definirse como

$$length \ = \ foldr\ (+)\ 0$$

De la misma forma, la función *append* añade los contenidos de una lista a otra lista

$$\begin{aligned} &append \ : \ [\alpha] \rightarrow [\alpha] \rightarrow [\alpha] \\ &append\ xs\ ys \ = \ foldr\ (:)\ ys\ xs \end{aligned}$$

y la función *concat* junta una lista de listas en una sola lista

$$\begin{aligned} &concat \ : \ [[\alpha]] \rightarrow [\alpha] \\ &concat \ = \ foldr\ append\ [] \end{aligned}$$

Otro ejemplo de función de orden superior es el el operador (.) de composición de funciones, que puede definirse (aunque ya está predefinido) como:

$$\begin{aligned} &(.) && : \ (\beta \ \rightarrow \ \gamma) \ \rightarrow \ (\alpha \ \rightarrow \ \beta) \ \rightarrow \ (\alpha \ \rightarrow \ \gamma) \\ &(.)\ f\ g && = \lambda\, x \ \rightarrow \ f\ (\ g\ x) \end{aligned}$$

[2] En realidad el operador $\rightarrow$ tiene asociatividad a la derecha, por lo que no se requieren los paréntesis en $Int \rightarrow (Int \rightarrow Int)$

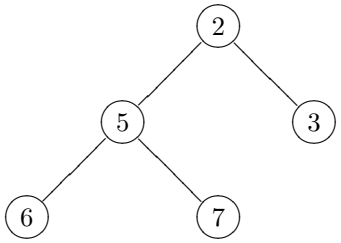


Figura B.1: Ejemplo de árbol binario

B.4 Sistema de Inferencia de tipos

Haskell es un lenguaje fuertemente tipado, realizando un chequeo de tipos en tiempo de compilación que garantiza que en tiempo de ejecución no se producirán errores de tipo. El sistema de tipos de *Haskell* se basa en el sistema de inferencia de tipos de Hindley-Milner. Las declaraciones de tipo son opcionales, en caso de que el programador las incluya, el sistema chequea en tiempo de compilación que los tipos declarados encajan con los tipos inferidos.

El usuario puede definir nuevos tipos de datos por inducción. Por ejemplo, la siguiente declaración define árboles binarios.

$$\textbf{data}\ \mathit{Tree}\ \alpha\ =\ \mathit{Leaf}\ \mid\ \mathit{Bin}\ \alpha\ (\mathit{Tree}\ \alpha)\ (\mathit{Tree}\ \alpha)$$

De esa forma, el árbol de la figura B.1 se representa en *Haskell* como

$$\mathit{Bin}\ 2\ (\mathit{Bin}\ 5\ (\mathit{Bin}\ 6\ \mathit{Leaf}\ \mathit{Leaf})\ (\mathit{Bin}\ 7\ \mathit{Leaf}\ \mathit{Leaf}))\ (\mathit{Bin}\ 3\ \mathit{Leaf}\ \mathit{Leaf})\ :\ \mathit{Tree}\ \mathit{Int}$$

Haskell permite definir tipos de datos cuyos campos tienen un nombre, los cuales se denominan *registros*.

Por ejemplo, en lugar de definir un punto como

$$\textbf{data}\ \mathit{Point}\ =\ P\ \mathit{Float}\ \mathit{Float}$$

es posible asignar nombres a los componentes.

$$\textbf{data}\ \mathit{Point}\ =\ P\ \{x : \mathit{Float},\ y : \mathit{Float}\}$$

El sistema añade las funciones de acceso $x, y : \mathit{Point} \rightarrow \mathit{Float}$, además, si $p : \mathit{Point}$, entonces $p\ \{x\ =\ 3\}$ indica un punto con los mismos valores que p salvo la x, que vale 3.

B.5 Evaluación perezosa

Mediante la evaluación perezosa, es posible definir funciones que utilizan estructuras de datos potencialmente infinitas.

Por ejemplo, la siguiente función genera la lista potencialmente infinita $[x,\ f\ x,\ f\ (f\ x),\ f\ (f\ (f\ x)),\ \ldots]$.

$$\mathit{iterate}\ f\ x\ =\ x\ :\ \mathit{iterate}\ f\ (f\ x)$$

B.6 Sobrecarga y clases de tipos

Una de las principales aportaciones del lenguaje *Haskell* fue la solución sistemática del problema de la sobrecarga mediante clases de tipos [237, 71].

Una clase de tipos puede definirse como un conjunto de tipos que tienen una serie de operaciones en común. El ejemplo más característico es la clase predefinida *Eq* que engloba a todos los tipos que sobre los que puede definirse el operador de comparación $==$.

La declaración de la clase de tipos *Eq* podría ser:

```
class Eq α
  where
    (==) : α → α → Bool
```

Cuando se define un nuevo tipo de datos, puede declararse que es una instancia de una clase de tipos. Por ejemplo, el tipo de datos *Arbol*:

```
instance Eq Tree
  where
    Leaf == Leaf = True
    (Bin x i d) == (Bin x' i' d') = x == x' && i == i' && d == d'
```

Cuando el sistema encuentra una utilización del operador $==$ infiere que los operandos deben pertenecer a la clase de tipos *Eq*. Por ejemplo, ante la función:

```
elem x []       = False
elem x (y : ys) = if x == y then
                    True
                  else
                    elem x ys
```

el sistema infiere el tipo $elem : (Eq\ \alpha) \Rightarrow \alpha \rightarrow [\alpha] \rightarrow Bool$.

A partir de la versión 1.3 se permiten las clases de constructores de tipos siguiendo el esquema propuesto por [106, 105]. Tanto las mónadas como los functores pueden definirse como una clase de constructores de tipos. Por ejemplo, las siguientes definiciones vienen predefinidas en *Haskell*:

```
class Functor f
  where
    map : (α → β) → (fα → fβ)

class Monad m
  where
    return : α → m α
    (>>=)  : m α → (α → m β) → m β
```

A partir de la declaración anterior, es posible declarar nuevas mónadas indicando que son una instancia de la clase *Monad*.

Por ejemplo, una mónada sobre listas puede declararse como:

```
instance Monad [ ]
  where
    return = λ x → [x]
    (>>=)  = λ xs f → concat (map f xs)
```

getChar : IO *Char*	Lee un carácter de la entrada estándar
putChar : *Char* → IO ()	Escribe un carácter en la salida estándar
getLine : IO *String*	Lee una línea de la entrada estándar
putStr : *String* → IO ()	Escribe una línea en la salida estándar
...	

Tabla B.1: Funciones de Entrada/Salida predefinidas en *Haskell*

B.7 Entrada/Salida y Mónadas

A partir de la versión 1.2, la Entrada/Salida en Haskell se realiza mediante mónadas. El lenguaje incluye una mónada especial IO. De esa forma, un valor de tipo IO α denota una computación que realiza Entrada/Salida y devuelve un valor de tipo α. El sistema incluye varias funciones predefinidas de Entrada/Salida, que se resumen en la tabla B.1.

A modo de ejemplo, el siguiente programa solicita un nombre y lo imprime en mayúsculas [3]:

```
main = do
         putStr "Name?"
         n ← getLine
         putStr ("Name is" ++ map toUpper n)
```

B.8 Extensiones

B.8.1 Tipos existenciales

El sistema Hugs98 proporciona de forma experimental tipos existenciales [167] que se pueden introducir mediante la palabra clave *forall*[4].

Combinando tipos existenciales con clases de tipos es posible utilizar valores heterogéneos en una misma estructura [140], permitiendo un patrón de programación similar al que se obtiene mediante el polimorfismo de herencia en lenguajes orientados a objetos.

A modo de ejemplo, supóngase que se desea calcular el área total de una lista de figuras. Se define una clase de los tipos que tienen una función que calcula su área:

```
class Figure f
  where
    area : f → Float
```

Es posible declarar que cada figura (círculo, rectángulo, etc.) es una instancia de la clase $Figura$ proporcionando la correspondiente función $area$

[3] Se utiliza el operador predefinido (++) que concatena dos cadenas y la función *toUpper* que convierte un carácter a mayúsculas

[4] en esta presentación se utilizará el signo ∀

```
Circle = C { radius : Float }

instance Figure Circle
  where
    area c = pi * radius c * radius c

Rectangle = R { width : Float, height : Float }

instance Figure Rectangulo
  where
    area r = width r * height r
```

Si se intenta definir una lista mezclando figuras de diferentes tipos, el sistema de tipos protesta porque los valores de una lista deben pertenecer todos al mismo tipo. Para evitar ese problema, la solución consiste en definir un tipo existencial:

```
Fig = ∀ f . (Figure f) ⇒ Mk f
```

Puede parecer paradójico que se denominen tipos existenciales cuando en realidad se utiliza el cuantificador universal. Sin embargo, obsérvese que el tipo del constructor de tipos *Mk* es:

$$Mk : \forall f.(Figure\ f) \Rightarrow f \rightarrow Fig$$

que equivale a:

$$Mk : (\exists f\ .\ (Figure\ f) \Rightarrow f) \rightarrow Fig$$

Si se declara la instancia

```
instance Figure Fig
  where
    area (Mk f) = area f
```

el sistema puede construir estructuras de datos de tipo *Fig*. Por ejemplo, una lista de figuras

```
fs : [Fig]
fs = [Mk (C {radius = 3}), Mk (R {width = 2, height = 5})]
```

y calcular la suma de las áreas llamando a la función *area*

```
lsArea : [Fig] → Float
lsArea = sum . map area
```

Con el esquema anterior, es posible definir nuevas figuras sin modificar la función genérica *lsArea*. Para ello, sólo hay que declarar que son instancia de la clase *Figure* proporcionando la correspondiente función *area*. Además, se consigue una mayor separación entre el usuario de la clase *Figure* que sólo podrá utilizar la función *area* y las definiciones de cada figura particular que pueden utilizar otros elementos propios de cada figura.

B.8.2 Polimorfismo de primera clase

En [107], M. P. Jones propone una extensión al sistema de tipos Hindley-Milner que admite la posibilidad de definir tipos de datos con valores polimórficos. Su propuesta fue implementada en el sistema Hugs y permite, entre otras cosas, implementar mónadas como un tipo de datos cualquiera, sin necesidad de recurrir a las clases de tipos.

Por ejemplo, la siguiente sería una definición posible de un tipo de datos que representa mónadas.

$$\begin{aligned}&\textbf{data}\ \mathit{Monad}\ m\ =\ M\ \{\ \mathit{return} : \forall\alpha\,.\,\alpha \to m\,\alpha\\ &\qquad\qquad\qquad\quad ,\ (\gg\!=)\ :\ \forall\alpha\,\beta\,.\,m\,\alpha\ \to\ (\alpha\ \to\ m\,\beta)\to\ m\,\beta\\ &\qquad\qquad\qquad\quad \}\end{aligned}$$

La mónada sobre listas podría definirse como

$$\begin{aligned}&\mathit{lsMonad}\ \ :\ \ \mathit{Monad}\,[\,]\\ &\mathit{lsMonad}\ =\\ &\quad M\ \{\ \mathit{return}\ =\ \lambda\,x\ \to\ [x],\\ &\qquad ,\ (\gg\!=)\ =\ \lambda\,\mathit{xs}\,f\ \to\ \mathit{concat}\,(\mathit{map}\,f\ \mathit{xs})\}\\ &\qquad \}\end{aligned}$$

La ventaja de esta codificación es que las mónadas pasan a ser valores de primera clase, pudiendo ser manipulados como cualquier otro valor.

B.8.3 Registros extensibles

En [60], se describe una nueva extensión al sistema de tipos que permite definir registros y variantes extensibles. En Hugs se implementan los registros extensibles, dejando las variantes extensibles fuera de la implementación.

Un registro extensible es un registro al que se le pueden añadir componentes. A continuación se define un registro extensible *Point* con dos coordenadas

$$\textbf{type}\ \mathit{Point}\ r\ =\ \mathit{Rec}\,(x\ :\ \mathit{Float},\ y\ :\ \mathit{Float}\mid r)$$

Un punto particular puede definirse como

$$\begin{aligned}&p\ :\ \mathit{Point}\,()\\ &p\ =\ (x\ =\ 3,\ y\ =\ 4)\end{aligned}$$

donde () denota el registro vacío (sin componentes). La función *dist* calcula la distancia al origen.

$$\mathit{dist}\ =\ \mathit{sqrt}((\#x\ p)^2\ +\ (\#y\ p)^2)$$

donde la notación $(\#x\ p)$ selecciona la componente x del registro p. El tipo inferido por el sistema es

$$\mathit{dist} : (r \setminus x, r \setminus y) \Rightarrow\ \mathit{Rec}\,(x : \mathit{Float}, y : \mathit{Float} \mid r) \to \mathit{Float}$$

Indicando que la función *dist* toma un registro extensible que tiene al menos las componentes x e y y devuelve un valor *Float*.

Mediante la combinación entre polimorfismo de primera clase y registros extensibles, es posible definir móandas y transformadores de mónadas prescindiendo de las clases de tipos. Un problema práctico que aparece al realizar la codificación es que la actual implementación de Hugs no admite componentes polimórficos en registros extensibles. La solución pasa por definir nuevos tipos de datos por cada componente.

$$\begin{aligned}&\textbf{type}\ \mathit{Monad}\ m\ r\ =\ \mathit{Rec}\,(\ \mathit{unit}\ :\ \mathit{Unit}\ m\\ &\qquad\qquad\qquad\qquad\qquad\quad ,\ \mathit{bind}\ :\ \mathit{Bind}\ m\\ &\qquad\qquad\qquad\qquad\qquad\quad \mid\ r\ )\end{aligned}$$

donde los tipos *Unit* y *Bind* se definen como

$$\begin{aligned}&\textbf{data}\ \mathit{Unit}\ m\ =\ U\ \{\ \mathit{unitOf}\ :\ \forall\alpha\,.\,\alpha\rightarrow\ m\,\alpha\ \}\\ &\textbf{data}\ \mathit{Bind}\ m\ =\ B\ \{\ \mathit{bindOf}\ :\ \forall\alpha\,\beta\,.\,m\,\alpha\ \rightarrow\ (\alpha\ \rightarrow\ m\,\beta)\ \rightarrow\ m\,\alpha\ \}\end{aligned}$$

junto con las siguientes funciones auxiliares que permiten extraer la componente polimórfica correspondiente en una mónada.

$$\begin{aligned}&\mathit{unit}\ m\ =\ \mathit{unitOf}\ (\#\mathit{unit}\ m)\\ &\mathit{bind}\ m\ =\ \mathit{bindOf}\ (\#\mathit{bind}\ m)\end{aligned}$$

La mónada sobre listas podría definirse como

$$\begin{aligned}&\mathit{lsMonad}\ \ :\ \ \mathit{Monad}\,[\,]\,()\\ &\mathit{lsMonad}\ =\\ &\quad(\ \mathit{unit}\ =\ U\ (\lambda\,x\rightarrow\ [x]),\\ &\quad,\ \mathit{bind}\ =\ B\ (\lambda\,\mathit{xs}\,f\ \rightarrow\ \mathit{concat}\ (\mathit{map}\,f\ \mathit{xs}))\}\\ &\quad)\end{aligned}$$

Mediante la utilización de registros extensibles, es posible especializar una mónada general añadiendo más componentes. Por ejemplo, la mónada con excepciones puede definirse como

$$\textbf{type}\ \mathit{ExcM}\ m\ r\ =\ \mathit{Monad}\ m\ (\mathit{err}\ :\ \mathit{Err}\ m\mid r)$$

donde

$$\textbf{data}\ \mathit{Err}\ m\ =\ E\ \{\ \mathit{errOf}\ :\ \forall\,\alpha\,.\,\mathit{String}\rightarrow\ m\,\alpha\ \}$$

$$\mathit{err}\ m\ =\ \mathit{errOf}\ (\#\mathit{err}\ m)$$

De esta forma, es posible definir un transformador de mónadas como una función entre dos mónadas. Por ejemplo, el transformador de mónadas que añade tratamiento de excepciones se podría definir de la siguiente forma.

Para ello se define el siguiente tipo de datos

$$\textbf{data}\ \mathit{Error}\ \alpha\ =\mathit{OK}\ \alpha\mid\mathit{Error}\ \mathit{String}$$

junto con la función auxiliar

$$\begin{aligned}&\mathit{ifError}\qquad\qquad\ :\ (\mathit{String}\ \rightarrow\ \beta)\rightarrow(\alpha\rightarrow\ \beta)\rightarrow\mathit{Error}\ \alpha\rightarrow\beta\\ &\mathit{ifError}\ f\ g\ (\mathit{OK}\ x)\quad =g\ x\end{aligned}$$

```
ifError f g (Error e) = f x


m2e : (r \ unit, r \ bind, r \ err) ⇒ Monad m r -> EMonad (Error m) r
m2e m@(unit = u, bind = b|r) =
  ( unit = U ((unit m) . OK)
  , bind = B (λ x f → bind m x (ifError (unit m . Error) f))
  , err  = E (unit m . Error)
  | r )
```

Una ventaja de esta implementación es la posibilidad de modificar en tiempo de ejecución la semántica de un lenguaje (siempre que los tipos sean consistentes con las operaciones del lenguaje). Sin embargo, esta técnica no permite realizar una elevación sistemática de todas las operaciones de la mónada que se está transformando, por lo que es necesario definir transformadores de mónadas específicos para cada combinación.

B.8.4 Parámetros implícitos

En [144] se introduce una extensión al sistema de tipos de *Haskell* que admite la definición de variables de ámbito dinámico mediente parámetros implícitos.

A modo de ejemplo, supóngase que se va a definir la semántica de un lenguaje de expresiones simples con declaraciones locales.

La sintaxis abstracta del lenguaje podría definirse mediante el tipo de datos *Expr*

```
Expr = Expr + Expr | Const Int | Var String | Let String Expr Expr
```

y la función de evaluación se definiría como

```
eval               : (?ρ : Env, Monad m) ⇒ Expr → m Int
eval (e1 + e2)     = do { v1 ← e1; v2 ← e2; return (v1 + v2) }
eval (Const n)     = return n
eval (Var x)       = return (lkp ?ρ x)
eval (Let x e1 e2) = do { v ← eval e1; eval e2 with ?ρ = upd ?ρ x v }
```

La expresión $?x$ representa un parámetro implícito x y la expresión e *with* $?x = v$ enlaza el parámetro implícito x al valor v en la evaluación de la expresión e.

Se han utilizado las siguientes definiciones sobre un entorno *Env*.

```
lkp     : Env → String → Int          — lkp ρ x busca x en ρ
upd     : Env → String → Int → Int    — upd ρ x v asigna a xel valor v en ρ
initial : Env                         — initial representa el entorno inicial
```

B.8.5 Dependencias funcionales

Una extensión reciente, propuesta e implementada por M. P. Jones en Hugs han sido las dependencias funcionales [108]. Esta extensión se ha inspirado en las dependencias funcionales de las bases de datos relacionales y permite definir relaciones de dependencia entre los parámetros de una clase de tipos con varios

parámetros facilitando la inferencia de tipos en situaciones en las que, de otra forma, habría ambigüedad.

Las dependencias funcionales pueden utilizarse en las clases que definen mónadas relacionados con un determinado tipo. Por ejemplo, la mónada transformadora de estado se define convencionalmente mediante la clase de tipos

```
class (Monad m) ⇒ StateMonad ς m
  where
    fetch : m ς
    set   : ς → m ς
```

De esa forma, la siguiente función

```
foo : (StateMonad ς m) ⇒ m ()
foo = do { ς ← fetch; return () }
```

es rechazada por el sistema de tipos convencional ya que el tipo de la variable ς no queda suficientemente delimitado. La solución es indicar al sistema que el tipo del estado depende de la mónada mediante la siguiente declaración

```
class (Monad m) ⇒ StateMonad m | m → ς
  where
    fetch : m ς
    set   : ς → m ς
```

Las dependencias funcionales han sido utilizadas en el sistema de prototipado de lenguajes y han sido recientemente implementadas en el compilador GHC de *Haskell*.

B.9 Limitaciones

En este apartado se relacionan algunas limitaciones o extensiones propuestas pero no implementadas del lenguaje *Haskell*.

B.9.1 Variantes extensibles

De forma similar a la definición de registros extensibles, en [60] se incluye un sistema de tipos con variantes extensibles. Sin embargo, esta capacidad no ha sido implementada en Hugs. Las variantes extensibles permiten definir uniones de valores y su aplicación sería especialmente interesante para la modelización de los dominios extensibles, evitando la necesidad de utilizar las clases de tipos definidas en 4.4.

B.9.2 Politipismo y genericidad

Como se ha indicado en la sección 5.10, la versión 5.00 del compilador GHC admite la definición de clases genéricas. Sin embargo, se admiten únicamente clases de tipos regulares, lo que impediría utilizar el functor exponencial utilizado en la definición del lenguaje Prolog. Tampoco admite la definición de clases de constructores de tipos ni con múltiples parámetros, lo cual impide las definiciones de las clases *Functor*, *MFunctor* y *BiFunctor* definidas en 5.10. Se espera que futuras versiones del compilador admitan este tipo de definiciones, aunque en la actualidad no existe un diseño efectivo de estas ampliaciones [84].

B.9.3 Clases disjuntas

En el sistema de tipos de Haskell no es posible declarar que dos clases de tipos son disjuntas. Por ejemplo, supóngase que se desea declarar la siguiente clase de tipos:

```
class Num a ⇒ Dividable a
  where
    dividedBy : a → a → a
```

Las clases predefinidas *Integral* y *Fractional* representan tipos que incluyen la división entera y real, respectivamente. Ambas podrían declararse instancias de *Dividable*.

```
instance Fractional a ⇒ Dividable a
  where
    dividedBy = (/)


instance Integral a ⇒ Dividable a
  where
    dividedBy = div
```

Sin embargo, *Haskell* no admite las declaraciones anteriores ya que existe un solapamiento de ambas instancias. En [61] se propone una solución mediante la utilización de reglas de gestión de restriccciones.

El sistema de prototipado de lenguajes se vería beneficiado de esta posibilidad para declarar que las F-álgebras que toman como portador un valor $\mathsf{M}\,\mathsf{V}$ como las F-álgebras monádicas pueden ser instancias de una misma clase de tipos de interpretación. Lo cual permitiría la inclusión incremental de componentes dejando que el sistema detectase qué tipo de operación utilizar para la interpretación.

Anexo C

Conversión de términos inglés-español

En numerosas ocasiones durante la redacción de esta tesis, ha sido necesario traducir un determinado término del inglés. Se ha intentado realizar dicha traducción respetando el significado original y buscando el concepto en castellano más cercano. Puesto que la literatura en castellano sobre el tema de la tesis es escasa, dicha traducción ha sido realizada sin poder consultar otras fuentes.

En la siguiente tabla se incluyen varios términos de difícil traducción.

Tabla C.1: Tabla de conversión de términos inglés-castellano

Término en inglés	**Término en castellano**	**Página**
Allocate	Reservar	51
Abstract State Machines	Máquinas de Estado Abstracto	46
Assertion	Aserción	38
Backtracking	Backtracking	62
Carrier	Portador	80
Category Theory	Teoría de Categorías	155
Continuation	Continuación	36
Constraint Logic Programming	Programación Lógica con Restricciones	134
Currying	Currificación	161
Deallocate	Liberar	51
Domain Specific Languages	Lenguajes de Dominio Específico	17
Dynamic binding	Enlace dinámico	123
Elevación	Lifting	66
End-user programming	Programación realizada por el usuario final	18

continúa en la página siguiente

Tabla C.1 – viene de la página anterior

Término en inglés	Término en castellano	Página
Extensible variants	Variantes extensibles	70
Fixpoint	Punto fijo	77
Higher Order Functions	Funciones de Orden Superior	161
Hoare triplet	Terna de Hoare	38
Lazy Evaluation	Evaluación perezosa	162
Many-Sorted Algebra	Álgebra multiclase o multiespecie	87
Monad	Mónada	59
Monad Transformer	Transformador de mónadas	66
Overlapping instances	Instancias solapables	70
Overloading	Sobrecarga	163
Overriding	Redefinición	123
Yielder	Productor	50
Pretty printing	Impresión	99
Powerdomains	Powerdomains	64
Resumption	Reanudación	64
Retract	Repliegue	42
Sort	Género	51
Subtyping	Subtipación	70
Type Classes	Clases de tipos	163
Thunk	No se ha traducido [a]	118

[a] El término *thunk* fue utilizado por los diseñadores de Algol-60 para denotar una técnica de implementación de paso de parámetros

Anexo D

Notación utilizada

En este anexo se resumen brevemente las principales convenciones notacionales utilizadas a los largo de la tesis.

D.1 Símbolos

$\triangleq$	Definición de tipo de datos
$=$	Definición de función
ς	Valor de tipo *State*
ρ	Valor de tipo *Env*
ϵ	Valor de tipo *Exc*
ω	Valor de tipo *Answer*
κ	Valor de tipo continuación
$\mathcal{C}$	Categoría
F, $\mathsf{G}, \ldots$	Functores
$\mathbb{F}, \mathbb{G}, \ldots$	Bifunctores
M	Mónada
$\mathcal{T}_x$	Transformador de mónadas para x
$\alpha, \beta, \gamma, \ldots$	Variables de tipo
φ_{F}	F-álgebra
ϖ_{F}	F-álgebra monádica
$(\![\ ]\!)$	Fold o Catamorfismo
$(\!(\![\ ]\!)\!)$	Fold o Catamorfismo Monádico
$\mathbb{C}$	Clase
τ	Objeto
$\tau \mapsto m$	Selección del método m del objeto τ
$\uplus$	Extensión de un registro
σ	substitución
ω	Unificador
$\mathcal{L}_x$	Lenguaje de programación x

D.2 Código Haskell

En el código *Haskell*, se han tomado ciertas libertades con la pretensión de facilitar la lectura y de buscar una solución de compromiso entre la nota-

ción matemática y el código de los programas. A continuación se resumen las principales decisiones:

- Los tipos de las funciones se indican con : en lugar de ::
- Se utilizan símbolos matemáticos en el código fuente. Dichos símbolos deben transcribirse por los identificadores correspondientes a la hora de construir los programas.
- Se utilizan las letras α, β, γ, ... para indicar variables de tipo cuantificadas universalmente
- En las declaraciones de tipos de datos algebraicos se evitan las palabras clave **data** o **type**. Para marcar este tipo de declaraciones se utiliza el símbolo $\triangleq$. Por ejemplo, el tipo predefinido *Either*

  data *Either a b* $=$ *Left a* | *Right b*

  se definirá como

  $\alpha \| \beta \triangleq L\,\alpha \mid R\,\beta$

- Salvo que sean estrictamente necesarios, se evitan los constructores de tipos. Por ejemplo, el tipo

  $ErrorT\ m\ a = E\ (m\ (Either\ String\ a))$

  se definirá como

  $\mathcal{T}_{Err}\ \mathsf{m}\ \alpha \triangleq \mathsf{m}\ (String \| \alpha)$

- Con algunos operadores matemáticos, se realiza una sobrecarga implícita, pudiendo aparecer como operadores de función y a la vez como tipos de datos.

  Por ejemplo, $\oplus$ se define en la página 79 como un tipo de datos parametrizado y en la página 82 como un operador.

- El código fuente se ha escrito utilizando identificadores en inglés, ya que este idioma es el más utilizado internacionalmente para la realización de programas y el Sistema de Prototipado de Lenguajes ha sido desarrollado íntegramente en inglés.

Bibliografía

[1] Glasgow Haskell Compiler. `http://www.haskell.org/ghc/`.

[2] Haskell page. `http://www.haskell.org`.

[3] Language Prototyping System. `http://lsi.uniovi.es/ labra/LPS/LPS.html`.

[4] Objective CAML. `http://caml.inria.fr/ocaml/index.html`.

[5] The Common Framework Initiative. `http://www.brics.dk/Projects/CoFI/`.

[6] M. Abadi y L. Cardelli. *A theory of objects*. Springer, 1998.

[7] M. Abadi, L. Cardelli, y R. Viswanathan. An interpretation of objects and object types. En *ACM Symposium on Principles of Programming Languages*, páginas 369–409. 1996.

[8] O. Agesen, J. Palsberg, y M. I. Schwartzbach. Type inference of SELF: analysis of objects with dynamic and multiple inheritance. En Oscar M. Nierstrasz, editor, *Proceedings of the European Conference on Object-Oriented Programming (ECOOP)*, tomo 707, páginas 247–267. Springer-Verlag, Berlin, Heidelberg, New York, Tokyo, 1993.

[9] D. Álvarez, L. Tajes, F. Álvarez, M. A. Díaz, R. Izquierdo, y J. M. Cueva. An object-oriented abstract machine as the substrate for an object-oriented operating system. En J. Bosch y S. Mitchell, editores, *Object Oriented Technology ECOOP'97*. Springer Verlag, LNCS 1357, Jÿvaskÿla, Finland, June 1997.

[10] K. R. Apt. *From Logic Programming to Prolog*. International Series in Computer Science. Prentice Hall, 1997.

[11] I. Attali, C. Courbis, P. Degenne, A. Fau, J. Fillon, D. Parigot, C. Pasquier, y C. Coen. SmartTools: a development environment generator based on XML. En *XML Technologies and Software Engineering, International Conference on Software Engineering*. Toronto, Canada, 2001.

[12] R. Backhouse. *Program Construction and Verification*. Prentice Hall International, Englewood Cliffs, NJ, 1986.

[13] R. Backhouse y T. Sheard, editores. *First International Workshop on Generic Programming*. Marstrand, Sweden, Jun 1998.

[14] Roland Backhouse, Patrik Jansson, Johan Jeuring, y Lambert Meertens. Generic programming - an introduction. En S. D. Swierstra, P. R. Henriques, y J.Ñ. Oliveira, editores, *Advanced Functional Programming*, tomo 1608 de *Lecture Notes in Computer Science*. Springer, 1999.

[15] J. Backus. Can Programming be Liberated from the von Neumann Style: A Functional Style and Its Algebra of Programs. *Communications of the ACM*, 21(8):613–641, August 1978.

[16] L. S. Barbosa. Components as processes: An exercise in coalgebraic modeling. En S. F. Smith y C. L. Talcott, editores, *FMOODS'2000 - Formal Methods for Open Object-Oriented Distributed Systems*, páginas 397–417. Kluwer Academic Publishers, Stanford, USA, September 2000.

[17] M. Barr y C. Wells. *Toposes, Triples and Theories*. Springer-Verlag, 1983. Updated on-line version.

[18] D. Basanta, C. Luengo, R. Izquierdo, J. E. Labra, y J. M. Cueva. Constructing language processors using object-oriented techniques. En *Proceedings of the 6th International Conference on Object-Oriented Information Systems (OOIS 2000)*, páginas 358–367. London – UK, 2000.

[19] D. Basanta, C. Luengo, R. Izquierdo, J. E. Labra, y J. M. Cueva. Improving the quality of compiler construction with object-oriented techniques. *ACM SIGPLAN*, 35(12):41–51, diciembre 2000.

[20] J. L. Bentley. Programming pearls: Little languages. *Communications of the ACM*, 29(8):711–721, August 1986.

[21] N. Benton, J. Hughes, y E. Moggi. Monads and effects. En *International Summer School On Applied Semantics APPSEM'2000*. Caminha, Portugal, 2000.

[22] R. Bird y Oege de Moor. *Algebra of Programming*. Prentice Hall, 1997.

[23] Richard Bird. *Introduction to Functional Programming using Haskell*. Prentice Hall, 2 edición, 1998.

[24] D. Bobrow, L. G. DeMichiel, R. P. Gabriel, S. E. Keene, G. Kiczales, y D. A. Moon. Common lisp object system specification. *SIGPLAN Notices*, 23, 1988.

[25] G. Boole. *El Análisis Matemático de la Lógica*. Ed. Cátedra, 1984. Original de 1847.

[26] E. Börger y D. Rosenzweig. A mathematical definition of full prolog. *Science of Computer Programming*, 1994.

[27] E. Börger y W. Schulte. Programmer friendly modular definition of the semantics of java. En J. Alver-Foss, editor, *Formal Syntax and Semantics of Java*, LNCS. Springer, 1998.

[28] A. H. Borning. Classes versus prototypes in object-oriented languages. En *Proceedings of the ACM-IEEE Fall Joint Computer Conference, Montvale (NJ), USA*, páginas 36–39. 1986.

[29] P. Borras, D. Clement, Th. Despeyrouz, J. Incerpi, G. Kahn, B. Lang, y V. Pascual. CENTAUR: The system. En *Proceedings of the ACM SIGSOFT/SIGPLAN Software Engineering Symposium on Practical Software Development Environments (PSDE)*, tomo 24, páginas 14–24. ACM Press, New York, NY, 1989.

[30] D. F. Brown, H. Moura, y D. A. Watt. Actress: an action semantics directed compiler generator. En *4th International Conference on Compiler Construction*, tomo 641 de *LNCS*, páginas 95–109. Springer-Verlag, 1992.

[31] K. B. Bruce, L. Cardelli, y B. C. Pierce. Comparing object encodings. En *Invited lecture at Third Workshop on Foundations of Object Oriented Languages (FOOL 3)*. 1996.

[32] L. Cardelli. A semantics of multiple inheritance. En G. Kahn, D. MacQueen, y G. Plotkin, editores, *Semantics of Data Types*, número 173 en Lecture Notes in Computer Science, páginas 51–67. Springer-Verlag, 1984.

[33] L. Cardelli. Type systems. En *Handbook of Computer Science and Engineering*, capítulo 103. CRC Press, 1997.

[34] Luca Cardelli y Rowan Davies. Service combinators for web computing. Informe Técnico 148, Digital Equipment Corporation Systems Research, June 1997.

[35] Luca Cardelli y Peter Wegner. On Understanding Types, Data Abstraction, and Polymorphism. *Computing Surveys*, 17(4):471–522, December 1985.

[36] Robert Cartwright y Matthias Felleisen. Extensible denotational language specifications. En *Symposium on Theoretical Aspects of Computer Science*. 1994.

[37] Pietro Ceciarelli y Engenio Moggi. A syntactic approach to modularity in denotational semantics. En *5th Biennial Meeting on Category Theory and Computer Science*, tomo CTCS-5. CWI Technical Report, 1993.

[38] A. Cernuda, J. E. Labra G., y J. M. Cueva. Itacio: A component model for verifying software at construction time. En *Third International Workshop on Component Based Software Engineering, 22nd. International Conference on Software Engineering*. Limerick, Ireland, junio 2000.

[39] Yoonsik Cheon y Gary T. Leavens. A quick overview of Larch/C++. *Journal of Object-Oriented Programming*, 7(6):39–49, 1994.

[40] K. Claessen. A poor man's concurrency monad. *Journal of Functional Programming*, 9(3):313–323, mayo 1999.

[41] Koen Claessen. *An Embedded Language Approach to Hardware Description and Verification*. Proyecto Fin de Carrera, Chalmers University of Technology, 2000.

[42] Koen Claessen y Peter Ljungl'of. Typed logical variables in haskell. En *Haskell Workshop*. ACM SIGPLAN, September 2000.

[43] T. Colon. *Programming in Parlog*. Addison-Wesley, 1989.

[44] Computer y Business Equipment Manufacturers Association, editores. *Programming Language FORTRAN*. American National Standard Institute, Inc., 1978. X3.9-1978, Revision of ANSI X3.9-1966.

[45] W. Cook y J. Palsberg. A denotational semantics of inheritance and its correctness. *Information and Computation*, 114(2):329–350, 1994.

[46] O. Dahl, E. W. Dijkstra, y C. A. R. Hoare. *Structured Programming*. Academic Press, 1972.

[47] O. Dahl y R. Nygaard. Simula, an algol-based simulation language. *Communications of the ACM*, 9(9):671–678, 1966.

[48] P. Deransart, A. Ed-Dbali, y L. Cervoni. *Prolog: The Standard*. Springer-Verlag, 1996.

[49] K. Doh y P. Mosses. Composing programming languages by combining action-semantics modules. En M. van den Brand, M. Mernik, y D. Parigot, editores, *First workshop on Language, Descriptions, Tools and Applications*. Genova, Italy, April 2001.

[50] Luc Duponcheel. Writing modular interpreters using catamorphisms, subtypes and monad transformers, 1995. Utrecht University.

[51] C. S. Scholten E. W. Dijkstra. *Predicate Calculus and Program Semantics*. Texts and Monographs in Computer Science. Springer Verlag, 1990.

[52] U. Eco. *La búsqueda de la lengua perfecta*. Ediciones Altaya S.A., 1993.

[53] Kerstin I. Eder. *EMA: Implementing the Rewriting Computational Model of Escher*. Tesis Doctoral, Department of Computer Science, University of Bristol, November 1998.

[54] David Espinosa. *Semantic Lego*. Tesis Doctoral, Columbia University, 1995.

[55] P.Ñaur et al. Revised report on the algorithmic language algol 60. *Communications of the ACM*, 3(5):299 – 314, 1960.

[56] A. E. Fischer y F. S. Grodzinsky. *The Anatomy of Programming Languages*. Prentice-Hall International, 1993.

[57] Jeroen Fokker. Functional parsers. En J. Jeuring y E. Meijer, editores, *First International School on Advanced Functional Programming*, tomo 925, páginas 1–23. LNCS, 1925.

[58] Maarten M. Fokkinga. *Law and Order in Algorithmics*. Tesis Doctoral, University of Twente, febrero 1992.

[59] Maarten M. Fokkinga. Monadic maps and folds for arbitrary datatypes. Memoranda Informatica 94-28, Dept. of Computer Science, Univ. of Twente, junio 1994.

[60] Benedict R. Gaster y Mark P. Jones. A Polymorphic Type System for Extensible Records and Variants. Informe Técnico NOTTCS-TR-96-3, Department of Computer Science, University of Nottingham, noviembre 1996.

[61] K. Glynn, M. Sulzmann, y P.J. Stuckey. Type classes and constraint handling rules. Informe técnico, 2000.

[62] J. A. Goguen y J. Meseguer. Order-sorted algebra I: Equational deduction for multiple inheritance, overloading, exceptions, and partial operations. *TCS: Theoretical Computer Science*, 105(2):217–273, 1992.

[63] Joseph A. Goguen y Grant Malcolm. *Algebraic Semantics of Imperative Programs*. Foundations in Computer Science. The MIT Press, 1996.

[64] A. Goldberg y D. Robson. *Smalltalk-80. The language and its implementation*. Addison-Wesley, 1983.

[65] B. M. González, J. E. Labra, y J. M. Cueva. Web navigability testing with remote agents. En *Second ICSE Workshop on Web Engineering, 22nd. International Conference on Software Engineering*. Limerick, Ireland, junio 2000.

[66] M. Gordon, R. Milner, y C. Wadsworth. Edinburgh lcf. *Lecture Notes in Computer Science, Springer-Verlag*, (78), 1979.

[67] D. Gries. *The Science of Programming*. Springer-Verlag, New York, 1981.

[68] Carl A. Gunter. *Semantics of Programming Languages: Structures and Techniques*. Foundations of Computing Series. The MIT Press, Cambridge, MA, 1992.

[69] Y. Gurevich. Evolving algebra 1993: Lipari guide. En E. Börger, editor, *Specification and Validation Methods*, páginas 9–36. Oxford University Press, 1995.

[70] Jr. Guy Lewis Steele. Growing a Language, octubre 1998. Transcript of invited talk at OOPSLA'98, Vancouver.

[71] C. V. Hall, K. Hammond, S. L. Peyton Jones, y P. L. Wadler. Type classes in Haskell. *ACM Transactions on Programming Languages and Systems*, 18(2):109–138, marzo 1996.

[72] M. Hamana. Algebraic semantics for higher-order functional-logic programming. En *2nd Fuji International Workshop on Functional and Logic Programming*. World Scientific, Singapore, 1996.

[73] William Harrison y Samuel Kamin. Modular compilers based on monad transformers. En *Proceedings of the IEEE International Conference on Computer Languages*. 1998.

[74] William Harrison y Samuel Kamin. Compilation as metacomputation: Binding time separation in modular compilers. En *5th Mathematics of Program Construction Conference, MPC2000*. Ponte de Lima, Portugal, June 2000.

[75] J. Heering y P. Klint. Semantics of programming languages: A tool-oriented approach. *ACM SIGPLAN Notinces*, 2000.

[76] Jan Heering. Application software, domain-specific languages and language design assistants. En *Proceedings SSGRR 2000 International Conference on Advances in Infrastructure for Electronic Business, Science, and Education on the Internet*. L'Aquila, Italy, 2000.

[77] I. Heim y A. Kratzer. *Semantics in Generative Grammar*. Blackwell Pub., 1998.

[78] Laurie Hendren. Joos - java object oriented subset. `http://www.sable.mcgill.ca/ hendren/520/`.

[79] F. Henglein. Operational semantics: Objects, methods, classes and inheritance, 1997. Lecture Notes.

[80] Jonathan M. D. Hill y Keith Clarke. An introduction to category theory, category theory monads, and their relationship to functional programming. Informe Técnico QMW-DCS-681, Department of Computer Science, Queen Mary and Westfield College, agosto 1994.

[81] P. M. Hill y J. W. LLoyd. *The Gödell Programming Language*. Logic Programming. The MIT Press, 1994.

[82] R. Hinze. A generic programming extension for Haskell. En E. Meijer, editor, *Proceedings of the 3rd Haskell Workshop, Paris, France*. septiembre 1999. The proceedings appear as a technical report of Universiteit Utrecht, UU-CS-1999-28.

[83] R. Hinze y S. P. Jones. Derivable type classes. En *Haskell Workshop*, páginas 94–105. Electronic Notes in Theoretical Computer Science, Montreal, 2000.

[84] R. Hinze y S. P Jones. Multiparameter generic classes. Comunicación personal por correo electrónico, Mayo 2001.

[85] Ralf Hinze. Monad transformers and lift, July 1998. Message 00669, Haskell mailing list.

[86] Ralf Hinze. Prological features in a functional setting — axioms and implementations. En Masahiko Sato y Yoshihito Toyama, editores, *Third Fuji International Symposium on Functional and Logic Programming (FLOPS'98), Kyoto University, Japan*, páginas 98–122. World Scientific, Singapore, New Jersey, London, Hong Kong, abril 1998.

[87] Ralf Hinze. Deriving backtracking monad transformers. En Roland Backhouse y J.Ñ. Oliveira, editores, *Proceedings of the 2000 International Conference on Functional Programming, Montreal, Canada*. septiembre 2000.

[88] C. A. R. Hoare. An axiomatic basis for computer programming. *Communications of the ACM*, 12(10):576–580, October 1969.

[89] C. A. R. Hoare. Hints on programming language design. Informe Técnico STAN-CS-73-403, Computer Science Department, Stanford University, 1973.

[90] F. Honsell, A. Pravato, y S. Ronchi della Rocca. Structured operational semantics of a fragment of the language scheme. *Journal of Functional Programming*, 8(4):335–367, 1998.

[91] Z. Hu y H. Iwasaki. Promotional transformation of monadic programs. En *Workshop on Functional and Logic Programming*. World Scientific, Susono, Japan, 1995.

[92] P. Hudak. Domain-specific languages. En Peter H. Salus, editor, *Handbook of Programming Languages*, tomo III, Little Languages and Tools. Macmillan Technical Publishing, 1998.

[93] P. Hudak y P. Wadler. Report on the Functional Programming Language Haskell. Informe Técnico YALEU/DCS/RR656, Department of Computer Science, Yale University, 1988.

[94] J. Hughes. Generalising monads to arrows. *Science of Computer Programming*, 2000.

[95] John Hughes. The design of a pretty-printing library. En J. Jeuring y E. Meijer, editores, *First International School on Advanced Functional Programming*, tomo 925 de *LNCS*. Springer Verlag, 1995.

[96] G. Hutton. Fold and unfold for program semantics. En *3rd ACM SIGPLAN International Conference on Functional Programming*. Baltimore, Maryland, 1998.

[97] G. Hutton y E. Meijer. Monadic parser combinators. Technical Report NOTTCS-TR-96-4, University of Nottingham, diciembre 1996.

[98] G. Hutton y E. Meijer. Functional perl: Monadic parsing in haskell. *Journal of Functional Programming*, 8(4):437–444, July 1998.

[99] H. Iwasaki, Z. Hu, y M. Takeichi. Towards manipulation of mutually recursive functions. En *3rd. Fuji International Symposium on Functional and Logic Programming (FLOPS'98)*. World Scientific, Kyoto, Japan, 1998.

[100] P. Jansson. Functional polytypic programming use and implementation. Informe técnico, Computing Science, Chalmers University of Technology, 1997.

[101] P. Jansson y J. Jeuring. PolyP - a polytypic programming language extension. En *ACM Symposium on Principles of Programming Languages*, páginas 470–482. ACM Press, 1997.

[102] P. Jansson y J. Jeuring. Polytypic unification. *Journal of Functional Programming*, 8(5):527–536, 1998.

[103] J. Jeuring. *Theories for Algorithm Construction*. Tesis Doctoral, Utrecht University, 1993.

[104] Johan Jeuring, editor. *Workshop on Generic Programming*. Ponte de Lima, Portugal, July 2000.

[105] Mark P. Jones. A system of constructor classes: overloading and implicit higher-order polymorphism. *Journal of Functional Programming*, 5(1):1–35, enero 1995.

[106] Mark P. Jones. *Qualified Types: Theory and Practice*. Distinguished Dissertations in Computer Science. Cambridge University Press, enero 1995.

[107] Mark P. Jones. First-class Polymorphism with Type Inference. En *Proceedings of the Twenty Fourth Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. Paris, France, January 15-17 1997.

[108] Mark P. Jones. Type Classes with Functional Dependencies. En *Proceedings of the 9th European Symposium on Programming ESOP 2000*, tomo LNCS 1782. Springer-Verlag, Berlin, Germany, 2000.

[109] Mark P. Jones y L. Duponcheel. Composing monads. YALEU/DCS/RR 1004, Yale University, New Haven, CT, USA, 1993.

[110] N.D̃. Jones, C. K. Gomard, y P. Sestoft. *Partial Evaluation and Automatic Program Generation*. Prentice Hall International, International Series in Computer Science, junio 1993.

[111] N. D. Jones y F. Nielson. Abstract interpretation: a semantics based tool for program analysis. En S. Abramsky, D. M. Gabbay, y T. S. Maibaum, editores, *Handbook of Logic in Computer Science*, tomo 4 – Semantic Modelling, páginas 527–636. Oxford Science Pub., 1995.

[112] S. P. Jones, J. Hughes, L. Augustsson, D. Barton, B. Boutel, W. Burton, J. Fasel, K. Hammond, R. Hinze, P. Hudak, T. Johnsson, M. Jones, J. Launchbury, E. Meijer, J. Peterson, A. Reid, C. Runciman, y P. Wadler. Haskell 98: A Non-strict, Purely Functional Language. Informe técnico, febrero 1999.

[113] Bill Joy, Guy Steele, Jame Gosling, y Gilad Bracha. *The Java Language Specification*. Addison-Wesley, 2 edición, 2000.

[114] B. L. Kurtz K. Slonneger. *Formal Syntax and Semantics of Programming Languages*. Addison-Wesley, 1995.

[115] G. Kahn. Natural semantics. En *Proceedings of the Symposium on Theoretical Aspects of Computer Science*, tomo 247, páginas 22–39. Springer-Verlag, 1987.

[116] A. Kaldewaij. *Programming: the derivation of algorithms*. International Series in Computer Science. Prentice Hall, 1990.

[117] S. Kamin. *Programming languages: an interpreter based approach*. Addison-Wesley, 1990.

[118] S. Kamin. Research on domain-specific embedded languages and program generators. *Electronic Notes in Theoretical Computer Science, Elsevier Press*, 12, 1998.

[119] Samuel Kamin y Uday Reddy. Two Semantic Models of Object-Oriented Languages. En Carl A. Gunter y John C. Mitchell, editores, *Theoretical Aspects of Object-Oriented Programming*, capítulo 13, páginas 463–495. MIT Press, 1994.

[120] A. C. Kay. The early history of smalltalk. *ACM SIGPLAN Notices*, 28(3), March 1993.

[121] Richard B. Kieburtz. Designing and implementing closed domain-specific languages, 2000. Invited talk at the Workshop on Semantics, Applications and Implementation of Program Generation (SAIG).

[122] D.J. King y P.Wadler. Combining Monads. En *Proc. of the Fifth Annual Glasgow Workshop on Functional Programming*. Springer Verlag, 1992.

[123] Bjorn Kirkerud. *Programming Language Semantics*. International Thompson Computer Press, 1997.

[124] Kevin Knight. Unification: a multidisciplinary survey. *ACM Computing, Springer Verlag (Heidelberg, FRG and NewYork NY, USA)-Verlag Surveys, ; ACM CR 9005-0423*, 21(1), 1989.

[125] D. Knuth. Computer programming as an art. En *ACM Turing Awards Lectures – The First Twenty Years 1966–1985*. ACM Press, 1987. Award Lecture of 1974.

[126] D. Knuth y L. T. Pardo. The early development of programming languages. En N. Metropolis, J. Howlett, y G.-C. Rota, editores, *A History of Computing in the Twentieth Century*. Academic Press, 1980.

[127] Donald E. Knuth. The genesis of attribute grammars. En *WAGA conference proceedings, Paris 1990*, páginas 1–12. Springer-Verlag, 1990. Lecture Notes in Computer Science 461.

[128] R. Kowalski. *Logic for Problem Solving*. North Holland, 1979.

[129] Thomas S. Kuhn. *The Structure of Scientific Revolutions*. University of Chicago Press, 3rd edition edición, 1993.

[130] Pablo E. Martínez L. Generic parsing combinators. En *II Conferencia Latinoamericana de Programaci'on Funcional*. III Congreso Argentino de Ciencias de la Computaci'on, La Plata, Argentina, October 1997.

[131] J. E. Labra. *Análisis y Desarrollo de un Sistema Prolog según borrador norma ISO 13211-I*. Proyecto Fin de Carrera, ETSIIIIG, 1994.

[132] J. E. Labra. An implementation of modular monadic semantics using folds and monadic folds. En *Workshop on Research Themes on Functional Programming, Third International Summer School on Advanced Functional Programming*. Braga - Portugal, 1998.

[133] J. E. Labra, J. M. Cueva, y C. Luengo. Language prototyping using modular monadic semantics. En *3rd Latin-American Conference on Functional Programming*. Recife - Brazil, March 1999.

[134] J. E. Labra, J. M. Cueva, y M. C. Luengo. Modular development of interpreters from semantic building blocks. En *The 12th Nordic Workshop on Programming Theory*. University of Bergen, Bergen, Norway, October 2000.

[135] J. E. Labra, J. M. Cueva, M. C. Luengo, y A. Cernuda. LPS: A language prototyping system using modular monadic semantics. En M. van den Brand y D. Parigot, editores, *First Workshop on Language Descriptions, Tools and Applications*, tomo 44. Electronic Notes in Theoretical Computer Science – Elsevier, Genova, Italy, abril 2001.

[136] J. E. Labra, J. M. Cueva, M. C. Luengo, y A. Cernuda. Specification of logic programming languages from reusable semantic building blocks. En *International Workshop on Functional and (Constraint) Logic Programming*. University of Kiel, Kiel, Germany, septiembre 2001.

[137] J. E. Labra, J. M. Cueva, y L. A. Oliveira. Harmony, a functional system for musical composition. En *II Conferencia Latinoamericana de Programación Funcional*. La Plata – Argentina, 1997.

[138] J. E. Labra, J. M. Cueva Lovelle, M. C. Luengo Díez, y B. M. González. A language prototyping tool based on semantic building blocks. En *Eight International Conference on Computer Aided Systems Theory and Technology (EUROCAST'01)*, Lecture Notes in Computer Science. Springer Verlag, Las Palmas de Gran Canaria – Spain, febrero 2001.

[139] P. J. Landin. The next 700 programming languages. *Communications of the ACM*, 9(3):157–166, 1966.

[140] Konstantin Laufer. Combining type classes and existential types. En *Proceedings of the Latin American Informatics Conference*. Mexico, 1994.

[141] John Launchbury y Tim Sheard. Warm fusion: Deriving build-catas from recursive definitions. En *Conf. Record 7th ACM SIGPLAN/SIGARCH Intl. Conf. on Functional Programming Languages and Computer Architecture, FPCA'95, La Jolla, San Diego, CA, USA, 25–28 June 1995*, páginas 314–323. New York, 1995.

[142] F. W. Lawvere y S. H. Schanuel. *Conceptual Mathematics. A First Introduction to Categories*. Cambridge University Press, 1997.

[143] Daan Leijen. Parsec, 2000. `http://www.cs.uu.nl/ daan/parsec.html`.

[144] Jeffrey R. Lewis, John Launchbury, Erik Meijer, y Mark B. Shields. Implicit parameters: dynamic scoping with static types. En *27th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, páginas 108–118. Boston, USA, January 2000.

[145] Sheng Liang. *Modular Monadic Semantics and Compilation*. Tesis Doctoral, Graduate School of Yale University, mayo 1998.

[146] Sheng Liang y Paul Hudak. Modular denotational semantics for compiler construction. En *Programming Languages and Systems – ESOP'96, Proc. 6th European Symposium on Programming, Linköping*, tomo 1058 de *Lecture Notes in Computer Science*, páginas 219–234. Springer-Verlag, 1996.

[147] Sheng Liang, Paul Hudak, y Mark P. Jones. Monad transformers and modular interpreters. En *22nd ACM Symposium on Principles of Programming Languages, San Francisco, CA*. ACM, enero 1995.

[148] T. Lindholm y F. Yellin. *The Java Virtual Machine Specification*. Java Series. Addison-Wesley, 2 edición, 1999.

[149] G. Link. *Algebraic Semantics in Language and Philosophy*. 74. CSLI Publishers, 1998.

[150] J. W. Lloyd. *Foundations of Logic Programming*. Springer-Verlag, 1993.

[151] C. Luengo, J. E. Labra, F. Domınguez, A. Pérez, N. Garcı́a, y J. M. Cueva. Desarrollo de compiladores en un sistema integral orientado a objetos. En *V Congreso Internacional de Investigación en Ciencias Computacionales CIICC´98*, páginas 291–301. noviembre 1998.

[152] Grant Malcolm. Data structures and program transformation. *Science of Computer Programming*, 14:255–279, 1990.

[153] J. Maraist, M. Ordersky, y P. Wadler. The call-by-need lambda calculus. *Journal of Functional Programming*, 8(3), 1998.

[154] J. McCarthy. Recursive functions of symbolic expressions and their computation by machine, part 1. *Communications of the ACM*, 3:184–195, 1960.

[155] John McCarthy. Towards a mathematical science of computation. En *IFIP Congress 1962*. North Holland, 1963.

[156] Bruce J. McLennan. *Principles of Programming Languages*. Oxford University Press, 2 edición, 1987.

[157] Lambert Meertens. Algorithmics - towards programming as a mathematical activity. En J. W. Bakker y J. C. van Vliet, editores, *Proceedings of the CWI Symposium on Mathematics and Computer Science*, páginas 289–334. North-Holland, 1986.

[158] E. Meijer, M. M. Fokkinga, y R. Paterson. Functional programming with bananas, lenses, envelopes and barbed wire. En *Functional Programming and Computer Architecture*, páginas 124–144. Springer-Verlag, 1991.

[159] E. Meijer y G. Hutton. Bananas in space: Extending fold and unfold to exponential types. En *Conf. Record 7th ACM SIGPLAN/SIGARCH and IFIP WG 2.8 Intl. Conf. on Functional Programming Languages and Computer Architecture, FPCA'95, La Jolla, San Diego, CA, USA, 25–28 June 1995*, páginas 324–333. New York, 1995.

[160] E. Meijer y J. Jeuring. Merging monads and folds for functional programming. En J. Jeuring y E. Meijer, editores, *Advanced Functional Programming*, Lecture Notes in Computer Science 925, páginas 228–266. Springer-Verlag, 1995.

[161] Erik Meijer. *Calculating Compilers*. Tesis Doctoral, University of Nijmegen, febrero 1992.

[162] Erik Meijer. More advice on proving a compiler correct: Improve a correct compiler. En *PHOENIX Seminar and workshop on declarative programming*. Springer Verlag, 1992. Volume 91 of Workshops in Computer Science.

[163] K. Meinke y J. V. Tucker. Universal algebra. En S. Abramsky, D. M. Gabbay, y T. S. E. Maibaum, editores, *Handbook of Logic in Computer Science*, tomo I. Oxford University Press, 1992.

[164] B. Meyer. *Object Oriented Software Construction*. Prentice-Hall, 2 edición, 1997.

[165] R. Milner. Fully abstract models of typed lambda calculi. *Theoretical computer science*, 4(1):1–22, 1977.

[166] R. Milner, M. Tofte, y R. Harper. *The definition of Standard ML*. MIT Press, Cambridge, Massachusetts, 1990.

[167] J. C. Mitchell y G. D. Plotkin. Abstract types have existential type. *ACM Transactions on Programming Languages and Systems*, 10(3):470–502, 1988.

[168] John C. Mitchell. *Foundations for Programming Languages*. The MIT Press, 1996.

[169] E. Moggi. Metalanguages and applications. En A. M. Pitts y P. Dybjer, editores, *Semantics and Logics of Computation*, Publications of the Newton Institute. Cambridge University Press, 1997.

[170] Eugenio Moggi. An abstract view of programming languages. Informe Técnico ECS-LFCS-90-113, Edinburgh University, Dept. of Computer Science, junio 1989. Lecture Notes for course CS 359, Stanford University.

[171] Eugenio Moggi. Notions of computacion and monads. *Information and Computation*, (93):55–92, 1991.

[172] R. Montague. *Ensayos de Filosofía Formal*. Alianza Editorial, 1974.

[173] G. V. Morrill. *Type Logical Grammar*. Kluwer Academic Press, 1994.

[174] C. Morris. *Writings on the general theory of signs*. Mouton and Co., 1971.

[175] Chriss Moss. *Prolog++, the power of Object Oriented and Logic Programming*. Addison-Wesley, 1994.

[176] P. Mosses. Foundations of modular SOS. Informe Técnico RS-99-54, BRICS, Dept. of Computer Science, University of Aarhus, 1999.

[177] Peter D. Mosses. *Action Semantics*. Cambridge University Press, 1992.

[178] Peter D. Mosses. A tutorial on action semantics. Informe técnico, Tutorial Notes for FME'96: Formal Methods Europe, Oxford, March 1996.

[179] G. Nadathur y D. Miller. An overview of λprolog. En K. A. Bowen y R. A. Kowalski, editores, *Fifth International Logic Programming Conference*, páginas 810–827. MIT Press, 1988.

[180] F. Nayeri. The modula-3 programming language. En Salus [206].

[181] T. Nicholson y N. Foo. A denotational semantics for prolog. *ACM Transactions on Programming Languages and Systems*, 11(4):650–665, 1989.

[182] H. R. Nielson y F. Nielson. *Semantics with Applications: A Formal Introduction*. John Wiley and Sons, revised on-line edición, 1999.

[183] Y. Onoue, Z. Hu, H. Iwasaki, y M. Takeichi. A calculational fusion system hylo. En *IFIP TC 2 Working Conference on Algorithmic Languages and Calculi*, páginas 76–106. Chapman & Hall, Le Bischenberg, France, 1997.

[184] P. Orbaek. Oasis: An optimizing action-based compiler generator. En *Proceedings 5th International Conference on Compiler Construction*, tomo 786 de *LNCS*, páginas 1–15. Springer-Verlag, 1994.

[185] F. G. Pagan. *Partial Computation and the construction of language processors*. Prentice Hall, 1991.

[186] Nikolaos S. Papaspyrou. *A Formal Semantics for the C Programming Language*. Tesis Doctoral, National Technical University of Athens, Department of Electrical and Computer Engineering, February 1998.

[187] Alberto Pardo. Monadic corecursion: Definition, fusion laws and applications. En B. Jacobs, L. Moss, H. Reichel, y J. Rutten, editores, *Selected Papers 1st Workshop on Coalgebraic Methods in Computer Science, CMCS'98, Lisbon, Portugal, 28–29 March 1998*, tomo 11. Amsterdam, 1998.

[188] Alberto Pardo. Towards Merging Recursion and Comonads. En *Workshop on Generic Programming, Ponte de Lima, Portugal*. julio 2000.

[189] L. C. Paulson. *ML for the working programmer*. Cambridge University Press, 1991.

[190] R. Peña. *Diseño de Programas: Formalismo y Abstracción*. Prentice-Hall, 2 edición, 1998.

[191] M. Pettersson. Rml – a new language and implementation for natural semantics. En M. Hermenegildo y J. Penjam, editores, *Proceedings of the 6th International Symposium on Programming Language Implementation and Logic Programming*, tomo 844, páginas 117–131. Springer-Verlag, 1994.

[192] B. C. Pierce. Type systems for programming languages, 2000. Working draft.

[193] B. C. Pierce y D. Turner. Simple type-theoretic foundations for object-oriented programming. *Journal of Functional Programming*, 4(2):207–247, abril 1994. Preliminary version in Principles of Programming Languages (POPL), 1993.

[194] M. Plezbert. Does just-in-time equal better-late-than-never? En *24th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. Paris, France, 1997.

[195] Gordon D. Plotkin. A structural approach to operational semantics. Informe Técnico DAIMI FN-19, Computer Science Department, Aarhus University, Aarhus, Denmark, 1981.

[196] Erik Poll. A coalgebraic semantics of subtyping. En *Coalgebraic Methods in Computer Sciece*. Berlin, Germany, 2000.

[197] T. W. Pratt y M. V. Zelkowitz. *Programming languages: Design and implementation*. Prentice Hall Intl., 1996.

[198] U. S. Reddy. Objects and classes in alogol-like languages. En *Fifth International Workshop on Foundations of Object-Oriented Languages*.

[199] U. S. Reddy. Objects as closures: Abstract semantics of object oriented languages. En *ACM Symposium on LISP and Functional Programming*, páginas 289–297. Snowbird, Utah, July 1988.

[200] John C. Reynolds. *Theories of Programming Languages*. Cambridge University Press, 1998.

[201] Niklas Röjemo. *Garbage collection, and memory efficiency, in lazy functional languages*. Tesis Doctoral, Chalmers University of Technology, Computing Science Department, 1995.

[202] Jan Rothe, Bart Jacobs, y H. Tews. The coalgebraic class specification language ccsl. En *4th Workshop on Tools for System Design and Verification*. Ulm, Germany, July 2000.

[203] Blas C. Ruiz, Francisco Gutiérrez, Pablo Guerrero, y José E. Gallardo. *Razonando con Haskell. Una Introducción a la Programación Funcional*. Universidad de Málaga, 2000.

[204] David E. Rydeheard y Rod M. Burstall. *Computational Category Theory*. Prentice Hall Series in Computer Science. Prentice Hall, 1988.

[205] J. I. Saeed. *Semantics*. Blackwell Pub., 1997.

[206] Peter H. Salus, editor. *Object Oriented Programming Languages*, tomo 1 de *Handbook of Programming Languages*. Macmillan Technical Publishing, 1998.

[207] D. Sannella y A. Tarlecki. Essential concepts of algebraic specification and program development. *Formal Aspects of Computing*, 9(3):229–269, 1997.

[208] J. Saraiva. *Purely functional implementation of attribute grammars*. Tesis Doctoral, University of Utrecht, 1999.

[209] D. A. Schmidt. *Denotational Semantics: A Methodology for Language Development*. Allyn and Bacon, Inc., Newton, MA, 1986.

[210] Tim Sheard y Leonidas Fegaras. A fold for all seasons. En *Proceedings 6th ACM SIGPLAN/SIGARCH Intl. Conf. on Functional Programming Languages and Computer Architecture, FPCA'93, Copenhagen, Denmark, 9–11 June 1993*, páginas 233–242. ACM, New York, 1993.

[211] Dorai Sitaram. Handling control. En *Proc. SIGPLAN '93 Conference on Programming Language Design and Implementation*, páginas 147–155. June 1993.

[212] Z. Somogyi, F. Henderson, y Thomas Conway. Mercury: an efficient purely declarative logic programming language. En *Australian Computer Science Conference*, páginas 499–512. Australia, February 1995.

[213] F. Spoto y G. Levi. A Denotational Semantics for Prolog. En M. Falaschi, M.Ñavarro, y A. Policriti, editores, *Proceedings of APPIA-GULP-PRODE '97 Conference*, páginas 201–212. Grado, Italy, 1997.

[214] Fausto Spoto. Operational and Goal-Independent Denotational Semantics for Prolog with Cut. *Journal of Logic Programming*, 42(1):1–46, January 2000.

[215] R. Stärk, J. Schmid, y E. Börger. *Java and the Java Virtual Machine: Definition, Verification, Validation*. Springer-Verlag, 2001.

[216] Guy L. Steele, Jr. Scheme: An interpreter for the extended lambda calculus. Informe técnico, MIT Artificial Intelligence Laboratory, 1975.

[217] Guy L. Steele, Jr. Building interpreters by composing monads. En *Conference record of POPL '94, 21st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages: Portland, Oregon, January 17–21, 1994*, páginas 472–492. ACM Press, New York, USA, 1994. ISBN 0-89791-636-0.

[218] L. Sterling y E. Shapiro. *The Art of Prolog*. MIT Press, 1986.

[219] A. Stoughton. *Fully abstract models of programming languages*. John Wiley and Sons, 1988.

[220] J. Stoy. *Denotational semantics: the Scott-Strachey approach to programming language theory*. The MIT Press, 1977.

[221] C. Strachey. Fundamental Concepts in Programming Languages. *Higher Order and Symbolic COmputation*, 13, April 2000. Reprinted from Lecture notes, International Summer School in Computer Programming at Copenhagen, 1967.

[222] C. Strachey y C. P. Wadsworth. Continuations: A Mathematical Semantics for Handling Full Jumps. *Higher Order and Symbolic COmputation*, 13, April 2000. Reprinted from a Technical Monograph at Oxford University Computing Laboratory, 1974.

[223] B. Stroustrup. *The C++ Programming Language*. Addison-Wesley, 1985.

[224] R. D. Tennent. Language design methods based on semantic principles. *Acta Informatica*, 8:97–112, 1977.

[225] R. D. Tennent. Denotational semantics. En S. Abramsky, D. M. Gabbay, y T. S. Maibaum, editores, *Handbook of Logic in Computer Science*, tomo 3. Oxford Science Publications, 1994.

[226] Delphine Terrasse. Encoding natural semantics in Coq. En V. S. Alagar, editor, *Proceedings of the Fourth International Conference on Algebraic Methodology and Software Technology*, páginas 230–244. Springer-Verlag LNCS 936, Montreal, Canada, 1995.

[227] Vladimir I. Ulogov. Language list. `http://oop.rosweb.ru/`.

[228] David Ungar y Randall B. Smith. Self: The power of simplicity. En Norman Meyrowitz, editor, *Proceedings of the Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA)*, tomo 22, páginas 227–242. ACM Press, New York, NY, 1987.

[229] F.W. v. Henke, H. Pfeifer, y H. Rueß. Guided tour through a mechanized semantics of simple imperative programming constructs. Ulmer Informatik-Berichte 96-11, Universität Ulm, Fakultät für Informatik, 1997.

[230] M. van den Brand et al. The asf+sdf meta-environment: a component-based language development environment. En *Compiler Construction*, LNCS. Springer-Verlag, 2001.

[231] Arie van Deursen, Paul Klint, y Joost Visser. Domain-specific languages: An annotated bibliography. *ACM SIGPLAN Notices*, 35(6):26–36, June 2000.

[232] P. Wadler. Theorems for free! En *Functional Programming Languages and Computer Architecture*, páginas 347–359. ACM Press, FPCA'89, Imperial College, London, 1989.

[233] P. Wadler. Comprehending monads. En *ACM Conference on Lisp and Functional Programming*, páginas 61–78. ACM Press, Nice, France, June 1990.

[234] P. Wadler. Comprehending Monads. *Mathematical Structures in Computer Science*, 2(4), 1992. (Special issue of selected papers from 6'th Conference on Lisp and Functional Programming.).

[235] P. Wadler. The essence of functional programming. En *POPL '92, Albuquerque*. 1992.

[236] P. Wadler. A formal semantics of patterns in xslt. En *Markup Technologies*. Philadelphia, December 1999.

[237] P. Walder y S. Blott. How to make *ad-hoc polymorphism* less *ad hoc*. En *16th ACM Symposium on Principles of Programming Languages*. 1989.

[238] D. C. Wang, A. W. Appel, J. L. Korn, y C. S. Serra. The zephyr abstract syntax description language. En J. C. Ramming, editor, *Proceedings of the USENIX Conference on Domain-Specific Languages*. USENIX Association, Berkeley, CA, October 1997.

[239] Keith Wansbrough. *A Modular Monadic Action Semantics*. Proyecto Fin de Carrera, Department of Computer Science, University of Auckland, febrero 1997.

[240] D. A. Watt y D. F. Brown. Formalising the dynamic semantics of java. En *3rd International Workshop on Action Semantics*, NS-00-6. BRICS, Dept. of Computer Science, Univ. of Aarhus, Recife, Brazil, 2000.

[241] David A. Watt. *Programming Language Syntax and Semantics*. C.A.R. Hoare Series in Computer Science. Prentice Hall International, 1991.

[242] David A. Watt. Why don't programming language designers use formal methods? En *Seminario Integrado de Software e Hardware - SEMISH'96*, páginas 1–6. University of Pernambuco, Recife, Brazil, 1996.

[243] David A. Watt. Joos action semantics. Informe técnico, Department of Computing Science, University of Glasgow, 1997.

[244] R. Wilhelm y D. Maurer. *Compiler Design*. Addison-Wesley, 1995.

[245] Glynn Winskel. *The Formal Semantics of Programming Languages: An Introduction*. Foundation of Computing Series. MIT Press, Cambridge, MA, 1993.

Índice