Notas para la asignatura Programación Declarativa Avanzada

Blas Carlos Ruiz Jiménez

Profesor del Departamento de Lenguajes y Ciencias de la Computación

Málaga, Marzo de 2001

Índice General

Capítulo 0

Programación Declarativa. Evolución y Tendencias

0.0 La Programación Declarativa

Mientras la palabra *programación* (de computadoras) tiene un significado claro[1], no parece tan claro el calificativo *declarativa* en el contexto de la programación. Por ello, trataremos de responder en esta sección a las siguientes preguntas:

¿Qué es la programación declarativa? ¿Cuáles son sus ejemplares?[2]

Un lenguaje declarativo es tal que sus frases describen relaciones entre los datos, ocultando los algoritmos dentro de la semántica del lenguaje. Esto es difícil de entender e intentaremos aclarar los términos.

Informalmente, la Programación Declarativa ($P_{RO}D_{EC}$) establece *qué* debe ser computado, pero no necesariamente *cómo* computarlo. En el ámbito de la programación, el término *declarativa* se suele introducir contraponiéndolo al término *imperativa* (o *procedimental*). En ésta las estructuras de control del lenguaje de programación permiten *seguir* perfectamente las acciones que debe realizar un programa. La ausencia de este tipo de estructuras de control es lo que caracteriza a los lenguajes declarativos. De esta forma, la $P_{RO}D_{EC}$ facilita la labor del programador, permitiéndole concentrarse en dar una formulación del problema, liberándolo del control del algoritmo: *cómo* la máquina, a partir de tal formulación, encontrará las soluciones del problema. Puede decirse que en la aproximación imperativa la construcción de un programa consiste en estudiar las secuencias de instrucciones que debo dar al computador para buscar la solución, mientras que la aproximación declarativa viene caracterizada por una actitud muy diferente: ¿cómo expresar la descripción del problema para que el computador pueda buscar las soluciones?. El primero de los estilos describe las *acciones* a realizar, mientras que el segundo describe el conocimiento

[1] En la literatura no es fácil encontrar definiciones concisas del concepto de programa; quizás una muy acertada, pero poco citada sea la de Ravi [Sethi, 1989]: "un programa es una especificación de un cómputo; un lenguaje de programación es una notación para escribir programas". El lector debería entonces preguntarse ¿qué entiende Ravi Sethi por especificación?

[2] Un informático escribiría ¿cuáles son sus instancias? Es usual utilizar el término *instancia* en lugar de *ejemplar*, pero el primero, que es correcto en español, no tiene la acepción deseada. Para el segundo, el Diccionario de la Real Academia Española propone la acepción: *cada uno de los individuos de una especie o de un género*; o también la acepción: *cada uno de los objetos de diverso género que forman una colección científica.*

acerca del problema. Así, en $P_{RO}D_{EC}$ un algoritmo aparece descompuesto a través de la famosa ecuación de [Kowalski, 1979a]:

$$\text{Algoritmo} = \text{Lógica} + \text{Control}^3$$

que indica que la componente (¿esencial?) de un algoritmo es la componente lógica o especificación de un problema en términos de cierta lógica. Tal ecuación ha sido considerada la *caja de Pandora* [Schreye y Denecker, 1999] de la $P_{RO}D_{EC}$, ya que permite aislar la especificación del problema de la estrategia de búsqueda de la solución. Hasta el punto es así que muchos autores han intentado generalizar la ecuación de múltiples formas con objeto de interpretar otros estilos de la $P_{RO}D_{EC}$. Más tarde, en la Sección 0.5, analizaremos la famosa ecuación y sus variantes.

Los dos principales paradigmas de la $P_{RO}D_{EC}$ son la Programación Lógica ($P_{RO}L_{OG}$) y la Programación Funcional ($P_{RO}F_{UN}$). Históricamente el término *declarativo* ha sido acuñado para cubrir la $P_{RO}F_{UN}$ y la $P_{RO}L_{OG}$, reconociendo que tienen muchos aspectos en común. Sin embargo, la comunidad científica que usa ambos estilos han reconocido trabajar de forma separada, con conferencias independientes (p.e., *International Logic Programming Conference*), y publicaciones también independientes (p.e., *Journal of Functional Programming*, Cambridge University Press, o *Journal of Logic Programming*). Afortunadamente, durante los últimos años se han consolidado varias conferencias sobre ambos paradigmas[4].

Entre los ejemplares de la $P_{RO}D_{EC}$ aparecen otros paradigmas además de las citados: Programación con Restricciones [Lassez, 1987; Maher, 1999], Programación Ecuacional [O'Donnell, 1985], Programación Inductiva [Lavrac y Dzeroski, 1997], Programación Algebraica [Bird y de Moor, 1997], o incluso mezcla de distintos paradigmas: Programación Concurrente con Restricciones [Saraswat, 1993], Programación Lógica/Concurrente con Restricciones [Ueda, 1999], y Programación Lógico/Funcional ($P_{RO}L_{OG}F_{UN}$) [Hanus, 1997; López-Fraguas y Sanchez-Hernandez, 1999][5].

[3] Durante la historia de la programación numerosos autores han querido expresar su *filosofía* o punto de vista de la programación a través de una ecuación que consideran la *caja de Pandora* del paradigma de la programación (aunque el personaje de la mitología griega no represente esta idea: *abrió la caja que se le prohibió y trajo todos los problemas al mundo*). Una de tales ecuaciones, quizás menos conocida, es *Algoritmo + Estrategia = Paralelismo* [Trinder et al., 1998], y otra muy conocida es *Algoritmos + Estructuras de Datos = Programas* de Niklaus [Wirth, 1976]. En general no estoy muy de acuerdo con el uso de tales ecuaciones o acertijos ya que en muchos casos no capturan de forma apropiada el paradigma de la programación, como es el caso de la ecuación de Wirth. En efecto, en algunos lenguajes el programa es un dato más y viceversa (y precisamente ésta es la idea genial de John von Neumann); considérese por ejemplo en LISP, donde un programa puede modificar el propio programa, o incluso puede interpretarlo; o en lenguajes del estilo de APL, o incluso en la concepción más actual de la programación OO con vinculación y sobrecarga dinámicas. Incluso podemos ir más lejos: en el λ–cálculo no existen datos o estructuras de datos, todos los objetos son funciones (los números, los valores lógicos, las estructuras arbóreas, ...), y un programa es una función. Y todavía podemos dar un ejemplo más complejo, como es el caso del lenguaje RUSSELL [Boehm et al., 1980; Hook, 1984], un lenguaje funcional polimórfico, cuyo sistema de tipos permite que un tipo pueda ser el argumento de una función.

De cualquiera de las formas, en el caso de la ecuación de Kowalsky la formulación de la ecuación está muy próxima a lo que pretende capturar.

[4] Como ejemplo podemos citar la *Joint Conference on Declarative Programming* (APPIA-GULP-PRODE), que lleva ya ocho ediciones, o la reciente *PADL 99: First International Conference on Practical Aspects of Declarative Languages* (*LNCS* 1551); además de otras conferencias donde ambos paradigmas están presentes. como la *International Conference on Programming Languages, Implementations, Logics, an Programs* (PLILP), que lleva ya 12 ediciones.

[5] Ver `www.informatik.uni-kiel.de/~curry/` y `{mozart,titan}.sip.ucm.es/toy`.

Aún lo dicho anteriormente, los principales paradigmas de la $P_{RO}D_{EC}$ son la $P_{RO}L_{OG}$ y la $P_{RO}F_{UN}$. Por esta razón, muchos autores han intentado localizar los aspectos que permiten unificarlas. Ello aclararía si realmente ambos estilos son ejemplares de una misma filosofía o punto de vista de la computación.

Quizás una primera visión interesante de partida sea: fórmulas como programas[6]. Desde esta perspectiva, un programa tiene dos lecturas. La primera es ver la fórmula como un programa desde el punto de vista operacional, con un modelo de reducción simple y conciso. La segunda es ver un programa como una fórmula de una lógica a la que exigiremos también una semántica declarativa simple. El término *declarativa* indica que se utiliza un dominio matemático conocido en el cual interpretar los objetos sintácticos[7].

La posible ventaja de esta visión viene dada cuando la interpretación semántica es sencilla, y de esta forma los programas son fáciles de manejar, transformar y verificar[8].

Así, si leemos el programa como una fórmula podemos razonar sobre su corrección centrando nuestra atención en un análisis lógico de la fórmula, a través de un formalismo que permita probar que el programa satisface la especificación. Obviamente, tanto la especificación como el programa son fórmulas [Apt et al., 1999]:76(pie de página)[9]. A finales de los años 70, CAR Hoare escribía:

> . . . una especificación es un predicado que describe todas las observaciones de un sistema complejo
> . . . un programa es un predicado expresado utilizando un subconjunto de conectivas, codificadas con un lenguaje de programación. [Hoare y Jones, 1989]

Otros autores establecen paralelismos entre ambas concepciones:

> . . . si sustituimos conectivas por cláusulas de Horn, la especificación es un programa lógico y no es necesaria una codificación. [Ringwood, 1988]

Normalmente en un programa intervienen varias fórmulas. Así, podemos formalizar algo más la concepción anterior en la forma siguiente. La idea de la $P_{RO}D_{EC}$ es que[10]:

- ✓ un programa es una teoría descrita en alguna *lógica* $\mathcal{L}$, y
- ✓ un cómputo es una deducción a partir de la teoría.

[6] En la interpretación de Curry–Howard–De Bruijn [Sørensen y Urzyczyn, 1998] las fórmulas son tipos y sus habitantes son las demostraciones. Aquí es muy diferente: no se intenta probar la fórmula, sino que ésta misma tiene una interpretación operacional.

[7] Algunos autores – yo en particular – prefieren el término *semántica denotacional*, que es más preciso. Así, la semántica denotacional es un objeto de cierto dominio, como puede ser un subconjunto de la base de Herbrand. Seguiremos utilizando también el término *semántica declarativa*, ya que es el que más aparece en la literatura. Realmente, la semántica operacional viene determinada por una relación de reducción; tal relación es de nuevo un objeto matemático definido normalmente a través de un sistema axiomático o inductivo [Hennessy, 1990; Winskel, 1993; Ruiz Jiménez, 1999], por lo que la semántica operacional puede ser vista como un caso particular de semántica denotacional. La diferencia esencial entre las visiones es puramente histórica, y puramente estilista.

[8] O al menos así debería ser, de forma que éste es un objetivo esencial de la $P_{RO}D_{EC}$.

[9] Esta aproximación es aplicable a la programación imperativa. Por ejemplo, si consideramos la semántica de Dijkstra de un lenguaje imperativo vía transformadores de predicados [Dijkstra y Scholten, 1990; Ruiz Jiménez, 1999], un programa $\mathcal{S}$ tiene asociado una fórmula: su transformador de predicados, que es identificado con $\mathcal{S}$; así, si X es un predicado, $\mathcal{S}.X$ es una fórmula, y una especificación [Hoare, 1969] o triple de Hoare $\{Y\}\mathcal{S}\{X\}$ pasa a ser una fórmula dentro del cálculo de predicados sobre un espacio de estados: $[Y \Rightarrow \mathcal{S}.X]$.

[10] Seguimos aquí la presentación dada en el primer capítulo de [Lloyd, 1995].

¿Qué lógica $\mathcal{L}$ es apropiada? Los principales requisitos de la lógica $\mathcal{L}$ es que debe tener: (1) un modelo teórico (semántica declarativa), (2) un mecanismo de deducción o cómputo (semántica procedural u operacional), y (3) un teorema de corrección (los cómputos o respuestas calculadas deben ser correctos). Ejemplos de tales teorías son la lógica de primer orden (PROLOG), ciertas lógicas polimórficas de primer orden (GÖDEL), o el λC simple tipificado (λPROLOG, ESCHER). Ejemplos de mecanismos de cómputo asociados pueden ser la resolución o la reescritura.

Dependiendo del lenguaje, se requieren otras propiedades, como la compleción, la confluencia, o la terminación. De esta forma, la división entre $\mathrm{P_{RO}F_{UN}}$ y $\mathrm{P_{RO}L_{OG}}$ puede ser vista desde un punto de vista histórico o sociológico, pero no técnico[11].

¿Cuales son los principios de la $\mathrm{P_{RO}D_{EC}}$?

El proceso en la tarea de la programación comienza formalizando el problema en la lógica para después escribir la componente lógica o programa. En $\mathrm{P_{RO}L_{OG}}$ tal componente es una teoría consistente en un conjunto de axiomas descrito como fórmulas de primer orden o cláusulas de Horn. En $\mathrm{P_{RO}F_{UN}}$ la componente lógica del programa puede ser una colección de ecuaciones para un λC tipificado. Una vez escrita la componente lógica, puede ser necesario que el programador añada una componente de control explícita. En este sentido, podemos distinguir entre $\mathrm{P_{RO}D_{EC}}$ en sentido *débil*, donde los programas son teorías pero el programador puede suministrar información (control) con objeto de obtener un programa eficiente. En la $\mathrm{P_{RO}D_{EC}}$ en sentido *fuerte* todo el mecanismo de control es implícito y el programador solo proporciona una teoría o programa.

El punto de vista descrito permite confirmar que los estilos funcional y lógico son ejemplares de una misma filosofía. Otra segunda visión puede consistir en unificar los paradigmas $\mathrm{P_{RO}L_{OG}}$ y $\mathrm{P_{RO}F_{UN}}$, por ejemplo vía métodos algebraicos proporcionados por la *Programación ecuacional* [O'Donnell, 1977][12]. Esta aproximación considera el uso de ecuaciones para definir funciones y su evaluación mediante reducción por reescritura [Klop, 1992], buscando en un espacio de estados una forma irreducible, que en el caso de la $\mathrm{P_{RO}F_{UN}}$ viene dada por la ausencia de redexes, y en el caso de la $\mathrm{P_{RO}L_{OG}}$ es la derivación de la cláusula vacía. Por tanto, al eliminar las conectivas lógicas y todo símbolo de predicado distinto de la igualdad, se obtiene un subconjunto restringido de la lógica de primer orden: la lógica ecuacional. La programación ecuacional incide en mayor medida en el contenido lógico de las computaciones y establece una conexión clara con otros estilos declarativos, como el lógico/funcional [Moreno Navarro y Rodríguez Artalejo, 1988; Hanus, 1994]. A tales estilos dedicaremos también otra sección en este capítulo.

Una tercera visión muy prometedora es la proporcionada por la *Semántica contextual*. La confluencia, propiedad esencial del lambda cálculo, se puede extender a otros sistemas de reescritura. Sin embargo, en muchos casos puede interesar que se pierda tal propiedad. Tales sistemas han sido modelados por [Abramsky, 1990] en el marco de la *semántica contextual*, que proporciona información de los términos (programas con o sin variables libres) a través de una relación de preorden contextual (un contexto es un término con un hueco)

[11] Curiosamente, Alan Robinson llama $\mathrm{P_{RO}L_{OG}}$ a la programación relacional, y la combinación de la programación relacional con la funcional la llama programación lógica.

[12] Aunque después dedicaremos una sección a este paradigma, el lector interesado puede encontrar una panorámica en [Bidoit et al., 1991; Wirsing, 1990], y más recientemente en [Lalement, 1993].

que permite definir una noción de equivalencia de programas. En tal modelo la confluencia no es importante, siendo ésta reemplazada por el concepto de transformación correcta. Tal modelo permite capturar el indeterminismo y la evaluación perezosa [Schmidt-Schaußy Huber, 2000], y la evaluación de un término puede realizarse en presencia de variables libres. Dedicaremos más tarde otra sección a esta aproximación.

A pesar de la profusión de la $P_{RO}D_{EC}$ en el currículum, normalmente su exposición está basada en ejemplos aislados y sin principios comunes. [Sabry, 1999] argumenta que el fundamento principal de la $P_{RO}D_{EC}$ es la abstracción: el uso de abstracciones que compatibilizan con el dominio del problema. En algunos casos, las abstracciones son esencialmente TADs (p.e., el TAD de las ecuaciones lineales en MATHEMATICA [Wolfram, 1999]). En otros casos las abstracciones son herramientas encapsuladas en una librería. En casos más complejos, las abstracciones son proporcionadas por un lenguaje orientado al dominio (*domain–specific language*)[13]. [Sabry, 1999] defiende que la clave del éxito de la enseñanza de la $P_{RO}D_{EC}$ es la enseñanza de las abstracciones en programación en todos los niveles imaginables[14].

La historia de las abstracciones es tan amplia como la historia de los lenguajes de programación. En su forma final, la $P_{RO}D_{EC}$ proporciona soluciones a los problemas usando únicamente el enunciado del problema como programa. Históricamente este nivel de abstracción ha sido asociado con lenguajes de programación especializados, pero es usado hoy en día de forma rutinaria por estudiantes y profesores en una variedad de situaciones, lenguajes y cursos.

0.1 Programación Funcional

La abstracción sobre la arquitectura de von Neuman ha influido en la evolución de los lenguajes, desde los ensambladores hasta los orientados a objetos. Una filosofía distinta a la empleada por estos lenguajes es la empleada en la programación funcional, donde no aparece (o no debería aparecer) el concepto de variable mutable que es reemplazado por el concepto de función matemática como entidad de primera clase: son pasadas como argumentos a otras funciones, o devueltas como resultado, aparecen en estructuras de datos, etc. Así, programar en un estilo funcional consiste básicamente en definir funciones a partir de otras más sencillas, componiéndolas. La semántica de una función se puede identificar con el conjunto de valores que computa a partir de ciertas entradas, y el valor de una función no depende del momento en que se evalúa. El concepto de variable es el de la matemática: una variable denota un mismo valor; las variables pueden utilizarse para denotar, tanto los argumentos de las funciones como otras funciones, y las funciones pueden definirse a partir de estas variables.

El uso de funciones para crear programas, junto con operaciones básicas (composición, recursión y condicional) es la esencia de la programación funcional; pero a ello hay que añadir una propiedad fundamental: la *transparencia referencial*, es decir, el valor de una función depende únicamente de los valores de sus argumentos, las variables no pueden representar valores diferentes, y una vez que a una variable se le asigna un valor, éste no puede cambiar. Los lenguajes funcionales con tal propiedad se llaman *aplicativos puros*

[13] Existe una conferencia sobre tales lenguajes, organizada por *USENIX Association*. Ver por ejemplo, [Leijhen y Meijer, 2000], donde referencia las actas de la conferencia de 2000.

[14] [Sabry, 1999] proporciona ejemplos para cursos introductorios y superiores descritos con herramientas que generan código JAVA [Arnold y Gosling, 1996; Naughton, 1996; Flanagan, 1996] (CUP, LALR y JLex).

aunque algunos autores incluyen tal propiedad en el término "lenguaje funcional"; esta es una diferencia fundamental con los llamados lenguajes imperativos ya que la ausencia de variables mutables prohibe los efectos laterales y el cómputo de una función puede realizarse por reescritura o reducción: sustitución de llamadas a funciones (según su definición) por los argumentos. Estamos hablando lógicamente de un lenguaje funcional puro; por razones de eficiencia algunos lenguajes (p.e. LISP) incluyen ciertas características imperativas, como las variables mutables.

La noción de igualdad es otra característica fundamental. Dos expresiones se consideran iguales si se pueden transformar una en otra por reescritura[15]. Para garantizar cómputos deterministas es necesario añadir la propiedad de confluencia, y ésta queda asegurada en aquellas clases de sistemas de reescritura que resultan ser *ortogonales* [Klop, 1992], en los cuales, la ortogonalidad puede verificarse fácilmente mediate condiciones sintácticas. Los sistemas de reescritura también se utilizan para modelar otras características en programación funcional.

La transparencia referencial garantiza que esta igualdad sea sustitutiva, de forma que las expresiones pueden ser manipuladas como entidades matemáticas, utilizando leyes algebraicas conocidas. Esto dota al programador de un marco conceptual simple y suficientemente expresivo que le permite verificar y transformar programas en forma algebraica. Así, los lenguajes funcionales basados en un λ–cálculo extendido junto con la correspondiente semántica operacional proporciona un concepto de igualdad junto con un comportamiento que puede ser utilizado para definir técnicas de transformación de programas en forma correcta.

0.1.1 El λ–Cálculo

El lambda cálculo (λC) fue concebido por el lógico/matemático norteamericano Alonzo Church en los años 30 [Church, 1932; Church, 1933; Church y Rosser, 1936] como parte de una teoría general para modelar las funciones y la lógica de orden superior. El resultado sería una teoría tan potente como la desarrollada por Alan Turing: las funciones computables por una máquina de Turing son las funciones representables en el λC. Así, el λC es el núcleo de cualquier lenguaje de orden superior no trivial, utilizable tanto para la lógica como para la programación.

Church comienza planteándose si el cálculo de los números de Betti para una variedad algebraica es "computable". Idea una teoría $\mathcal{T}$ basada en la noción de función para fundamentar la matemática y la noción de cómputo. Añadiría posteriormente algunos axiomas para capturar nociones lógicas. Desgraciadamente, dos alumnos excepcionales de Church (Kleene y Rooser) probaron la inconsistencia de tal teoría [Kleene y Rosser, 1935], y [Church, 1936] aisla el λC: la parte esencial de $\mathcal{T}$ que trata de las funciones.

Lo primero que llama la atención es la sencillez del λC, cuyos términos son: variables, abstracciones ($\lambda x.M$, para representar la función $x \mapsto M$) y aplicaciones ($A\ B$)[16].

El concepto de reducción en el λC viene determinado por la relación β

[15] Una revisión actualizada de los sistemas de reescritura puede verse en [Klop, 1992; Dershowitz y Jouannaud, 1990; Meseguer, 2000]).

[16] El origen de símbolo para denotar la abstracción es muy antiguo; la notación en los *Principia Mathematica* de [Russell y Whitehead, 13] para la función $x \mapsto 2x+1$ es $2\widehat{x}+1$, que fue transformada por Church en la forma $\widehat{x}.2x+1$; por problemas de mecanografiado la notación derivó en $\wedge x.2x+1$, que se transformó posteriormente en la conocida $\lambda x.2x+1$.

$$(\ (\lambda x.M) \ N, \ M[x := N] \ ) \in \beta$$

que captura la aplicación de una función a un argumento. A partir de β se define $\rightarrow_\beta$ (β–reducción en un paso) como el cierre compatible de β, $\twoheadrightarrow_\beta$ como el cierre reflexivo y transitivo de $\rightarrow_\beta$, $=_\beta$ como el cierre simétrico y transitivo de $\twoheadrightarrow_\beta$. Otra forma de introducir el λC es como una teoría con una igualdad $=$, donde tal igualdad resulta ser una equivalencia compatible que contiene a la relación β. De tal forma

$$M =_\beta N \qquad \Leftrightarrow \qquad \lambda C \vdash M = N$$

[Church y Rosser, 1936] prueban que tal sistema es consistente a través de la confluencia de la β–reducción. Básicamente, la demostración de la consistencia es la utilizada hoy en día (ver, p.e., [Barendregt, 1984]:63).

La noción de λ–definibilidad es conceptualmente la base de la programación funcional. Church introduce la noción de λ–definibilidad de funciones $f : \mathbb{N}^k \rightarrow \mathbb{N}$ con objeto de capturar la noción de computabilidad[17]. Church establece que las funciones *intuitivamente* computables deben ser λ–definibles (*tesis de Church*). Esta tesis motiva el desarrollo de lo que hoy en día se conoce como *teoría de la computabilidad* o teoría básica de las funciones recursivas [Rogers, 1967] (puede verse un punto de vista más actual en [Constable y Smith, 1993]).

Al principio, Church solo logró probar la λ–definibilidad de ciertas funciones elementales. [Kleene, 1935], tras encontrar la λ–definibilidad de la función predecesor, prueba que las funciones recursivas primitivas son λ–definibles, lo que fortalecía la tesis de Church. Las funciones primitivas recursivas son trasladadas al λC usando, por ejemplo, el combinador

$$\mathbf{Y} \equiv \lambda f.(\lambda x.f \ (x \ x)) \ (\lambda x.f \ (x \ x))$$

donde tal término tiene la propiedad de ser un *combinador para puntos fijos*, es decir, verifica, para todo término F, la propiedad $\mathbf{Y}F =_\beta F(\mathbf{Y}F)$. Ackermann encuentra en 1928 una función 'intuitivamente computable' pero no recursiva primitiva [Ackermann, 1937]. Para resolver el problema se añade un esquema de *minimización*, y la clase de las funciones *recursivas parciales* sería el cierre para la minimización a partir de las primitivas. Kleene, en 1936, prueba que las funciones de tal clase son λ–definibles. Turing prueba en 1937 que también la clase de funciones λ–definibles coincide con la clase de funciones Turing–computables, y establece que esta clase coincide con las 'mecánicamente' computables (tesis de Turing).

Realmente, el cálculo introducido en [Church, 1936] es el $\lambda\mathbf{I}$C, en el cual solo se permiten abstracciones $\lambda x.M$ si x aparece libre en M; al eliminar tal condición se obtiene el λC [Church, 1941], que es el cálculo desarrollado bajo la influencia de Curry. Las funciones recursivas parciales son definibles en el $\lambda\mathbf{I}$C. La diferencia fundamental es que en el $\lambda\mathbf{I}$C todo término con forma normal es fuertemente normalizante; desgraciadamente, en el λC esta propiedad falla. Luego, si se usa el $\lambda\mathbf{I}$C como modelo para la programación funcional, la estrategia de evaluación impaciente (*eager*) siempre encuentra la forma normal.

[17] Barendregt sostiene que la 'verdadera' noción de computabilidad fue formalizada por primera vez en términos de λ–definibilidad para numerales; L. Wittgenstein, en un tratado escrito en 1920, ya representaba los numerales (de Church) como fórmulas; no está claro si Church se inspiró en ello.

No solo la aritmética y la lógica es representable (λ–definible) en el λC, sino también otras estructuras de datos (lineales y arbóreas) como muestran [Böhm y Gross, 1966] y también otros trabajos más recientes [Böhm et al., 1994; Mogensen, 1992].

La Lógica Combinatoria ($\mathcal{CL}$ o *Combinatory Logic*) [Hindley y Seldin, 1980] fue concebida con los mismos propósitos que el λC: construir una fundamentación de la matemática basada en las funciones. La idea básica la describió Moses Schönfinkel en la Sociedad Matemática de Göttingen en diciembre de 1920, y fue publicada como [Schönfinkel, 1924]; en su trabajo ya representa fórmulas lógicas con cuantificadores a través de expresiones combinatorias independientes de los axiomas de la lógica, en las cuales aparecen los combinadores **S** y **K**; por tanto introduce la notación parcializada[18]

La $\mathcal{CL}$ fue redescubierta por [Rosser, 1935], y convenientemente axiomatizada por Haskell Curry a partir de 1929 [Curry, 1934], tratando de utilizarla para construir una lógica similar a $\mathcal{T}$, pero encuentra también inconsistencias ([Curry, 1942], [Barendregt, 1984]:apéndice B). La $\mathcal{CL}$ es desarrollada posteriormente junto a Robert Feys [Curry y Feys, 1958; Curry et al., 1972], haciendo de ésta una técnica potente en la fundamentación de la matemática [Curry, 1977].

En la lógica combinatoria no aparecen las abstracciones y el lenguaje de términos es más simple que el del λC: constantes, variables y aplicaciones. Las constantes **S** y **K** permiten definir un concepto de igualdad verificando $\mathbf{K}AB = A$, y $\mathbf{S}ABC = AC(BC)$.

Es posible definir un concepto de abstracción similar al del λC, de suerte que toda expresión del λC es representable en la $\mathcal{CL}$. Curry demostró que añadiendo un conjunto de axiomas adicionales, las teorías λC y $\mathcal{CL}$ resultan equivalentes. El problema es que el proceso de conversión de un λ–término a una expresión combinatoria produce una expresión extraordinariamente grande (incluso para λ–términos simples); se introducen entonces nuevos combinadores y axiomas o reglas de reducción entre combinadores para simplificarlas (algoritmos de Curry y Turner) ([Hindley, 1985], [Revesz, 1988]:51–57, [Peyton Jones, 1987]:265–274). Estas expresiones combinatorias fueron utilizadas para implementar el lenguaje funcional Miranda [Turner, 1985; Thompson, 1995] y algunas máquinas de reducción, como la *SKIM* y la *NORMA* (Normal Order Reduction Machine)[19].

0.1.2 LisP y los trabajos de John McCarthy

Históricamente, el principal ejemplo de lenguaje funcional es LisP (LISt Processor); en cierto sentido, el precursor de LisP fue IPL V (Information Processing Language) desarrollado entre 1954 y 1958; es un lenguaje para proceso de listas específicamente diseñado para problemas de inteligencia artificial[20] y su principal aportación es representar los datos y programas con listas.

[18] Erróneamente atribuida a Haskell Curry, y que hoy se conoce como *currying*. Muy pocos autores citan a Schönfinkel como el descubridor de la notación parcializada. Una excepción es Christopher [Strachey, 2000], que además la usa en el lenguaje de programación CPL, denominándola *Schönfinkel's form*.

[19] Una descripción precisa de estas máquinas abstractas puede verse en [Field y Harrison, 1988].

[20] A propósito de la Inteligencia artificial he aquí una opinión un tanto particular de [Hofstadter, 1979] sobre el tema:

> "Hay un Teorema relativo al progreso en IA: una vez programada determinada función mental, la gente deja muy pronto de considerarla un ingrediente esencial del pensamiento real. El núcleo irrefutable de la inteligencia siempre reside en esa zona contigua que todavía no ha sido programada ... IA es todo aquello que todavía no ha sido concretado."

LISP [McCarthy, 1959; McCarthy et al., 1965] fue desarrollado por John McCarthy alrededor de 1960 en el MIT (Instituto Tecnológico de Massachusetts) también para resolver problemas de Inteligencia Artificial; causó sorpresa la aparición de este lenguaje, profunda y conceptualmente diferente a ALGOL 60, y más aún si tenemos en cuenta el hecho de que McCarthy perteneciera al comité que diseñó ALGOL 60.

Aparentemente el lenguaje es muy simple al estar formado por dos únicos objetos: átomos y listas, con los que se escriben las *s*–expresiones (*symbolic expressions*). A partir de una serie de funciones elementales ($atom, eq, cdr, car$ y $cons$), y utilizando una notación funcional, se definen las funciones de *s*-expresiones y las expresiones resultantes (p.e. $car[cons[(A.B); x)]]$) llamadas *m*-expresiones (*meta- expressions*), permiten especificar la forma en que son procesadas las *s*-expresiones. De hecho [McCarthy, 1960]:187 describe con este lenguaje todas las funciones computables al introducir expresiones condicionales y definiciones recursivas. McCarthy utiliza las expresiones condicionales deterministas con guardas: $[p_1 \rightarrow e_1; p_2 \rightarrow e_2; \ldots; p_n \rightarrow e_n]$, donde p_i es una expresión booleana y e_i representa una expresión arbitraria. El significado es el esperado: si p_1 entonces e_1 en otro caso si p_2 entonces $e_2, \ldots$, en otro caso si p_n entonces e_n. Tal mecanismo que describe el valor devuelto por una función dependiendo de ciertas guardas será utilizado posteriormente en otros lenguajes: ALGOL 60, C, CSP [Hoare, 1978; Hoare, 1985]. Así, la función para concatenar dos listas la escribe en la forma:

$$append[xs; ys] = [null[xs] \rightarrow ys; T \rightarrow cons[\, car[xs]; \, append[cdr[xs]; ys]]]$$

La función anterior se escribiría en un lenguaje funcional moderno con notación parcializada en la forma

$$\begin{array}{l} append\ xs\ ys\ =\ \textbf{if}\ null\ xs\ \textbf{then}\ ys \\ \qquad\qquad\qquad\qquad\qquad\qquad \textbf{else}\ \ head\ xs\ :\ append\ (tail\ xs)\ ys \end{array}$$

(donde el operador (:) denota el constructor *cons*) o también, en un lenguaje que disponga de patrones en la forma

$$\begin{array}{l} append\ xs\ ys\ =\ \textbf{case}\ xs\ \textbf{of}\ [\,] \qquad\ \rightarrow ys \\ \qquad\qquad\qquad\qquad\qquad\qquad\quad x : xs' \ \rightarrow x\ :\ append\ xs'\ ys \end{array}$$

(donde [ ] es la lista vacía) e incluso, si utilizamos un lenguaje con patrones a la izquierda de la ecuación, del estilo de HASKELL, podrían escribirse en la forma

$$\begin{array}{l} append\ [\,] \qquad ys\ = ys \\ append\ (x : xs)\ ys\ = x\ :\ append\ xs\ ys \end{array}$$

McCarthy utiliza además un mecanismo basado en el λ–cálculo de Alonzo [Church, 1941] para representar una *s*-función por una *s*-expresión (con funciones como argumentos).

Así, en LISP no solamente las estructuras de datos se implementan con listas, sino también las de control, ya que las funciones, e incluso los programas, son listas; un entorno de programación facilita las funciones más útiles que se llaman primitivas. El programador puede ampliar y definir su propio entorno (programa) añadiendo las primitivas necesarias, así como construir y mantener grandes bases de datos. Esta representación uniforme permite introducir la *s*-función universal *apply*, que permite definir un intérprete de LISP utilizando un sublenguaje [Horowitz, 1987]:355–361, técnica utilizada en la definición y desarrollo de posteriores lenguajes. [McCarthy, 1960] escribiría:

> ... la s-función universal *apply* juega el papel teórico equivalente al de una máquina de Turing universal y el papel práctico de un intérprete.

Desde la aparición de LISP aparecieron gran cantidad de dialectos y un gran esfuerzo en su estandarización finaliza con la aparición de COMMONLISP [Steele, 1984], al cual se deben ajustar todos las versiones de LISP que deseen considerarse dialectos, aunque algunas versiones puedan incluir otras características especiales.

La amplia difusión de la programación orientada a objetos (POO) y su metodología han dado lugar a lenguajes y/o sistemas híbridos, que toman a LISP y su entorno como base y a la filosofía de la POO como fundamento en su diseño (paso de mensajes, objetos, etc.); citemos entre ellos a: COMMONLOOPS [Bobrow y Kahn, 1985], FLAVORS [Moon, 1987] y OAKLISP [Lang y Pearlmutter, 1987].

Otro dialecto importante es SCHEME, diseñado por G. Steele y G. Sussman en 1975 en el MIT [Clinger y Rees, 1991]. Para éste se ha desarrollado un entorno pedagógico llamado DRSCHEME [Findler et al., 1997] para distintas plataformas (Windows95/NT, MacOS, X-Windows).

0.1.3 Los sistemas funcionales de John Backus

Desde los primeros años ha existido cierta controversia entre los lenguajes imperativos y funcionales; tales controversias no se reflejan simplemente en el modelo de cómputo y eficiencia, sino también en cuestiones conceptuales y de estilo. Como ejemplo de ello citemos el célebre artículo de John [Backus, 1978] *Can Programming be liberated from the von Neumann style?*. Este trabajo, además de criticar el estilo de von Neumann por razones *conceptuales* y de *claridad* en los programas, propone una alternativa clara a los lenguajes imperativos, base de posteriores investigaciones en lenguajes y sistemas.

En este trabajo Backus clasifica (de forma desafortunada como él mismo admite) los modelos de cómputo en *operacionales* (máquinas de Turing, p.e.), *aplicativos* (λ–cálculo, LISP y FPS-*Functional Programming Systems* – que él mismo propone) y *modelos de von Neumann* (ALGOL 68); tal clasificación será desafortunada ya que no permitirá incluir estilos como el orientado a objetos, el orientado al flujo de datos o la programación lógica.

En los modelos imperativos la semántica viene determinada por transición de estados complejos y en los aplicativos por reducción/sustitución. Así mismo, John [Backus, 1978]:614 considera[21]:

> ... los programas imperativos son poco claros y conceptualmente inútiles

aunque añade que la semántica axiomática [Hoare, 1969] permite recuperar la falta de elegancia de los programas imperativos (transformación de estados) bajo la forma de transformadores de predicados [Dijkstra, 1976].

Backus propone como alternativa los lenguajes funcionales basados en los sistemas funcionales o FPS. En un FPS los programas son funciones sin variables que son escritas utilizando objetos, funciones primitivas y formas funcionales para combinar objetos o funciones. La diferencia con el λ–cálculo es que éste utiliza un conjunto de reglas de sustitución para variables, mientras que un FPS no utiliza variables o reglas de sustitución,

[21] Llama la atención tal afirmación si tenemos en cuenta que John Backus diseñaría 20 años antes el lenguaje imperativo por excelencia: FORTRAN.

sino las propiedades de un álgebra de programas asociado; así, todas las funciones son del mismo tipo: objetos sobre objetos. El álgebra de programas verifica algunas leyes con las cuales se puede demostrar la corrección y la equivalencia de programas.

La principal limitación de un FPS es que no puede computar un programa, puesto que una función no es un objeto; luego no existe en tales sistemas una función *apply* equivalente a la de LisP; para resolver tal problema Backus introduce los sistemas formales para la programación funcional (FFPS o *Formal Systems for Functional Programming*) en los cuales las formas funcionales se representan en forma sistemática como listas de objetos (la cabeza determina la funcional representada y la cola los argumentos) teniendo cada forma funcional (o expresión) e una semántica μe, determinada por una función μ llamada función semántica; el resultado μe es un objeto, por lo que puede aparecer como argumento de una función.

La importancia del trabajo de Backus es que propone alternativas a la programación imperativa de forma justificada dentro del marco de los lenguajes funcionales. [Guezzi y Jazayeri, 1982], en el capítulo 8, parten de la definición del estilo FP enfatizando las características que debe tener un lenguaje funcional (transparencia referencial, composición de funciones, descripciones lambda, etc.) para después ejemplificar en dos posibles candidatos, como LisP o APL.

Un análisis parecido realiza [Horowitz, 1984] aunque no enfatiza demasiado los conceptos. El trabajo de J. Backus ha tenido claros defensores del estilo funcional tanto académico como práctico [Harrison y Khoshnevisan, 1985], y ha servido de base de modernas metodologías, como por ejemplo, la metodología basada en la transformación de programas (*program transformation*) orientada a lenguajes que soportan transparencia referencial [Darlington, 1985].

0.1.4 Otras notaciones funcionales

En la década del desarrollo de AlgoL 60 y sus derivados aparecieron lenguajes basados en conceptos muy diferentes y que obedecían a necesidades también muy dispares. De la necesidad del uso de lenguajes interactivos (*on-line*) surgen dos lenguajes que han tenido importancia en entornos educativos y profesionales; uno de ellos es BASIC[22] [Kemeny y Kurtz, 1967], considerado un dialecto interactivo de ForTran y diseñado para introducir la programación elemental, aunque hoy en día es considerado un lenguaje no menos que "peligroso"por los malos hábitos que crea al no soportar directamente una metodología basada en la estructuración. El otro lenguaje interactivo es APL[23] [Iverson y Falkoff, 1973] descrito inicialmente por Kenneth Iverson en el famoso texto *A Programming Language* [Iverson, 1962] que será implementado como lenguaje interactivo en 1967, alcanzando gran popularidad entre matemáticos e ingenieros.

Se trata de un lenguaje funcional con un potente conjunto de operadores, algunos de los cuales son extensiones de los operadores escalares para las operaciones con vectores y matrices; así mismo dispone de las estructuras de control convencionales de los lenguajes imperativos. Según [Horowitz, 1987], uno de los motivos de la tardanza en disponer de un intérprete es por la falta de estandarización en el conjunto de caracteres que denotan los operadores.

[22] *Beginners All purpose Symbolic Instruction Code.*
[23] A Programming Language

La potencia del lenguaje está en su expresividad y buen diseño lo que permite describir cálculos muy complejos con expresiones muy simples. Algunos autores [Wegner, 1976] consideran que no está claro si la popularidad de APL fue debida a la calidad de su sistema de programación o bien a la calidad del diseño del lenguaje. Añadiremos además como curiosidad que APL fue utilizado para describir formalmente la nueva familia de ordenadores IBM/360.

Un derivado importante de LisP es Logo, desarrollado por Seymour Papert[24] alrededor de 1970 [Abelson, 1982; Allan, 1984]. Además del proceso de listas de LisP, Logo incorpora una serie de primitivas para manejar una tortuga gráfica (o bien otros dispositivos) lo que permite un aprendizaje ameno de la programación. La facilidad de uso y su aproximación a un lenguaje natural ha permitido que numerosos autores se hayan preocupado por un uso sistemático del lenguaje en la enseñanza dentro de varios campos: estudio de propiedades geométricas, robótica, inteligencia artificial [Abelson y diSessa, 1981; Roselló, 1986], o incluso explorar campos atractivos como por ejemplo la *Geometría Fractal* de Benoit Mandelbrot [Hofstadter, 1979; Mandelbrot, 1982; Dewdney, 1987b; Dewdney, 1987a; Dewdney, 1988; Dewdney, 1989]. Aunque el diseño de gráficos es su principal ventaja (hasta el punto de que muchos lenguajes posteriores incorporan las primitivas para el manejo de la tortuga gráfica), el proceso de listas es más versátil que en LisP. Eficientes versiones para PCs han hecho de Logo un entorno interesante para la enseñanza de un lenguaje funcional. Una variante de Logo es SOLO [Eisenstadt, 1983], que ya introduce la comparación de patrones (*pattern matching*).

Otros lenguajes funcionales que caben destacar, pero con menos éxito que los anteriores, son Fun [Cardelli y Wegner, 1995], SOL[25] [Mitchell y Plotkin, 1985], y TALE[26] [Barendregt y van Leeuwen, 1986], interesantes al menos desde el punto de vista teórico ya que están basados en el λ–cálculo con tipos de segundo orden.

0.1.5 Características de los lenguajes funcionales modernos

Dentro de la comunidad educativa existe un acuerdo bastante generalizado sobre las características idóneas que debe tener un *buen lenguaje funcional*, sobre todo si es el primer contacto con la programación funcional. Entre tales características caben destacar: (0) la posibilidad de definir tipos o datos algebraicos (inductivos) de forma simple e intuitiva, (1) la posibilidad de definir funciones vía patrones dependiendo de la forma de los datos algebraicos, (2) el tipo de evaluación que soporte (impaciente o perezosa), (3) que disponga de funciones de primer orden o de orden superior, (4) inferencia de tipos y polimorfismo versus lenguajes libres de tipos (lenguajes no tipificados) o con tipificación dinámica (al estilo de LisP, Scheme o Erlang [Armstrong et al., 1996; Armstrong, 1997]), (5) concurrencia implícita versus concurrencia explícita, etc.[27]

[24] También en el MIT, donde coincide con McCarthy y Marvin Minsky. Realmente, aunque como *director* del proyecto figura S. Papert, el lenguaje es desarrollado entre 1966 y 1968 por un grupo dirigido por Wally Fuerzeig. En definitiva, Logo es desarrollado a la vez que LisP.

[25] Second-Order Lambda calculus.

[26] Typed Applicative Language Experiment.

[27] En [Hudak, 1989] puede encontrarse una comparación de las características de los principales lenguajes funcionales.

$$(ax) \quad \frac{}{\vdash x{:}\sigma} \qquad x : \sigma \in \Gamma$$

$$(\rightarrow e) \quad \frac{\Gamma \vdash f{:}\sigma \rightarrow \tau \qquad \Gamma \vdash a{:}\sigma}{\Gamma \vdash fa{:}\tau}$$

$$(\rightarrow i) \quad \frac{\Gamma, x{:}\sigma \vdash b{:}\tau}{\Gamma \vdash \lambda x.b{:}\sigma \rightarrow \tau}$$

Figura 0: El sistema $\lambda_{\rightarrow \text{Curry}}$

Tipos y Polimorfismo no restringido

El uso de tipos para modelar la lógica se remonta al trabajo de Bertrand Russell, *Mathematical logic as based on a theory of types* [Russell, 1908], donde desarrolla lo que llama *ramified theory of types*, aplicándola para resolver algunas paradojas clásicas. Esta teoría fue posteriormente adaptada a los *Principia Mathematica* [Russell y Whitehead, 13] y simplificada por otros matemáticos, como [Hilbert y Ackermann, 1928].

Alonzo Church, una década después de desarrollar el λC, e inspirado en las ideas de [Russell y Whitehead, 13], desarrolla una teoría simple de tipos (*simple theory of types*) [Church, 1940] como un primer intento para resolver las inconsistencias en su propuesta para la lógica, que fueron encontradas por Kleene y Rosser en 1936. En forma paralela, Haskell [Curry, 1934] desarrolla una teoría de tipos para la lógica combinatoria[28].

Desde entonces, han sido establecidas varios estilos para la descripción de los sistemas de tipos, distinguiéndose hoy en día esencialmente dos: sistemas de tipos *à la Church* y sistemas de tipos *à la Curry* [Barendregt, 1992]. Los primeros se suelen llamar también sistemas *explícitos* ya que en las abstracciones se tipifica en forma explícita la variable (p.e., en la forma $\lambda x : A.b$) mientras que los segundos se conocen como *implícitos* y en éstos los términos a tipificar son esencialmente los del λ–cálculo. Estos últimos tienen su origen en los trabajos de Haskell [Curry, 1934] para la teoría de combinadores.

Para asignar tipos a los términos combinatorios partimos de una colección numerable de variables de tipos $\mathbb{V}$ $(\alpha, \beta, \dots)$. El conjunto de tipos $\mathbb{T}$ se define con la sintaxis abstracta $\mathbb{T} ::= \mathbb{V} | \mathbb{T} \rightarrow \mathbb{T}$, de forma que $\alpha \rightarrow \beta$ denota el conjunto de las funciones de α en β.

Entre las notaciones *modernas* para representar la tipificación de los términos se suele utilizar $A : \sigma$, una variante de la utilizada por [Hindley y Seldin, 1986; Barendregt, 1984]. Un conjunto Γ de fórmulas de la forma $x : \sigma$ sin variables repetidas se llama una *base* o *contexto*. El sistema de tipos $\lambda_{\rightarrow \text{Curry}}$[29] viene dado por las reglas de la Figura 0.

El sistema de tipos $\lambda_{\rightarrow \text{Curry}}$ verifica una serie de propiedades muy interesantes. La primera de tales propiedades es la conservación del tipo bajo la presencia de reducciones del sujeto:

$$\Gamma \vdash a : \sigma \wedge a \twoheadrightarrow_\beta a' \qquad \Rightarrow \qquad \Gamma \vdash a' : \sigma$$

propiedad que es denominada *subject β–reduction* y que nosotros denotaremos con $S\beta$. La propiedad anterior tiene una aplicación inmediata a los lenguajes de programación: si Γ denota el conjunto de declaraciones de un programa y a es una expresión a evaluar en el

[28] Véase una excelente introducción en [Hindley, 1997].
[29] Una adaptación del definido en [Curry y Feys, 1958].

contexto del programa, la propiedad viene a decir que la evaluación de a (β–reducción) no cambia su tipo. Otra propiedad esencial de $\lambda_{\rightarrow \text{Curry}}$ es que *todo término tipificable es fuertemente normalizante*. Es decir, si es posible inferir $\Gamma \vdash a : \sigma$, entonces a es fuertemente normalizante: cualquier secuencia de reducciones $a \rightarrow_\beta a_1 \rightarrow_\beta a_2 \rightarrow_\beta \ldots$ es finita (resaltemos que esta propiedad no es válida en el λ–cálculo) y por tanto, (por el teorema de estandarización) la forma normal de a es efectivamente computable, así como la relación de igualdad $=_\beta$ es decidible para términos tipificables. Una aplicación inmediata de esta propiedad es que, en un lenguaje de programación con núcleo el sistema $\lambda_{\rightarrow \text{Curry}}$, los programas (términos tipificables) siempre terminan. Por otro lado, cualquier subtérmino de un término bien tipificado también es tipificable; lo que significa que cualquier subtérmino de un programa también es un programa (con o sin variables).

En el sistema $\lambda_{\rightarrow \text{Curry}}$, y en otros sistemas de tipos, son esenciales los siguientes problemas:

- *comprobación de tipos* (*type checking*), denotado con $\vdash^? a : \tau$
- *tipificación* (*typability*), denotado con $\vdash a :?$
- *habitabilidad* (*inhabitation*), denotado con $\vdash ? : \tau$

El primero consiste en, dados a y τ, probar si es posible inferir $\vdash a : \tau$. El segundo, también denominado *reconstrucción de tipos*, estudia si para cierto a existe τ verificando $\vdash a : \tau$. El tercero consiste en, dado τ, probar si existe a verificando $\vdash a : \tau$.

Un hecho interesante, que demuestran en forma independiente Haskell [Curry, 1969] y J.R. [Hindley, 1969], y redescubierto por Robin [Milner, 1978], es que si un término a es tipificable, entonces admite un *par principal* (Γ_a, τ_a); es decir, un *contexto principal* Γ_a y un *tipo principal* τ_a tales que $\Gamma_a \vdash a : \tau_a$, y además, si $\Gamma \vdash a : \tau$ entonces existe una sustitución $^* \equiv [\alpha_1, \ldots \alpha_n := \sigma_1, \ldots, \sigma_n]$ tal que $\Gamma_a^* \subseteq \Gamma$ y $\tau \equiv \tau_a^*$. Además, tal par principal es computable; es decir, existe una función recursiva (o algoritmo) que calcula a partir de un término su par principal, o falla, si el término no es tipificable.

Por tanto, en el sistema $\lambda_{\rightarrow \text{Curry}}$, la comprobación de tipos y la tipificación son problemas decidibles[30].

El lenguaje ML

El polimorfismo en la $\text{P}_{\text{RO}}\text{F}_{\text{UN}}$ es introducido por Strachey en 1967 ([Strachey, 2000]), y sería incorporado *como idea* al lenguaje funcional/imperativo CPL (*Combined Programming Language*). Tal lenguaje no sería nunca implementado de forma completa, ya que las ideas que introdujo necesitaban técnicas muy avanzadas.

El polimorfismo previsto por Strachey sería posteriormente formalizado por Robin [Milner, 1978], y utilizado en la definición de ML (*Meta Language*) [Milner et al., 1990; Paulson, 1991], un lenguaje con tipificación estática y evaluación impaciente. Su sistema de tipos es esencialmente $\lambda_{\rightarrow \text{Curry}}$ extendido con dos tipos especiales (Nat y $Bool$), algunas funciones (*suc*, *pred*, ...) para representar la aritmética natural y la lógica, y un generador de puntos fijos $\mathbf{Y}$ cuyo tipo asociado es $\mathbf{Y} : (\tau \rightarrow \tau) \rightarrow \tau$. Lo más interesante de ML es

[30] Utilizando la correspondencia de Howard–Curry–de Bruijn, puede probarse que la habitabilidad también es decidible; sin embargo, para ciertas extensiones de $\lambda_{\rightarrow \text{Curry}}$ algunos de estos problemas pueden ser indecidibles, e incluso algunos son aún problemas abiertos; por ejemplo en $\lambda 2$ – el λ–cálculo polimórfico – el problema de la habitabilidad es indecidible.

que permite representar cualquier función computable. Hoy día la implementación más utilizada es SML (Standard ML) [Milner, 1984; Milner et al., 1990].

Muchos investigadores admiten que ML está fuertemente influenciado por CPL, incorporando e implementando muchas ideas de éste.

ML posee una implementación muy eficiente y ha sido un estándar en ámbitos educativos[31]. Quizás el éxito de ML sea el haber sido el primer lenguaje moderno cuyo diseño e implementación están directamente basados en el λ–cálculo. Por ejemplo, usando técnicas derivadas directamente del λ–cálculo, Peter Landin diseña a principios de los 60 la máquina SECD [Landin, 1963], una máquina abstracta que resulta ser combinación de la notación del matemático holandés Nikolas [de Bruijn, 1972], junto a la notación posfija. Tal máquina será utilizada para implementar ML, y para (re)implementar también muy eficientemente LISP[32]. Sin embargo, su semántica es estricta, aunque existe una versión posterior con semántica perezosa llamada LML (*Lazy ML*)[33].

ML ha sido vastamente utilizado en aplicaciones industriales, así como en el desarrollo de aplicaciones educativas. Por ejemplo, [Jung y Michaelson, 2000] expone un sistema gráfico (implementado en SML) para visualizar la tipificación de expresiones durante su evaluación en presencia de polimorfismo. También ha sido utilizado para implementar distintos *proof checker* (como LCF, HOL, ISABELLE, NUPRL y COQ).

Ecuaciones vía patrones

En LISP (y sus derivados), FP o APL la definición de funciones es esencialmente por composición, por lo que la expresividad queda bastante limitada. A partir de las formas guardadas de LISP, surgen distintas notaciones para expresar la forma de calcular el valor de una función a partir de la forma de sus argumentos. El primer lenguaje funcional que utiliza esta característica es HOPE [Burstall et al., 1980; Bailey, 1985], desarrollado en 1979 en la Universidad de Edimburgo por Burstall, MacQueen y Sannella.

HOPE es un lenguaje aplicativo determinista con tipificación fuerte y funciones de orden superior. Soporta tipos definidos por el usuario (datos algebraicos) y polimorfismo. Para la definición de una función pueden utilizarse varias ecuaciones y comparación de patrones en los argumentos de la parte izquierda de las ecuaciones. He aquí la función *map* descrita en HOPE:

$$\begin{array}{l}\textbf{dec}\ map\ :\ (alpha\ \rightarrow\ beta)\ \#\ list(alpha)\ \rightarrow\ list(beta);\\ ---\ map(f, nil)\ <=\ nil;\\ ---\ map(f, x :: l)\ <=\ f(x) :: map(f, l);\end{array}$$

[31] Incluso existen editores *inteligentes* para la enseñanza de SML. Por ejemplo [Whittle et al., 1997] expone el editor *CYNTHIA* que permite en forma interactiva corregir errores al hacer un uso incorrecto del sistemas de tipos, de la recursión, o de los patrones, sugiriendo en algunos casos posibles soluciones; el sistema incorpora una interfaz gráfica para un uso más atractivo del sistema. Desgraciadamente, hasta al momento no se han desarrollado tales herramientas para HASKELL.

[32] Durante los años 60, Landin colaboraría estrechamente con Strachey, llegando a utilizar la máquina SECD para implementar un subconjunto del lenguaje CPL. [Field y Harrison, 1988] dedica un capítulo a la descripción de la máquina SECD. Otra referencia interesante de la máquina SECD es [Aït-Kaci, 1999b].

[33] Rober Harper ha preparado una página con información adicional interesante sobre ML (`http://www.cs.cmu.edu/~rwh/introsml/`) muy actualizada (Noviembre 2000), donde podemos encontrar implementaciones y una breve introducción a SML. Así mismo relata su experiencia docente en [Harper, 1999], enfatizando el papel de la PROFUN en el paradigma *programación como disciplina* à la [Dijkstra, 1976].

A diferencia de HASKELL y MIRANDA, HOPE no incorpora inferencia de tipos y el usuario debe describir el tipo de las funciones. Quizás lo más desafortunado de HOPE sea la notación para describir la aplicación, escribiendo los argumentos tras la función, entre paréntesis y separados por comas[34].

Las características más interesantes de HOPE también están presentes en MIRANDA [Turner, 1985], otro lenguaje que ha tenido una fuerte influencia en el diseño de lenguajes y la enseñanza de la programación funcional. Una diferencia fundamental entre HOPE y MIRANDA es el tratamiento de las funciones de orden superior; mientras en HOPE una función es creada utilizando expresiones "lambda", en MIRANDA se construyen por aplicación parcial de funciones existentes. Por ejemplo, la función HOPE $map(lambda\ x\ =>\ 1+x, xs)$, que incrementa en una unidad los elementos de la lista xs, se escribirá en MIRANDA en la forma $map\ ((+)1)\ xs$, donde la función $(+)1$ es una parcialización de la operación $(+)1\ x$, y siendo:

$$\begin{array}{ll} map\ f\ [\,] & = [\,] \\ map\ f\ (x :: xs) & = f\ x\ ::\ map\ f\ xs \end{array}$$

El uso de patrones proporciona un mecanismo extraordinariamente cómodo para la simplificación de código. Por ejemplo, consideremos el tipo *Temp* para representar temperaturas en grados Fahrenheit o Celsius, y la siguiente función de conversión de temperaturas vía patrones (escrita en HASKELL):

$$\begin{array}{l} \textbf{data}\ Temp\ =\ Far\ Float \mid Cel\ Float \\ convTemp\ ::\ Temp\ \rightarrow\ Temp \\ convTemp\ (Far\ x)\ =\ Cel\ (((x-32)/9)*5) \\ convTemp\ (Cel\ x)\ =\ Far\ ((x*9)/5+32) \end{array}$$

donde observamos que la conversión es directa e inmediata, y se codifica en función de la forma o patrón del argumento. El ejemplo también muestra que en $Far\ f$, f denota la magnitud absoluta, y Far la unidad de medida.

Dentro de la familia de lenguajes de orden superior caben destacar HASKELL, HOPE, MIRANDA y ML, todos ellos basados en un estilo ecuacional, con definiciones recursivas y patrones en la parte izquierda de las ecuaciones.

David Turner propuso el lenguaje MIRANDA como consecuencia de una larga experiencia en el estudio de notaciones y conceptos; el resultado es un lenguaje con comprobación estricta de tipos basado en ecuaciones recursivas de segundo orden, con una notación altamente expresiva y cercana a la notación usual de la Matemática. Actualmente, todavía se usa MIRANDA en las universidades, e incluso se escriben algunos libros con base este lenguaje [Thompson, 1995].

El diseño de casi todos los lenguajes funcionales posteriores ha estado muy influenciado por MIRANDA. Entre todos cabe destacar ORWELL. Diseñado por Philip Wadler en la Universidad de Oxford [Wadler, 1985], este lenguaje recoge y amplia muchos conceptos introducidos en MIRANDA. Basado en ORWELL, Richard Bird y Philip Wadler escriben su *Introduction to Functional Programming*, que se convirtió rápidamente en un *clásico* de

[34] Quizás tal notación desafortunada sea una herencia de LISP, que denota $f[x;y]$ la aplicación que en HOPE se escribe $f(x,y)$, mientras que en lenguajes con notación parcializada se escribe $f\,x\,y$. Llama la atención este hecho ya que la notación parcializada, popularizada por Curry en los años 30, es del siglo XIX, siendo utilizada ya por [Cauchy, 1821]

la programación funcional. También es cierto que otros textos basados en otros lenguajes han tenido mucho éxito. Por ejemplo, [Field y Harrison, 1988] utiliza HOPE, mientras que [Reade, 1989], muy influenciado por el anterior, utiliza SML.

La evaluación impaciente fue más fácil de implementar que la evaluación perezosa hasta la aparición de las técnicas de reducción de grafos introducidas por [Wadsworth, 1971]. El primer lenguaje perezoso implementado por tal técnica fue SASL [Turner, 1976], un lenguaje sin tipos, que sería el precursor de MIRANDA.

A principio de los 80 se perfeccionan los métodos de implementación de los LF vía reescritura (o reducción) de grafos, que es formalizada en [Barendregt et al., 1987b], lo que da lugar a la implementación de LEAN, sucesor de DACTL[35] [Glauert et al., 1987], y precursor del actual CLEAN, desarrollado en la Universidad de Nijmegen. La formalización de éstas técnicas para la aplicación a los lenguajes perezosos aparece en [Barendregt et al., 1987a], aunque a principio de los ochenta surge la máquina G (G–machine), que sería utilizada por un equipo de la Universidad de Chalmers para el desarrollo de LML (Lazy ML), una versión perezosa de ML. La descripción estándar de la máquina abstracta aparece en [Peyton Jones, 1987].

A partir de la máquina abstracta G y del estándar [Peyton Jones, 1987], se desarrollan a finales de los 80 otros lenguajes que tienen una fuerte influencia académica. Por un lado, CLEAN[36] [van Eekelen y et al., 1989; Nocker et al., 1991; Plasmeijer y van Eekelen, 1998], un lenguaje perezoso polimórfico con primitivas para explotar el paralelismo, que es implementado utilizando una versión paralela de la máquina G [Plasmeijer y van Eekelen, 1993]. Y finalmente por otro lado HASKELL, al que dedicamos la siguiente sección.

Evaluación perezosa y redes de procesos

Así, una clasificación de los lenguajes funcionales puede realizarse atendiendo al orden de evaluación de las expresiones, distinguiendo entre impacientes (o estrictos), y perezosos (no estrictos). HASKELL, MIRANDA y CLEAN son perezosos, mientras que SML, CAML, ERLANG y SCHEME son estrictos. Aunque las implementaciones actuales son igualmente eficientes, la programación perezosa introduce un concepto potencial: permite trabajar con estructuras *infinitas*.

Un ejemplo típico que permite ilustrar la potencia de la programación perezosa es el paradigma de las *redes de procesos*. Éste considera los procesos como funciones que consumen datos (desde una o varias listas) y producen valores para otras funciones (otra lista). Los procesos (expresiones) son suspendidos hasta que se solicita la evaluación (por algún proceso) de cierta expresión en la entrada (cierto elemento de una de las listas argumento). Normalmente la petición de evaluación la hace la comparación de patrones y una expresión se evalúa sólo parcialmente.

A menudo es interesante el uso de una red de procesos para resolver un problema describiendo la red con definiciones recursivas en un lenguaje funcional perezoso. Cada elemento de la red le hacemos corresponder a un proceso, y a éste una función. La interconexión entre procesos puede así mismo definirse vía llamadas entre las funciones asociadas a los procesos.

[35] Declarative Alvey Compiler Target Language

[36] El estado del lenguaje puede verse en `http://www.cs.kun.nl/~clean/`.

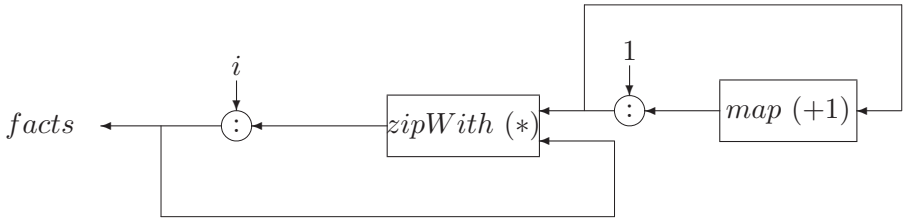
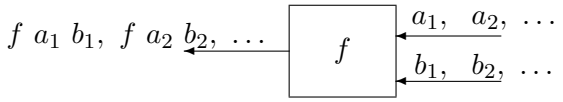


Figura 1: Red de procesos para la lista de factoriales

A modo de ejemplo sea el proceso que toma la información de dos canales de entrada (dos listas) y devuelve sobre el canal de salida (otra lista) la aplicación de cierta función f dos a dos:



A tal proceso le hacemos corresponder la función recursiva:

$$\begin{aligned} &zipWith :: (a \to b \to c) \to [a] \to [b] \to [c] \\ &zipWith\ f\ (x : xs)\ (y : ys) = f\ x\ y\ :\ zipWith\ f\ xs\ ys \\ &zipWith\ \_\ \_\ \quad\ \_\ \quad = [\,] \end{aligned}$$

El caso más interesante es cuando la salida de un proceso está conectada a una de las entradas del mismo, directamente o indirectamente a través de otro proceso intermedio. A modo de ejemplo consideremos la función HASKELL que calcula la lista *facts* de los factoriales multiplicados por un valor inicial $(i, 1! * i, 2! * i, \dots)$:

$$facts = i\ :\ zipWith\ (*)\ [1..]\ facts$$

La evaluación de la función anterior es posible representarla vía la red de procesos de la Figura 1, en la cual se ha descrito la lista $[1..]$ $(\equiv [1, 2, 3, \dots])$ como una subred. En la figura observamos que el proceso $zipWith\ (*)$ necesita evaluar el primer elemento de la lista *fact* para poder devolver a la salida el valor $1 * i$. Tal valor es proporcionado por realimentación de la red para el cálculo del siguiente: $2 * (1 * i)$ $(\equiv 2! * i)$, y así sucesivamente.

Un estudio de la potencia de la programación perezosa como herramienta puede encontrarse en [Friedman y Wise, 1976; Henderson y Morris, 1976; Field y Harrison, 1988; Hudak, 1989; Hughes, 1990], y múltiples ejemplos clásicos podemos obtener en [Ruiz Jiménez et al., 2000].

Donald [Michie, 1968] introdujo una técnica que llamó *memoization*, bajo la cual una vez que una función es evaluada (aunque sea parcialmente) su valor es conservado en memoria para usos posteriores. Esta técnica, junto con la evaluación perezosa, potencian el uso de los lenguajes funcionales en aplicaciones prácticas[37].

[37] Para una discusión puede verse [Okasaki, 1998], donde tales técnicas son utilizadas masivamente para la descripción de estructuras de datos muy eficientes.

0.1.6 Concurrencia y paralelismo en los lenguajes funcionales

La concurrencia es una abstracción en los lenguajes de programación, mientras que el paralelismo es un fenómeno físico ([Smolka, 1995]:328). Quizás [Shapiro, 1989] aclara algo más los conceptos. Así, considera los sistemas con comportamientos transformacionales como aquellos que a partir de una entrada terminan computando o devolviendo un valor a la salida, de forma que los sistemas paralelos son aquellos sistemas concurrentes con un comportamiento transformacional.

Debido a la propiedad de confluencia de los lenguajes funcionales, el paralelismo implícito es semánticamente transparente y permite la paralelización automática. Está disponible en varios sistemas, como el compilador de CLEAN de la universidad de Nijmegen o el sistema GPH (Glasgow parallel HASKELL)[38].

Según [Gelernter y Carriero, 1992], los lenguajes constan de dos piezas: el modelo de computación y el modelo de coordinación (*coordination model*), donde el segundo se encarga de enlazar las actividades o cómputos separados que puede producir un programa. No es necesario que los dos modelos describan el mismo paradigma, de forma que el modelo de computación puede ser imperativo mientras que el de coordinación puede tener una naturaleza declarativa. Un ejemplo de lenguaje de coordinación es PCN[39] [Foster y Taylor, 1992], que permite entrelazar módulos C con FORTRAN. Sin embargo, el prototipo de lenguaje de coordinación es LINDA [Gelernter, 1985; Carriero y Gelernter, 1989], cuya filosofía está basada en la comunicación a través de una estructura compartida de objetos (*tuples*)[40].

Los lenguajes concurrentes (con paralelismo explícito) contienen un modelo de coordinación sofisticado ya que proporcionan construcciones o primitivas para la creación en forma dinámica de procesos así como el intercambio de mensajes para coordinar las actividades de los procesos. Así, hablamos de añadir un lenguaje de coordinación a un lenguaje ya existente. Tales son los casos de PFL, CML, FACILE, CONCURRENT HASKELL, ERLANG, GOFFIN y EDEN.

PFL (Parallel Functional Language) [Holmstrom, 1982] es la primera extensión concurrente de ML a través de CCS. FACILE [Giacalone et al., 1989] extiende SML con procesos de orden superior basado en CCS, mientras que CML (Concurrent ML) [Reppy, 1991] extiende SML con mensajes asíncronos sobre canales tipificados y los cómputos se enlazan vía continuaciones; GOFFIN [Chakravarty et al., 1995] extiende HASKELL con restricciones y concurrencia. EDEN [Breitinger et al., 1998] añade un lenguaje de coordinación a HASKELL.

Un modelo suficientemente general es el sistema para lógica de reescritura MAUDE [Meseguer, 2000], en el que los cómputos paralelos son estructurados bajo módulos [Meseguer y Winkler, 1992; Meseguer, 1992].

0.1.7 Haskell, el estándar de la programación funcional moderna

Trataré de justificar en esta sección la elección de HASKELL como uno de los lenguajes más atractivos, tanto para la enseñanza, como para la investigación y desarrollo de la

[38] Para este sistema recientemente se ha diseñado GRANSIM, una herramienta para el estudio del comportamiento dinámico de programas HASKELL bajo el sistema GPH.

[39] Program Composition Notation

[40] Tanto éxito ha tenido tal filosofía que durante la última década se han desarrollado muchos lenguajes basados en LINDA, como C++ LINDA (1990), FORTRAN-LINDA, GLENDA, LINDALISP, LUCINDA (1991), MELINDA (1990), PROLOG-LINDA.

programación funcional moderna. Los motivos que a continuación exponemos *obligan* a su elección.

Entre los años 88 y 92, un equipo formado por importantes expertos de varias universidades tratan de reunir en un único lenguaje las principales características de la programación funcional moderna. HASKELL [Hudak et al., 1992], el resultado de este esfuerzo, es considerado hoy el lenguaje académico para la programación funcional, tanto para la enseñanza como para la investigación de otros conceptos y formalismos (concurrencia, objetos, indeterminismo, etc.). El lenguaje recibe este nombre en honor al matemático norteamericano *Haskell B. Curry*, que desarrolla durante los años 40, junto con Alonzo Church, los fundamentos de la lógica combinatoria y el λ–cálculo.

HASKELL, además de recoger las ideas mas importantes de los lenguajes funcionales modernos, amplia el sistema de tipos introduciendo un nuevo concepto (*el sistema de clases de tipos*)[41], y añade otros aspectos como *módulos* (lo que permite la definición de TADs) o entrada-salida por medio de *continuaciones* o *mónadas*.

Además de ser un estándar académico, el uso de HASKELL se está extendiendo cada vez más a la comunidad industrial, lo que ha llevado en pocos años a la definición de un estándar: HASKELL 98 [Peyton Jones y Hughes, 1999a]. Para este lenguaje se han diseñado gran cantidad de librerías [Peyton Jones y Hughes, 1999b; Day et al., 1999], y actualmente se está trabajando en su ampliación a librerías gráficas (p.e., la librería FRAN [Elliot y Hudak, 1997; Thompson, 2000a]), de cálculo numérico, de acceso a bases de datos. Incluso existe un proyecto interesante como es VISUALHASKELL[42].

Polimorfismo restringido en Haskell

En MIRANDA o en ORWELL, las funciones pueden ser *monomórficas* o *polimórficas*. En la expresión de tipo de las primeras no aparecen variables de tipo, mientras que en las segundas debe aparecer alguna variable de tipo. Ejemplos típicos son:

$$
\begin{array}{l}
(.) \quad :: (b \rightarrow c) \rightarrow (a \rightarrow b) \rightarrow (a \rightarrow c) \\
\mathit{length} :: [a] \rightarrow \mathit{Integer} \\
\\
(.)\, f\ g\ x\ = f(g\ x) \\
\\
\mathit{length}\ [\,] \quad\quad = 0 \\
\mathit{length}\ (\_ : \mathit{xs}\ ) = 1 + \mathit{length}\ \mathit{xs}
\end{array}
$$

Las funciones polimórficas son más reutilizables que las monomórficas ya que pueden actuar con cualquier tipo que se obtenga al sustituir las variables de tipo por tipos concretos. Además, la misma función puede actuar con distintos tipos en una expresión. Por ejemplo, en la expresión $\mathit{length}\ [1,2]\ +\ \mathit{length}[\mathit{True}, \mathit{False}]$, podemos entender que el mismo identificador de función es usado con distinto tipo. Lo importante es que el código de una función polimórfica no depende de las variables de tipo.

[41] A pesar de que se admite que el sistema de tipos de HASKELL es extraordinariamente complicado, recientemente han aparecido propuestas para simplificarlo. Incluso existen propuestas donde la tipificación puede *modificarse* vía HASKELL [Jones, 1999], aunque el trabajo anterior es una implementación del algoritmo de inferencia escrito en HASKELL.

[42] Ver `www.cs.uu.nl/~erik`.

Existe un tercer grupo de funciones que se encuentran a medio camino entre las funciones monomórficas y las polimórficas: las *sobrecargadas*[43]. Estas funciones tienen sentido para más de un tipo, pero no para cualquiera. Además, el código de una función sobrecargada puede depender del tipo de alguno de sus argumentos o del resultado.

Un ejemplo simple de función sobrecargada es el operador (+). En ciertos lenguajes (funcionales, lógicos e imperativos) es posible utilizarlo con *sobrecarga*: $1 + 3$, $1.2 + 4.5$. Aunque algunos lenguajes permiten esta sobrecarga, lo que normalmente no está permitido es ampliar la sobrecarga o alterarla[44], de forma que podamos escribir $1 + 1.3$, $True + False$, o también $'a' + 'b'$.

La solución adoptada por HASKELL para modelar la sobrecarga es a través del concepto de *clases de tipos*. Una *clase* es un conjunto de tipos para los que tiene sentido un conjunto de funciones. La declaración de las funciones que van a pertenecer a una clase se realiza con una *declaración de clase*, mientras que para incluir cierto tipo en una clase se utiliza una *declaración de instancia*[45].

En HASKELL existen distintas clases predefinidas en el módulo PRELUDE. Para ilustrar el mecanismo de clasificación veremos algunos ejemplos. Por ejemplo, la clase *Eq* cuyas instancias pueden compararse a través del operador de igualdad (==) o desigualdad (/ =):

```
class Eq a where
  (==) :: a → a → Bool
  (/=) :: a → a → Bool
  -- Mínimo a implementar: (==) o bien (/=)
  x == y = not (x /= y)
  x /= y = not (x == y)
```

Los operadores de la clase son (==) y (/=). El resto de declaraciones son *métodos por defecto* para evaluar un operador en función del otro. Otro ejemplo puede ser la clase de los puntos definida por:

```
class Punto p where
  origen       :: p
  coordenada_x :: p → Float
  asigna_x     :: p → Float → p
  ...
```

Para expresar que un tipo es miembro de una clase se utiliza un contexto: *Eq Bool*. *Eq* [*Integer*]. Los contextos permiten introducir restricciones en una expresión de tipo. Cada (función) miembro de una clase recibe implícitamente en su tipo el contexto apropiado. Por ejemplo, los tipos *completos* de las funciones anteriores son:

```
(==)     :: Eq a ⇒ a → a → Bool
(/=)     :: Eq a ⇒ a → a → Bool
asigna_x :: Punto a ⇒ a → Float → a
```

Los contextos pueden aparecer (y deben en algunos casos) en las expresiones de tipo. Por ejemplo, la expresión $\lambda\, q \rightarrow$ **if** $p == q$ **then** *coordenada_x p* **else** ... tiene por tipo

[43] Con más precisión deberíamos decir *sobrecargada de significado*.

[44] El lenguaje ADA sí permite alterar la sobrecarga, pero únicamente de los operadores simbólicos.

[45] Utilizaremos los términos *instanciar, instancia, etc.* – ausentes en castellano– como sinónimos de ejemplificar, ejemplar, etc.; así, una clase está formada por sus ejemplares o instancias.

$(Eq\ a, Punto\ a)\ =>\ a \rightarrow\ Float$, y resulta ser una función polimórfica restringida (aunque no sea miembro de ninguna clase).

Al instanciar un tipo para una clase es posible definir la forma especial en que se evaluará alguna de las funciones de la clase. Por ejemplo, para el tipo *Temp* (temperaturas en grados Fahrenheit o Celsius) descrito anteriormente podemos escribir la declaración de instancia siguiente:

```
instance Eq Temp where
  Far x == Far y  =  x == y
  Cel x == Cel y  =  x == y
  t     == t'     =  convTemp t == t'
```

donde vemos que a la izquierda de las ecuaciones se utiliza la igualdad (==) para el tipo Float. Al no aparecer la definición del operador (/ =) en la instancia, se utilizará el dado por defecto. Otro ejemplo puede ser:

```
data Cartesiano  =  Car Float Float
data Polar       =  RadioÁngulo Float Float


instance Punto Cartesiano where
  origen  =  Car 0.0 0.0
  coordenada_x (Car x _)    =  x
  asigna_y     (Car x _) y  =  Car x y
  ...


instance Punto Polar where
  origen  =  RadioÁngulo 0.0 pi
  coordenada_x (RadioÁngulo r theta)    =  r * cos theta
  asigna_y     (RadioÁngulo r theta) y  =  RadioÁngulo r' theta' where
    x      =  r * cos theta
    r'     =  sqrt (x^2 + y^2)
    theta' =  atan (y/x)
    ...
```

Los contextos permiten generar instancias en forma paramétrica; por ejemplo, si un tipo a dispone de la igualdad, es deseable que el tipo $[a]$ disponga también de una igualdad; para ello se declaran las listas como instancias genéricas restringidas:

```
instance Eq a  => Eq [a] where
  []     == []     =  True
  (x:xs) == (y:ys) =  x== y && xs== ys
  _      == _      =  False
```

donde observamos que el contexto *Eq a* permite el uso de la igualdad de los elementos del tipo base. Otro ejemplo puede ser

```
pinta :: Punto p  =>  p  ->  String
pinta p  =  "(" ++ coordenada_x p ++"," ++ coordenada_y p ++ ")"
```

Así mismo, los contextos pueden aparecer en los tipos de los miembros de otra clase. Esto ocurre cuando el tipo asociado a una función g de una clase B necesite el contexto de otra clase A cuando su cómputo utiliza una función f de ésta:

```
class A a where
  f :: a → Bool
class B a where
  g :: A a ⇒ a → a → a
  g x y | f x       = y
        | otherwise = x
```

En este caso el tipo de la función g será $g :: (A\ a,\ B\ a) \Rightarrow a \rightarrow a \rightarrow a$

Otro uso de los contextos es para definir subclases. Así, una declaración de clase precedida de un contexto permite utilizar los miembros de las clases del contexto en la declaración de los miembros de la clase. Por ejemplo tenemos:

```
class Punto p ⇒ PuntoColoreado p where
  color :: p → Color
```

Se dice que la clase *PuntoColoreado* es una subclase de la clase *Punto*. Siguiendo con el ejemplo de la clase B, podríamos haber incorporado el contexto $A\ a$ en el preámbulo:

```
class A a ⇒ B a where
  g :: a → a →a
  g x y | f x       = y
        | otherwise = x
```

y ahora el tipo de g es $B\ a \Rightarrow a \rightarrow a \rightarrow a$, y B es subclase de A.

Las clases se organizan en una jerarquía, y Prelude establece una jerarquía inicial aunque ésta puede ser ampliada por el programador introduciendo nuevas clases. Otro ejemplo típico de Prelude es la clase *Ord*:

```
class Eq a ⇒ Ord a where
  compare              :: a → a → Ordering
  (<), (≤), (≥), (>)   :: a → a → Bool
  max, min             :: a → a → a
  -- Mínimo a implementar: (≤) o compare
  compare x y | x== y     = EQ
              | x≤ y      = LT
              | otherwise = GT
  x ≤ y = compare x y /= GT
    ...
  max x y | x ≥ y      = x
          | otherwise  = y
  ...
```

El mecanismo de subclasificación no implica una herencia para las instancias, sino solamente introduce una noción de herencia en visibilidad, que, por otro lado, se consigue con un contexto. Por ello, el uso del término *subclase* quizás no sea apropiado, ya que no se corresponde con el uso normal en la POO. Además, las instancias no se generan automáticamente, y siempre hay que dar una declaración de instancia (aunque el cuerpo pueda ser vacío). Así, para el tipo *Temp* ya descrito, no podemos derivar una instancia por defecto de *Ord* y, al igual que con *Eq*, debemos crear la instancia manualmente:

```
instance Ord Temp where
  Far x ≤ Far y  =  x ≤ y
  Cel x ≤ Cel y  =  x ≤ y
  t1    ≤ t2     =  convTemp t1 ≤ t2
```

Ahora es posible comparar dos temperaturas, ordenar los elementos de una lista de temperaturas, construir un árbol binario de búsqueda cuyos elementos son temperaturas, etc.

De la misma forma, si tenemos:

```
data Color = Blanco | ...
data CartesianoColoreado = CC Cartesiano Color
instance PuntoColor CartesianoColoreado where
  color (CC _ c) = c
```

los datos de tipo *CartesianoColoreado* no son instancias automáticas de la clase *Punto*, y al contrario que en los lenguajes OO hay que definir las instancias convenientemente:

```
instance Punto CartesianoColoreado where
  origen = CC origen Blanco
  coordenada_x (CC p _)   = coordenada_x p
  asigna_y     (CC p c) y = CC (asigna_y p y) c
  ...
```

Obsérvese cómo se desvían los métodos a las componentes de un punto coloreado, pero en estos casos hay que hacerlo manualmente, y no existe la herencia automática de la POO.

En Haskell es posible modelar el mecanismo de clasificación y la herencia del paradigma OO. Para ello existen varias propuestas. Una posibilidad es modelar la herencia vía subtipos, como en [Odersky, 1991]. Otra muy interesante utiliza directamente el mecanismo de clases del lenguaje para implementar la herencia, haciendo corresponder una clase Haskell al interfaz de una clase OO y una o varias instancias Haskell a la implementación de cada interfaz. Este mecanismo es expuesto en [Hughes y Sparud, 1995], y ha sido utilizada para diseñar un sistema basado en objetos reactivos en un entorno concurrente [Gallardo et al., 1997]. De este modo es posible utilizar dos identificadores especiales como *self* y *super*, y es posible modelar la vinculación postergada: enviar mensajes a *self* en un método de una clase.

Hemos de resaltar que el sistema de módulos y clases de Haskell permite la creación de unidades independientes que pueden importar, exportar y ocultar selectivamente funciones y tipos. Estos mecanismos utilizados conjuntamente proporcionan la expresividad suficiente para la descripción de tipos abstractos de datos con toda la genericidad necesaria. Ejemplos de ello pueden verse en [Bird, 1998; Rabhi y Lapalme, 1999; Thompson, 1999; Ruiz Jiménez et al., 2000]. Más tarde, en la Sección **??**, daremos más detalles.

Lenguajes funcionales concurrentes basados en Haskell

Anotaciones y paralelismo explícito

Debido a la propiedad de confluencia, los lenguajes funcionales pueden utilizar directamente el paralelismo implícito. Otra opción es añadir al lenguaje anotaciones para especificar la concurrencia; trabajos pioneros en este sentido son [Burton, 1983] (anotaciones para evaluación estricta, perezosa y paralela) y la tesis de [Hughes, 1983] (anotaciones para

evaluar el cuerpo de una función antes que su argumento) y otro relativamente reciente es [Jones y Hudak, 1993]; el resultado es un lenguaje muy próximo al original de forma que no es oportuno en estos casos hablar de un lenguaje de coordinación asociado. Un ejemplo evolucionado de esta técnica aparece en [Trinder et al., 1998], donde proponen una serie de anotaciones suficientemente flexible y consistente con el lenguaje HASKELL, que finalmente es la adoptada en el GPH, y que describimos a continuación.

A partir de dos primitivas es posible describir la estrategia de evaluación de un programa. La expresión p `` `par` `` e, tiene el mismo valor que e pero el comportamiento dinámico es indicar que p puede ser evaluado en paralelo generando para ello un nuevo proceso (*sparked*) (se trata por tanto de una creación *lazy* de un proceso). Por el contrario la expresión e' `` `seq` `` e tiene el valor de e pero indica que e' debe ser reducido solamente a una forma débil–normal por la cabeza (WHNF) antes de evaluar e. Por ejemplo, el cálculo de los números de Fibonacci se puede anotar en la forma:

```
fib :: Int → Int
fib n | n ≤ 1      = 1
      | otherwise  = n1 `par` n2 `seq` 1 + n1 + n2
  where n1 = fib (n − 1)
        n2 = fib (n − 2)
```

El programador puede además utilizar distintas estrategias de evaluación. Por ejemplo,

```
type Strategy a = a → ()
rwhnf :: Strategy a
rwhnf x = x `seq` ()
```

de forma que la estrategia *rwhnf* es utilizada para evaluar el argumento a WHNF, y ésta podría ser la estrategia por defecto para cierta función que permite parametrizar la estrategia de evaluación:

```
class NFData a where
  rnf :: Strategy a
  rnf = rwhnf
```

de tal forma podemos hacer que la estrategia sea heredada a una estructura de datos vía la estrategia utilizada para los elementos básicos de la estructura:

```
instance NFData a ⇒ NFData [a] where
  rnf []     = ()
  rnf (x:xs) = rnf x `seq` rnf xs


instance (NFData a, NFData b) ⇒ NFData (a, b) where
  rnf (x, y) = rnf x `seq` rnfy
```

Así, es posible calcular un valor a partir de una estrategia:

```
using :: a → Strategy a → a
using x s = s x `seq` x
```

ya que al ser la variable x una variable compartida devolverá x evaluado según la estrategia deseada. Esto permite mejorar la expresividad, separando la estrategia, como por ejemplo en:

```
quickSort (x : xs) = i ++ x : d `using` miEstrategia
  where i = quickSort [y | y ← xs, y < x]
        d = quickSort [y | y ← xs, x ≤ y]
        miEstrategia z = rnf i `par` rnf d `par` rnf z
```

Lo más interesante de este sencillo mecanismo es que, ya que las estrategias de evaluación son funciones, es posible combinar estrategias:

```
seqList :: Strategy a → Strategy [a]
seqList s []       = ()
seqList s (x : xs) = s x `seq` (seqList s xs)


parList :: Strategy a → Strategy [a]
parList s []       = ()
parList s (x : xs) = s x `par` (parList s xs)
```

De esta forma es posible separar el algoritmo (por ejemplo, el generado por *map*) de la estrategia utilizada en su aplicación:

```
parMap :: Strategy b → (a → b) → [a] → [b]
parMap s f xs = map f xs `using` parList s
```

lo que viene a decir: evaluar *map f xs* usando la estrategia *parList s*, que será transmitida a todos los objetos de la lista.

[Trinder et al., 1998] comparan la filosofía de las anotaciones anteriores con los modelos de coordinación, como LINDA o PCN, ya que éstos proporcionan, tanto el aspecto algorítmico como la estrategia de evaluación, pero en el caso de la propuesta comentada, la estrategia de evaluación puede incorporase de forma sencilla al lenguaje.

Mónadas y procesos comunicantes. Haskell como lenguaje de coordinación

El modelo I/O basado en mónadas e implementado en HASKELL proporciona otra colección de lenguajes de coordinación. Normalmente se utilizan estructuras de datos específicas para los procesos y los canales. Por ejemplo CONCURRENT HASKELL [Peyton Jones et al., 1996], o nuestras propuestas para la incorporación del paralelismo [Gallardo et al., 1994; Gallardo et al., 1995b; Gallardo et al., 1995a; Ruiz Jiménez et al., 1996; Gallardo et al., 1996], basadas en CONCURRENT HASKELL. Otra propuesta, algo más elaborada, con herencia, objetos reactivos y selección de mensajes es ISMDCH[46] [Gallardo et al., 1997]. En ésta exponemos cómo es posible diseñar un mecanismo de sincronización sencillo y expresivo de forma que el objeto selector de un mensaje pueda *responder* en forma indeterminista a un mensaje utilizando un mecanismo llamado *lazy rendezvous* [Ruiz Jiménez et al., 1996]. Para ello se utiliza una mónada especial para encapsular objetos mutables al estilo de [Launchbury y Peyton Jones, 1995].

En la propuesta ISMDCH una interfaz es una colección de funciones o métodos, donde alguno puede venir dado por defecto. La mónada *IOC* es utilizada para capturar los cambios de estados y describir las respuestas en forma monádica y a través de acciones:

[46] Inheritance and Selective Method Dispatching in Concurrent Haskell.

```
interface ICollection where
  isEmpty, notEmpty :: IOC Bool
  isFull, notFull   :: IOC Bool
  len               :: IOC Int
  isEmpty   = self ! notEmpty >>= λ b → return (not b)
  notEmpty  = self ! isEmpty >>= λ b → return (not b)
  isFull    = return False
  ...
```

El término reservado **self** denota el receptor del mensaje, mientras que el operador (!) se usa para enviar (en forma perezosa) un mensaje a un objeto. Así, *isEmpsty* realiza las acciones: enviar (a sí mismo) el mensaje de selector *notEmpty*, y con la respuesta (b) a este mensaje devolver en forma monádica como respuesta *not b*. Obsérvese que los métodos *isEmpty* y *notEmpty* aparecen definidos por defecto uno en función del otro, mientras que *isFull* se define por defecto como una acción constante. En ISMDCH una interfaz puede extenderse, por ello, puede utilizar los métodos de la que extiende, además de definir métodos nuevos:

```
interface IBuffer a extends ICollection where
  put  :: a → IOC ()
  get  :: IOC a
  move :: Self → IOC ()


  move buff = buff ! isEmpty >>= λ b →
              if b then return()
              else buff ! get >>= λ item → (self ! put) item >>
              (self ! move) buff
```

donde **Self** denota el tipo del objeto receptor.

En ISMDCH una clase es una plantilla genérica. En esta se define la memoria privada que tendrán las instancias de la clase (objetos mutables), así como la forma en que los métodos la manipulan:

```
object class Buffer a
  use ICollection implement isEmpty, len
  use IBuffer     implement put, get
 where

  instanceVars xs :: [a]   size :: Int
  methods


  put x    = become (Buffer (xs ++ [x]) (size + 1) )
  get      = when (self ! notEmpty) >>
             let (y : ys) = xs in
             become (Buffer ys (size − 1)) >> return y
  isEmpty  = return (size == 0)
  len      = return size


newBuffer = new (Buffer [] 0)
```

La construcción **when** *msg* >> *ac* denota una forma guardada, y suspende la acción *ac* si la respuesta al mensaje *msg* es falsa. Las formas guardadas pueden componerse al estilo de Dijkstra. Por ejemplo, en la siguiente clase la respuesta al mensaje *adquire* se realiza en forma indeterminista utilizando una forma guardada:

```
interface IAgent where
  free, acquire :: Int → IOC ()
  timesEmpty :: IOC Int


object class Agent
  use IAgent   implement free, acquire, timesEmpty where


  instanceVars resources :: Int   Empty :: Int
  methods


  timesEmpty = return tEmpty
  acquire n = when (n < resources) >>
                become (Agent (resources − n) tEmpty)
          alt
              when (n == resources) >>
                become (Agent 0 (tEmpty + 1))
```

El mecanismo hereditario de ISMDCH permite que una subclase extienda otra, heredando métodos de la interfaz de la segunda o redefiniendo ciertos métodos. El mecanismo sintáctico es muy expresivo:

```
object class BoundedBuffer extends Buffer
  use IBuffer     inherit     get
                  implement put{notFull}
  use ICollection inherit     isEmpty, len
                  implement isFull


 where


  instanceVars maxLen :: Int
  methods
  isFull = self ! len >>= λ l → return(l >= maxLen)
  put n  = when (self ! notFull) >>
             (super ! put) n >>
             become (BoundedBuffer maxLen)
```

El significado de (**super** ! *put*) *n* es bien conocido en la POO: el receptor de tal mensaje es el propio objeto que está reduciendo la respuesta. La anotación *put*{*notFull*} permite introducir restricciones de sincronización para resolver los problemas derivados de la *anomalía de la herencia* [Matsuoka y Yonezawa, 1993], al estilo de [Mitchell y Wellings, 1994]. La implementación del lenguaje (ver [Gallardo et al., 1997]:sec. 4) utiliza una modificación del modelo expuesto en [Hughes y Sparud, 1995], pero adaptado a objetos mutables.

Otra extensión de HASKELL con objetos es O'HASKELL [Nordlander y Carlsson, 1997], donde extienden HASKELL con objetos reactivos, utilizando un modelo parecido al que expusimos

en [Gallardo et al., 1997], aunque desde nuestro punto de vista resulta ser menos expresivo. Sin embargo sí está implementado, e incluso existe una versión bajo HUGS, llamada O'HUGS.

Otro lenguaje de coordinación basado en HASKELL es EDEN[47], un lenguaje diseñado entre la Universidad de Marburg (Germany) y la Complutense de Madrid. EDEN usa el tipo *Process i o*, donde las variables representan las entradas y las salidas, que pueden ser canales tipificados, o estructuras de datos donde intervienen canales: *Process* [< *i* >] [< *o* >]. Un proceso es una función especial que es creada con una abstracción *process* ... **where** ...:

$$\begin{aligned} &proc :: (Int -> Process\ i\ o) -> Process\ [< i >]\ [< o >] \\ &proc\ p = process\ inp \rightarrow out \\ &\qquad \textbf{where}\ out = f\ p\ inp \end{aligned}$$

Al igual que para GPH, para EDEN se está diseñando el sistema PARADISE[48] [Hernández et al., 1999], un simulador para estudiar la evolución de programas EDEN, similar al sistema GRANSIM.

Haskell en la enseñanza de la $P_{RO}D_{EC}$

Desde el punto de vista educativo, un aspecto muy importante para la elección de un lenguaje es la existencia de intérpretes y compiladores eficientes y de libre disposición[49].

Casi simultáneamente, junto a la aparición del primer compilador eficiente y completo para HASKELL [Hall et al., 1992] desarrollado únicamente para plataformas UNIX, el primer sistema disponible para PCs surge a partir de 1992 con el desarrollo de GOFER[50] por Mark Jones en las universidades de Oxford, Yale y Nottingham. Éste ha sido el sistema utilizado en nuestra Escuela de Informática para las prácticas de laboratorio de las asignaturas de programación declarativa desde el año 1993[51]. Basado en la versión 2.30 de GOFER [Jones, 1994], desarrollamos en nuestra Escuela el primer sistema de programación para GOFER basado en Windows [Gallardo, 1994]. Dicho entorno fue sucesivamente incorporado a las distintas versiones de GOFER, y es el actualmente utilizado en las últimas versiones, como HUGS 98[52] [Jones, 1998], un sistema basado en HASKELL 98. Pensamos que el éxito de un lenguaje es el disponer de un entorno de desarrollo adecuado. Muchos lenguajes no se usan por carecer de tal entorno (p.e., λPROLOG). Por ello creemos que parte del éxito de GOFER ha sido su buen entorno.

HUGS 98 proporciona características adicionales que HASKELL 98 no presenta en su definición original, como la posibilidad de utilizar una librería de funciones gráficas, o ampliaciones del sistema de tipos para describir datos vía registros extensibles [Gaster y Jones, 1996]. No obstante, en una primera fase, para las prácticas de las asignaturas de $P_{RO}D_{EC}$ no se harán uso de estas características y nos limitaremos al uso de HASKELL 98 en su forma pura[53].

[47] `http://www.mathematik.uni-marburg.de/inf/eden/index.html`.

[48] PARAllel DIstribution Simulator for Eden.

[49] Es de suponer que todos nuestros alumnos disponen de un PC en su casa. Por ello, el aprendizaje de un lenguaje *debe* facilitarse en tres niveles fundamentales: económico, calidad y eficiencia.

[50] GOod For Equational Reasoning. Realmente, GOFER es un subconjunto extendido de HASKELL.

[51] Anteriormente se utilizó, desde el curso 1987/88, HOPE, y posteriormente MIRANDA.

[52] Haskell Users Gofer Systems.

[53] Puede verse el estado actual del lenguaje en la dirección `www.haskell.org`, donde encontraremos la documentación necesaria así como versiones de HUGS 98 para distintas plataformas.

Creo que existen motivos que aconsejan el uso de HASKELL en favor de otro lenguaje funcional muy utilizado en las universidades, como LISP, o también su derivado, SCHEME. Quizás se podría argumentar que LISP, además de ser considerado el primer lenguaje funcional, es uno de los lenguajes más desarrollados [Winston y Horn, 1981; Steele, 1984], e incluso es aún un lenguaje *vivo* desde el punto de vista educacional [Winston, 1992] y sigue utilizándose en muchas universidades pero esencialmente para aplicaciones dentro de la *Inteligencia Artificial*[54].

Sin embargo, ni LISP ni SCHEME presentan la pureza ni algunas de las ventajas de HASKELL. Entre éstas se podrían destacar la descripción de ecuaciones vía patrones a la izquierda, el polimorfismo restringido y el sistema de clases de tipos, que obligan a hacer un uso del lenguaje en forma disciplinada[55].

HASKELL es el lenguaje con más posibilidades entre los actualmente existentes, es de amplia aceptación en muchas universidades, y está siendo utilizado en la industria cada vez con más profusión. Cabe resaltar que la Universidad de Málaga es pionera en su uso para la docencia y la investigación, desde la extensa divulgación realizada por la ACM [Hudak et al., 1992; Hudak y Fasel, 1992].

Los motivos citados justifican la elección del lenguaje HASKELL, y pensamos que éste debe quedar reflejado en los contenidos de un curso de programación funcional. Esto no es una originalidad y muchos autores (que comparten nuestro criterio) van más lejos: además del uso de un determinado lenguaje para la descripción de un paradigma de la programación, el lenguaje utilizado queda reflejado en el título de muchos libros de programación funcional. Citemos por ejemplo [Bird, 1998]: *Introduction to Functional Programming using* HASKELL[56], una revisión y adaptación a HASKELL del clásico [Bird y Wadler, 1988]. O también, [Thompson, 1999], *Haskell: The Craft of Functional Programming*. Incluso nuestro texto [Ruiz Jiménez et al., 2000], *Razonando con* HASKELL. *Una Introducción a la Programación Funcional*. O el último e interesante de [Hudak, 2000], *The Haskell School of Expression. Learning Functional Programming through Multimedia*.

Haskell como primer lenguaje para la enseñanza de la programación

Como hemos discutido, la $P_{RO}F_{UN}$ está ya aceptada como disciplina propia en los currícula. Durante los 8 últimos años se viene experimentando con HASKELL como primer contacto con la programación[57]. En la Tabla 1 podemos ver algunas universidades que lo llevan a

[54] Las razones que exponen los defensores de LISP no están suficientemente argumentadas. Por ejemplo, el amplio desarrollo de aplicaciones industriales en HASKELL (o en lenguajes *modernos*) en los últimos años es comparable en calidad y en cantidad a las aplicaciones actuales realizadas en LISP.

[55] Se suele argumentar que un serio inconveniente de LISP es la *engorrosa* notación que contiene un excesivo uso de paréntesis. Este inconveniente puede evitarse con el uso de entornos adecuados, como el ya citado DRSCHEME. Sin embargo, el principal inconveniente de LISP es la presencia de la tipificación dinámica, que da demasiada libertad al programador. Ésta libertad facilita su uso en aplicaciones de Inteligencia Artificial – o al menos esto sostienen sus defensores, entre los que se encuentra el equipo de compañeros de nuestro Departamento que imparte tales asignaturas – pero facilita una programación *no disciplinada*.

[56] Traducido recientemente al castellano por la misma editorial (1999).

[57] Similares experiencias han sido ya realizadas con SML (véase por ejemplo [Cousineau, 1997]), con MIRANDA (véase el monográfico [JFP, 1993]), y con SCHEME (véase [Aerts y de Vlaminck, 1999; Lapidt et al., 1999]). En la universidad de Twente (Holanda) se introdujo en 1986 el lenguaje MIRANDA como primer lenguaje [Joosten y van den Bergand G. van der Hoeven, 1993] en los estudios de CC; los autores

cabo[58]. Además, en todas ellas se enseña programación con únicamente unos conocimientos de matemáticas discretas elementales y sin otros prerrequisitos.

[Giegerich et al., 1999; Page, 1999] defienden el uso de HASKELL como primer lenguaje, ya que la programación funcional está íntimamente relacionada con la esencia de la computación y permite desarrollar los fundamentos de forma lógica, rigurosa y *excitante*. Propone modelos simples para la Música y la Genética Molecular. Un argumento parecido exponen [Jeuring y Swierstra, 1999], de la Universidad de Utrecht, donde HASKELL es utilizado para ilustrar e implementar los conceptos de la programación, enfatizando la idoneidad de la evaluación perezosa y de la *programación genérica* [Backhouse et al., 1999] obtenida vía funciones de orden superior, tales como los plegados (*fold*)[59].

Otro defensor de HASKELL como primer lenguaje es Paul Hudak[60], que en numerosos trabajos ha expuesto su visión de la enseñanza de la $\mathrm{F}_{\mathrm{UN}}^{\mathrm{RO}}$ enfatizando una orientación alejada de las típicas aplicaciones a la matemática. Un compendio importante de su aproximación personal a la introducción de HASKELL como primer lenguaje es el reciente texto publicado como [Hudak, 2000], donde podemos encontrar aplicaciones llamativas y actuales orientadas a multimedia: gráficos y música computacional[61].

En [Page, 1999] podemos encontrar otro estudio de experiencias con HASKELL como primer lenguaje, donde realiza una comparación con otros lenguajes: SCHEME, SML, etc., o incluso JAVA (aunque limita la exposición de las experiencias fundamentalmente a los Estados Unidos, donde prima el uso de SCHEME y SML). Además de esta comparación, el autor analiza el impacto de la $\mathrm{F}_{\mathrm{UN}}^{\mathrm{RO}}$ como primer paradigma a estudiar, criticando algunas ideas de Alan Tucker (quizás admitidas *alegremente* por una amplia audiencia), en lo

relatan una experiencia describiendo en un grupo la introducción a la programación en MODULA2, y en otro grupo vía MIRANDA; a través de varias prueban observaron la mejor preparación de los alumnos del segundo grupo. [Lambert et al., 1993], en la misma línea que los anteriores ha sido más atrevidos; relatan su experiencia con el uso de MIRANDA como primer lenguaje en las universidades de New South Wales y Queensland (Australia) desde 1989. El éxito condujo a utilizar el mismo lenguaje en un segundo curso (TADs), y en un tercer curso (Programación Concurrente), donde se implementó un subconjunto de CSP en MIRANDA. Es obvio que tales experiencias podrían haberse llevado a cabo en forma similar con HASKELL, ya que MIRANDA es un subconjunto de HASKELL. [Aerts y de Vlaminck, 1999] relata una experiencia sobre la enseñanza de la programación a estudiantes de ciencias aplicadas (no informáticos) vía SCHEME; en esta experiencia enfatiza la importancia del papel de los *juegos* en la enseñanza ya que motiva la introducción de conceptos (como los TADs) de forma natural. En [Lapidt et al., 1999] se relata una experiencia con DRSCHEME en Israel, y en la educación secundaria. [Findler et al., 1997] expone que en USA, durante 1995, los lenguajes mas utilizados en cursos introductorios son: PASCAL (35%), ADA (17%), C/C++ (17%) y SCHEME (11%), mientras que SML es solo utilizado en un 2%. A pesar de todo, estudios recientes (ver `www.haskell.org`) muestran que en Europa los lenguajes funcionales más extendidos son SML, MIRANDA y HASKELL.

[58] Actualmente (finales de 2000) todos estos centros usan HASKELL 98 o HUGS 98, y los textos utilizados son los actualizados a estos lenguajes [Thompson, 1999; Bird, 1998].

[59] No debe confundirse el concepto introducido por [Backhouse et al., 1999] con el utilizado por otros autores. En la literatura podemos encontrar los términos *generic functional programming*, *structural polymorphism*, *type parametric*, *shape polymorphism*, etc. Todos ellos tratan sobre la posibilidad de ampliar el concepto de clase de tipos para resolver problemas típicos como la generación de instancias en forma automática a partir de tipos de datos con constructores de primer orden . En este sentido, y orientado a una futura extensión de HASKELL, aparecen los trabajos de [Hinze, 1999; Hinze, 2000].

[60] Profesor de la Universidad de Yale, propulsor del lenguaje HASKELL desde hace más de 15 años, y autor de más de 100 trabajos sobre diseño y aplicaciones de los lenguaje de programación.

[61] Aunque la intención original de Hudak es dar una introducción comprensible sin la necesidad de herramientas matemáticas, el resultado final de su texto, como él mismo reconoce, es muy diferente. Así, propone temas como *un álgebra musical* (cap. 21), o también *música fractal* (cap. 20).

Universidad	Curso	Lenguaje	Textos
Kent University (UK)	Introductory programming	HUGS	[Thompson, 1996]
Utrecht University (The Netherlands)	Programming I	HUGS	www.cs.uu.nl/docs/vakken/go
Brisbane Queensland (Australia)	Programming I	HUGS, GHC	[Thompson, 1999]
Oxford University Computing Laboratory	Functional Programming	GOFER	www.comlab.ox.ac.uk /igdp/text/course06.html
Nottingham	Functional Programming	HUGS 1.3	[Bird, 1998] y transparencias: www.cs.nott.ac.uk /Modules/9899/G51FUN.html
UNC Chapel Hill	Introduction to Functional Programming	HUGS 1.4, lib. Fran y Haskore	[Thompson, 1996] y transparencias: www.cs.unc.edu /∼prins/Classes/15/
Gothenburg y Chalmers	Programmering för Naturvetare	HUGS 1.4	[Thompson, 1996], www.md.chalmers.se /Cs/Grundutb/Kurser/nptah/
Hull (UK)	Functional and Logic Programming	HUGS 1.4	[Thompson, 1996]
Konstanz (Germany)	Deklarative Programmierung	HUGS 1.4	[Bird y Wadler, 1988; Peyton Jones, 1987] y transparencias www.fmi.uni-konstanz.de /dbis/Courses-old /Courses-ss98 /decl-ss98.html
New South Wales	Computing 1A	HUGS 1.4	[Thompson, 1996], `www.cse.unsw.edu.au /∼cs1011`
Bielefeld, Bonn (Germany)	Introduction to Computer Science	HUGS	class notes by Robert Giegerich
Málaga	Introducción a la Informática (Licenciatura de Matemáticas)	GOFER	[Ruiz Jiménez et al., 1995] y notas de clase

Tabla 1: HASKELL como primer lenguaje (setiembre de 1999)

referido a la introducción de la recursión en los primeros cursos. Recordemos que, mientras algunos autores ([Davie, 1992; Thompson, 1999]) sugieren la introducción de la recursión en forma *explícita* con *abundantes*ejemplos, otros ([Bird y Wadler, 1988; Bird, 1998]) aconsejan capturarla en un *pequeño* conjunto de funciones de orden superior (recursores, plegados, iteradores, etc.), con el objetivo esencial de enfatizar el pragmatismo de la $P_{RO}F_{UN}$; esto es: $P_{RO}F_{UN} \equiv$ composición de funciones. Este punto de vista es el adoptado fundamentalmente en nuestro texto [Ruiz Jiménez et al., 2000].

Futuras extensiones de Haskell 98: Haskell 2

HASKELL 98 es un lenguaje en continuo desarrollo. En el último Workshop de 1999 se propuso la necesidad de desarrollar un sucesor[62]: HASKELL 2. Entre las diferentes ex-

[62]En `www.pms.informatik.uni-muenchen.de/forschung/haskell-wish-list/` pueden verse las últimas

tensiones aparecen: clases multiparamétricas, generalización del concepto de guardas (por Simon Peyton Jones), generalización de las clases numéricas para incorporar álgebras arbitrarias, y objetos. Al respecto de esta última extensión cabe citar O'Haskell [Nordlander y Carlsson, 1997], desarrollado en la Universidad de Chalmers, que extiende Haskell con objetos y otras posibilidades, como una mónada para encapsular objetos reactivos en un entorno concurrente, en forma parecida a como hemos propuesto en [Gallardo et al., 1997]. Existe una versión para Hugs, llamada O'Hugs.

Además de los *registros extensibles* de [Gaster y Jones, 1996], ya incorporados a Hugs 98, se está trabajando en un nueva propuesta para añadir al futuro Haskell 2 la posibilidad de *registros extensibles con variantes* [Jones y Peyton Jones, 1999]. Tales extensiones permitirían la construcción de herramientas en Haskell que permitan la comunicación con sistemas *proof checker*, del estilo de Coq o Lego, o incluso que permitan escribir en Haskell teorías matemáticas. Por ejemplo, en [Pollack, 2000] se dan ideas de como modelar productos y sumas dependientes. También puede estudiarse en [Augustsson y Carlsson, 2000] un intérprete *bien–tipificado* y descrito en Cayenne [Augustsson, 1999], una extensión de Haskell con tipos dependientes. Otras trabajos pueden encontrarse en el *Workshops on Dependent Types in Programming* (Marzo, 1999).

Programación distribuida y servidores Web

Sobre aplicaciones de Haskell a la Programación distribuida y la Web[63] existen numerosos trabajos. Basado en el modelo estándar de Microsoft COM (*Component Object Model*) [Rogerson, 1997], podemos encontrar varias propuestas para integrar componentes COM en Haskell, como por ejemplo [Peyton Jones et al., 1998; Leijhen, 1998][64]. Sobre el uso de tales componentes para escribir aplicaciones híbridas con Haskell, Java o VisualBasic, puede verse [Leijen et al., 1999], donde utiliza un lenguaje que está convirtiéndose en el estándar para estas aplicaciones: HaskellScript (expone ejemplos vía *Microsoft Internet Explorer*).

Citemos otros trabajos recientes con aplicaciones de Haskell a servidores Web e interconexión con Java.

Sigbjorn Finne y Erik Meijer han diseñado el sistema Lambada, que permite integrar programas Haskell con programas Java a través de JNI y H/Direct[65].

En [Meijer, 2000] se propone la librería Haskell/CGI (*Common Gateway Interface*) para generar documentos dinámicos sobre servidores Web a través de una serie de combinadores generadores de HTML. Una función *wrapper* se encarga de trasladar las peticiones vía un script CGI de bajo nivel. En [Meijer y van Velzen, 2000] se propone el sistema HSP (*Haskell Server Pages*) que permite generar páginas dinámicamente, de forma que el código Haskell es incorporado directamente dentro de fragmentos XML. Existe un prototipo para Apache sobre Linux[66].

Por otro lado, [Wakeling, 1999] propone un modelo suficientemente general para compilar programas funcionales sobre la máquina virtual de Java, en el cual la eficiencia es del

propuestas para el futuro Haskell.

[63] Ver `www.haskell.org/active/activehaskell.html`.

[64] Haskell no es el único lenguaje funcional utilizado para aplicaciones distribuidas. Por ejemplo, [Hayden, 2000] presenta una experiencia utilizando Objective CaML, un dialecto de ML.

[65] La situación actual del sistema puede verse en `http://www.cs.uu.nl/~erik/`.

[66] `www.microsoft-lab.org/HSP`.

mismo orden que los modelos tradicionales basados en intérpretes de *bytecode*.

Otra propuesta de lenguaje para escritura de guiones (*scripting*) es MONDRIAN, propuesto recientemente (Octubre de 2000) por Erik Meijer y Arjan van Ijzendoorn, de la Universidad de Amsterdam. Se trata de un subconjunto extendido de HASKELL, disponible como una extensión de HUGS98, y que usa JAVA como *target*. Así, MONDRIAN es visto como un lenguaje de dominio específico empotrado en JAVA, de tal forma que es posible escribir en JAVA código MONDRIAN a mano. Ejemplos de uso y un tutorial puede verse en `www.cs.uu.nl/~erik/`.

0.1.8 Los lenguajes funcionales en la industria

Quizás uno de los argumentos que suelen utilizarse para justificar el poco uso de los lenguajes funcionales en la industria *software* sea su falta de eficiencia. Sin embargo, el código generado por los compiladores actuales es en términos comparativos parecido al código generado por los compiladores de otros lenguajes imperativos. Además, los compiladores de lenguajes funcionales suelen tener un código fuente más compacto ya que (como es habitual en otros lenguajes) los compiladores para lenguajes funcionales son escritos utilizando un subconjunto del lenguaje a compilar. Así, el compilador de SML tiene 130K líneas de código SML, mientras que el compilador de HASKELL escrito en la Universidad de Glasgow tiene 90K de código HASKELL, y algo parecido ocurre para SCHEME y ERLANG.

Muchos observadores han resaltado las numerosas aplicaciones industriales de los lenguajes funcionales[67] [Hudak y Jones, 1994; Odersky y Wadler, 1998; Wadler, 1999], así como las interesantes ideas que aportan al desarrollo de otros lenguajes. Por ejemplo, los conceptos de genericidad y clases de tipos de HASKELL están siendo incorporadas a JAVA [Odersky y Wadler, 1997].

Como es de suponer, una línea de investigación reciente es la aplicación de los lenguajes funcionales al desarrollo de servidores Web e Internet. Así, ciertas componentes de servidores HTTP, o TCP/IP han sido escritos en una variante de SML. También, como hemos comentado, HASKELL se está utilizando para el diseño de guiones o plantillas (*script*) [Meijer, 2000; Meijer et al., 1999]. Otro ejemplo es ERLANG (ERicsson LANGuage) [Armstrong et al., 1996; Armstrong, 1997], un lenguaje funcional que está siendo utilizado en las mismas aplicaciones que ha sido utilizado JAVA o C++. ERLANG comparte características con JAVA: mecanismo de recolección de basura (que asegura una ejecución segura), librerías independientes del sistema operativo, se compila sobre una máquina virtual (asegurando la transportabilidad sobre distintas arquitecturas), etc.

Otro aspecto interesante de los lenguajes funcionales es su uso para la descripción de prototipos, que permite explorar rápidamente varias fases del ciclo de vida *software*. Así, en un experimento realizado por el *Naval Surface Warfare Center* en 1993 para el desarrollo de un prototipo de *geo–server* [Hudak y Jones, 1994] (que básicamente consistía en computar vía patrones las relaciones entre objetos móviles: barcos, aviones, etc.) participaron distintas universidades y expertos en programación, que desarrollaron su prototipo en distintos lenguajes: HASKELL, ADA, ADA9X, C++, LISP y otros lenguajes evolucionados utilizados para simulación. El resultado fue espectacular. Mientras las versiones descritas en ADA y C++ contenían por encima de 800 líneas de código, y emplearon por

[67] Ver `www.cs.bell.labs.com/~wadler/realword`.

encima de 10 horas en el desarrollo, las versiones funcionales eran más rápidas y sencillas. En el caso de HASKELL se emplearon 85 líneas de código[68] y 10 horas de desarrollo, mientras que para la solución LISP se utilizaron 280 líneas de código y 3 horas.

La construcción de prototipos cercanos a la solución final es también enfatizada por diferentes autores. Por ejemplo, [Peterson et al., 1999] describen el sistema FROB (*F*unctional *ROB*otics), un lenguaje orientado al dominio[69] que resulta ser un subconjunto de HASKELL, y permite la programación de aplicaciones en robótica independientes del sistema *hardware*; el resultado es un sistema para el prototipado rápido, tan eficiente como las clásicas aplicaciones en C/C++, y extraordinariamente fácil de programar y mantener.

Otras aplicaciones interesantes de los lenguajes funcionales son en conexión con bases de datos. Por ejemplo, CLP (Collection Programming Language) es un lenguaje funcional de orden superior cercano a SQL, y que comparte ciertas características con otros lenguajes funcionales modernos, como estructuras por comprensión, subtipos y registros. Para tal lenguaje se ha desarrollado el sistema KLEISLI [Wong, 2000] (un interfaz entre el lenguaje CLP y SQL), que ha sido utilizado en el proyecto Cromosoma 22, y está siendo utilizado en proyectos similares. No menos sorprendente es que el sistema KLEISLI está escrito en SML. Un sistema parecido se está desarrollando para HASKELL. Así, [Leijhen y Meijer, 2000] proponen un modelo para expresar las abstracciones de los lenguajes con dominios específicos (*Domain–specific languages*) utilizando lenguajes de orden superior. Como ejemplo desarrolla en HASKELL un interfaz con SQL utilizando el modelo, aunque también cita otros proyectos similares para manejadores de correo[70].

Un grupo notablemente importante de aplicaciones es el cálculo simbólico. Además del muy conocido MATHEMATICA, podemos añadir XFUN [Dalmas, 1992], un lenguaje fuertemente tipificado diseñado para álgebra computacional, que está basado en SML y RUSSELL.

Citemos finalmente otro grupo de aplicaciones de los lenguajes funcionales al desarrollo de sistemas de demostración asistida y demostradores de teoremas. Por ejemplo, SML ha sido utilizado para desarrollar los sistemas ISABELLE[71] y HOL. Son conocidas las amplias aplicaciones de estos dos demostradores de teoremas [Barendregt y Geuvers, 1999]. Recientemente, estos sistemas han sido utilizados para modelar y verificar los sistemas multiprocesador HP 9000, o los señuelos para misiles del departamento de defensa australiano. Señalemos otros demostradores y sus implementaciones: COQ ha sido implementado en CAML; VERITAS en MIRANDA; YARROW en HASKELL; ALF, ELF y LEGO en SML. Como vemos, salvo algunos sistemas implementados en LISP, casi todos los demostradores están escritos en "verdaderos"(modernos) lenguajes funcionales.

Las anteriores son solamente algunas de las aplicaciones[72]. Sin embargo, salvo las aplicaciones a la inteligencia artificial, y básicamente al desarrollo del subárea de los sistemas expertos, que históricamente se han desarrollado en LISP, no son demasiadas las aplicaciones *reales*, y podemos plantearnos la siguiente pregunta:

¿Cuales son las razones del poco uso industrial de los lenguajes funcionales?

[Wadler, 1999] cita varias razones, entre las cuales podemos destacar:

[68] De las cuales, 20 eran para describir el código I/O, y 29 para describir tipos.
[69] Domain-specific language.
[70] `www.dimac.net/`.
[71] Ver `www.cl.cam.ac.uk/Research/HVG/Isabelle`.
[72] Otras aplicaciones industriales son analizadas en [Wadler, 1999].

Compatibilidad	La industria tiende a enfatizar los estándares.
Transportabilidad	Muchos proyectos usan C por su *transportabilidad* y no necesariamente por su eficiencia.

Los problemas debidos a la compatibilidad tienden hoy en día a resolverse. Así, como hemos comentado, existen varios proyectos que usan HASKELL o SML como *scripting language* para navegadores web, o para componentes CORBA o COM. La segunda razón es todavía más absurda ya que las máquinas abstractas para HASKELL o JAVA son tan populares que están escritas en C.

Otras razones que se suelen alegar sobre el poco uso de los lenguajes funcionales son: (1) la poca disponibilidad de compiladores, (2) la falta de costumbre, y (3) la poca popularidad. La primera razón es sencillamente falsa. Las dos siguientes son importantes y considero que el marco idóneo para resolverlas es comenzando a divulgar, estudiar, investigar, enseñar, etc. los lenguajes funcionales en las universidades. Como ya hemos dicho, aunque al menos en nuestro país la enseñanza de la programación declarativa en las diferentes universidades es ya una realidad, hay que hacer un esfuerzo algo mayor, sobre todo en la preparación de los futuros ingenieros en informática.

0.2 Programación Lógica

0.2.1 Orígenes de la Programación Lógica y lenguajes

> Para los no iniciados, los orígenes y relaciones entre los lenguajes para la programación lógica pueden aparecer con un misterio similar a las religiones Judeo-Cristianas; ambos tienen sus profetas, herejes y cismas. [Ringwood, 1988]:23

La $P_{RO}L_{OG}$ está basada en la idea de que la computación puede ser vista como un proceso deductivo controlado, siendo una secuela de la investigación en demostración automática de teoremas, atribuida principalmente a Robinson.

La $P_{RO}L_{OG}$ tiene su origen durante el desarrollo de la lógica matemática. Está basada en la sintaxis de la lógica de primer orden, propuesta originalmente por Gottlob Frege sobre los años 1850, y modificada hasta su forma actual por Giuseppe Peano y Bertrand Russell. En 1930, Kurt Gödel y Jacques Herbrand introducen la noción de computabilidad basada en derivaciones. Herbrand aborda en su tesis doctoral un conjunto de reglas para la manipulación de ecuaciones algebraicas que puede verse como el esqueleto de un algoritmo de unificación. Estos hechos son admitidos por todos los computacionistas. Sin embargo, a partir del desarrollo de las ideas de Herbrand, la historia de la $P_{RO}L_{OG}$ parece entrar en un mar de confusiones, y muchos autores quieren atribuirse ciertos méritos[73].

El comienzo de la $P_{RO}L_{OG}$ moderna, visto por Alain Robinson[74] es el siguiente. [McCarthy, 1958b; McCarthy, 1958a] propone una estrategia (para aplicar en inteligencia artificial) basada

[73] Es curioso que muchos autores no enfatizan la importancia de las contribuciones de Frege, Peano o Russell [1872-1970] a la actual $P_{RO}L_{OG}$, aunque sí enfatizan los trabajos de otros matemáticos del siglo XX, como el matemático alemán Gerhard Gentzen [09-45], o de Ronald Harrop [26-60], o del propio Jacques Herbrand [08-31] (desgraciadamente, todos fallecidos en edades muy tempranas, sobre todo el caso de Herbrand, muy poco después de la lectura de su tesis doctoral). La principal contribución de Frege es la simplificación de la sintaxis (que no es poco), y la de Herbrand, la simplificación semántica: una sucesión creciente de interpretaciones ..., que además dará lugar al principio de resolución de Robinson.

[74] El siguiente punto de vista, que muchos textos citan sin referir, lo he extraído del prólogo escrito por J.A. Robinson para el libro de [Deville, 1990], páginas v-vii.

en una colección de asertos que expresan el conocimiento relevante de algún problema expresado en la lógica de primer orden (*base de conocimiento*), añadiendo a éste un *procedimiento de demostración* eficiente. Por otro lado, Marvin Minsky opina que es mejor basar el conocimiento en una colección de procedimientos. Estas diferencias motivaron dos corrientes de opinión enfrentadas: ¿el conocimiento es declarativo o procedural?.

Casi simultáneamente, aparece el estudio de técnicas para la demostración automática de teoremas tales como los trabajos desarrollados por Alain Robinson y Donald Loveland a mediados de los años 60; estos son el principio de resolución, la noción de unificación y el algoritmo de unificación de [Robinson, 1965], así como el principio de eliminación [Loveland, 1969][75]. [Green, 1969], en la conferencia sobre IA (Mayo de 1969) propone dos sistemas (QA2 y QA3) basados en el principio de resolución, lo que fortalecía la idea de McCarthy; simultáneamente, Carl Hewitt mantenía la posición de Minsky describiendo el sistema PLANNER.

A principios de la década de los 70 seguirá la controversia. El estilo procedural seguirá defendido por Marvin Minsky, John McCarthy y Seymout Papert, todos en el MIT.

A través del principio de resolución es posible probar teoremas de la lógica de primer orden, pero es necesario mecanizarlos. Donald Kuehner y Rober Kowalski muestran que el modelo de eliminación y una forma especial de resolución (lineal) pueden sintetizarse en un método común llamado SLD–resolución [Kowalski y Kuehner, 1970], estableciéndose el embrión de un lenguaje. Esta forma de resolución es mas restrictiva que la propuesta por Robinson ya que solamente se permiten reglas con limitaciones sintácticas.

A partir de 1972, y durante un periodo de colaboración entre Alain Colmerauer (en la Universidad de Marsella) y Robert Kowalski (en la Universidad de Edimburgo) se producen los desarrollos más importantes. Colmerauer y sus colaboradores desarrollan *Un système de communication homme-machine en Français* [Colmerauer et al., 1973] capaz de realizar deducciones desde un conjunto de sentencias escritas en lenguaje natural; tal sistema es implementado por Philippe Roussel (alumno de Colmerauer) en 1972 en forma de intérprete escrito en ALGOL W, y en él aparecen ya ciertas características comunes de PROLOG: uso de anotaciones para reducir el espacio de búsqueda (precursor del corte) y la negación por fallo.

La implementación posterior de tal intérprete en FORTRAN y la disponibilidad de un compilador desarrollado por David Warren en 1977 ([Warren, 1983]) para un PDP-10, que además podía competir en eficiencia con programas LISP, ayudó a la difusión del lenguaje.

Robert [Kowalski, 1974] establece las cláusulas de Horn como base para la programación lógica y posteriormente, [Kowalski, 1979b], junto con van Emden, define la semántica de los programas basados en cláusulas de Horn[76]. De esta forma se obtiene un punto de vista más general: estrategias de búsqueda ascendente y/o descendente, e interpretación procedural indeterminista y potencialmente concurrente[77]. De esta forma Kowalski vino a

[75] Puede verse unas notas sobre los orígenes de la PROLOG en [Robinson, 1983] y en [Kowalski, 1985], y un estudio detallado en [Cohen, 1988]. Bibliografía complementaria se encuentra en [Ringwood, 1988].

[76] [Andréka y Németi, 1976] prueban que el fragmento de la lógica basada en cláusulas de Horn es completo. Incluso, puede limitarse la forma de las cláusulas a cláusulas unitarias con cuerpos atómicos [Devienne et al., 1996]. Además, toda teoría de Horn admite un modelo mínimo de Herbrand, de donde podemos obtener una semántica declarativa de PROLOG, o como mejor dicen algunos autores, una *semántica denotacional*.

[77] Puede verse un estudio elemental pero muy clarificador que relaciona la PROLOG basada en cláusulas de Horn con PROLOG en [Davis, 1985]. Desgraciadamente, tanto el PROLOG de Edimburgo como el de Marsella son deterministas y con motor de inferencia hacia atrás.

reconciliar las ideas iniciales de McCarthy y de Minsk, mostrando que ambos tenían parte de razón, presentando la doble interpretación que se puede dar a un aserto: declarativa y procedimental. Así mismo mostró que esta última la hace efectiva como lenguaje de programación.

Robert [Kowalski, 1979a] indica la relación importante entre especificación de un problema en términos de predicados y la $\mathrm{P_{RO}L_{OG}}$ a través de su famosa ecuación $A = L + C$:

$$\text{Algoritmo} = \text{Lógica} + \text{Control}$$

dando a entender la posibilidad de aislar la estrategia de solución de la especificación.

Hoy día la $\mathrm{P_{RO}L_{OG}}$ es un paradigma suficientemente consensuado en sus elementos básicos dentro de las diferentes metodologías de programación existentes. Sus atractivos son evidentes, y cualquier programador que se inicia en ésta encuentra, superadas las primeras dificultades de adaptación, que el nuevo estilo resulta natural y flexible. La ecuación $A = L + C$ muestra uno de los aspectos que hace a la $\mathrm{P_{RO}L_{OG}}$ más atractiva: la clara separación entre el conocimiento que se requiere para resolver un problema (lógica), y la forma en que ese conocimiento debe ser aplicado para resolverlo (control). De esta forma, un lenguaje lógico, invita (aunque no exige) al programador a olvidar el control de ejecución para concentrarse tan solo en la lógica subyacente al problema a resolver. Es responsabilidad del sistema proporcionar una estrategia de control por defecto que explore de forma sistemática los asertos especificados por el programador, siguiendo un orden uniforme. Por otro lado, el conocimiento de esta estrategia de control por parte del programador, le permite escribir programas eficientes.

Los programas lógicos descritos con cláusulas de Horn maximizan la legibilidad a expensas de la eficiencia. Por ello, los sistemas para la $\mathrm{P_{RO}L_{OG}}$ suelen incorporar mecanismos para incrementar tal eficiencia. Explotando adecuadamente estos mecanismos se pueden diseñar programas que, manteniendo las ventajas que ofrece el estilo declarativo, no supongan un coste computacional excesivo. No obstante, es necesario cuidar la forma en que estas posibilidades, ajenas a la lógica, son utilizadas, porque pueden degenerar fácilmente en programas ininteligibles y con un comportamiento que no es el deseado inicialmente. Por lo tanto, es preciso establecer una metodología de programación que regule la utilización de los recursos que ofrecen estos sistemas.

Como dice [Apt et al., 1999] en el prefacio, la programación lógica es un paradigma atípico de las Ciencias de la Computación en el cual se cruzan áreas autónomas. Su uso puede encontrarse en:

- Bases de Datos, con el modelo datalog (Logic as Data Model) [Ullman, 1988]:cap.3,
- Compiladores y escritura de compiladores, como propone [Wilhelm y Maurer, 1995],
- Inteligencia Artificial [Shoham, 1994],
- Procesamiento del Lenguaje Natural [Gazdar y Mellish, 1989], y
- Enseñanza Asistida [Mitchell, 1997; Dahl, 1999].

Hoy en día el número de aplicaciones industriales continúa en aumento, potenciadas por el uso de otros paradigmas, como la Programación Lógica con Restricciones[78] (CLP), y la Programación Lógica Inductiva [Raedt, 1999][79].

[78] Muchos autores sostienen que el nacimiento de la CLP surge para cubrir las deficiencias de PROLOG.

[79] Ver el proyecto (del Computing Laboratory de la Oxford University) *Machine Learning at the Computing Laboratory* en `http://web.comlab.ox.ac.uk/oucl/research/areas/machlearn/`.

0.2.2 Prolog y Programación Lógica

Hay cierto misticismo en considerar a PROLOG un lenguaje lógico, y no solamente por el determinismo implícito; PROLOG es una manifestación de la $P_{RO}L_{OG}$, una aproximación; en PROLOG la famosa ecuación $A = L + C$ es violada por el uso del corte[80]. [Robinson, 1983] ya avisaba del profundo peligro en tal confusión:

> Tengo miedo en la rápida difusión de PROLOG ... que puede tener aspectos desagradables ... lamento, al igual que lo hacía Dijkstra con la sentencia *goto*, que disponga del predicado corte (*cut*).

Aún así, PROLOG representa el avance más claro del término $P_{RO}L_{OG}$ durante los primeros años de la década de los ochenta, como muestran los siguientes hechos. Por un lado, el programa japonés *ordenadores de la quinta generación*[81] oferta un importante papel a PROLOG; aparecen a su vez potentes compiladores y versiones de PROLOG que conducen a una rápida difusión del paradigma $P_{RO}L_{OG}$, pensando seriamente si es el lenguaje adecuado para los problemas de IA. D. Borrow, en un artículo titulado *If Prolog is the Answer, What is the Question?, or What if Take to Support AI Programming Paradigms* [Bobrow, 1985] defiende que el paradigma $P_{RO}L_{OG}$ es compatible con otros, como la programación orientada al objeto, a la regla y al acceso.

Hoy en día hay muchísimas versiones de PROLOG, casi todas ellas derivadas de los PROLOG de Edimburgo y de Marsella, por lo que las diferencias fundamentales entre ellos son sintácticas y muy pequeñas, y sobre todo relativas a la escritura de listas:

```
"Marsella"                          /* Edimburgo */
sublista(nil, v) ←;                 sublista([ ], V).
sublista(x.l, v) ←                  sublista([X|L], V) :-
    miembro(x, v)                       miembro(X, V),
    sublista(l, v);                     sublista(L, V).

miembro(x, x.l) ← /;                miembro(X, [X|_]) :- !.
miembro(x, y.l) ←                   miembro(X, [_|L]) :-
    miembro(x, l);                          miembro(X, L).
```

Otra notación a destacar es MICRO-PROLOG, desarrollada por F. McCabe y K. Clark con vista a un uso educacional; existe una versión de esta notación preparada por Hugh de [Saram, 1985] y supervisada por Robert Kowalski que ha sido experimentada con gran éxito en ambientes no universitarios; utiliza una notación infija tal como:

```
( x | l ) es_sublista_de w si
    x es_elemento_de w y
    l es_sublista_de w

x es_elemento_de ( x | l)
x es_elemento_de ( y |l) si
    x es_elemento_de l
```

[80] Puede verse un estudio amplio de los predicados de control de PROLOG en [Naish, 1986].

[81] Sobre el proyecto *The Japanese Fifth Generating Computer systems* (FGCS) desarrollado en el *Institute for New Generation Computer Technology* (ICOT) de Tokyo, puede verse [Robinson, 1984; Meyrowitz, 1986; ACM, 1993]. Trata sobre el uso en los 90 de máquinas especializadas para la ingeniería del conocimiento y sus aplicaciones en IA.

Colmerauer propondrá rápidamente la extensión de PROLOG vía otros dominios de términos con ciertas relaciones. Así, importantes derivados del PROLOG de Marsella son PROLOG II [Colmerauer et al., 1982; Giannesini et al., 1985], con operaciones sobre los racionales; PROLOG III [Colmerauer, 1987; Colmerauer, 1990], con restricciones sobre racionales y booleanos; PROLOG IV [Benhamou y Touraïvane, 1995], con restricciones sobre intervalos. Sin embargo, las implementaciones más difundidas para ordenadores personales utilizan la notación de Edimburgo, como TURBO-PROLOG [Borland, 1986; Borland, 1988] y ARITY-PROLOG [Arity Corp., 1986b; Arity Corp., 1986a], que han sido utilizadas vastamente por distintas universidades. Ésta última además permite definir precedencia de operadores y evaluación simétrica, al estilo de [Clocksin y Mellish, 1981]. Posteriormente, las versiones de Edimburgo han evolucionado hasta las dos más difundidas hoy día: QUINTUS-PROLOG [Quintus, 1997] y SICSTUS-PROLOG [Carlsson y Widen, 1988; ISL, 1997].

Una extensión de PROLOG con corrutinas es MU-PROLOG, desarrollado por Lee [Naish, 1986] en la Universidad de Melbourne. Basada en éste han aparecido otros, como NU-PROLOG [Thom y Zobel, 1986] y PNU-PROLOG (con extensiones para la concurrencia).

Actualmente hay un grupo de trabajo ISO (ISO/IEC JTC1 SC22 WG17) que acomete la tarea de definir un estándar internacional para PROLOG y que consta de dos partes: N110 (parte 1, Núcleo General) y N111 (parte 2, Módulos). La parte 1 ya está terminada y aprobada [Deransart et al., 1996]. A principios de 1995, el documento se publicó como un ISO/IEC (*ISO CD 13211-T*). La parte 2 (*ISO CD 13211-II*) progresa mucho más lentamente siendo objeto de un intenso debate dentro del WG17. Algunas extensiones (como restricciones e interfaz con otros lenguajes) se han dejado conscientemente fuera del estándar para evitar retrasos en su terminación. Existe un borrador que recoge el estado actual de las discusiones (X3J17 96/7 ANSI Committee Working Draft WD9.1).

El *Department of Social Science Informatics* (SWI) de la Universidad de Amsterdam comenzó a desarrollar SWI-PROLOG, basado en un subconjunto de la máquina de Warren [Warren, 1983; Aït-Kaci, 1991] formado únicamente por 7 instrucciones [Bowen y Byrd, 1983]. Actualmente, la última versión disponible es la 3.4.3 (Diciembre de 2000[82]), y ésta, así como las anteriores, han sido validadas por [Hodgson, 1998] de acuerdo al estándar ISO antes mencionado, de forma que implementa todos los predicados de [Deransart et al., 1996]. SWI-PROLOG, además de ser un estándar, es compatible con las versiones más difundidas de PROLOG, como SICSTUS-PROLOG, de tal forma que será la versión elegida para las prácticas de $\mathrm{P_{RO}L_{OG}}$ de las asignaturas objeto del presente proyecto docente.

0.2.3 Sistemas de tipos para Prolog

Un problema muy conocido por los docentes de PROLOG (y por los alumnos) es la dificultad en localizar ciertos errores debido a la ausencia de un sistema para la declaraciones de modo o de tipos, lo que permite un estilo poco disciplinado en la programación. Por ejemplo, la detección de variables que aparecen una sola vez en una regla, o la no concordancia de la aridad entre reglas y objetivos. Así, si existe un predicado *inserta*/3, en ARITY-PROLOG el objetivo $inserta(3, X, I, Z)$ simplemente no tiene éxito, sin más información. Para corregir este defecto, algunos versiones de PROLOG incorporaron un sistema de localización de tales errores basado en ciertas declaraciones de tipo; por ejemplo, en TURBO-PROLOG los predicados pueden ir precedidos de una declaración de tipo: $inserta(int, int*, int*)$; pero

[82] `www.swi.psy.uva.nl/proyects/SWI-Prolog/`.

ya que no está permitido el polimorfismo, es necesario definir un predicado para cada versión. Otros sistemas (como SWI-PROLOG) incorporan un mecanismo de información al usuario (para el objetivo anterior indicaría que no existe ninguna versión de *inserta*/4), aunque no incorpora un sistema de tipos (ni siquiera monomórfico).

El problema de la tipificación en PROLOG es introducido en [Bruynooghe, 1982]. Según [Reddy, 1988][83], la tipificación de un programa PROLOG puede ser descriptiva (*descriptive*) o prescriptiva (*prescriptive*)[84]. La primera está relacionada con la tipificación à la Curry, y un programa tipificado es visto como una parte de la base de Herbrand, de forma que la semántica se obtiene estableciendo una relación entre partes de la base de Herbrand. Ejemplo de tal sistema es *Regular Tree Typed Prolog* [Mishra, 1984], y también, añadiendo polimorfismo, el propuesto en [Yardeni y Shapiro, 1991]. En tales estilos de tipificación no se habla de programa bien tipificado (estáticamente) y el sistema es flexible. Propuestas posteriores con ligeras modificaciones permiten encontrar algoritmos de reconstrucción de tipos en presencia de polimorfismo, como el expuesto en [Pyo y Reddy, 1989].

El punto de vista prescriptivo está relacionado con la tipificación (fuerte) à la Church, y un sistema de inferencia de tipos libera al programador de un uso incorrecto. Normalmente tales sistemas son modificaciones de sistemas utilizados en el λ–cálculo con tipos, como por ejemplo el propuesto por [Mycroft y O'Keefe, 1984], que es una adaptación del sistema de tipos de ML [Milner, 1978], utilizando una *many–sorted logic* extendida con polimorfismo; tales sistemas de tipos para PROLOG los denominaremos à la ML.

El sistema de [Mycroft y O'Keefe, 1984] ha dado lugar a diferentes controversias[85], aunque ha sido utilizado como herramienta para la verificación de programas, e incluso, ha sido el punto de partida de otros sistemas de tipos, como los adoptados para MERCURY, λPROLOG, GÖDEL y ESCHER[86].

Los sistemas de tipos (prescriptivos) à la ML tienen una clara ventaja: en tiempo de compilación se descubren prácticamente todos los errores, más aún, si es obligatorio la tipificación de todos los predicados. La desventaja es que ciertas prácticas de la $\mathrm{P_{RO}L_{OG}}$ dejan de ser válidas. A modo de ejemplo, es usual encontrarnos con el siguiente predicado para *aplanar* una lista:

$$
\begin{array}{lll}
aplana\ [\,] & [\,] & :-\ !.\\
aplana\ [L|L']\ A & & :-\ !,\ aplana\ L\ AL,\ aplana\ L'\ AL',\ conc\ AL\ AL'\ A.\\
aplana\ E & [E]. &
\end{array}
$$

Tal descripción es válida ya que en un PROLOG sin tipos las estructuras pueden ser heterogéneas, y tales descripciones son esenciales en metaprogramación. En un PROLOG à la ML, como λPROLOG, habría que escribir algo como lo siguiente para aplanar una estructura arbórea:

[83] ver también [Ridoux, 1998]:53, para una discusión histórica del tema.

[84] Simplificaremos de esta forma la exposición sobre sistemas de tipos en la $\mathrm{P_{RO}L_{OG}}$. Existen otras propuestas no clasificables con esta filosofía, como la expuesta recientemente en la tesis de J. Boye, *Directional Types* (1996, Linköping University), bajo la cual un predicado p es visto como una (multi)función entre tuplas de tipos, siguiendo una idea de A. Aiken de 1994. Referencias esenciales, así como una introducción a este punto de vista puede obtenerse en [Charatonik y Podelski, 1998].

[85] Citadas, por ejemplo en [Ridoux, 1998] y en [Pfenning, 1992]

[86] Además, ML ha sido utilizado para implementar tales lenguajes; por ejemplo, la implementación TERZO del lenguaje λPROLOG está realizada en SML. `www.cis.upenn.edu/~dale/lProlog`.

```
kind árbol type → type.
type hoja F → (árbol F).
type nodo (árbol F) → (árbol F) → (árbol F).
type aplanaÁrbol (árbol F) → (list F) → o.


aplanaÁrbol (hoja F) [F].
aplanaÁrbol (nodo I D) Fs :- aplanaÁrbol I Ai, aplanaÁrbol D Ad, conc Ai Ad A.
```

En un PROLOG tipificado à la ML, en una misma cláusula todas las ocurrencias de la misma variable deben admitir un tipo común (de la misma forma que en lenguajes funcionales derivados de ML). Esta condición, llamada *genericidad definicional* (o también *head condition*) se viola en muchos programas clásicos, y varios autores obligan a su uso ya que consideran que los programas que la violan no pueden verse como programas declarativos [O'Keefe, 1990; Ridoux, 1998].

En [Fernández Leiva, 1994; Ruiz Jiménez y Fernández Leiva, 1995; Ruiz Jiménez y Fernández Leiva, 1996] proponíamos un sistema de clases de tipos para PROLOG al estilo de HASKELL, con polimorfismo paramétrico. Esencialmente proponíamos una sintaxis à la HASKELL para describir clases de tipos que permitía controlar la sobrecarga. La semántica operacional viene dada por una nueva estrategia de resolución en la cual el significado de un objetivo viene determinado por el contexto en que aparece. La sintaxis es similar a la de HASKELL:

```
class Igual a where
  igual/2 :: a → a.
  distinto/2 :: a → a.
  distinto(X, Y) :- not igual(X, Y).


instance Igual a ⇒ Igual [a] where
  igual([], []).
  igual([X|Xs], [Y|Ys]) :- igual(X, Y), igual(Xs, Ys).


data Conjunto a = vacío | conj(a, Conjunto a).
pertenece/2 :: Igual a ⇒ a → Conjunto a.
pertenece(X, conj(X', C)) :- igual(X, X').
pertenece(X, conj(_, C)) :- pertenece(X, C).
```

El sistema permite definir predicados por defecto, subclases y predicados de orden superior. Así mismo proponíamos un concepto de parcialización, de forma que para el predicado $suma/3 :: Int \rightarrow Int \rightarrow Int$, el término $suma(4, 6)$ representa un predicado[87]. Así mismo es posible introducir el concepto de clases de constructores de tipos. Desgraciadamente el sistema no fue implementado completamente.

Mercury

Una propuesta relativamente reciente es MERCURY [Somogyi et al., 1995], un lenguaje lógico puro con tipos y módulos. Al decir puro queremos decir que no incorpora construcciones no

[87] [Nadathur y Pfenning, 1992] proponen un concepto de parcialización muy diferente ya que el sistema de tipos lo permite. Tal sistema es el adoptado para λPROLOG.

lógicas (como el corte), aunque proporciona sustitutos declarativos para las características no lógicas, como por ejemplo la entrada/salida. Incorpora además declaraciones de modos y de determinismo, así como predicados de orden superior.

MERCURY es parecido en su sintaxis a PROLOG, pero semánticamente es muy diferente. Su sistema de tipos, al igual que el sistema de tipos de GÖDEL, esta basado en una lógica heterogénea por lo que resulta esencialmente equivalente al sistema de tipos propuesto por [Mycroft y O'Keefe, 1984]. La sintaxis para las declaraciones de tipos está inspirada en la de NU-PROLOG:

$$:- \textit{type bool} \; --> \; \textit{true}; \textit{false}.$$
$$:- \textit{type lista}(T) \; --> \; []\,; \, [T \mid \textit{lista}(T)].$$
$$:- \textit{type árbol}(T) \; --> \; \textit{vacío}\,; \, \textit{nodo}(\textit{árbol}(T), T, \textit{árbol}(T)).$$
$$:- \textit{pred permutación}\,(\textit{list}(T), \textit{list}(T)).$$
$$:- \textit{pred aplana}\,(\textit{árbol}(T), \textit{lista}(T)).$$

Incluye también declaraciones de modo (*mode declaration constraint*) y polimorfismo:

$$:- \textit{pred append}(\textit{list}(T), \textit{list}(T), \textit{list}(T)).$$
$$:- \textit{mode append}(\textit{in}, \textit{in}, \textit{out}).$$
$$:- \textit{mode append}(\textit{out}, \textit{out}, \textit{in}).$$

(donde T es una variable de tipo) incorporando en el compilador procedimientos para comprobar si las reglas las satisfacen[88]. Otra característica es el tratamiento del determinismo, incorporando declaraciones para ello, lo que mejora notablemente la eficiencia. Así, tenemos la siguiente versión determinista:

$$:- \textit{mode append}(\textit{in}, \textit{in}, \textit{out})\ \textit{is det}.$$
$$\textit{append}(A, B, C) \; :- \; A == [], \; C := B.$$
$$\textit{append}(A, B, C) \; :- \; A == [H|T], \; \textit{append}(T, B, NT), \; C := [H|NT].$$

o también la siguiente versión no determinista:

$$:- \textit{mode append}(\textit{out}, \textit{out}, \textit{in})\ \textit{is nondet}.$$
$$\textit{append}(A, B, C) \; :- \; A := [], \; B := C.$$
$$\textit{append}(A, B, C) \; :- \; C == [H|NT], \; \textit{append}(T, B, NT), \; A := [H|T].$$

Las entradas/salidas conservan la transparencia ya que incluyen variables para capturar los efectos laterales. El siguiente ejemplo ilustra el uso de módulos:

$$:- \textit{module ejemplo}.$$

$$:- \textit{interface}.$$
$$:- \textit{import\_module io}.$$
$$:- \textit{pred main}(\textit{io\_state}, \textit{io\_state}).$$
$$:- \textit{mode main}\,(\textit{destr\_in}, \textit{unique\_out})\ \textit{is det}.$$

$$:- \textit{implementation}.$$
$$:- \textit{import\_module string}.$$
$$\textit{main}\,(S0, S) \; :- \; \textit{write\_string}(\text{"}\textit{Hola}, \text{"}, \; S0, S1),$$
$$\qquad\qquad\qquad\qquad \textit{write\_string}(\text{"}\textit{Adios}\backslash\, n.\text{"}, \; S1, S).$$

[88] El comprobador de tipos de MERCURY es un módulo de 2600 líneas de código MERCURY.

donde vemos cómo quedan encapsulados los efectos laterales en las variables $S0, S1$ y S, utilizando un mecanismo parecido a las *continuaciones* utilizadas en HASKELL (aunque en este caso se tiene *visibilidad* sobre las variables que capturan los efectos laterales, y es necesaria pasarlas entre los predicados).

El código generado con las anotaciones de modo y determinismo permite que MERCURY sea uno de los sistemas más eficientes[89]. Pero quizás más interesante es la facilidad para la programación disciplinada vía las declaraciones de tipos, modos y determinismo, ya que el programador será advertido de cualquier violación de tales restricciones.

La implementación actual del sistema[90] soporta clases de tipos à la HASKELL [Jeffery et al., 1998][91]. Por ejemplo, tenemos las siguientes declaraciones de clases MERCURY:

$$
\begin{array}{l}
:- \mathit{typeclass}\ \mathit{punto}(T)\ \mathit{where}\ [\\
\quad \mathit{pred}\ \mathit{coordenada\_x}(T, \mathit{float}), \\
\quad \mathit{pred}\ \mathit{asigna\_x}(T, \mathit{float}, T) \\
\quad \ldots \\
].
\end{array}
$$

$$
\begin{array}{l}
:- \mathit{typeclass}\ \mathit{mostrable}(T)\ \mathit{where}\ [\\
\quad \mathit{pred}\ \mathit{muestra}(T, \mathit{io\_state}, \mathit{io\_state}). \\
].
\end{array}
$$

La declaraciones de instancias siguen de nuevo el esquema de HASKELL:

$$
\begin{array}{l}
:- \mathit{type}\ \mathit{par}\ -\!\!-\!\!> \ \mathit{cart}(\mathit{float}, \mathit{float}). \\
:- \mathit{pred}\ \mathit{primero}(\mathit{par}, \mathit{float}). \\
\mathit{primero}(\mathit{par}(X, \_), X).
\end{array}
$$

$$
\begin{array}{l}
:- \mathit{pred}\ \mathit{asigna\_segundo}(\mathit{par}, \mathit{float}, \mathit{par}). \\
\mathit{asigna\_segundo}(\mathit{par}(\_, Y), X, \mathit{par}(X, Y)).
\end{array}
$$

$$
\begin{array}{l}
:- \mathit{instance}\ \mathit{punto}(\mathit{par})\ \mathit{where}\ [\\
\quad \mathit{pred}(\mathit{coordenada\_x})\ \mathit{is}\ \mathit{toma\_primero}, \\
\quad \mathit{pred}(\mathit{asigna\_y})\ \mathit{is}\ \mathit{asigna\_segundo} \\
\quad \ldots \\
].
\end{array}
$$

De la misma forma se permiten incluir *contextos* à la HASKELL en la declaración de tipo de un predicado:

$$
\begin{array}{l}
:- \mathit{pred}\ \mathit{pinta}(T, \ldots) <= \mathit{punto}(T). \\
\mathit{pinta}(P, \ldots) :- \mathit{coordenada\_x}(P, X), \ldots
\end{array}
$$

[89] En [Somogyi et al., 1995] aparece un estudio comparativo con otras versiones de PROLOG, entre las que aparecen SWI-PROLOG, SICSTUS-PROLOG, NU-PROLOG y AQUARIUS [van Roy, 1992]. Realmente el compilador *base* de MERCURY (es decir, de un subconjunto básico de MERCURY) es una traslación a código NU-PROLOG. El compilador completo está formado por 42500 líneas de código MERCURY.

[90] `http://www.cs.mu.oz.au/research/mercury/`.

[91] Realmente, el sistema de clases está implementado en MERCURY vía un argumento auxiliar en los predicados para capturar una estructura de diccionario que encapsula las predicados reales que se deben invocar.

$$
\begin{array}{l}
:-instance\ mostrable(lista(T))\ <=\ mostrable(T)\ where\ [\\
\quad pred(muestra/3)\ is\ muestra\_lista\\
].
\end{array}
$$

$$
\begin{array}{l}
:-\ muestra\_lista(lista(T), io\_state, io\_state)\ <=\ mostrable(T).\\
muestra\_lista([\,], S0, S3).\\
muestra\_lista([X|Xs], S0, S3)\ :-\ muestra(X, S0, S1),\\
\qquad\qquad\qquad\qquad\qquad\quad write(",\ ",\ S1,\ S2),\\
\qquad\qquad\qquad\qquad\qquad\quad muestra\_lista(Xs, S2, S3).
\end{array}
$$

También se permiten definir subclases:

$$
\begin{array}{l}
:-\ typeclass\ punto\_coloreado(T)\ <=\ punto(T)\ where\ [\\
\quad pred\ color(T, color).\\
].
\end{array}
$$

Recientemente se ha propuesto el sistema MCORBA [Jeffery et al., 1999], un sistema basado en MERCURY que permite interaccionar con componentes CORBA. De la misma forma que HASKELL representa interfaces COM sobre clases de tipos de HASKELL [Peyton Jones et al., 1998; Leijen et al., 1999; Leijhen, 1998], MCORBA representa interfaces CORBA sobre clases de tipos de MERCURY.

0.2.4 Programación Lógica con restricciones

PROLOG tiene dos problemas importantes derivados del principio de unificación sintáctica; por un lado la pérdida de la simetría en parámetros de entrada–salida y por otro lado la imposibilidad en casi todos los casos de representar un conjunto infinito de soluciones; consideremos algunos ejemplos bien conocidos; si tomamos el factorial:

$$
\begin{array}{l}
fact(0, 1).\\
fact(N, F)\ :-\ N > 0,\ M = N - 1, fact(N - 1, G),\ F\ =\ N * G.
\end{array}
$$

mientras el objetivo $:-fact(3, F)$ es resuelto sin problemas, el objetivo inverso $:-fact(N, 6)$ no es posible satisfacerlo ya que la invocación al predicado $M = N - 1$ no puede resolverse al no estar N instanciada; problemas semejantes tenemos con sistemas de ecuaciones tales como:

$$
\begin{array}{l}
ec1(S, T)\ :-\ S + T\ =\ 8\\
ec2(S, T)\ :-\ U - V\ =\ 8
\end{array}
$$

al intentar satisfacer el objetivo $:-\ ec1(X, Y),\ ec2(X, Y)$ ya que la coerción lógica $S + T = 8$ no es manejada por PROLOG durante el mecanismo de inferencia a no ser que tales variables estén instanciadas. Si volvemos a la formulación del factorial y al objetivo $:-fact(3, F)$ vemos que puede ser resuelto si podemos resolver el siguiente conjunto de restricciones lógicas:

$$
\begin{array}{l}
C1 = \{3 > 0, F = 3 * M1\}\\
C2 = C1 \wedge \{2 > 0, M1 = 2 * M2\}\\
C3 = C2 \wedge \{1 > 0, M2 = 1 * M3\}\\
C2 = C1 \wedge \{2 > 0, M3 = 1\}
\end{array}
$$

La idea del uso de una restricción (*constraint*) como fórmula lógica es antigua. La

programación con restricciones ($P_{RO}R_{ES}$, CP o *Constraint Programming*) es introducida en la tesis de Guy [Steele, 1980], aunque normalmente ha estado íntimamente ligada a la $P_{RO}L_{OG}$, y será aplicada a la construcción *freeze* para suspender cómputos en PROLOG II [Colmerauer et al., 1982; Colmerauer, 1982].

Esencialmente, la CP considera una restricción que puede evolucionar incorporando restricciones sucesivas más fuertes a través del mecanismo *ask-tell*. Existe una relación entre restricciones: $C \rightrightarrows D$ (esta relación no es necesariamente la implicación lógica estándar, y captura la idea de que C es compatible con D). En cierto espacio de la memoria (*constraint store*) aparece ubicada una restricción especial S que controla los cómputos (la llamaremos restricción global). Una nueva restricción C puede añadirse a la global a través del mecanismo *telling* para formar $S \wedge C$ (siempre que sea compatible), lo que significa que la restricción S avanza a la conjunción $S \wedge C$. Se supone que es posible sincronizar una acción o tarea t con una restricción C llamada su guarda. La tarea será reducible si su guarda es compatible con la restricción global, y en ese caso ésta es añadida. Así el modelo es monótono, y la restricción global evoluciona en la forma $true \wedge C_1 \wedge C_2 \wedge \ldots \wedge C_n$. La semántica declarativa de tales lenguajes está basada en universos de Herbrand con restricciones[92].

El esquema de la *Programación Lógica con Restricciones* (CLP o *Constraint Logic Programming*) [Jaffar, 1987; Jaffar y Maher, 1994] consiste en sustituir el concepto de unificación por el de resolución por coerción en un dominio específico con operadores apropiados; al igual que PROLOG, el esquema CLP usa una sintaxis basada en cláusulas de Horn:

$$p :- t_1,\ t_2,\ \ldots,\ t_n.$$

donde los términos t_i pueden ser predicados o restricciones. En la familia CLP el motor de inferencia está basado en un resolvedor de restricciones. El *backtracking* se genera al detectar una restricción inconsistente. Dentro de la familia aparecen varios sistemas, como por ejemplo CLP(R), que incorpora restricciones sobre los reales.

El número de lenguajes que incorporan restricciones no ha parado de crecer, y muchos añaden algún otro paradigma, aunque esencialmente sea el paradigma lógico, como en los lenguajes ECHIDNA (extensión del PROLOG de Edimburgo), GAPLOG (*General Amalgamated Programming with Logic*) (que utiliza varios dominios), PROLOG III y PROLOG IV (intervalos), CHIP (*Constraint Handling In Prolog*) (de la familia CLP(Bool,Q), además de dominios finitos), CIAL (*Interval constraint logic language*) (de la familia CLP(R), incorpora un resolvedor de restricciones lineales – Gauss/Seidel – y *narrowing*), y CLP(sigma*), que incorpora restricciones sobre conjuntos.

Otros son extensiones de COMMONLISP [Gold Hill Computers, 1987], como CONSTRAINTLISP y SCREAMER. Otros extienden CP con objetos, como KALEIDOSCOPE (con construcciones imperativas), y SIRI.

Otros extienden CP con funciones y lógica, como TRILOGY (es un CLP(N) con mezcla de PROLOG, LISP y PASCAL), OZ, LIFE, CURRY y TOY; estos últimos serán comentados en otra sección específica.

Finalmente, otros lenguajes extienden el paradigma CP con la concurrencia (o el pa-

[92] Puede verse una introducción en [de Boer et al., 1993; Saraswat, 1993], aunque el segundo introduce un lenguaje para la comunicación asíncrona basada en restricciones. Una referencia suficientemente actual y muy didáctica es [Marriott y Stuckey, 1998]. En ésta podemos encontrar numerosos ejemplos, árboles de resolución, etc., así como el uso de estructuras de datos, aplicaciones diversas y técnicas avanzadas.

	Lógica	Funciones	Tipos	Restricciones	Concurrencia	Objetos	Referencias
AKL	x			CLP(R)	x		[Janson, 1992]
CHIP	x			CLP(B,Q)			[Dincbas et al., 1988; van Hentenryck, 1989]
CIAL	x			CLP(R), intervalos			[Chiu, 1994]
CLP(SIGMA*)	x			*sets*			[Walinsky, 1989]
CONSTRAINTLISP		x		x		x	[Liu, 1992]
ECHIDNA	x			x			
GAPLOG	x			varios dominios			[Maluszynski et al., 1993]
GOFFIN		x	x	x	x		[Chakravarty et al., 1995]
JANUS, CC(...),	x			x	distrib.		[Saraswat, 1990; Saraswat, 1989]
KALEIDOSCOPE				x		x+imp.	[Freeman-Benson, 1990; Lopez, 1994]
LIFE	x	x	x	x	x	x	[Aït-Kaci y Lincoln, 1989; Aït-Kaci et al., 1994]
LUCY	x			x	actores		[Kahn, 1990]
OZ	x	x	x	x	x	x	[Smolka, 1995]
PROCOL				x	x	x	[van den Bos, 1991]
PROLOG III	x			CLP(B,Q)			[Colmerauer, 1987]
SCREAMER, extensión de COMMONLISP		x		x		x	
SIRI				x		x	[Horn, 1992]
TRILOGY	x	x + PASCAL		CLP(N)			[Voda, 1988]

Tabla 2: Principales Lenguajes que incorporan restricciones

ralelismo). En la Sección 0.3.0 daremos otros detalles.

Una restricción puede ser vista como una fórmula que proporciona información parcial, de forma que puede competir con el concepto de sustitución de la $\mathrm{P_{RO}L_{OG}}$.

Comentemos dos ejemplos de tales lenguajes; uno de ello es CLP(R) [Lassez, 1987; Jaffar, 1990], un lenguaje experimental tipo Edimburgo que utiliza el estilo CLP en el dominio de los números reales. En CLP(R) un circuito eléctrico (ideal) como el de la Figura 2 puede ser descrito con los siguientes predicados (al igual que en otro PROLOG):

$$ley\_de\_Ohm\,(V, I, R) :- V = I * R.$$
$$ley\_de\_Kirchoff(L) :- suma(L, 0).$$

$$suma([\,], 0).$$
$$suma([H|T], N) :- H + M = N, suma(T, M).$$

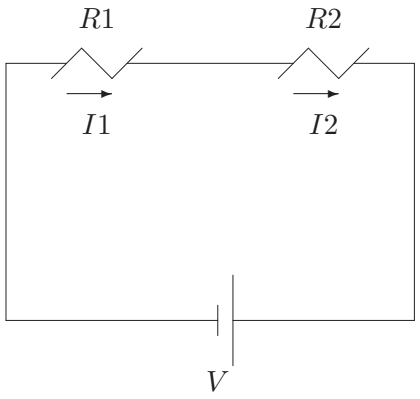


Figura 2: Un circuito eléctrico

$$
\begin{array}{l}
circuito(V1, I1, R1, V2, I2, R2, V) :- \\
\quad ley\_de\_Ohm(V1, I1, R1), \\
\quad ley\_de\_Ohm(V2, I2, R2), \\
\quad ley\_de\_Kirchoff([I1, -I2]), \\
\quad ley\_de\_Kirchoff([-V, V1, V2]).
\end{array}
$$

Supongamos ahora que tenemos una serie de resistencias disponibles y de fuentes, descritas por:

$$
\begin{array}{ll}
resistencia(10). & resistencia(14). \\
resistencia(27). & resistencia(60). \\
fuente(10). & fuente(20).
\end{array}
$$

El problema que queremos resolver es determinar la forma de construir circuitos con los elementos disponibles de forma que el potencial en la resistencia $R2$ esté en ciertos límites, p.e., $14.5 < V2 < 16.25$; en un PROLOG estándar el objetivo:

$$
\begin{array}{l}
:- 14.5 < V2,\ V2 < 16,25, \\
\quad resistencia(R1),\ resistencia(R2), \\
\quad fuente(V), \qquad circuito(V1, I1, R1, V2, I2, R2, V).
\end{array}
$$

no puede ser resuelto, mientras en CLP(R) el sistema resuelve apropiadamente el sistema de ecuaciones:

$$
\begin{array}{lcll}
V1/R1 & - & V2/R2 & = & 0 \\
V1 & + & V2 & = & V
\end{array}
$$

con ayuda de las restricciones impuestas por los elementos disponibles y la desigualdad deseada, obteniéndose las soluciones:

$$
\begin{array}{l}
V = 20, R1 = 10, R2 = 27 \\
V = 20, R1 = 14, R2 = 67
\end{array}
$$

Otro lenguaje que implementa CLP es PROLOG III [Colmerauer, 1987; Colmerauer, 1990], donde el mecanismo de unificación es potenciado con restricciones . En este lenguaje una variable puede denotar objetos atómicos (números, caracteres, booleanos), listas o árboles;

existen cuatro constructores para listas y árboles; constructor de listas: $a =< b, c, d >$ (los elementos pueden ser árboles), constructor de árboles: $a = b(c, d, e) - b$ (debe ser una hoja), constructor general de árboles: $a = b[c]$ (donde b es una hoja y c una lista) y concatenador de listas: $a = b \cdot c$ (b y c deben ser listas). Así, la descomposición $[X|L]$ se escribe $< x > \cdot l$. Además de las relaciones binarias normales, existen una serie de relaciones unarias tales como: $a : list$ (el árbol a es una lista), $a : 10$ (el árbol a es una lista de longitud 10), $a : leaf$ (el árbol a es una hoja). En PROLOG III cada regla es de la forma:

$$t_0 \rightarrow t_1 \dots t_n, S;$$

donde t_i son términos (en sentido ordinario) y S es un conjunto de restricciones. La máquina PROLOG III es una máquina indeterminista descrita con tres fórmulas:

$$\begin{array}{ll} (1) & (W, t_0 t_1 \dots t_n, S) \\ (2) & s_0 \rightarrow s_1 \dots s_n, R \\ (3) & (W, t_0 \dots t_n s_0 \dots s_n, S \cup R \cup \{t_0 = s_0\}) \end{array}$$

La fórmula (1) representa el estado de la máquina en cierto instante: W es un conjunto de fórmulas, $t_0 t_1 \dots t_n$ es una secuencia de términos y S un conjunto de restricciones; la fórmula (2) indica una regla del programa que puede cambiar el estado de la máquina y la fórmula (3) indica el nuevo estado después de aplicar la regla. Por ejemplo, sea el programa que describe un menu bajo en calorías (el segundo parámetro de los *platos* indica un valor de calorías):

$$\begin{array}{ll} 1 & menu(e, s, p) \leftarrow \\ & \quad entrada(e, i) \quad segundo(s, j) \quad postre(p, k), \\ & \quad \{i \geq 0,\ j \geq 0,\ k \geq 0,\ i + j + k \leq 10\}; \\ 2 & segundo(s, j) \leftarrow carne(s, j); \\ 3 & segundo(s, j) \leftarrow pescado(s, j); \\ 4 & entrada(rábano, 1) \leftarrow; \\ 5 & entrada(ensalada, 6) \leftarrow; \\ 6 & carne(cordero, 5) \leftarrow; \\ 7 & carne(ternera, 7) \leftarrow; \\ 8 & pescado(lenguado, 2) \leftarrow; \\ 9 & pescado(atún, 4) \leftarrow; \\ 10 & postre(piña, 2) \leftarrow; \\ 11 & postre(helado, 6) \leftarrow; \end{array}$$

Para el objetivo $menu(e, s, p)$? consideramos el estado inicial:

$$(\{e, s, p\},\ \text{menu}(e, s, p), \{\})$$

aplicando la regla 1 tenemos el siguiente estado (después de simplificar):

$$(\{e, s, p\}, ent(e, i) seg(s, j) pos(p, k), \{i \geq 0, j \geq 0, k \geq 0, i + j + k \leq 10\})$$

Aplicando la regla 5 tenemos el nuevo estado:

$$(\{e, s, p\}, seg(s, j) pos(p, k), \{e = ensalada, j \geq 0, k \geq 0, j + k \leq 4\})$$

y aplicando sucesivamente las reglas 3 y 9 se llega a

$$(\{e, s, p\}, pos(p, k), \{e = ensalada, s = atun, k \geq 0, k \leq 2\})$$

que conduce al siguiente estado al aplicar la regla 10:

$$(\{e, s, p\}, , \{e = ensalada, s = atun, p = piña\})$$

Otro ejemplo viene determinado por la siguiente formulación; sea una lista de pagos anuales para amortizar una deuda d al 10% de interés anual; si la lista de pagos es $< a >\cdot$ (a es la amortización del primer pago y p son los plazos pendientes) entonces la deuda d se transforma en $(1 + 10/100)d - a$, es decir, tenemos la siguiente formulación:

$plazos(<>,\ 0) \leftarrow;$
$plazos(< a > \cdot p,\ d\ ) \leftarrow\ plazos(\ p,\ (1 * 10/100) * d - a\ );$

Así, podemos resolver problemas tales como el cálculo de una amortización variable, tal como:

$$plazos(< i, 2i, 3i >, 1000)?$$

Tenemos que el estado inicial es

$$(\{i\}, plazos(< i, 2i, 3i >, 1000), \{\})$$

y al aplicar la regla 2 tenemos el siguiente estado:

$$(\{i\},\ \text{plazos}(p, (1 * 10/100) * d - a), \{plazos(< i, 2i, 3i >, 1000) = plazos(< a > \cdot p, c\}$$

que se simplifica en:

$$(\{i\}, plazos(< 2i, 3i >, 1100 - i, \{\})$$

y aplicando sucesivamente la regla 2 (dos veces) y la 1 se llega al estado final:

$$(\{i\}, , \{i = 207 + 413/641\})$$

Para PROLOG puro, o para CLP, no disponemos de un método intuitivo y simple que permita verificar los programas. El problema, y aquí quizás estemos todos de acuerdo, es la recursión, clave de la $\text{P}_{\text{RO}}\text{L}_{\text{OG}}$ y de la CLP. Combinar recursión con fallo (que permiten expresar realmente los programas) puede conducir a programas no manejables para un análisis formal (es el eterno vicio del programador). Para resolver el problema, [Apt y Bezem, 1999] proponen sustituir la recursión por la cuantificación acotada [Voronkok, 1992], lo que da lugar a un nuevo paradigma. Basado en éste proponen el lenguaje ALMA-0, un lenguaje que extiende la programación imperativa con la lógica.

0.2.5 Programación Lógica Inductiva

Un campo hoy en día atractivo y en continuo desarrollo es la Programación Lógica Inductiva (ILP o *Inductive Logic Programming*), cuya principal motivación es la enseñanza automática (*machine learning*).

En $\text{P}_{\text{RO}}\text{L}_{\text{OG}}$ un paso de resolución es visto como un paso fundamental de la computación, que puede ser implementado eficientemente a través de la WAM. Por otro lado, la inducción es mucho más ineficiente debido a la aparición del indeterminismo.

El término ILP fue acuñado por Stephen Muggleoton en 1990, y surge como una verdadera escuela en 1991 en Portugal durante la celebración del *First International Workshop*

on Inductive Logic Programming [Muggleton, 1991]. Hoy en día tiene su propia conferencia anual (ver por ejemplo, [Lavrac y Dzeroski, 1997; Dzeroski et al., 1998])

El marco lógico de la ILP es muy simple [Furukawa, 1999]: dado un conocimiento de fondo B, un conjunto de ejemplos positivos E (que no sea derivable a partir de B) y un conjunto de restricciones I, la tarea de la ILP es encontrar una hipótesis H verificando $B \wedge H \Rightarrow E$, manteniendo las restricciones I[93]. Desde el punto de vista computacional, el objetivo de la ILP es encontrar algoritmos eficientes para computar la implicación inversa: es decir, encontrar una hipótesis apropiada H verificando $B \wedge H \Rightarrow E$, o equivalentemente, $B \wedge \neg E \Rightarrow \neg H$, lo que sugiere el cómputo de $\neg H$ en forma deductiva. Para ello, A. Yamamoto introduce el concepto de hipótesis más específica o $MSH(B,E)$[94] de forma que a partir de una solución de $B \wedge \neg E \Rightarrow \neg MSH(B,E)$ cualquier hipótesis válida pueda obtenerse en la forma $H \Rightarrow MSH(B,E)$. Yamamoto sugiere la idea de que tal MSH pueda obtenerse vía la negación $\neg MSH(B,E)$ que consiste en todos los literales sin variables positivos y negativos que son ciertos en todos los modelos de Herbrand de la fórmula $B \wedge \neg E$[95]. Sin embargo, existen ejemplos donde tal definición no funciona, como por ejemplo, para el siguiente ejemplo extraído de [Furukawa, 1999], descrito en PROLOG:

$$\begin{array}{ll} B & : even(0). \\ & \quad even(s(x)) \ :- \ odd(x). \\ E & : odd(s(s(s(0)))). \\ H & : odd(s(x)) \ :- \ even(x). \end{array}$$

En ese caso tenemos $MSH(B,E) \equiv odd(s(s(s(0)))) : -even(0)$, pero sin embargo, siendo H una hipótesis correcta, no puede ser computada a través de $MSH(B,E)$ (ya que $H \not\models MSH(B,E)$). El problema queda resuelto imponiendo restricciones a las fórmulas que proporcionan el conocimiento B, utilizando en este caso el principio de SB–resolución introducido por Plotkin en su tesis de 1971.

La ILP ha sido utilizada con éxito dentro de muchos campos: bioquímica, redes bayesianas, interfaz hombre–máquina, análisis musical, ecología, etc. Referencias interesantes sobre los últimos avances y la situación actual de la ILP pueden encontrarse en [Raedt, 1999].

0.2.6 Programación Lógica y Programación Concurrente

La incorporación de otros paradigmas a la programación lógica se realizará muy pronto; por ejemplo, [McCabe, 1985; Miller y Nadathur, 1986] desarrollan λPROLOG y [Moreno Navarro y Rodríguez Artalejo, 1988] desarrollan BABEL, ambos lenguajes lógico–funcionales (estos paradigmas serán analizados en otra sección). Sin embargo, serán los paradigmas de la programación orientada a objetos, la programación concurrente y la programación con restricciones los que faciliten el desarrollo más importante de la programación lógica en los últimos años. Veremos en esta sección la problemática de la programación lógica concurrente.

Existen varias formas de integrar la concurrencia en la $\mathrm{P_{RO}L_{OG}}$, y la literatura es muy amplia, aunque caben destacar dos trabajos esenciales: [Shapiro, 1989] y [de Kergommeaux y Codognet, 1994]. Esencialmente los modelos parten de considerar cada objetivo o átomo como un proceso, mientras que los procesos se comunican a través de variables compartidas

[93] En inglés suelen utilizarse los términos *background knowledge* para B y *positive example* para E.

[94] *Most Specific Hypothesis.*

[95] Estamos suponiendo que la hipótesis es una regla simple, de donde su negación es una conjunción de hechos sin variables.

a través de un mecanismo llamado *mensajes incompletos*.

Para ilustrar los problemas de la incorporación del paralelismo a la $P_{RO}L_{OG}$ consideremos el siguiente conjunto de reglas para la ordenación rápida o *quicksort* de CAR [Hoare, 1962] descritas con la sintaxis del PROLOG de Edimburgo:

$$
\begin{array}{l}
ordenar([\,],[\,]).\\
ordenar([H|T],\ Ordenada]\ :-\\
\qquad partir(T,H,L,G),\ ordenar(L,LO), ordenar(G,GO),\\
\qquad concatenar(LO,[H|GO],\ Ordenada).
\end{array}
$$

$$
\begin{array}{l}
partir([\,],X,[\,],[\,]).\\
partir([H|T],X,[H|L],G)\ :-\ H\ <=\ X,\ partir(T,X,L,G).\\
partir([H|T],X,L,[H|G])\ :-\ H\ >=\ X,\ partir(T,X,L,G).
\end{array}
$$

$$
\begin{array}{l}
concatenar([\,],X,X).\\
concatenar([X|R],\ Y,\ [X|Z])\ :-\ concatenar(R,\ Y,\ Z).
\end{array}
$$

Para la ejecución de un objetivo tal como $:-ordenar([3,4,2,6,8,1], Ordenada)$ con dos procesadores se puede pensar en, una vez partida la lista inicial, asignar a cada procesador la tarea de ordenar cada sublista L y G en paralelo para después concatenarlas; podemos introducir una notación especial para este paralelismo llamado y-paralelismo (*and-parallelism*) y describir la regla en la forma:

$$
\begin{array}{l}
ordenar([H|T],\ Ordenada]\ :-\\
\qquad partir(T,H,L,G),\\
\qquad ordenar(L,LO)\ \|\ ordenar(G,GO),\\
\qquad concatenar(LO,[H|GO],\ Ordenada).
\end{array}
$$

Es evidente que el y-paralelismo no es extensible a todos los subobjetivos ya que en una descripción en la forma:

$$
\begin{array}{l}
ordenar([H|T],\ Ordenada]\ :-\ partir(T,H,L,G)\ \|\ ordenar(L,LO)\ \|\\
\qquad\qquad\qquad\qquad\qquad ordenar(G,GO)\ \|\ concatenar(LO,[H|GO],\ Ordenada).
\end{array}
$$

las variables compartidas por distintos subobjetivos dan lugar a conflictos; es decir, el segundo proceso (subobjetivo) no puede satisfacerse hasta que la variable L, compartida con este subobjetivo y el primero no sea instanciada. Por otro lado, si un subobjetivo puede unificar con distintas reglas, podemos pensar en satisfacer éstas en paralelo; esta forma de paralelismo se llamará en forma natural o-paralelismo. Esto de nuevo plantea varios problemas; por ejemplo, si en el predicado *partir* (descrito con y-paralelismo):

$$
\begin{array}{l}
partir([\,],X,[\,],[\,]).\\
partir([H|T],X,[H|L],G)\ :-\ H\ <=\ X\ \|partir(T,X,L,G).\\
partir([H|T],X,L,[H|G])\ :-\ H\ >=\ X\ \|\ partir(T,X,L,G).
\end{array}
$$

se aplican en forma paralela las reglas 2 y 3 se puede llegar a una explosión combinatoria innecesaria si el primer predicado ($H <= X$) falla.

Casi todos los intentos para resolver tales problemas parten de la idea de incorporar notaciones especiales en la reglas, tales como las guardas y las anotaciones en variables, todo ello acompañado de restricciones en el uso del mecanismo de unificación (p.e., secuencial). Dos modelos surgen de estas ideas: PARLOG [Clark y Gregory, 1984] y CONCURRENT PROLOG

[Shapiro, 1983]. Ambos modelos han sido utilizados para otras aplicaciones. Así, PARLOG ha sido utilizado para la implementación de un sistema operativo para una máquina paralela como ALICE. W. Silverman y M. Hirsh, del Weizmann Institute of Science (Israel), desarrollaron un sistema operativo y entorno de programación (llamado *Logix*) para un subconjunto de CONCURRENT PROLOG: FLATCONCURRENT PROLOG [Shapiro, 1987]:(Vol. 2, cuarta parte).

CONCURRENT PROLOG extiende la sintaxis de las cláusulas añadiendo el símbolo guardián |, llamado *commit*. La regla:

$$H : -G_1, G_2, ..., G_m \mid B_1, B_2, ..., B_n$$

significa que el proceso H puede ser reducido a $B_1, \ldots, B_n$ si el conjunto de procesos guardianes $G_1, \ldots, G_m$ terminan con éxito; tal notación está inspirada en la notación de secuencias guardadas de [Dijkstra, 1976]. Se dice que un proceso puede declararse con una regla si es posible la unificación y las guardas terminan; si existen varias reglas se selecciona una en forma indeterminista; si no existe ninguna se dice que el proceso falla[96]. CONCURRENT PROLOG no es capaz de retornar atrás (backtracking) para buscar todas las soluciones.

La búsqueda de una regla para unificar puede realizarse en paralelo (o-paralelismo). La unificación en CONCURRENT PROLOG se extiende con el uso de anotaciones de solo-lectura sobre las variables (terminándolas en ?). Una espera a instanciar una variable con una notación de solo-lectura puede suspender el proceso, que se activa al instanciar la variable; ello permite que no se use el valor de la variable hasta que algún proceso ejecutándose concurrentemente le asigne un valor.

Desde el punto de vista de la lectura de la regla, las anotaciones de solo lectura pueden ignorarse. Aún así, obsérvese que de esta forma se introducen de nuevo mecanismos de control en las reglas, al igual que en PROLOG.

Así, el método de ordenación rápida se escribe en CONCURRENT PROLOG en la forma:

$ordenar([\,], [\,]).$
$ordenar([H|T],\ Ordenada] \ :-$
$\quad partir(T?, H, L, G),$
$\quad ordenar(L?, LO), ordenar(G?, GO),$
$\quad concatenar(LO?, [H|GO?],\ Ordenada).$

$partir([\,], X, [\,], [\,]).$
$partir([H|T], X, [H|L], G) \ :- H \ <= \ X \mid partir(T?, X, L, G).$
$partir([H|T], X, L, [H|G]) \ :- H \ >= \ X \mid partir(T?, X, L, G).$

$concatenar([\,], X, X).$
$concatenar([X|R],\ Y,\ [X|Z]) \ :- concatenar(R?, Y, Z).$

Las dificultades de implementación y los problemas provocados por las variables anotadas como de solo lectura en CONCURRENT PROLOG dan lugar a un subconjunto del lenguaje llamado FLAT–CONCURRENT PROLOG, en el cual las guardas solamente pueden contener primitivas.

[96] En [Dijkstra, 1976] si todas las guardas de un conjunto de secuencias guardadas **if** $b_1 \rightarrow S_1 \,\square\, \ldots \,\square\, b_n \rightarrow S_n$ **fi** fallan se *aborta* la sentencia **if**.

Un derivado de PARLOG y CONCURRENT PROLOG es GHC (*Guarded Horn Clauses*) [Ueda, 1985], desarrollado en el instituto japonés ICOT; en GHC no se utilizan variables de solo lectura y una llamada es suspendida si espera una instanciación de otra; al igual que para FLAT–CONCURRENT PROLOG, las dificultades de implementación del mecanismo de suspensión da lugar a un subconjunto llamado FLAT–GHC, que ha sido utilizado para el desarrollo de la *parallel inference machine* (PIM) del proyecto japonés de ordenadores de la quinta generación [Meyrowitz, 1986].

Un derivado de PARLOG y GHC es PARLOG 86. En PARLOG 86 la unificación está restringida a variables de entrada; si en una llamada alguna variable no está instanciada para poder unificar, la llamada es suspendida hasta que otra llamada en cooperación la instancie. Esta separación de entrada/salida tiene la ventaja de que puede eliminarse el problema del o-paralelismo durante la espera hasta que una guarda de una cláusula sea verificada. Debemos especificar con una declaración **mode** si se pasa información a la regla (?) o si una regla devuelve información (↑). Tales anotaciones se pueden escribir sin más precedidas de algún identificador para información del programador; así, si tenemos:

$mode\ característica(persona?, atributo\uparrow).$
$característica(juan, tolerante).$ $característica(pedro, listo).$
$característica(juan, pesado).$ $característica(ana, hermosa).$

el objetivo $\leftarrow\ característica(juan, X)$ selecciona en forma indeterminista cualquiera de las reglas que puedan unificar (en este caso la 1 y la 3) e instancia la variable X a *tolerante* o *pesado*. Es de notar que una vez seleccionada una de las reglas las restantes son descartadas. Un objetivo como $\leftarrow\ característica(N, X)$ suspende el proceso hasta que la variable N esté instanciada. Se puede especificar que las reglas se seleccionen en forma secuencial (como en PROLOG) cambiando el carácter "." por el carácter ";" en la forma:

$característica(juan, tolerante);$ $característica(pedro, listo);$ ...

y en ese caso la selección es determinista.

Todos los operadores primitivos tienen un modo por defecto; así tenemos para el test de igualdad $mode\ ==(?,?)$, al igual que para el resto de comparadores $<, >, \ldots$ y la negación por fallo $mode\ not(?)$. Esto significa que el objetivo $\leftarrow X == 3$ quedaría suspendido. En el conjunto de objetivos paralelos:

$$\leftarrow característica(juan, X), not(X == tolerante).$$

se deja suspendido el segundo proceso hasta que X ha sido instanciada; podemos forzar la invocación secuencial sustituyendo "," por "&":

$$\leftarrow característica(juan, X)\&not(X == tolerante).$$

La simplificación de PARLOG en unificación tiene algunos inconvenientes; por ejemplo, la formulación:

$mode\ absoluto(entero?, valor\_absoluto).$
$absoluto(N, N)\ \leftarrow\ N >= 0.$
$absoluto(N, M)\ \leftarrow\ N < 0,\ M\ is\ -1 * N.$

no funciona correctamente ya que un objetivo tal como $\leftarrow\ absoluto(3, M)$ puede fallar si se selecciona la segunda regla. La solución a estos casos viene descrita, al igual que para CONCURRENT PROLOG, por guardas en las reglas:

$absoluto(N, N) \leftarrow N >= 0 : true.$
$absoluto(N, M) \leftarrow N < 0 : M\ is\ -1 * N.$

Redes de procesos

En PARLOG es posible describir los canales para la comunicación a través de mensajes vía procesos recursivos. A modo de ejemplo, sea el siguiente predicado que genera una lista de números enteros (aleatorios):

$mode\ lista\_azar(cuantos?, lista).$
$lista\_azar(N, [\,]) \leftarrow N <= 0 : true.$
$lista\_azar(N, [A|R]) \leftarrow N > 0 : A\ is\ aleatorio,\ M\ is\ N + 1,\ lista\_azar(M, R).$

Se observa que, puesto que el segundo argumento es de salida, el proceso $lista\_azar(5, L)$ es suspendido hasta que es generado el primer aleatorio (el valor A en la regla); una vez generado el primer valor, si algún proceso cooperante con él está suspendido esperando la lista de salida, podrá proseguir si necesita tal valor.

Veamos un ejemplo (inspirado en [Conlon, 1989]:118). Sea el predicado que comprueba si los elementos de una lista de enteros son todos positivos:

$mode\ ent\_pos(lista?).$
$ent\_pos([\,]).$
$ent\_pos([B|R]) \leftarrow B > 0, ent\_pos(R).$

y sea ahora el objetivo $\leftarrow lista\_azar(20, L),\ ent\_pos(L)$. En ese caso, L actúa a modo de canal de comunicación entre los procesos; el proceso $lista\_azar(L)$ envía información a $ent\_pos(L)$ cada vez que ésta es parcialmente generada; por consiguiente, si en algún paso se genera un número negativo, puede fallar el objetivo antes de generar toda la lista. Así, el problema del productor-consumidor puede expresarse como una conjunción paralela de dos objetivos $\leftarrow prod(C),\ cons(C)$ donde:

$prod([\,A \mid R\,]) \leftarrow producir(A),\ prod(R).$
$cons([\,B \mid R\,]) \leftarrow consumir(B),\ cons(R).$

Finalmente, el problema de ordenación antes comentado se escribe en PARLOG 86 en la forma:

$mode\ ordenar(lista?, listaord\uparrow).$
$ordenar([\,], [\,]).$
$ordenar([H|T],\ Ordenada] \leftarrow partir((T, H, L, G),$
$\qquad ordenar(L, LO),\ ordenar(G, GO),$
$\qquad concatenar(LO, [H|GO],\ Ordenada).$

$mode\ partir(lista?, pivote?, izq\uparrow,\ der\uparrow).$
$partir([\,], X, [\,], [\,]).$
$partir([H|T], X, [H|L], G) \leftarrow H <= X : partir(T, X, L, G).$
$partir([H|T], X, L, [H|G]) \leftarrow H >= X : partir(T, X, L, G).$

$concatenar([\,], X, X).$
$concatenar([X|R],\ Y,\ [X|Z]) \leftarrow concatenar(R,\ Y, Z).$

Programación Concurrente	Programación Lógica
sistema de comunicación	petición con variables compartidas
buffer (mensajes)	variable lógica
canal	lista
indeterminismo	multiples reglas
sincronización	unificación en entrada vía *pattern matching*

Tabla 3: Programación Concurrente versus Programación Lógica

Como curiosidad puede verse en [Ringwood, 1988] el problema de los cinco filósofos resuelto en PARLOG 86.

Como resumen podemos observar en la Tabla 3 la interpretación de algunos tópicos de la programación concurrente desde la $P_{RO}L_{OG}$.

Es evidente que el *commit choice* provoca que se pierda la completitud de los programas lógicos. En resumen, como dice [Ueda, 1999]

$$\text{Concurrent LP} = \text{LP} + \text{committed choice} = \text{LP - completeness}$$

Los lenguajes para la Programación concurrente con restricciones [Saraswat, 1989] o CCP (*Concurrent Constraint Programming*) son también mecanismos potentes: descripción de procesos distribuidos y en redes, tiempo real y computación móbil (*mobile computation*). El esquema que se obtiene al añadir los tres paradigmas es la Programación lógica concurrente con restricciones o CCLP (*Concurrent logic/Constraint Programming*). Como defiende [Ueda, 1999], la CCLP tiene sus propios desafíos: (1) una contrapartida al λ–cálculo en el campo de la concurrencia, (2) una plataforma común para varias formas de computación no secuenciales, (3) sistemas de tipos que cubren tanto los aspectos físicos de la computación, como los lógicos.

0.2.7 Programación lógica y Programación Orientada a Objetos

Las componentes del paradigma Programación Lógica son: cláusulas, objetivos, variables lógicas, unificación, etc., y las del paradigma Programación OO son: objetos, clases, métodos, mensajes, herencia, etc. Para unir estos paradigmas hay que establecer correspondencias entre tales componentes, al igual que hicimos anteriormente entre $P_{RO}L_{OG}$ y programación concurrente. Una forma de empotrar objetos en PROLOG es como conjunto de reglas en una base de datos:

$$nombre(o,\ juan). \quad edad(o,\ 24). \quad madre(o,\ o2).$$

Así, o será una referencia a un objeto con variables de instancia *nombre*, *edad* y *madre*. Esta aproximación tiene problemas conceptuales (diferencia entre variables de instancia y métodos, etc.) y prácticos (puesto que los objetos existen como asertos, la recolección de basura es difícil).

Durante los últimos 15 años se han propuesto varias formas de integrar la $P_{RO}L_{OG}$ y la POO[97]. Los diferentes métodos empleados dan visiones diferentes para la representación de estados, clases, herencia, paso de mensajes, etc. Según [Davison, 1992], podemos clasificar los modelos en tres grupos: (1) metainterpretación de clases y objetos, (2) clases como conjuntos de cláusulas, y (3) objetos como procesos perpetuos.

[97] Puede verse una visión general en el Cap. 3 de [Pimentel, 1995].

Dentro del grupo (1) aparecen MANDALA [Furukawa et al., 1986], un lenguaje que vía un metaintérprete (que genera código para alguna extensión de PROLOG) permite clases y métodos con alta expresividad. El problema de estas propuestas es la dificultad para conseguir implementaciones eficientes.

Dentro del grupo (2) aparecen distintas extensiones de PROLOG, donde un mensaje a un objeto consiste en la resolución de un objetivo dentro de un conjunto de cláusulas donde interviene una referencia al objeto. La herencia se implementa enlazando conjuntos de cláusulas. Un ejemplo de éste es SPOOL [Fukunaga y Hirose, 1987].

Dentro del grupo (3) aparecen las propuestas que contienen a CONCURRENT PROLOG [Shapiro, 1983; Shapiro y Takeuchi, 1983] como lenguaje base, en las cuales un objeto es identificado con un proceso perpetuo [Lloyd, 1984]. Esta concepción ha evolucionado hacia diferentes propuestas, entre las que podemos citar VULCAN [Kahn et al., 1987], POLKA [Davison, 1989] y $L^2||O^2$ [Pimentel, 1993]. También cabe citar las propuestas que integran la Programación Concurrente con restricciones vía lógica y objetos, una de las cuales es OZ.

Comencemos con las extensiones para la concurrencia en PROLOG ya que permiten una asociación de conceptos más simple. Así, CONCURRENT PROLOG puede verse como un lenguaje basado en objetos si asociamos la identidad de objetos ciertos procesos perpetuos y los canales de comunicación como variables compartidas que transportan mensajes. La variable lógica representando el canal es tratada como un puntero al objeto; por ello tales variables se llaman variables de flujo (*stream variables*) y se denotan con el prefijo 'New' (típicamente será la variable usada en recursividad por la cola). Un mensaje sobre tal objeto será identificado con [*Mensaje* | *Variable*] y el uso de funtores permite seleccionar el método a aplicar, sus parámetros y el flujo sobre el que actúa:

$$[depositar(Cantidad)|NewCuenta] \qquad [saldo(Saldo)|NewCuenta]$$

Esto permite construir plantillas para métodos con varias reglas. [Kahn et al., 1987] exponen como ejemplo las siguientes reglas para definir una serie de métodos (análogos a los utilizados para SMALLTALK o ADA) que pueden actuar sobre canales (objetos) asociados a una cuenta bancaria :

$$\begin{array}{l} contabilizar\ (\ [depositar(Cantidad)|NewCuenta], Saldo,\ Nombre,\ \ldots\ ) :- \\ \quad plus(Saldo,\ Cantidad,\ NewSaldo), \\ \quad contabilizar(\ NewCuenta?,\ NewSaldo,\ Nombre,\ \ldots\ ). \end{array}$$

$$\begin{array}{l} contabilizar\ (\ [saldo(Saldo)|NewCuenta], Saldo,\ Nombre,\ \ldots\ ) :- \\ \quad contabilizar(\ NewCuenta?,\ Saldo,\ Nombre,\ \ldots\ ). \end{array}$$

$$\begin{array}{l} contabilizar\ (\ [sacar(Cantidad)|NewCuenta], Saldo,\ Nombre,\ \ldots\ ) :- \\ \quad Saldo\ >=\ Cantidad\ |\ plus(Cantidad,\ NewSaldo,\ Saldo), \\ \qquad\qquad\qquad\qquad contabilizar(\ NewCuenta?,\ NewSaldo,\ Nombre,\ \ldots\ ). \end{array}$$

$$\begin{array}{l} contabilizar\ (\ [sacar(Cantidad)|NewCuenta], Saldo,\ Nombre,\ \ldots\ ) :- \\ \quad Saldo\ <\ Cantidad\ |\ informarNumerosRojos(\ Nombre,\ Saldo,\ Cantidad\ ), \\ \qquad\qquad\qquad\qquad contabilizar(\ NewCuenta?,\ Saldo,\ Nombre,\ \ldots\ ). \end{array}$$

Por ejemplo, si tenemos el objetivo (hemos suprimido el resto de datos, *Nombre*, etc.):

$$contabilizar(A?, 100),$$
$$A = [depositar(100), saldo(S), sacar(200)|NewCuenta].$$

el proceso *contabilizar* queda suspendido; el segundo instancia inmediatamente parte de la variable A por lo que el primer proceso anterior continua, unificando solamente con la primera regla y queda reducido a:

$$plus(Cuenta, Cantidad, NewC1),$$
$$contabilizar([saldo(S), sacar(200)|NewCuenta], NewC1?].$$

Así, tales objetos en CONCURRENT PROLOG son activos (al contrario que en FLAVORS, LOOPS o SMALLTALK) y procesan mensajes de uno en uno (al contrario que en los actores). La técnica utilizada en la comunicación se llama *mensajes incompletos* (*incomplete messages*) [Shapiro y Takeuchi, 1983]; es la forma más común de enviar mensajes y recibir respuestas.

El punto de vista anterior es complicado de utilizar directamente en POO ya que utiliza una verbosidad excesiva; se han dado varias soluciones al respecto describiendo preprocesadores para CONCURRENT PROLOG, tales como VULCAN [Kahn et al., 1987] o MANDALA [Furukawa et al., 1986]. Por ejemplo, en VULCAN las cláusulas *class* declaran los nombres de las variables de estado de todas las instancias de la clase, como en *class*(*ventana*, [*X*, *Y*, *Alt*, *Anc*, *Contenido*]). Para generar instancias se utilizan cláusulas *make*:

$make(ventana,\ [X, Y, Alt,\ Anc,\ Contenido],\ MiVentana) :-$
$\quad ventana(MiVentana?,\ X, Y, Alt,\ Anc,\ Contenido]).$
$/* \ crea\ un\ proceso\ asoci\acute{a}ndole\ el\ flujo\ MiVentana \ */$

Las cláusulas *method* permiten generar métodos. Por ejemplo

$$method(\ ventana,\ origen(X, Y))$$

Las expresiones para alterar el estado de un objeto son de la forma

$$become(\ (Var1,\ valor1),\ (Var2, valor2),\ \ldots)$$

y son interpretadas como asignaciones concurrentes. Así, la posición de una ventana puede alterarse con la declaración de método:

$method(\ ventana,\ desplazarSegún(Dx, Dy)\ ) :-$
$\quad plus(\ X,\ Dx,\ Nx),\ plus(\ Y,\ Dy,\ Ny),\ become((X, Nx),\ (Y, Ny)\ ).$

la cual generaría el siguente flujo:

$ventana(\ [desplazarSegún(Dx, Dy) \mid NewVentana], X, Y, Alt, Anc,\ Contenido)\ :-$
$\quad plus(X, Dx, Nx),$
$\quad plus(Y, Dy, Ny),$
$\quad ventana(NewVentana?,\ Nx?,\ Ny?,\ Alt,\ Anc,\ Contenido).$

El envío de mensajes también se especifica de forma especial: *send*(*ObjetoReceptor*, *Mensaje*). Por ejemplo, tenemos la versión mejorada para desplazar una ventana borrando de la pantalla el contenido anterior:

$method(\ ventana,\ desplazarSegún(Dx, Dy)\ )\ :-$
$\quad send(Self,\ borra),$
$\quad plus(\ X,\ Dx,\ Nx),\ plus(\ Y,\ Dy,\ Ny),$
$\quad become(\ (X, Nx),\ (Y, Ny)\ ),\ send(\ Self,\ muestra).$

donde suponemos que *borra* y *muestra* hacen lo apropiado. La pseudo-variable *Self* colocada después de *send* o *become* hace referencia al nuevo objeto (receptor del mensaje).

A diferencia de Smalltalk, Concurrent Prolog no tiene punteros a objetos (*inmutable streams vs. pointers to mutable objects*). Si dos procesos envían diferentes mensajes al mismo flujo, el primero actúa normalmente pero el segundo falla al no poder unificar.

Vulcan implementa la herencia de dos formas; la primera por subclasificación y la segunda por delegación. Por ejemplo: *class*(*ventanaEtiquetada*, [*etiqueta*], *ventana*) donde el tercer argumento es la superclase, que es creada copiando todos los métodos de la superclase. La delegación consiste en suponer que una *ventanaEtiquetada* contiene a una ventana como una parte: $class(ventanaEtiquetada, [Ventana, Etiqueta])$. de forma que un método no aceptado explícitamente por *ventanaEtiquetada* es *delegado* al contenido *ventana* referenciado en la variable de estado, aunque esto tiene algunos inconvenientes al heredar métodos generados por el preprocesador.

Comentemos otras notaciones para la POO basadas en Prolog. Una de ellas es Orient 84/K [Ishikawa y Tokoro, 1987], un lenguaje de programación concurrente para describir sistemas de conocimiento. En Orient 84/K un objeto es una unidad modularizada de conocimiento, llamada KO (*Knowledge Object*) y se compone de un *comportamiento*, una *base de conocimiento* (conocimiento local) y un *monitor*; el monitor controla el paso de mensajes y supervisa el comportamiento. La sintaxis de la KO es parecida a Prolog y la sintaxis y semántica del comportamiento parecida a Smalltalk.

Otra notación mezcla de Prolog y Smalltalk es SPOOL [Fukunaga y Hirose, 1987]; de éste artículo extraemos el siguiente ejemplo para ilustrar las notaciones:

$class\ empleado\ has$
$super - class\quad root - class;$
$instance - var\ nombre;$
$methods$
$\quad peticion(Sujeto)\ :-\ nombre\ ::\ M\ \&\ M\ <<\ consulta(Sujeto, Resp);$
$end.$

$class\ manejador\ has$
$super - class\ empleado;$
$methods$
$\quad consulta(S, admitido)\ :-\ importante(S);$
$\quad consulta(S, rechazado)\ :-\ indeseable(S);$
$\quad consulta(S, pendiente);$

$\quad importante(jefe);$
$\quad importante(sereno);$
$\quad indeseable(muy - costoso);$
$end.$

donde se describe una clase *empleado* y una subclase suya *manejador*; el mensaje $nombre\ ::\ M$

es satisfecho si el receptor del mensaje tiene una variable de instancia de identificador *nombre* que puede ser instanciada con M. En el mensaje $M << selector$, M representa el receptor. Para crear instancias se utiliza una sintaxis tal como:

$$instance\ trabajador\ is - a\ empleado\ where\ nombre\ :\ juan$$

Así pues, el mensaje (objetivo) $trabajador << peticion(sereno)$ es satisfecho ya que la invocación $peticion(sereno)$ sobre la instancia *trabajador* también es satisfecha unificando la variable *Resp* al valor *admitido*.

0.3 Integración de los paradigmas lógico y funcional

La unificación o la combinación de los paradigmas lógico y funcional ha sido muy abundante en la literatura[98] [Darlington, 1983; Hoare y Jifeng, 1998].

¿Qué ventajas puede tener la integración de ambos paradigmas?

La primera es que un lenguaje con tales características establece un puente entre la $\mathrm{P_{RO}L_{OG}}$ y la $\mathrm{P_{RO}F_{UN}}$, y los investigadores pueden cooperar uniendo sus esfuerzos.

La segunda es que puede facilitar la enseñanza de la programación, ya que los estudiantes trabajarían con un único lenguaje declarativo, en lugar de dos, como actualmente. Esto permite introducir la $\mathrm{P_{RO}D_{EC}}$ en forma incremental, comenzando con la visión funcional, pero pudiendo introducir la componente lógica desde el comienzo. Todo ello queda reforzado si vemos que casi todas las propuestas utilizan un estilo ecuacional para la descripción[99]

La principal desventaja es que las principales propuestas actuales son ineficientes, y no están consensuadas. Sobre la ineficiencia [Lloyd, 1995] opina que tal aspecto puede ser compensado con la economía de esfuerzo y tiempo dedicado a la programación y el mantenimiento.

Durante los últimos 15 años, la comunidad docente que enseña la $\mathrm{P_{RO}L_{OG}}$ no discute sobre la posibilidad de PROLOG como primer lenguaje para la enseñanza, y la razón principal que aducen es que las características no lógicas[100] complican la práctica de la programación para los no iniciados, además de la pérdida de otras características de los lenguajes declarativos, como puede ser el paralelismo implícito, que queda destruido por los predicados *var*, *nonvar*, *assert*, etc. Sin embargo, los docentes de la $\mathrm{P_{RO}F_{UN}}$ opinan lo contrario: un lenguaje moderno, como HASKELL, facilita el aprendizaje. En este sentido los investigadores defienden que el lenguaje que integre ambos paradigmas debe disponer de tipos, módulos, etc., y debe eliminar las características no lógicas de PROLOG, sin perder la expresividad de los lenguajes lógicos (p.e., ESCHER).

[98] Una revisión puede verse en [Hanus, 1994], además de la interesante página mantenida por M. Hanus `http://www-i2.informatik.rwth-aachen.de/ ~hanus/ FLP/ implementations.html`.

[99] Al igual que John [Lloyd, 1995], considero que ésta es la clave de la integración. Así, la componente lógica en la $\mathrm{P_{RO}L_{OG}}$ son fórmulas ($p : -obs$); en la $\mathrm{P_{RO}F_{UN}}$ también son fórmulas: $f\ x\ y = x + y$; pero en todas las propuestas para los lenguajes lógico/funcionales la descripción viene dada por fórmulas; bien descritas en forma pura al estilo de ESCHER, como $finSemana(x) \Rightarrow x = Dom \vee x = Sab.$, donde el símbolo $\Rightarrow$ *captura* la reescritura (no la forma clausal); bien condicionales al estilo (por ejemplo) de TOY, como $f\ a\ b = a + b <== a > 0, q(b).$, o al estilo de LIFE, como $fact(N) \rightarrow N * fact(N-1) | noNegativo(N).$ La elección del símbolo para denotar reescritura es trascendente ya que *refleja* la intención o la semántica.

[100] De hecho, la práctica con GÖDEL prueba que es falso el argumento de que ciertas características no lógicas son necesarias para hacer que los lenguajes lógicos sean eficientes.

La combinación de ambos paradigmas en un único lenguaje puede realizarse básicamente de dos formas:

1. Sumergir un lenguaje lógico dentro de uno funcional trasladando cada predicado a una función, como ya propusieron [Robinson y Sibert, 1982; Robinson, 1988] para los lenguajes LOGLISP y SUPER [Robinson y Greene, 1987].

2. Definir un nuevo lenguaje donde la estrategia de evaluación incluya resolución/unificación y reescritura.

Dentro del primer grupo aparecieron las primeras propuestas, y además de las citadas, existen otras propuestas muy recientes como las descritas en [Seres y Spivey, 1999; Day et al., 1999]. Desgraciadamente estas propuestas no están suficientemente experimentadas ni aceptadas. El problema es modelar la variable lógica y sacar partido al indeterminismo que ello conlleva. Este aspecto será tratado también en la Sección 0.4.

Dentro del segundo grupo aparecen las propuestas más prometedoras[101] entre las cuales describiremos las más difundidas (aunque por cierto, no hay consenso en éstas).

La forma natural de integración es considerar la unión de un conjunto H de cláusulas de Horn y un conjunto E de ecuaciones condicionales para obtener un programa P. Así, la semántica declarativa de P viene dada por una lógica de primer orden con igualdad o teoría ecuacional[102] descrita por E. Mendelson en 1979. Estas teorías extienden la lógica con un símbolo de igualdad junto a sus axiomas estandarizados, y la semántica operacional viene dada por un conjunto de reglas de inferencia, de forma que el sistema es completo y correcto [Lloyd, 1984]. Es posible extender el principio de resolución pero el espacio de búsqueda resultante es enorme, de forma que otra solución es extender la unificación sintáctica módulo la teoría ecuacional para obtener la *unificación semántica* [Siekmann y Szabó, 1984; Siekmann, 1989]. Tal mecanismo resultó ser muy limitado.

Una segunda posibilidad es considerar a E como un conjunto de ecuaciones orientadas o reglas de reescritura de forma que los argumentos en una llamada podrían reducirse en forma perezosa o impaciente. Quizás la primera propuesta en ese sentido fue FUNLOG [Subrahmanyam y You, 1984], que modifica la idea de unificación semántica.

Otras propuestas con un mecanismo de unificación parecido son LEFUN[103] [Aït-Kaci y Nasr, 1987], LIFE [Aït-Kaci y Lincoln, 1989] y L&O [McCabe, 1993].

La tercera posibilidad es una extensión natural del concepto de reducción llamada estrechamiento (*narrowing*) [Hullot, 1980b; Hullot, 1980a], y operacionalmente tal mecanismo es una variante de la SLD-resolución con resolventes de ecuaciones, y que al parecer fue introducido por primera vez por John [Slagle, 1974][104]. El estrechamiento consiste en aplicar

[101] Entre los años 80 y 90 se describieron/desarrollaron al menos las propuestas: FUNLOG, LOG(F), SLOG, EQLOG, LEAF, LEFUN, BABEL y λPROLOG, mientras que en la década de los 90 el número de propuesta es enorme.

[102] Puede verse una introducción corta (unas 10 páginas) en el capítulo 13 de [Nilsson y Maluszynski, 1995]. Sin embargo, la cita obligada en este caso es [Lassez et al., 1988], al menos desde un punto de vista histórico.

[103] Logic, Equations and Functions.

[104] No parece claro si la idea la toma J. Slagle al finalizar el diseño del sistema SAINT (Symbolic Automatic INTegrator), escrito en LISP en el MIT en la década de los 60 [Sammet, 1969] donde coincide con J. McCarthy, S. Papert y M. Minsky, y precisamente durante la época previa al nacimiento de PROLOG. Si esto fuera así, al igual que ocurriría con la controversia ALGOL–SIMULA–PASCAL, de nuevo hemos retrocedido casi 20 años. En mi opinión y la de otros autores, el desarrollo de la $\text{P}_{\text{RO}}\text{R}_{\text{ES}}$, que actualmente

la sustitución mínima con objeto de hacer reducible una expresión para reducirla posteriormente. Tal sustitución se obtiene unificando con el miembro izquierdo de una regla de reescritura. Por ejemplo, para el programa[105]:

$$\begin{array}{lcl} suma(O, Y) & = & Y \\ suma(s(X), Y) & = & s(suma(X, Y)) \end{array}$$

la expresión $suma(Z, W)$ puede ser contraída (*narrowed*) a W vía la sustitución $\{Z/0\}$, y también puede contraerse a $s(suma(V, W)$ vía la sustitución $\{Z/s(V)\}$. Da tal forma el estrechamiento puede servir para dotar de una semántica operacional a los lenguajes funcionales [Reddy, 1985], o también simula la resolución si el conjunto de cláusulas H es visto como un conjunto de reglas de reescritura. Tal mecanismo o similares han servido para implementar varias propuestas y lenguajes, como LEAF [Barbuti et al., 1986], SLOG [Fribourg, 1985] y EQLOG [Goguen y Meseguer, 1986].

Babel, Toy y Curry

De los anteriores lenguajes, quizás sea BABEL[106] [Moreno Navarro y Rodríguez Artalejo, 1988; Moreno Navarro y Rodríguez Artalejo, 1992] uno de los que sigue *vivo*, mientras que otros han derivado a otros lenguajes.

BABEL está basado en estrechamiento. Otros ejemplos notables que usan distintas especializaciones de *narrowing*, como *needed narrowing* [Antoy et al., 1994; Antoy et al., 2000] o *lazy narrowing* [Loogen et al., 1993], son CURRY [Hanus, 1997] y TOY [López-Fraguas y Sanchez-Hernandez, 1999; Abengózar et al., 2000][107]. Una revisión de la técnica *lazy narrowing* puede estudiarse en [Cheong y Fribourg, 1993], donde además expone como simularla en PROLOG.

Los tres lenguajes citados usan una sintaxis mixta heredada de PROLOG. El siguiente ejemplo (extraído de la página web de BABEL) ilustra como calcular los números de Fibonacci vía listas infinitas:

$$\begin{array}{lcl} fibnbs\ X\ Y & := & [X|\, fibnbs\ Y\ (X + Y)]. \\ fiblist & := & fibnbs\ 1\ 1. \\ fib\ X & := & toma\ X\ fiblist. \\ toma\ X\ [Y|Ys] & := & \textbf{if}\ (X = 0)\ \textbf{then}\ Y\ \textbf{else}\ toma\ (X - 1)\ Ys. \end{array}$$

donde observamos que, salvo la sintaxis (variables en mayúsculas, sintaxis para lista à la PROLOG de la forma $[\,X|\,\ldots]$, etc.) es código estándar para un lenguaje funcional. El siguiente ejemplo resuelve el problema de la N–reinas, mediante el uso de funciones y predicados:

---

se acepta conceptualmente más interesante que la unificación pura, ha sido paralelo al desarrollo de la PROLOGFUN, al menos, a la integración de los dos paradigmas, e incluso al desarrollo de PROLOG. La aparición de la máquina WAM quizás inclinó el esfuerzo de los investigadores hacia PROLOG.

[105] Los ejemplos que siguen están extraídos de [Cheong y Fribourg, 1993]. Una introducción corta a la técnica de estrechamiento puede verse en [Hanus, 1997], donde expone una semántica operacional sencilla para CURRY.

[106] `http://www-i2.informatik.rwth-aachen.de/babel/`.

[107] El estado actual de TOY puede verse en `{titan,mozart}.sip.ucm.es/toy`, y la situación actual del lenguaje CURRY puede verse en `http://www.informatik.uni-kiel.de/~curry/`.

$$queens(N, Ds) :- queens1(range\ 1\ N, [\,], Ds).$$
$$range\ N1\ N2\ :=\ \textbf{if}\ (N1\ =\ N2)\ \textbf{then}\ [N1].$$
$$range\ N1\ N2\ :=\ \textbf{if}\ (N1\ \sim=\ N2)\ \textbf{then}\ [N1|range\ (N1+1)\ N2].$$

y ¡los argumentos de predicados pueden sustituirse por llamadas a funciones! La función *range* utiliza formas guardadas, y devuelve la lista de reinas a colocar en el tablero, mientras el predicado *queens*1 las coloca:

$$queens1([\,], Ds, Es)\ :- Ds = Es.$$
$$queens1(Dsunp, Dssicher, Ds) :- select(D,\ Dsunp,\ Ds1unp),\ noattack(D, Dssicher),\ queens1(Ds1unp, [D|Dssicher], Ds).$$
$$select(Y, [X|Xs], Zs)\ :- X = Y,\ Zs = Xs.$$
$$select(Y, [X|Xs], [Z|Zs])\ :- X = Z,\ select(Y, Xs, Zs).$$

El predicado *select* (para listas no vacias) permite probar con cada reina, y el siguiente es el clásico:

$$noattack(X, Xs)\ :- noattack1(X, 1, Xs).$$
$$noattack1(X, N, [\,]).$$
$$noattack1(X, N, [Y|Ys]) :- (\ (X\ \sim=\ (Y+N),\ (X\ \sim=\ (Y-N))\ ),\ noattack1(X, (N+1), Ys).$$

Tomando la regla $queens(N, Ds) :- queens1(range\ 1\ N, [\,], Ds), false.$ obtendríamos todas las soluciones vía backtracking.

En Toy, la técnica de estrechamiento perezoso permite implementar la evaluación perezosa de los lenguajes funcionales y la SLD-resolución de los lenguajes lógicos, aunque el lenguaje está implementado en Sicstus-Prolog. Además añade otras características, como funciones indeterministas y patrones de orden superior. El sistema de tipos del lenguaje está basado en el sistema polimórfico de [Damas y Milner, 1982]. En Toy las definiciones de funciones toman la forma

$$f\ t_1\ \ldots\ t_n\ =\ r\ <==\ C_1, \ldots, C_n$$

y la sección $<==\ C_1, \ldots, C_n$ es opcional. Al igual que Haskell, la parte izquierda no puede contener variables repetidas y las C_i son condiciones (restricciones) de la forma $a \Diamond b$, donde $\Diamond \in \{==, /=, <, >, <=, >=\}$. Sin embargo, la sintaxis es una mezcla de Prolog y notación parcializada (que por otro lado no tiene mucho sentido como rápidamente veremos). Así, la función *append* se escribe en la forma:

$$append\ [\,]\ \ Ys\ =\ Ys$$
$$append\ [\ X\ |\ Xs\ ]\ Ys\ =\ [\ X\ |\ append\ Xs\ Ys\ ]$$

Un objetivo puede ser una expresión ordinaria (sin variables) o con variables pero en el marco de una restricción; por ejemplo:

```
Toy> append Xs [2, 3] == [1, 2, 3]
  yes
  Xs == [ 1 ]
Toy> append [1, 2, 3] [4, 5, 6] == Xs
  yes
  Xs == [1, 2, 3, 4, 5, 6]
Toy> append [1, 2, 3] [4, 5, 6]
[ 1, 2, 3, 4, 5, 6 ]
```

```
TOY> append [1] Xs /= [1,2]
  yes
  { Xs /= [2] }
TOY> [X, 3 | R] == [Y | S]
  yes
  X == Y, S == [3 | R]
```

Así pues, salvo algunos detalles sintácticos (desafortunados a mi entender), la ecuaciones para la descripción de funciones siguen un estilo parecido a HASKELL. Los predicados son funciones especiales de tipo $\tau_1 \rightarrow \ldots \tau_n \rightarrow bool$, y se pueden escribir las cláusulas à la PROLOG en la forma típica $p\ t_1\ \ldots\ t_n\ :-\ C_1,\ \ldots,\ C_n$, aunque ésta son una forma abreviada de $p\ t_1\ \ldots\ t_n\ =\ true\ <==\ C_1,\ \ldots,\ C_n$. Todo esto es algo incómodo:

```
type persona = string
padreDe, madreDe :: (persona, persona) -> bool
padreDe ("pedro","juan") :- true
...
madreDe("pedro","maria") :- true
...

progenitorDe, antepasadoDe :: (persona, persona) -> bool
progenitorDe (X, Y) :-padreDe (X, Y)
progenitorDe (X, Y) :- madreDe (X, Y)
antepasadoDe (X, Y) :- progenitorDe (X, Y)
antepasadoDe (X, Y) :- progenitorDe (X, Z), antepasadoDe(Z, Y)
```

En la escritura de ecuaciones en forma clausal se permiten variables repetidas:

```
inserta X [Y|Ys]   [X, Y|Ys]   :- true
```

ya que la traducción a la forma funcional elimina las repeticiones:

```
inserta X [Y|Ys]   [X', Y'|Ys]   = true <== X == X', Y == Y'
```

La construcción de funciones indeterministas (multievaluadas) utiliza una declaración especial $(--{>})$:

```
padreDe, madreDe, progenitorDe, antepasadoDe :: persona -> persona
padreDe "pedro" = "juan"
...
madreDe "pedro" = "maria"
...
progenitorDe X --> padreDe X
progenitorDe X --> madreDe X
antepasadoDe X --> progenitorDe X
antepasadoDe X --> progenitorDe ( antepasadoDe X )
```

de forma que podemos escribir el objetivo:

```
TOY> antepasadoDe "juan" == A
```

Los constructores de tipos de HASKELL y los funtores de PROLOG son descritos con una sintaxis *mixta*:

```
data nat = z | s nat
esNat :: nat -> bool
esNat z     :- true
esNat (s N) :- esNat N


infixr 10 +.
(+.) :: nat -> nat -> nat
X +. z     = X
X +. (s Y) = s (X +. Y)
```

de forma que tenemos el diálogo:

```
TOY> esNat N
  yes
  N == z
more solutions [y]? y
  yes
  N == (s z)
...

TOY> s z +. (s (s z)) == R
  yes
  R == (s(s(s z)))
more solutions [y]? y
  no
```

TOY utiliza muchas ideas de HASKELL, pero no su sintaxis, ya que las ecuaciones deben tener una guarda (aunque esta sea *true*); al mezclar la sintaxis confunde al programador. Por otro lado la escritura de predicados à la PROLOG no conlleva que el sistema utilizará únicamente SLD-resolución, y de nuevo, el programador puede sentirse confundido. Otro inconveniente adicional es la imposibilidad de evaluar una expresión (funcional) sin introducir una restricción, lo que no contenta a los programadores funcionales. En definitiva, pienso que el lenguaje, independientemente de que su semántica sea más o menos acertada, desde el punto de vista pragmático no contenta a ambos.

L&O

Otra propuesta interesante es L&O [McCabe, 1993], un lenguaje lógico con funciones y objetos. La sintaxis para las reglas es parecida a la de RELFUN y otros lenguajes para la $P_{RO}L_{OG}F_{UN}$. Existe una única forma para describir las reglas. El lenguaje incorpora módulos y plantillas para clases (*class templates*) parametrizadas:

```
simple(Order) : {
  sort([]) = [].
  sort [E1|Ls]) = insert(E1, sort(Ls)).
  insert(E1, []) = [E1].
  insert(E1, [E|Ls]) = [E1, E|Ls] :- Order : less(E1, E).
  insert(E1, [E|Ls]) = [E|insert(E1, Ls] :- not Order : less(E1, E)
  }.
```

$descending : \{$
$\quad less(I1, I2) \ :- \ I1 > I2 \ \}.$

de tal forma que un posible objetivo puede ser $simple(descending) : sort([3,2,1]) =?$. Obsérvese el mecanismo de sobrecarga. Así mismo incorpora un mecanismo de especialización para la herencia; por ejemplo, si tenemos:

$animal : \{$
$\quad mode(walk).$
$\quad mode(hop). \ \}.$

$bird : \{$
$\quad mode(fly).$
$\quad legs(2). \quad \}.$
$bird \ <= \ animal.$

Entonces el objetivo $bird : mode(X)?$ devuelve como respuestas $X = fly$, $X = walk$ y $X = hop$. La semántica del lenguaje está basada en reglas de unificación donde interviene un sistema de distribución de etiquetas para las clases.

λProlog

Otra propuesta que integra la $P_{RO}L_{OG}$ y la $P_{RO}F_{UN}$ es λPROLOG; es desarrollado por [Nadathur y Miller, 1988] e implementado en su primera versión siguiendo el esquema de la máquina WAM [Nadathur y Jayaraman, 1989][108]. Utiliza un esquema de unificación de segundo orden para los términos del λ–cálculo, basado en las ideas de [Huet, 1975]. λPROLOG es hoy día un lenguaje ampliamente utilizado[109] en diferentes aplicaciones, como puede ser el campo de los demostradores de teoremas [Felty, 1997] (esta última referencia contiene un interesante tutorial de λPROLOG, del cual extraemos algunos ejemplos).

λPROLOG extiende a PROLOG de varias formas. En primer lugar, utiliza notación parcializada:

$append \ nil \ K \ K.$
$append \ (X :: L) \ K \ (X :: M) \ :- \ append \ L \ K \ M.$

λPROLOG> $append \ (1 :: nil) \ (2 :: nil) \ L.$
$L \ == \ (1 :: 2 :: nil).$

aunque el constructor de listas :: (desgraciadamente) no sea el adoptado para otros lenguajes. λPROLOG permite utilizar cuantificadores: $pi \ x \backslash \ \phi \ (\equiv \forall x.\phi)$, $sigmax \backslash \ \phi \ (\equiv \exists x.\phi)$, y éstos pueden aparecer en la hipótesis y en la conclusión. Por ejemplo, la sentencia "si todas las bacterias que contienen un recipiente están muertas, entonces el recipiente es estéril", se escribe en λPROLOG en la forma[110].

[108] Aunque actualmente existen máquinas abstractas particulares para la implementación, como la expuesta en [Nadathur y Mitchell, 1999].

[109] El estado del lenguaje puede verse en `http://www.cse.psu.edu/~dale/lProlog/`. Un *delicioso* informe sobre λPROLOG es [Ridoux, 1998].

[110] Aunque λPROLOG utiliza la palabra reservada *pi* para el cuantificador ∀, y *sigma* para el cuantificador existencial, para mejorar la expresividad, aquí utilizaremos los cuantificadores simbólicos. También, el separador \ será reemplazado por .; de la misma forma, la abstracción $\lambda x.t$ es descrita en λPROLOG en la

$est\acute{e}ril\ R\ :-\ \forall\ B \setminus (\ (bacteria\ B,\ contenida\ B\ R)\ =>\ muerta\ B).$

El siguiente código contiene la descripción de caminos en un grafo, mostrando las diferentes posibilidades para la cuantificación explícita:

$type\ arco,\ camino\quad nodo\ \rightarrow\ nodo\ \rightarrow\ o.$
$camino\ X\ Y\ :-\ \exists\ z\ .\ (arco\ X\ z,\ camino\ z\ Y).$
$\forall\ x\ .\ (\forall\ y\ .\ (\ camino\ x\ y\ :-\ \exists\ z\ .(arco\ x\ z,\ camino\ z\ y))).$

λPROLOG> $\exists\ x\ .\ (camino\ a\ x).$
...

Por ejemplo, los siguientes predicados definen las reglas de derivación (del cálculo diferencial) para la derivada de la función identidad, y la derivada de una suma:

$d\ \lambda\ x\ .\ x\qquad\qquad\lambda\ x\ .\ (s\ 0)\ .$
$d\ \lambda\ x\ .(\ (A\ x)\ +\ (B\ x))\quad\lambda\ x\ .(\ (A'\ x)\ +\ (B'\ x))\ :-\ d\ A\ A',\ d\ B\ B'.$

El siguiente ejemplo es una versión λPROLOG de un predicado que reconstruye una lista aplicando cierto predicado, o también la versión de ∀:

$type\ mappred\quad (A\ \rightarrow\ B\ \rightarrow\ o)\ \rightarrow\ list\ A\ \rightarrow\ list\ B\ \rightarrow\ o.$
$mappred\ P\ nil\ nil.$
$mappred\ P\ (X : L)\ (Y : K)\ :-\ P\ X\ Y,\ mappred\ P\ L\ K.$

$type\ forevery\quad (A\ \rightarrow\ o)\ \rightarrow\ list\ A\ \rightarrow\ o.$
$forevery\ P\ nil.$
$forevery\ P\ (X : L)\ :-\ P\ X,\ forevery\ P\ L.$

Lógicamente, los predicados son inversibles; así, si tenemos:

$type\ edad\quad persona\ \rightarrow\ int\ \rightarrow\ o.$
$edad\ pepe\ 23.$
$edad\ ana\ 24.$
$edad\ jose\ 23.$

entonces tendremos el siguiente diálogo:

λPROLOG> $mappred\ edad\ (jose : pepe : ana : nil)\ L.$
$L\ ==\ (23 : 23 : 24 : nil).$

λPROLOG> $mappred\ edad\ L\ (23 : 23 : 24 : nil).$
$L\ ==\ (jose : pepe : ana : nil);$
$L\ =\ (pepe : jose : ana : nil).$

λPROLOG> $mappred\ (\lambda\ x\ .\ \lambda\ y\ .(edad\ y\ x))\ (23 : 24 : nil)\ K.$
$K\ ==\ (pepe : ana : nil);$
$K\ ==\ (jose : ana : nil).$

λPROLOG> $forevery\ (\lambda\ x\ .\ (\exists\ y\ .\ (edad\ x\ y)))\ (jose : pepe : ana : nil).$
$yes.$

forma $x \backslash t$, y para mejorar la expresividad añadiremos una λ, y cambiaremos el separador: $\lambda x\ .\ t$; también cambiaremos el constructor de listas :: por el constructor :

λPROLOG> $forevery\ (\lambda\ x\ .\ (edad\ x\ A))\ (jose : pepe : nil).$
$A\ ==\ 23.$

El siguiente ejemplo ilustra la potencia del concepto de unificación de segundo orden. Por ejemplo, dada la siguiente versión funcional de *map*:

$type\ mapfun\quad (A\ \rightarrow\ B)\ \rightarrow\ list\ A\ \rightarrow\ list\ B\ \rightarrow\ o.$
$mapfun\ F\ nil\ nil.$
$mapfun\ F\ (X : L)\ ((F\ X) : K)\ :-\ mapfun\ F\ L\ K.$
$type\ g\quad i\ \rightarrow\ i\ \rightarrow\ i.$
$type\ a, b\quad i.$

tendremos el siguiente diálogo:

λPROLOG> $mapfun\ (\lambda\ x\ .\ (g\ a\ x))\ (a : b : nil)\ L.$
$L\ ==\ ((g\ a\ a\ ) : (g\ a\ b) : nil).$

donde vemos que el intérprete forma los términos $(\lambda\ x\ .\ (g\ a\ x))a$ y $(\lambda\ x\ .\ (g\ a\ x))b$ reduciéndolos a través de la β–regla. La unificación de segundo orden permite reconstruir la función:

λPROLOG> $mapfun\ F\ (a : b : nil)\ ((g\ a\ a\ ) : (g\ a\ b) : nil).$
$F\ ==\ \lambda\ x\ .\ (g\ a\ x);$
$no.$

El cómputo con λ–términos envuelve unificación y conversión, pero no es posible reconstruir una función por patrones:

λPROLOG> $mapfun\ F\ (a : b : nil)\ (c : d : nil).$
$no.$

He aquí la correspondiente versión en λPROLOG de la función *fold*:

$type\ reducefun\quad (A \rightarrow\ B \rightarrow\ B)\ \rightarrow\ list\ A\ \rightarrow\ B\ \rightarrow\ B\ \rightarrow\ o.$
$reducefun\ F\ nil\ Z\ Z.$
$reducefun\ F\ (H : T)\ Z\ (F\ H\ R)\ :-\ reducefun\ F\ T\ Z\ R.$

de forma que tendremos el siguiente diálogo, donde observamos que la unificación no reduce la suma, pero ésta es reducida vía *is*:

λPROLOG> $reducefun\ (\lambda\ x\ .\ \lambda\ y\ .\ (x + y))\ (3 : 4 : 8 : nil)\ 6\ R,\ S\ is\ R.$
$R\ ==\ 3\ +(4\ +(8 + 6)).$
$S\ ==\ 21.$

La reconstrucción de la función de plegado puede obtenerse por unificación:

λPROLOG> $reducefun\ F\ (4 : 8 : nil)\ 6\ (1 + (4 + (1 + (8 + 6)))).$
$F\ ==\ \lambda\ x\ .\ \lambda\ y\ .\ (1 + (4 + (1 + (8 + 6))));$
$F\ ==\ \lambda\ x\ .\ \lambda\ y\ .\ (1 + (x + (1 + (8 + 6))));$
$F\ ==\ \lambda\ x\ .\ \lambda\ y\ .\ (1 + (x + y));$
$no.$

λPROLOG> $\forall\ z\ .\ (\ reducefun\ F\ (4 : 8 : nil)\ z\ (1 + (4 + (1 + (8 + z)))).$
$F\ ==\ \lambda\ x\ .\ \lambda\ y\ .\ (1 + (x + y));$
$no.$

λProlog dispone de un sistema de módulos con posible ocultación. Así, tenemos la siguiente definición de un módulo para trabajar con pilas:

```
module stack.
kind stack            type → type.
type empty            stack A → o.
type enter, remove    A → stack A → stack A → o.

local emp   stack A.
local stk   A → stack A → stack A.

emp.
enter X S (stk X S).
remove X (stk X S) S.
```

de forma que los predicados locales *emp* y *stk* permiten representar de forma interna las estructuras de datos. Al utilizar el anterior, las cláusulas de éste estarán disponibles durante el proceso de inferencia:

```
module int_stack.
import stack.
type test   int → int → o.
test A B :-
  ∃ S1 . (∃ S2 . (∃ S3 . (∃ S4 . (∃ S5 .
  (empty S1, enter 1 S1 S2, enter 2 S2, S3, remove A S3 S4, remove B S4 S5 ))))).

λProlog> test A B.
A == 2, B == 1.
```

Las aplicaciones más interesantes de λProlog son aquellas donde está presenta la *meta-interpretación*. Un análisis pragmático del papel de λProlog en la meta-interpretación puede verse en [Ridoux, 1998]:20-29, y ejemplos prácticos de usos podemos encontrarlos en el tutorial [Felty, 1997], donde relata de forma simple aplicaciones en demostradores de teoremas para la lógica de primer orden, cálculo de secuentes y sistemas de Gentzen, demostradores de teoremas con tácticas (al estilo de HOL, Isabelle, Coq o NuPRL), o incluso sistemas de reescritura de orden superior.

Gödel y Escher

Gödel [Hill y Lloyd, 1994] es en cierto sentido un sucesor de Prolog. Gödel permite el uso de fórmulas arbitrarias en el cuerpo de la cláusula, y dispone de un sistema de tipos con polimorfismo paramétrico basado en una lógica ecuacional heterogénea. Incorpora además un resolvente de restricciones sobre dominios finitos de enteros y racionales, así como facilidades metalógicas (uno de los puntos más débiles de Prolog puro) que le permite soportar metaprogramas y transformación de programas con una semántica declarativa clara. En particular, incorpora versiones declarativas de predicados tales como *setof* y *assert*, por lo que reduce el nivel entre teoría y práctica de la $P_{RO}L_{OG}$. Así mismo contiene otras extensiones interesantes, como módulos y construcciones para la concurrencia (un operador de poda que generaliza el operador *commit* de los lenguajes lógicos concurrentes). La principal desventaja de Gödel es que apenas está difundido fuera del marco académico.

Sin embargo, si PROLOG está lejos de la teoría, GÖDEL está demasiado alejado de la práctica.

Dentro de una línea muy distinta aparece ESCHER[111] [Lloyd, 1995]. ESCHER es un lenguaje que combina las principales características de GÖDEL, HASKELL y λPROLOG, con E/S declarativa, módulos (además de modulo del estilo del PRELUDE de HASKELL), orden superior y llamadas a predicados parcialmente instanciados. Tiene una semántica operacional clara y una lógica que es una extensión de la teoría de tipos simple de Church. La notación para el ecuaciones en ESCHER es $finSemana(x) \Rightarrow x = Dom \vee x = Sab.$, que tiene una ventaja: captura el indeterminismo vía disyunciones explícitas. La principal desventaja es la necesidad del uso de modos:

$$
\begin{array}{l}
FUNCTION\ Concat\ :\ List(a) * List(a) \rightarrow List(a).\\
MODE\ Concat(NONVAR, \_).\\
Concat(Nil, x) \Rightarrow x.\\
Concat(Cons(u, x), y) \Rightarrow Cons(u, Concat(x, y)).
\end{array}
$$

$$
\begin{array}{l}
FUNCTION\ Split\ :\ List(a) * List(a) * List(a) \rightarrow Boolean.\\
MODE\ Split(NONVAR, \_, \_).\\
Split(Nil, x, y) \Rightarrow x = Nil\ \&\ y = Nil.\\
Split(Cons(x, y), v, w) \Rightarrow\\
\quad (v = Nil\ \&\ w = Cons(x, y)) \vee\\
\quad SOME\ [z]\ (v = Cons(x, z)\ \&\ Split(y, z, w)).
\end{array}
$$

ESCHER no tiene un modelo computacional basado en resolución y utiliza reescritura. El indeterminismo se captura en la respuesta computada en forma disyuntiva:

$$
\begin{array}{l}
\text{ESCHER}> Split([L, M], x, y)\\
(x = [\,]\ \&\ y = [L, M]) \vee\\
(x = [L]\ \&\ y = [M]) \vee\\
(x = [L, M]\ \&\ y = [\,]).
\end{array}
$$

Recientemente, [Nadathur y Miller, 1998] retoman la programación lógico/funcional dentro del marco de la programación lógica de orden superior, como una teoría de tipos a la Church basada en cláusulas de Horn. Permite explotar de forma *real* la metaprogramación vía las fórmulas hereditarias de Ronald Harrop; éstas resultan ser una extensión de las fórmulas de Horn y son utilizadas como modelo de λPROLOG. Además, admiten un principio de resolución[112].

Alf

ALF[113] [Hanus, 1991a] es un lenguaje basado en cláusulas de Horn con igualdad [Hanus, 1990; Hanus, 1991b]. La semántica operacional mezcla resolución, reflexión (*reflection*), estrechamiento y reescritura [Hanus y Schwab, 1995]:15, resultando ser más eficiente que la SLD–resolución; incorpora también *backtracking* y la implementación se realiza en una extensión de la máquina WAM. Así, para programas PROLOG (sin funciones) el código

[111] En honor al pintor holandés M.C. Escher.
[112] Es muy interesante el tratamiento dado por [Ridoux, 1998] a estas fórmulas.
[113] Algebraic Logic Functional; ver `www-i2.informatik.rwth-aachen.de/~hanus/systems/ALF.html`.

es idéntico al código original de la máquina WAM (lo que asegura a los programadores PROLOG trabajar del mismo modo).

El sistema de tipos de ALF está basado en un lógica multievaluada con igualdad [Padawitz, 1988] y incorpora un sistema de módulos:

```
module nats.
  export 0, s, +, < .
  datatype nat = { 0, s(nat)}.
  func + : nat, nat → nat infixleft 500.
  pred < : nat, nat infix.


rules.
  N + 0    = N.
  N + s(M) = s(N + M).
  0    < s(M).
  s(N) < s(M) :- N < M.
end nats.
```

ALF permite la definición de módulos genéricos:

```
module list(a).
  export [ ], '.', append, ...
  use nats.
  datatype list = { '.'(a, list); [ ] }.
  func append : list, lists → list;
  pred member : a, list.


rules.
  append ([ ], L)      = L.
  append ([H|T], L) = [H| append(T, L)].
  member (E, [E|_]).
  member (E, [_|L]) :-member(E, L).
end list.


module qsort (a, (pred lesseq : a, a)).
  use list(a).
  func qsort : list → list.
  pred split : a, list, list, list.


rules.
      qsort ([ ]) = [ ].
      qsort([E|L]) = append(qsort(L1), [E|qsort(L2)]) :- split(E, L, L1, L2).


  split(E, [ ], [ ], [ ]).
  split(E, [E1|L], [E1|L1], L2) :- lesseq(E1, E), split(E, L, L1, L2).
  split(E, [E2|L], L1, [E2|L2]) :- lesseq(E, E2), split(E, L, L1, L2).
```

RELFUN

El lenguaje RELFUN[114] [Boley, 1996] es un lenguaje lógico que incluye llamadas a funciones que son evaluadas en forma impaciente, pero pueden ser indeterministas. Por tanto utiliza una resolución especial (*SLV-resolution*). De hecho existe una única declaración para funciones y predicados $p :- guarda \,\&\, valor.$ donde, o bien la guarda es vacía (y tenemos la declaración de una función) y se simplifica la declaración en la forma $p :\& valor$, o bien la parte $\&\ valor$ es vacía y obtenemos una cláusula ordinaria; lo interesante es que ambas secciones no sean vacías:

$fib(0) :\& 1.$
$fib(1) :\& 1.$
$fib(N) :- >(N,1) \,\&\, +(fib(-(N,2)); fib(-(N,1))).$

Una función, como extensión de un predicado, puede ser indeterminista:

$f(N) :\& N.$
$f(N) :\& sqrt(N).$
$f(N) :- Aux\ is\ f(-(N,1)) \,\&\, tree[N, Aux, N].$

de forma que la evaluación de la expresión $f(3)$ produce las soluciones 4, 2.0, $tree[4,3,2]$, $tree[4, 1.17, 4]$, ... El operador *tupof* permite recopilar es una lista las soluciones de la llamada a una función indeterminista.

Es posible utilizar estructuras pasivas $[P]$ y activas (A). Se incorpora una primitiva $.=$ que generaliza la unificación de una expresión con una estructura, así como se incorporan operadores especiales para cortes, etc. El lenguaje no dispone de un sistema de tipos, aunque sí permite trabajar con una jerarquía de *sorts*. Para ilustrar el lenguaje exponemos algunos ejemplos extraídos de la página Web de RELFUN. Así tenemos la siguiente descripción:

%% find[P](L) : type
%% P : type −− > Bool
%% L : [type]
%% aplica P a cada elemento de una lista devolviendo el primer valor
%% evaluado a True, o devuelve el indefinido si no existe ninguno

$find[P]([\,]) :\& none.$
$find[P]([H|T]) :- P(H)\,!\,\&\, H.$
$find[P]([H|T]) :\& find[P](T).$

Otro ejemplo que ilustra la notación es la siguiente función parametrizada que permite ordenar una estructura:

%% sort[P](L) : [type]
%% P : type type −− > Bool
%% L : [type]

[114] Relational-Functional language; `www.dfki.uni-kl.de/~vega/relfun.html`, `www.relfun.org/`.

$$
\begin{array}{l}
sort[P]([\,]) \ :\& \ [\,]. \\
sort[P]([H|T]) \ :- \ [L, G]. = partition[comp[P, H]](T), \\
\qquad\qquad\qquad\quad L - sorted. = sort[P](L), \\
\qquad\qquad\qquad\quad G - sorted. = sort[P](G) \\
\qquad\qquad\qquad\quad \& \ app(L - sorted, [H|G - sorted]).
\end{array}
$$

$$
comp[P, X](Y) \ :\& \ P(Y, X).
$$

$$
\begin{array}{l}
partition[P]([\,]) \ :\& \ [[\,], [\,]]. \\
partition[P]([H|T]) \ :- \ P(H), \ [L1, L2]. = partition[P](T), \ M. = tup(H|L1) \\
\qquad\qquad\qquad\quad ! \ \& \ [M, L2]. \\
partition[P]([H|T]) \ :- \ [L1, L2]. = partition[P](T), \ M. = tup(H|L2) \\
\qquad\qquad\qquad\quad \& \ [L1, M].
\end{array}
$$

Observamos que la notación es engorrosa (los paréntesis son necesarios, la parametrización – orden superior– no está incorporada al lenguaje, etc.). En el tutorial [Boley et al., 1996] aparecen otros ejemplos de uso de la jerarquía de sorts en forma dinámica. El sistema está implementado vía una máquina GWAM en un subconjunto de CommonLisP.

Life

Life[115] [Aït-Kaci y Lincoln, 1989; Aït-Kaci y Podelski, 1991] es un lenguaje experimental que integra los paradigmas: restricciones, lógica, funcional y objetos. Mezcla ideas de LeFun (lógica+funcional) [Aït-Kaci y Nasr, 1987] y de LOGIN (lógica+herencia) [Aït-Kaci y Nasr, 1986]. Está basado en una modificación del paradigma CLP (*Constraint Logic Programming*) [Jaffar y Maher, 1994]. Actualmente podemos disponer de Wild Life[116] [Aït-Kaci et al., 1994], un intérprete de Life escrito en C.

Del mismo modo que Prolog usa términos de Herbrand, Life está basado en ψ–términos[117]. Así, los (ψ)términos son estructuras que denotan conjuntos de valores y permiten introducir un concepto de cómputo con información parcial. Ejemplos de éstos son:

$$
\begin{array}{l}
fecha(viernes, "XIII"), \ fecha(1 \ \Rightarrow \ viernes, 2 \ \Rightarrow \ "XIII"), \\
rectángulo(ancho \ \Rightarrow \ S : int, \ largo \ \Rightarrow \ S), \\
X : persona(casa \ \Rightarrow \ dirección(ocupantes \ \Rightarrow \ [X]))
\end{array}
$$

Vemos que se usan etiquetas o atributos (*features*) para capturar los argumentos, y se permiten referencias cíclicas. Los (ψ) términos son registros extensibles y pueden refinarse. Por ejemplo, $círculo(origen \ \Rightarrow \ P)$ se refina en la forma $círculo(origen \ \Rightarrow \ P, \ radio \ \Rightarrow \ R)$. Es posible asociar un término t a una variable X en la forma $X : t$. La última asociación viene a decir que X es una *persona*, donde aparece un campo que debe ser una dirección, en la cual aparece otro campo que contiene una lista de personas; $X : [42|X]$ asocia a la variable X la lista infinita $[42, 42, \ldots]$.

[115] Logic, Inheritance, Functions, and Equations.
[116] `www.isg.sfu.ca/life/`.
[117] Para simplificar, en esta sección usaremos la palabra *término* para referirnos a un ψ–término.

Es posible asociar un *sort* a un término, que se llama sort principal del término. Los sorts forman una jerarquía; $s \lhd s'$ se leerá s es un subsort de s'. Así tenemos el siguiente diálogo:

LIFE> $camión \lhd vehículo.$
$***$ *Yes*

LIFE> $móvil(vehículo).$
$***$ *Yes*

LIFE> $útil(camión).$
$***$ *Yes*

LIFE> $móbil(X), útil(X)?$
$***$ *Yes*
$X = camión.$

Los números, los tipos, etc. son sorts, y se cumple $0 \lhd int$, $int \lhd real$, ... Existen dos sorts especiales (@ y {}) que denotan el mayor y el menor en la jerarquía. Los sorts pueden ocupar posiciones en los términos, como en $L : [int, int, int|L]$, que representa una lista cíclica de enteros. Entre términos existe una operación para el cálculo del $glp(t, t')$ (*greatest lower bound*) de dos términos, que será utilizado en la unificación; por ejemplo para la jerarquía:

$bici \lhd dosRuedas.$
$bici \lhd vehículo.$
$camión \lhd cuatroRuedas.$
$camión \lhd vehículo.$
$coche \lhd cuatroRuedas.$
$coche \lhd vehículo.$

se tiene: $glb(dosRuedas, vehículo) = bici$, $glb(dosRuedas, vehículo) = \{coche; camión\}$, $glb(bici, vehículo) = bici$, $glb(dosRuedas, cuatroRuedas) = \{\}$. La unificación de términos extiende la unificación de PROLOG, y por tanto también extiende el concepto de SLD-resolución. Para unificar dos términos se calcula el *glb* de sus sorts principales ligando las variables en la raíz, se añade a éste los atributos comunes y no comunes, y se unifica en forma recursiva los términos asociados. Por ejemplo la unificación de $coche(ruedas \Rightarrow 4)$ con $vehículo(ruedas \Rightarrow N : int)$ produce como resultado $coche(ruedas \Rightarrow N : 4)$.

La definición de términos y relaciones jerárquicas es fácil:

$:: rectángulo(alto \Rightarrow L : real,\ ancho => W : real,\ área \Rightarrow L * W).$
$:: cuadrado(alto \Rightarrow S,\ ancho \Rightarrow S).$
$cuadrado \lhd rectángulo.$

La última declaración permite extender una nueva característica de un *cuadrado* (su área), aunque, debido a que las dimensiones son iguales, el área sería siempre $(cuadrado.alto)\ \hat{}\ 2$ (obsérvese el uso del selector).

Es posible añadir restricciones a un sort; por ejemplo, la declaración $:: I : int \mid R$ obliga a que todos los enteros tengan la restricción R. Si ésta restricción es un objetivo, la evaluación de una expresión entera permitiría ejecutar el objetivo; así si queremos trazar todos los cálculos con enteros:

LIFE> $:: I : int \mid write(I, " ").$
$***$ *Yes*

LIFE> $A = 5 * 7$?
5 7 35
$***$ *Yes*
$A = 35$.

También es posible definir herencia múltiple con una restricción asociada; por ejemplo, la declaración:

$$padre := \{hijo(a); hija(b)\} \mid rest\ .$$

añade a la jerarquía los elementos $hijo \lhd padre$, $hija \lhd padre$, así como las restricciones asociadas: $hijo \mid rest$, $hija \mid rest$. Esto permite definir estructuras de datos:

$$árbol := \{\ hoja;\ nodo(izdo \Rightarrow árbol,\ dcho \Rightarrow árbol)\ \}.$$

y la declaración anterior equivale a las declaraciones:

$$hoja \lhd árbol.$$
$$nodo(izdo \Rightarrow árbol,\ dcho \Rightarrow árbol) \lhd árbol.$$

Los términos son (registros) extensibles; por ejemplo, el término $árbol(info \Rightarrow I : int)$ refina al término *árbol* añadiendo información a las hojas y nodos.

En LIFE las funciones son perezosas, por lo que una función puede ser invocada antes que sean evaluados sus argumentos; una llamada *suspendida* se llama un *residuo*. También se permite parcializar una llamada. La definición de funciones sigue la forma estándar:

LIFE> $fact(0) \rightarrow 1$.

LIFE> $fact(N : int) \rightarrow N * fac(N - 1)$.

LIFE> $write(fact(5))$?
120
$***$ *Yes*

y como se observa los paréntesis son necesarios. Si seguimos con el diálogo anterior:

LIFE> $A = fact(B)$?
$***$ *Yes*
$A = @,\ B = @\sim$.

LIFE> $B = real$?
$***$ *Yes*
$A = @,\ B = real\sim$.

LIFE> $B = 5$?
$***$ *Yes*
$A = 120,\ B = 5$.

La restricción $A = @$ significa que A está libre, mientras que la restricción $B = real\sim$ indica que B está ligado a un número real no determinado.

LIFE> $map(F,\ [\,]\,) \rightarrow [\,]$.

LIFE> $map(F, [H|T]) \rightarrow [F(H)|map(F,\ T)]$.

LIFE> $R = map(F, [4, 5, 6])$?
$***$ *Yes*
$F = @\sim,\ R = [@, @, @]$

LIFE> $F = fact?$
$***$ *Yes*
$F = fact,\ R = [24, 120, 720]$

El resultado de una función puede ser un término que representa un predicado. Si es booleano, puede ocupar la posición de un predicado. También es posible asociar una guarda a una ecuación, y ésta será ejecutada antes de devolver el valor de la función:

LIFE> $fact2(0) \rightarrow 1.$

LIFE> $fact2(N) \rightarrow N * fact2(N-1) \mid write(N, \text{" "}).$

LIFE> $7 = fact2(3)?$
1 2 3
$***$ *No*

La parcialización sigue una notación especial:

LIFE> $desayuno(tostada \ \Rightarrow T,\ cafe \ \Rightarrow C) \rightarrow [T, C].$

LIFE> $A = desayuno\,(tostada \ \Rightarrow panConAceite),\ write(A)?$
$desayuno\,(tostada \ \Rightarrow panConAceite)$
$***$ *Yes*

LIFE> $R = A(café \ \Rightarrow cafésolo),\ R' = A(café \ \Rightarrow caféConLeche),\ write(R, R')\ ?$
$[panConAceite, cafésolo]\ [panConAceite, caféConLeche]$

donde observamos que la llamada parcial $desayuno\,(tostada \ \Rightarrow panSolo)$ no se reduce, pero sí las totales. Así, los argumentos son consumidos por nombre y no por posición como en el λ–cálculo.

Las funciones se invocan vía comparación de patrones y los predicados vía unificación. Sin embargo, es posible modificar los argumentos en una llamada a través de guardas. [Aït-Kaci et al., 1994] expone el siguiente ejemplo:

$conca1([\,], L) \rightarrow L.$
$conca1([X \mid W], L) \rightarrow [X \mid conca1(W, L)].$

$conca2([\,], L, L).$
$conca2([X \mid W], L, [X \mid L']) \ :- \ conca2(W, L, L').$

$conca3([\,], L, R) \rightarrow true \mid R = L.$
$conca3([X \mid W], L, R]) \rightarrow true \mid R = [X \mid L'],\ conca3(W, L, L').$

De forma que la llamada $conca3(P, Q, R)$ espera a que el primer argumento sea una lista; si ésta no es vacía unifica el resultado con el tercer argumento.

Una función puede invocar un predicado directamente:

LIFE> $f(A : int) \rightarrow write(A).$ *% espera hasta que A sea un entero*

LIFE> $f(23)\ ?$
23
$***$ *Yes*

```
LIFE> X, X = f(Y), Y = 23?
23
*** Yes
X = write(Y), Y = 23.
```

Las restricciones son tratadas como predicados en programas. Así, tenemos la siguiente solución al puzzle SEND + MORE = MONEY:

```
puzzle :-
            1   = M,
  C3 + S +  M   = O  + 10,
  C2 + E +  O   = N  + 10 * C3,
  C1 + N +  R   = E  + 10 * C2,
       D +  E   = Y  + 10 * C1,


  diferentes([S, E, N, D, M, O, R, Y]),
  C1 = acarreo, C2 = acarreo, C3 = acarreo,
  S = dígito, E = dígito, N = dígito, D = dígito, O = dígito, R = dígito, Y = dígito,

  write(' ', S, E, N, D), nl,
  write('+', M, O, R, E), nl, write('- - - - -'), nl,
  write( M, O, N, E, Y), fail.


dígito → {0; 1; 2; 3; 4; 5; 6; 7; 8; 9}.
acarreo → {0, 1}.


diferentes([]).
diferentes ([H|T]) :- difCon(H, T), diferentes(T), H <= 9, H >= 0.
difCon(H, []).
difCon(H, [A|T]) :- difCon(H, T), A = / = H.
```

Obsérvese que *dígito* es una función multievaluada (indeterminista), y si aparece en una restricción puede provocar *backtracking*: $\ldots, E = dígito, \ldots$

Es importante decir que el orden de aparición de las restricciones no es relevante. Así, observamos que la restricción $C3 + S + M = O + 10$ aparece antes que las restricciones $\ldots, E = dígito, \ldots$.

El proceso de listas infinitas es también estándar. A modo de ejemplo consideremos las siguientes funciones auxiliares para el cálculo de los números primos por el método de la *criba de Eratóstenes*:

```
criba([])       → [].
criba([P|Ns]) → [P|criba(quitaMúltiplos(Ns, P))].


quitaMúltiplos([], P) → [].
quitaMúltiplos([X|Xs], P) → cond( (X mod P = / = 0),
                                  [X|quitaMúltiplos(Xs, P)],
                                  quitaMúltiplos(Xs, P) ).
```

$primos\ (N)\ \rightarrow\ criba(enteros(2, N))$.
$enteros(P, Q)\ \rightarrow\ cond(N > Q, [\,], [P|enteros(P + 1, Q)])$.

Tendremos entonces el siguiente diálogo:

LIFE> $A = criba([2, 3, 4, 5|L])$?
$***$ *Yes*
$A\ =\ [2, 3, 5|@],\ L = @ \sim$.

LIFE> $L = [6, 7, 8, 9|L']$?
$A\ =\ [2, 3, 5, \_A : 7|@],\ L = [6, \_A, 8, 9|L'],\ L' = @ \sim$.

donde observamos que en el primer objetivo (restricción) la lista A está parcialmente evaluada, esperando que una restricción posterior permita refinar algo más la lista L.

LIFE contiene algunas funciones estandarizadas muy útiles; por ejemplo aquellas que permiten extraer información de un término (*proyect*, o .), o la función *features* que permite extraer la lista de los identificadores de los campos de un término:

LIFE> $A = s(a, b)$?
$***$ *Yes*
$A = s(a, b)$

LIFE> $B\ =\ A.2$
$***$ *Yes*
$A = s(a, b),\ B\ =\ b$

LIFE> $C\ =\ A.B$?
$***$ *Yes*
$A = s(a, B, b\ \ \Rightarrow\ C),\ B\ =\ b,\ C\ =\ @$

LIFE> $Q\ =\ features(A)$?
$***$ *Yes*
$A = s(a, B, b\ \ \Rightarrow\ C),\ B\ =\ b,\ C\ =\ @,\ Q = [1, 2, b]$

Así, la función $features\_values(X)\ \rightarrow\ map\ (proyect(2\ \Rightarrow\ X), features(X))$ calcula la lista de los valores de los campos de X.

LIFE tiene dos operadores para la asignación, uno *no–persistente* ($<-$) y otro persistente ($<<-$) que no permite backtracking:

LIFE> $X = 5,\ (X < -6;\ X < -7;\ succeed),\ write(X), fail$?
6 7 5
$***$ *No*

El siguiente ejemplo, extraído de [Aït-Kaci et al., 1994]:84, ilustra la potencia de los operadores de asignación; se pretende calcular el tamaño de un término LIFE:

```
tamaño(X) → N |
  V <<− @,                    % crea un término persistente anónimo
  ( V <<− explora(X, Vistos), % calcula el tamaño de X
  fail                        % y elimina los efectos del cálculo
  ;
  N = copy_term(V)            % copia el resultado en N
  ).
explora(X, Vistos) → V |
  (X === Vistos, !, V = 0     % comprueba si tal nodo ha sido visitado
  ;
  FV = features_values(X),
  X <− Vistos,
  V = 1 + sum( map( explora(2 ⇒ Vistos) , FV) )
  ).
```

donde la expresión $X === Y$ comprueba si los términos son idénticos. En el ejemplo se ilustra una pseudo–programación imperativa. Es más, la secuencia formada por el proyector (.) y la asignación no persistente permite manejar las tablas o *arrays* de los lenguajes imperativos; todo ello ayudado por el uso de variables globales:

```
global(criba).
global(límite).


eliminaMúltiplos(P, M) :− cond( M < limite,
                                criba.M <− multiploDe(P),
                                eliminaMúltiplos(P, M + P))).
...
```

Otros ejemplos de programas completos en LIFE pueden verse en [Aït-Kaci, 1999b], donde a modo de ejemplo desarrolla un emulador para la máquina SECD en LIFE.

LIFE usa una sintaxis (para PROLOG) esencialmente igual que la ISO Standard. Además, la SLD-resolución de PROLOG es un caso particular, de forma que podríamos pensar en el uso de LIFE como sustituto de PROLOG. Esto tendría claras ventajas frente a PROLOG ya que LIFE sustituye los términos de la base de Herbrand por los ψ–términos, y añade la *call-by-matching* típica de los lenguajes funcionales modernos. Todo ello permite utilizar características muy sugerentes: funciones, orientación a objetos con tipos y herencia múltiple, datos extensibles y posiblemente cíclicos, variables globales, asignaciones no persistentes, además de la característica más natural de LIFE: restricciones con acciones asociadas (o acciones guardadas con restricciones).

Todo lo anterior permite pensar que LIFE puede ser un buen candidato para sustituir a PROLOG; de hecho sus diseñadores afirman ([Aït-Kaci et al., 1994]:102) que una vez que los programadores usan LIFE, ya no hacen uso de PROLOG.

De todas las formas, aunque el lenguaje lleva desarrollándose 10 años, no conocemos experiencias educativas sobre el uso de LIFE. Además no se han publicado textos de programación para LIFE del estilo de los *buenos* libros que sí tenemos las versiones de los lenguajes es estado puro: lógicos, funcionales y con restricciones.

Desde un punto de vista formativo, si la evaluación de una función puede producir ligaduras (o efectos laterales) sobre sus argumentos, desaparece la principal característica de la $\mathrm{P_{RO}F_{UN}}$, y como consecuencia de ello todas las estrategias de razonamiento sobre los

programas funcionales basados en la trasparencia (transformación, inducción, etc.). Por ello, y como resumen, podría decir que estos lenguajes son potentes, muy interesantes, pero es necesario que el programador conozca básicamente los estilos funcional, lógico y con restricciones. Por ello, tales lenguajes pueden y deben experimentarse en cursos avanzados.

Borneo

Hassan Aït-Kaci está desarrollando actualmente el sistema para la programación BORNEO [Aït-Kaci, 1999a], un sistema más versátil que WILD LIFE para la implementación de LIFE. Para ello utiliza JACC (*Just Another Compiler Compiler*), una herramienta similar a `yacc` pero que genera código JAVA. Todo ello facilita el desarrollo de BORNEO en Internet.

BORNEO puede ser visto como una familia de lenguajes y paradigmas, ya que proporciona un generador de analizadores muy potente y un cálculo de máquinas abstractas para diseñar la semántica del lenguaje deseado[118].

0.3.0 Programación Concurrente con restricciones

El paradigma de la programación con restricciones es introducido en la tesis de Guy [Steele, 1980], y casi desde su aparición ha sido aplicado junto al paradigma de la $P_{RO}L_{OG}$. Sin embargo, la combinación de restricciones y concurrencia ha tenido un fuerte impacto desde la tesis de Vijay [Saraswat, 1989], publicada posteriormente como [Saraswat, 1993].

En esta tesis introduce la familia de lenguajes CC (*Concurrent Constraint*), donde el mecanismo de comunicación/sincronización se realiza vía una restricción global[119], aunque la idea ya había sido expuesta anteriormente por [Maher, 1987]. La comunicación basada en restricciones tiene las siguientes características:

las restricciones especifican información parcial de forma que un agente puede esperar hasta que otro complete la información de un objeto.

la comunicación es aditiva de forma que únicamente son añadidas las restricciones consistentes; si una restricción es añadida será derivable en el futuro.

[118] Es posible que esta idea de Hassan Aït-Kaci fue concebida durante la implementación del sistema WILD LIFE, donde observó que varias técnicas y paradigmas de la programación podían ser implementadas siguiendo ideas derivadas del profundo estudio de la máquina WAM. Así, la extensión del concepto de unificación (de PROLOG) sobre los ψ–términos le permitió desarrollar de forma más amplia el concepto de restricción, que a su vez permite, como hemos visto en LIFE, el desarrollo de los paradigmas aplicativo, imperativo, lógico, etc.; de todos formas, como acepta Aït-Kaci de forma indirecta, al parecer la idea ya la desarrollaron [Höhfeld y Smolka, 1988], lo que demuestra que, aunque el desarrollo de la $P_{RO}R_{ES}$ fue lento, ha sido una idea acertada. Recordemos que el concepto de restricción es introducido en la tesis de Guy L. [Steele, 1980], y un concepto cercano (*freeze*) ya aparecía en PROLOG II [Colmerauer et al., 1982], por lo que podemos considerar que la idea lleva desarrollándose 20 años. También llama la atención que Steele no sacara algo de más partido a la idea, teniendo en cuenta su experiencia como programador en SCHEME y en COMMONLISP.

[119] Una introducción interesante y actualizada de estos mecanismos puede encontrarse en las primeras secciones de [Breitinger et al., 1995] y de [Smolka, 1995], aunque estos trabajos están dedicados a los lenguajes EDEN y OZ. Otra recopilación de conceptos e implementaciones se encuentra en [Jaffar y Maher, 1994].

los canales son genéricos; si un agente P comparte una variable X con otro agente Q, y P realiza la restricción $X = f(a)$, donde f es una función inyectiva, entonces, el agente Q puede obtener la información del canal tratando de satisfacer $X = f(Z)$, lo que produciría $Z = a$ automáticamente.

Saraswat trata de capturar las características de varios lenguajes que en la década de los 80 tuvieron mucho éxito, y fundamentalmente CONCURRENT PROLOG [Shapiro, 1983] y VULCAN [Kahn et al., 1987]. La familia base CC está formada por aquellos lenguajes basados en restricciones con *tell* atómico. Posteriormente se pueden añadir otras características, como '$\downarrow$' que representa el *blocking ask*, '$\rightarrow$' representa el indeterminismo *don't care* (DC), y ' $\Rightarrow$ ' que representa el indeterminismo *don't know* (DK) (estos dos últimos con *commit* implícito). Así, la familia de lenguajes más completa sería CC($\downarrow, \rightarrow, \Rightarrow$). Utilizando el sistema de restricciones HERBRAND($\downarrow, \rightarrow$), con una implementación particular (secuencial y bajo PROLOG) es posible obtener GHC, PARLOG o PROLOG.

CC proporciona una serie de operadores para anidar cláusulas; uno de ellos es una generalización del operador de alternancia $\square$ de [Dijkstra, 1976]; otros operadores *commit* permiten proteger acciones básicas (*stop*, *fail*, ...) o acciones con guardas e indeterminismo DK paralelo ($\Rightarrow$), indeterminismo DK secuencial ($\Longrightarrow$), e indeterminismo DC ($\rightarrow$).

Consideremos el programa:

$$\begin{array}{l} frontera(X, Y) \ :: \ tell(X = españa, Y = francia) \ \Rightarrow \ stop. \\ frontera(X, Y) \ :: \ tell(X = españa, Y = portugal) \ \Rightarrow \ stop. \\ frontera(X, Y) \ :: \ tell(X = francia, Y = italia) \ \Rightarrow \ stop. \end{array}$$

Obsérvese que en los cuerpos no aparecen guardas. Si en la RG (*store* o restricción global) no aparece la variable X, para un objetivo como $frontera(X, Y)$ se realizarán tres copias del RG de forma que las restricciones producidas por la evaluación de cada una de las alternativas no afecte a las restantes[120]. Así, las respuestas al objetivo $tell(X = españa), frontera(X, Y)$ deberían proporcionar de forma indeterminista todos los países fronterizos con *españa* y en un orden indeterminado. Es evidente que se permiten guardas:

$$\begin{array}{l} max(X, Y, Z) \ :: \ (X \leq \ Y) \ : \ tell(Y = Z) \ \Rightarrow \ stop. \\ max(X, Y, Z) \ :: \ (Y \leq \ X) \ : \ tell(X = Z) \ \Rightarrow \ stop. \end{array}$$

de forma que el objetivo $max(X, Y, Z)$ quedará suspendido hasta tener suficiente información para evaluar las guardas, y en un solo paso, de forma indeterminista posiblemente, se incluye la restricción $Y = Z$ o la restricción $X = Z$ (o ambas, para las dos copias de la RG).

Otra forma de escribir el programa anterior utilizando el operador $\square$ sería:

$$\begin{array}{lll} frontera(X, Y) \ :: & tell(X = españa, Y = francia) & \Rightarrow \ stop \\ & \square \, tell(X = españa, Y = portugal) & \Rightarrow \ stop \\ & \square \, tell(X = francia, Y = italia) & \Rightarrow \ stop. \end{array}$$

[120] En [Saraswat, 1993]:51 aparece gráficamente cómo evoluciona el RG en el tiempo dependiendo de las operaciones realizadas sobre éste.

$$max(X, Y, Z) :: X \leq Y \Rightarrow igual(Y, Z) \ \square \ Y \leq X \Rightarrow igual(X, Z).$$
$$igual(Y, Z) :: tell(Y = Z) \Rightarrow stop.$$

Incluso es posible suprimir el procedimiento *igual*:

$$\begin{aligned} max(X, Y, Z) :: \quad & X \leq Y \Rightarrow tell(Y = Z) \Rightarrow stop \\ & \square \ Y \leq X \Rightarrow tell(X = Z) \Rightarrow stop. \end{aligned}$$

La secuencia $tell(c) \Rightarrow stop$ es muy utilizada y es abreviada como c:

$$\begin{aligned} frontera(X, Y) :: \quad & (X = espa\tilde{n}a, Y = francia) \\ & \square (X = espa\tilde{n}a, Y = portugal) \\ & \square (X = francia, Y = italia). \end{aligned}$$

$$max(X, Y, Z) :: X \leq Y \Rightarrow Y = Z \ \square \ Y \leq X \Rightarrow X = Z.$$

En muchas situaciones es posible predecir que cierto procedimiento tendrá un sola solución; en ese caso es preciso utilizar el indeterminismo DC $\rightarrow$, como en

$$max(X, Y, Z) :: X \leq Y \rightarrow Y = Z \ \square \ Y \leq X \rightarrow X = Z.$$

Obsérvese que al final el resultado es un programa à la Dijkstra, con indeterminismo DC, que precisamente es la semántica operacional del operador $\square$ de [Dijkstra, 1976].

La familia de lenguajes CC proporcionan otras construcciones, como bloques, estructuras de acceso, datos basados en restricciones, etc. Otros ejemplos de programas escritos con la familia CC, como CC(↓,—,&), pueden verse en [Saraswat, 1993], capítulos 3, 5 y Apéndice B.

La semántica operacional de los lenguajes de la familia CC viene dada por un sistema de reglas de transición à la [Plotkin, 1981] muy completo. Además, tal estilo permite probar propiedades de los programas (secciones 4.3, 4.4 y 4.5).

El interesante trabajo de Vijay Saraswat ha sido utilizado para implementar y definir muchos lenguajes. Ejemplos de éstos son: JANUS [Saraswat, 1990], un lenguaje distribuido con la construcción *ask/tell* (existe una implementación en SICSTUS-PROLOG realizada en la Univ. de Arizona); LUCY [Kahn, 1990], un subconjunto de JANUS basado en actores; PROCOL [van den Bos, 1991], un lenguaje concurrente OO con protocolos y restricciones; AKL[121]. Otra propuesta es GOFFIN [Chakravarty et al., 1995], que extiende HASKELL con restricciones y concurrencia.

Así mismo se han propuestos varias máquinas abstractas para la implementación. Por ejemplo, la *K-machine* [Lopez, 1994], para KALEIDOSCOPE, o la máquina BERTRAND [Leler, 1988], que proporciona un lenguaje basado en reglas de reescritura donde el usuario puede especificar como se propagan las restricciones. O incluso la máquina CLAM (*Constraint Language Abstract Machine*) [Jaffar, 1992], una máquina abstracta basada en la máquina WAM, que es utilizada para la implementación de CLP(R).

[121] *Agents Kernel Language*, previamente llamado *Andorra Kernel Language* [Janson, 1992], que mezcla restricciones (sobre dominios finitos), el indeterminismo *dont'n know* de PROLOG y el indeterminismo *dont'n care* de los lenguajes lógicos concurrentes

Oz

El modelo de Oz [Smolka, 1995] está basado en la $P_{RO}F_{UN}$ de orden superior y la programación lógica concurrente (*committed-choice*) con objetos y restricciones, y es el resultado de una década de investigación en la integración de estos paradigmas. Por tanto se trata de un lenguaje de la familia de los CCPLs (*Concurrent Constraint Programming Languages*) que proporciona $P_{RO}F_{UN}$ de orden superior junto a CLP, además de objetos concurrentes. Las funciones son casos especiales de relaciones y la semántica operacional es similar a la residuación (*residuation*). Actualmente podemos experimentar con el sistema MOZART[122] [van Roy, 1999], un sistema para la computación distribuida basado en restricciones y que implementa Oz. Al igual que con LIFE, la programación à la PROLOG es posible desarrollarla en Oz [van Roy, 1999][123]. Desde un punto de vista didáctico, es interesante añadir que MOZART proporciona dos herramientas gráficas (Browser y Explorer) muy útiles para desarrollar programas lógicos.

Curiosamente, [Smolka, 1998] prueba que es posible extender el lenguaje SML para describir las características de Oz, lo que muestra el alto contenido expresivo de los lenguajes funcionales modernos.

0.4 Semántica contextual y Transformación de Programas

La propiedad sintáctica esencial del lambda cálculo puro es la confluencia. Tal propiedad se extiende a sistemas de reescritura confluentes, pero sin embargo también son interesantes aquellos donde se pierde la confluencia. Tales sistemas han sido modelados por [Abramsky, 1990] en el marco de la *semántica contextual* que permite introducir operacionalmente un cálculo perezoso con indeterminismo.

Esencialmente, la semántica contextual proporciona información de los términos (programas con o sin variables libres) a través de una relación de preorden contextual, de forma que un término s contiene menos información que otro t si para todo contexto $C[]$, $C[s]$ da más información (p.e., termina) que $C[t]$. Este preorden permite definir una noción de equivalencia de programas que a su vez proporciona un concepto de transformación correcta. Así, en tal modelo semántico la confluencia no es importante, siendo ésta reemplazada por el concepto de transformación correcta. Además, el cálculo puede ser extendido con construcciones indeterministas, evaluación perezosa y recolección de basura, como exponen [Schmidt-Schaußy Huber, 2000], donde la evaluación de un término puede realizarse en presencia de variables libres.

Basado en las lógicas con igualdad de [Hoare y Jifeng, 1998] es posible extender el papel del álgebra [Bird y de Moor, 1997], lo que permite extender las *leyes de la programación funcional* a la lógica; esta es una idea muy interesante ya que establece un puente metodológico en el marco de la $P_{RO}L_{OG}F_{UN}$.

[122] `http://www.mozart-oz.org/`.

[123] Según [Smolka, 1995]:4, Oz desplazará a los lenguajes secuenciales como LISP, PROLOG o SMALLTALK. De cualquier forma, no conocemos experiencias docentes de su uso como sustituto de PROLOG.

Programa	= Algoritmo + Estructuras de datos	[Wirth, 1976]
Algoritmo	= Lógica + Control	[Kowalski, 1979a]
Programa	= Formalismo + Control	[Gregory, 1987]
Algoritmo	= Conocimiento + Control	[Abbott, 1987]
AlgorithmS	= Logic + ControlS	[Schreye et al., 1997]
Paralelismo	= Algoritmo + Estrategia	[Trinder et al., 1998]
Concurrent LP	= LP + committed choice = LP - completeness	[Ueda, 1999]

Figura 3: Diferentes *cajas de Pandora* en la literatura

0.5 La caja de Pandora $\boxed{A = L + C}$ y sus variantes

Muchos investigadores se han ayudado de *ecuaciones* para explicar diferentes paradigmas. En la Figura 3 aparecen las ecuaciones más conocidas.

Comencemos con la de [Kowalski, 1979a]: $A = L + C$. Casi todos los textos contienen un capítulo dedicado a este principio, bien para ilustrar el paradigma de la $\mathrm{P_{RO}L_{OG}}$, o bien para ilustrar el paradigma de alguna de sus variantes, como la concurrencia y el indeterminismo *don't care*. Sin embargo, para el paradigma de la $\mathrm{P_{RO}F_{UN}}$ no conozco ningún texto que utilice una ecuación similar para ilustrar el paradigma.

Esencialmente la ecuación enfatiza la independencia de la parte lógica de un programa de la parte de control. [Gregory, 1987] prefiere formularla de un modo sutilmente diferente:

$$\text{Programa declarativo} = \text{Formalismo} + \text{Control}$$

de forma que en el formalismo podemos indicar cualquier teoría que proporcione la semántica *declarativa* a los programas. El control determina la semántica operacional. Los dos principales ejemplares de la $\mathrm{P_{RO}D_{EC}}$ se obtienen tomando, bien como teoría las cláusulas de Horn para obtener la $\mathrm{P_{RO}L_{OG}}$, bien el λ-cálculo para obtener la $\mathrm{P_{RO}F_{UN}}$.

Como sigue indicando [Gregory, 1987], la componente de control puede a su vez descomponerse según la fórmula:

$$\textit{Control} = \textit{Evaluador} + \textit{Lenguaje de Control}$$

El evaluador permite fijar distintas estrategias de evaluación; a veces opera de forma autónoma, mientras que otras puede quedar afectado por medio de algún lenguaje de control impuesto por el programa declarativo.

En $\mathrm{P_{RO}F_{UN}}$ el evaluador está basado en mecanismos de reducción por reescritura y el lenguaje de control determina una estrategia de reducción; por ejemplo, evaluación perezosa, que podría ser impaciente para ciertas subexpresiones si tal expresión es anotada vía el lenguaje.

En $\mathrm{P_{RO}L_{OG}}$ el evaluador viene determinado normalmente por SLD(NF)–resolución, y el lenguaje de control selecciona las reescrituras de las cláusulas.

Realmente, el término $\mathrm{P_{RO}L_{OG}}$ supone una interpretación procedural para las fórmulas, y la noción de programación solo existe si el modelo de ejecución es conocido en el desarrollo de la programación. En la práctica, podemos disponer simultáneamente de diferentes resolvedores actuando sobre la misma base de conocimiento para resolver el problema: inductivo, deductivo o basado en restricciones. Por tanto cabría suponer un evaluador mixto para obtener la CLP, o un lenguaje de control que permita el paralelismo, etc.

Supongamos la ecuación $A = L + C$, donde C es la SLD(NF)–resolución. Entonces se crea la ilusión de que podemos representar un problema únicamente vía cláusulas de Horn. La práctica dice que esto solo funciona en problemas simples ya que en muchos casos necesitamos otras técnicas más que la mera deducción para resolver problemas donde el espacio de búsqueda es enorme.

O sea, el estilo de programación utilizado puede ser declarativo en sentido formal, pero no es declarativo en sentido metodológico. Por ello [Kowalski, 1995] ha replanteado su ecuación motivado por el hecho de que los programadores enfatizan más la lectura procedural de los programas lógicos que la lectura declarativa. Como expresa [Schreye y Denecker, 1999], en los textos de $\mathrm{P_{RO}L_{OG}}$ los algoritmos son expresados siguiendo una tendencia al estilo PASCAL.

También puede ocurrir que el dominio del problema no sea codificado en átomos y reglas, sino, por ejemplo, en listas que representan teorías del conocimiento. En ese caso la ecuación pasa a ser $A = L' + C$, donde L' no es la lógica del dominio del problema. Una forma de resolver esto es codificar la lógica en una teoría separada, y escribir un metaprograma M de forma que la ejecución de L bajo M (en SLD(NF)–resolución) sea lo mismo que la ejecución de L' bajo SLD(NF)–resolución. Quedaría pues $A = L+M+C$. De esta forma, un programa lógico representa de alguna forma la abstracción del conocimiento del problema; [Abbott, 1987] aconseja el uso de la ecuación:

$$\text{Algoritmo} = \text{Conocimiento} + \text{Control}$$

La justificación que da es la dificultad en controlar el uso del conocimiento en los lenguajes tradicionales. Se han intentado algunas soluciones extendiendo un lenguaje de programación del conocimiento (como PROLOG) con algún lenguaje procedural como C (SILOGICPROLOG) o MODULA (MODULAPROLOG [Muller, 1986]).

La ecuación $A = L + C$ ha motivado diversos trabajos en transformación y síntesis de programas. Alan [Bundy, 1988] dice que un uso ingenuo de la ecuación lleva a programas ineficientes e inaceptables, y defiende la idea de extender la expresividad de la lógica además de usar métodos de síntesis para transformar especificaciones (dentro de una lógica más rica) en programas basados en cláusulas de Horn.

Muchos autores han querido establecer un puente entre las dos componentes motivando la transformación y la síntesis de programas [Deville, 1990]. Esto sugiere una nueva ecuación $AlgorithmS = Logic + ControlS$. En este sentido aparece planteado el proyecto *PL+: a second generation Logic Programming language* [Schreye et al., 1997]

Capítulo 1

La asignatura *Programación Declarativa Avanzada*

Objetivos:

- Complementar las técnicas de la PROLOG y de la PROFUN estudiadas en la asignatura obligatoria *Programación Declarativa*.
- Profundizar en los fundamentos teóricos de ambos paradigmas.
- Estudiar las técnicas de implementación de lenguajes lógicos y funcionales.
- Presentar aspectos alternativos dentro de los contextos aplicativo y relacional, y la integración de ambos paradigmas.
- Describir las extensiones más interesantes de los modelos lógico y funcional que permitan integrar características adicionales: *concurrencia, objetos, eventos, modularidad, orden superior, restricciones, ...*, o incluso otras de relevancia actual: *servidores WEB, aplicaciones en Internet, interface con* JAVA, ...

1.0 Temas monográficos

En la Tabla 1.2 aparecen algunos de los temas realizados por alumnos de esta asignatura en cursos anteriores.

1.1 Otros Temas Monográficos y/o Proyectos de Fin de Carrera

Objetivo: considerar los trabajos realizados como núcleo inicial para la elaboración del Proyecto de Fin de Carrera. Entre las posibles líneas podemos considerar las siguientes:

- Emuladores de la máquina abstracta WAM [Aït-Kaci, 1991], o de la máquina SECD [Aït-Kaci, 1999b].
- Intérpretes funcionales para lenguajes imperativos indeterministas [Goguen y Malcolm, 1996; Ruiz Jiménez et al., 2000]
- Aplicaciones para Internet.
- Intérpretes para lenguajes lógico/funcionales del estilo de CURRY o LIFE.
- Bases de datos deductivas [Das, 1992].
- Analizadores semánticos para programas lógicos o funcionales basados en interpretación abstracta [Jones y Nielson, 1995; Burn, 1991].
- Sistemas de tipos para PROLOG (MERCURY [Somogyi et al., 1995], λPROLOG [Ridoux, 1998]).

Tema Lecciones (horas)	Horas T+PPiz+PLab
Fundamentos de la PRODEC	19
Fundamentos de la PROLOG. Semántica declarativa de programas definidos (2). Negación y manipulación dinámica de programas (2). Otras lógicas como lenguajes de programación (1).	5
Fundamentos de la PROFUN. Introducción al λ–cálculo (1). λ–teorías (2). Lambda definibilidad (1). Semántica denotacional del λC (1).	5
Lógica ecuacional y Sistemas de reescritura. Lógica ecuacional (2). Sistemas de reescritura (2).	4
Sistemas de tipos. El λ-cálculo con tipos (3). Sistemas de tipos para PROLOG (2).	5
Extensiones y otros paradigmas de la PRODEC	19
Concurrencia en los lenguajes lógicos. Modos de Paralelismo en los lenguajes lógicos (1). Lenguajes lógicos concurrentes (2). Técnicas de la PROLOG concurrente (2).	5
PROLOG con restricciones. El esquema CLP (1). Extensiones de la PROLOG con restricciones (1).	2
Mónadas y continuaciones. Mónadas algebraicas y mónadas à la HASKELL (2). Programación con mónadas (3).	5
Concurrencia en los lenguajes funcionales. Programación funcional concurrente (1). Técnicas de la PROFUN concurrente en HASKELL (2).	3
Lenguajes lógico–funcionales. Resolución, Unificación y Reescritura (1). Lenguajes lógico–funcionales (2).	4
Otras extensiones. Objetos en los lenguajes declarativos (2). Extensiones para aplicaciones en Internet (2).	4
Técnicas avanzadas en PRODEC	13
Técnicas avanzadas de PROLOG. Técnicas de búsqueda (2). Técnicas de meta-interpretación (3). Reconocimiento de lenguajes (2).	7
Técnicas avanzadas de programación funcional. Aplicación de la evaluación perezosa (1). Técnicas de búsqueda en grafos (2). Algoritmos numéricos y simulación (3).	6
Implementación de lenguajes declarativos	9
Implementación de lenguajes lógicos. Máquina abstracta de Warren (WAM) (3). Técnicas adicionales de traducción (1).	4
Implementación de lenguajes funcionales. Traducción al λC (1). Modelos de Compilación (2). Reducción de grafos (2).	5

Tabla 1.1: Contenidos de la asignatura *Programación Declarativa Avanzada*

•Modelado de entrada/salida en los lenguajes funcionales. •Implementación de lenguajes funcionales. La máquina G. •Programación en PARLOG ([Conlon, 1989], intérprete y manuales [Conlon y Gregory, 1990]) •La máquina abstracta de Warren (`http://www.isg.sfu.ca/~hak`) •Herencia, objetos reactivos y selección de mensajes en CONCURRENT HASKELL [Gallardo et al., 1997]. •Algoritmos parciales y estructuras en diferencia en PROLOG [Sterling y Shapiro, 1994]. •Analizadores monádicos [Hutton y Meijer, 1998]. •Estructuras de datos monádicas [Ruiz Jiménez et al., 1995]. •Expresividad de los lenguajes lógico/funcionales •El teorema de confluencia de Church–Rosser [Hindley y Seldin, 1986]. •$\mathrm{P_{RO}L_{OG}}$ y concurrencia. •Sistemas de tipos para PROLOG. λPROLOG (`http://www.cse.psu.edu/~dale/lProlog/`). •Una panorámica de la $\mathrm{P_{RO}D_{EC}}$ y sus aspectos educacionales. •Programación lógico/funcional con CURRY [Hanus, 1997].	•Estructuras de datos en el λC puro [Ruiz Jiménez, 1994]. •Concurrencia en ERLANG. •Un modelo gráfico basado en la máquina G [Jiménez Santana, 1996]. •Objetos y herencia en HASKELL modelados en HASKELL [Hughes y Sparud, 1995; Gallardo et al., 1997]. •El lenguaje funcional CLEAN (información internet). •Aspectos educativos de la $\mathrm{P_{RO}F_{UN}}$ en diferentes universidades (internet). •Módulos y TADs en HASKELL. •Gráficos en HASKELL. •Sistemas de tipos à la Curry y à la Church. •El lenguaje de Programación TOY. •Mecanismos de reducción de grafos basados en la máquina G. •Programación con estructuras de datos incompletas en PROLOG. •Programación con eventos y restricciones en el lenguaje lógico TEMPO ([Gregory, 1996; Gregory y Ramírez, 1995; Gregory, 1997], `http://star.cs.bris.ac.uk/ software/ tempo.html`) •Modelos neuronales en $\mathrm{P_{RO}F_{UN}}$.

Tabla 1.2: Temas monográficos realizados en la asignatura *Programación Declarativa Avanzada*

— Programación con restricciones (MOZART [van Roy, 1999], WILD LIFE [Aït-Kaci et al., 1994]).

— Programación lógica inductiva [Apt et al., 1999].

— Implementación de un lenguaje lógico basado en uno funcional [Eisinger et al., 1997].

— Emuladores de autómatas [Thompson, 2000b].

Sobre el desarrollo de aplicaciones en Internet, caben citar algunas líneas. Por ejemplo, aquellas aproximaciones basadas en $\mathrm{P_{RO}L_{OG}}$, del estilo de JASPER [Davies et al., 1995] (interfaz PROLOG–JAVA), o servidores lógicos del estilo de *PiLLow/CIAO Library for Internet* [Cabeza et al., 1996], o incluso JINNI [Tarau, 1999] (un lenguaje lógico diseñado como una herramienta para mezclar objetos JAVA con servidores lógicos para aplicaciones en una red cliente/servidor). En [Pearce, 1997] podemos encontrar un informe sobre el impacto de la lógica computacional en el mercado software enfatizando las aplicaciones orientadas a Internet. Sobre las aplicaciones basadas en el paradigma de la $\mathrm{P_{RO}F_{UN}}$ caben citar PIZZA[1] [Odersky y Wadler, 1997; Odersky y Wadler, 1998], y las ya comentadas desarrolladas para HASKELL: integración de componentes COM [Peyton Jones et al., 1998; Leijhen, 1998], aplicaciones híbridas HASKELL–JAVA [Naughton, 1996; Flanagan, 1996], o bajo HASKELLSCRIPT [Leijen et al., 1999], y basadas en LAMBADA o MONDRIAN (`www.cs.uu.nl/~erik/`). Incluso, podemos encontrar otras propuestas basadas en lenguajes lógico/funcionales como TOY o CURRY [Hanus, 2000].

[1] PIZZA es un intento para sumergir JAVA dentro de un lenguaje funcional extendido, añadiendo a éste ideas de los lenguajes funcionales modernos, como tipos paramétricos.

Otro campo interesante es la Programación Lógica con Restricciones (CLP). Para CLP podemos estudiar sus fundamentos teóricos y algún lenguaje práctico para la solución de problemas reales. Un estudio actual sobre los fundamentos de la CLP puede verse en en [Maher, 1999], donde podemos encontrar otras referencias sobre distintas formas de integrar las cláusulas de Horn con las restricciones. Sobre las posibilidades actuales (lenguajes, y aplicaciones prácticas) puede verse [van Emden, 1999; Cohen, 1999] (todos estos trabajos contienen bibliografía actualizada sobre otras aplicaciones industriales: cálculo numérico, problemas combinatorios, biología molecular, etc.).

Finalmente, otro campo cada día más atractivo es la Programación lógica inductiva. Algunas líneas interesantes aparecen descritas en [Bergadano y Gunetti, 1995; Nienhuys-Cheng y de Wolf, 1997], aunque la panorámica más actual aparece en la colección [Apt et al., 1999] que hemos comentado en el Capítulo 0.

Bibliografía

Abbott, R. J. (1987). Knowledge Abstraction. *CACM*, 30(8):664–671.

Abelson, H. (1982). *Apple LOGO*. McGraw-Hill. Traducido al castellano por McGraw-Hill, México (1984).

Abelson, H. y diSessa, A. (1981). *Turtle Geometry. The Computer as a Medium for Exploring Mathematics*. MIT Press, Cambridge. Traducido al castellano en Anaya Multimedia, Madrid (1986).

Abengózar, M., Arenas, P., Caballero, R., Gil, A., González, J., Leach, J., López, F., Martí, N., Rodríguez, M., Ruz, J., y Sanchez-Hernandez, J. (2000). TOY: A Multiparadigm Declarative Language. Version 1.0. Informe técnico SIP 106.00, Dep. de Sistemas Informáticos y Programación, Univ. Complutense de Madrid.

Abramsky, S. (1990). The Lazy Lambda Calculus. En Turner, D. (ed.), *Research Topics in Functional Programming*, pp. 100–200. Addison–Wesley.

Ackermann, W. (1937). Mathematische Annalen. *Volumen 114*, pp. 305–315.

ACM (ed.) (1987). *Object-Oriented Programming Systems, Languages and Applications (OOPSLA'86)*, volumen 21(11) de *ACMSN*.

ACM (1993). Special Issue on The Fifth Generation Project. *CACM*, 36(3).

Aerts, W. y de Vlaminck, K. (1999). Games Provide Fun(ctional Programming Task)! En Felleisen, M., Hanus, M., y Thompson, S. (eds.), *Proceedings of the Workshop on Functional and Declarative Programming in Education, FDPE'99*. Technical Report of Rice University, Rice COMP TR99-346. `http://www.cs.rice.edu/~matthias/FDPE99`.

Allan, B. (1984). *Introducing LOGO*. Granada Publ. L. Traducido al castellano en Díaz de Santos, Madrid (1985).

Andréka, H. y Németi, I. (1976). The Generalised Completeness of Horn Predicate-Logic as a Programming Language. Tech. Report DAI RR 21, DAI, University of Edinburgh.

Antoy, S., Echahed, R., y Hanus, M. (1994). A Needed Narrowing Strategy. En *ACM Symp. on Principles of Programming Languages (POPL'94)*, pp. 268–279. ACM Press.

Antoy, S., Echahed, R., y Hanus, M. (2000). A needed narrowing strategy. *JACM*, 47(4):776–822.

Apt, K. R. y Bezem, M. (1999). Formulas as Programas. En Apt, W. R., Marek, V. W., Truszczynski, M., y Warren, D. S. (eds.), *The Logic Programming Paradigm. A 25–Year Perspective*, capítulo 2, pp. 75–108. Springer.

Apt, W. R., Marek, V. W., Truszczynski, M., y Warren, D. S. (1999). *The Logic Programming Paradigm. A 25–Year Perspective*. Springer.

Arity Corp. (1986a). *Arity/Prolog Reference Manual*.

Arity Corp. (1986b). *An Introduction to Arity/Prolog.*

Armstrong, J. (1997). The development of Erlang. *ACMSN*, 32(8):196–215.

Armstrong, J., Virding, R., Wikström, C., y Williams, M. (1996). *Concurrent Programming in Erlang.* Prentice Hall, 2ª edición.

Arnold, K. y Gosling, J. (1996). *The Java Programming Language.* Addison-Weley.

Aït-Kaci, H. (1991). *Warren's Abstract Machine: A tutorial reconstruction.* MIT Press. Existe una reimpresión con correcciones de 1999. `http://www.isg.sfu.ca/~hak/documents/wam.html`.

Aït-Kaci, H. (1999a). Borneo: Life and after born anew in Java. Tech. report, Intelligent Software Group. Simon Fraser University, Burnaby, Canada. `www.isg.sfu.ca/~hak`.

Aït-Kaci, H. (1999b). Programming Languages. Foundations, Design and Implementation. Incomplete draft, Intelligent Software Group. Simon Fraser University, Burnaby, Canada. `www.isg.sfu.ca/~hak`.

Aït-Kaci, H., Dumant, B., Meyer, R., Podelski, A., y van Roy, P. (1994). The Wild LIFE Handbook (version 1.02). Prepublication edition, Digital Equipment Corporation (Paris Research Laboratory). `www.isg.sfu.ca/life/`.

Aït-Kaci, H. y Lincoln, P. (1989). LIFE – A natural language for natural language. *Traitement Automatique des Langues – Informations*, 30(1-2):37–67. Paris.

Aït-Kaci, H. y Nasr, R. (1986). LOGIN: A Logic Programming Language with Built-In Inheritance. *JLP*, 3(3):185–215.

Aït-Kaci, H. y Nasr, R. (1987). LeFun: Logic, Equations and Functions. En *1987 Symp. on Logic Programming.* San Francisco.

Aït-Kaci, H. y Podelski, A. (1991). Toward a meaning of LIFE. En Maluszynski, J. y Wirsing, M. (eds.), *3rd International Symposium on Programming Languages: Implementations, Logics and Programs (PLILP'91)*, volumen 528 de *LNCS*, pp. 255–274. Springer-Verlag.

Augustsson, L. (1999). Cayenne – a Language with Dependent Types. *ACMSN*, 34(1):239–250.

Augustsson, L. y Carlsson, M. (2000). An exercise in dependent types: A well–typed interpreter. Tech. rep., Dep. of Computing Sciencies, Chalmers University of Technology. `www.cs.chalmers.se/~augustss`.

Backhouse, R., Jansson, P., Jeuring, J., y Meertens, L. (1999). Generic Programming – An Introduction. En Swierstra, S. D., Henriques, P., y Oliveira, J. (eds.), *Advanced Functional Programming*, volumen 1608 de *LNCS*, pp. 28–115. Springer-Verlag. `http://www.cs.uu.nl/~johanj/publications/portugal.ps`.

Backus, J. (1978). Can programming be liberated from the von Neumann style? *CACM*, 21(8):613–641. Reimpreso en [Horowitz, 1987]:174-202.

Bailey, R. (1985). A Hope Tutorial. *Byte Magazine*, Agosto:235–258.

Barbuti, R., Bellia, M., Levi, G., y Martelli, M. (1986). LEAF: A Language which Integrates Logic, Equations and Functions. En DeGroot, D. y Lindstrom, G. (eds.), *Logic Programming: Functions, Relations, and Equations*, pp. 201–238. Prentice-Hall, Englewood Cliffs, NJ.

Barendregt, H. P. (1984). *The Lambda Calculus, Its Syntax and Semantics*, volumen 103 de *Studies in Logic and the Foundations of Mathematics.* North-Holland, Amsterdam. Edición revisada de la primera (1981).

Barendregt, H. P. (1992). Lambda Calculi With Types. En Abramsky, S., Gabbay, D., y Maibaum, T. S. (eds.), *Handbook of Logic in Computer Science*, capítulo 2.2, pp. 117–309. Oxford University Press.

Barendregt, H. P. y Geuvers, H. (1999). Proof–assistants using Dependent Type Systems. En Robinson, A. y Voronkov, A. (eds.), *Handbook of Automated Reasoning*, capítulo 1, pp. 1–84. Elsevier.

Barendregt, H. P., Kennaway, J. R., Klop, J. W., y Sleep, M. R. (1987a). Needed Reduction and Spine Strategies for the Lambda Calculus. *Information and Computation*, 75(3):191–231.

Barendregt, H. P., van Eekelen, M. C. J. D., Glauert, J. R. W., Kennaway, J. R., Plasmeijer, M. J., y Sleep, M. R. (1987b). Term Graph Rewriting. En de Bakker, J. W., Nijman, A. J., y Treleaven, P. C. (eds.), *Proc. PARLE'87 Conference, vol.II*, volumen 259, pp. 141–158, Eindhoven, The Netherlands. También como Tech. Rep. 87, University of Nijmegen.

Barendregt, H. P. y van Leeuwen, M. (1986). Functional Programming and the Language TALE. En *Current Trends in Concurrency*, volumen 224, pp. 122–207, Eindhoven, The Netherlands. También como Tech. Rep. 412, University of Utrecht Dept. of Mathematics.

Benhamou, F. y Touraïvane (1995). Prolog IV: Langage et Algorithmes. En *Quatrièmes Journées Francophones de Programmation en Logique, JFPL'95*, pp. 51–65.

Bergadano, F. y Gunetti, D. (1995). *Inductive Logic Programming: From Machine Learning to Software Engineering*. The MIT Press.

Bidoit, M., Kreowski, H.-J., Lescanne, P., Orejas, F., y Sannella, D. (1991). *Algebraic system specification and development: a survey and annotated bibliography*, volumen 501 de *LNCS*. Springer-Verlag.

Bird, R. (1998). *Introduction to Functional Programming using Haskell*. Prentice Hall, 2 edición. Esta es una versión revisada de [Bird y Wadler, 1988]. Existe también una traducción al castellano, publicado por la misma editorial, en 1999.

Bird, R. y de Moor, O. (1997). *Algebra of Programming*. CAR Hoare Series Editor. Prentice Hall, Hemel Hempstead, England.

Bird, R. y Wadler, P. (1988). *Introduction to Functional Programming*. Prentice–Hall.

Bobrow, D. G. (1985). If Prolog is the Answer, What is the Question?, or What if Take to Support AI Programming Paradigms. *IEEE Transactions on Software Engineering*, SE-11(11):1401–1408.

Bobrow, D. G. y Kahn, K. (1985). CommonLoops: Merging CommonLisp and Object Oriented Programming. Informe Técnico núm. ISL–85–8, XEROX, Palo Alto Research Center: Intelligent Systems Lab. También en [ACM, 1987]:17–29.

Boehm, H., Demers, A., y Donahue, J. (1980). An Informal Description of Russell. Informe Técnico núm. TR 80-430, Department of Computer Science, Cornell University.

Böhm, C. y Gross, W. (1966). Introduction to the CUCH. En Caianiello, E. R. (ed.), *Automata Theory*, pp. 35–65. Academic Press, New York.

Böhm, C., Piperno, A., y Guerrini, S. (1994). Lambda-definition of function(al)s by normal forms. *LNCS*, 788:135–154. ESOP'94.

Boley, H. (1996). Functional-Logic Integration via Minimal Extensions. En Breitinger, S., Kroeger, H., y Loogen, R. (eds.), *5th International Workshop on Functional and Logic Programming*. University of Giessen.

Boley, H., Andel, S., Elsbernd, K., Herfert, M., Sintek, M., y Stein, W. (1996). RELFUN Guide: Programming with Relation and Functions Made Easy. Document D-93-12, DFKI, Universität Kaiserslautern.

Borland (1986). *Turbo-Prolog. Reference Manual*. Borland International, Scotts Valley, California.

Borland (1988). *Turbo Prolog 2.0*. Borland International, Scotts Valley, California.

Bowen, D. y Byrd, L. (1983). A Portable Prolog compiler. En Pereira, L. (ed.), *Proc. of the Logic Programming Workshop 1993*. Universidade nova de Lisboa.

Breitinger, S., Klusik, U., y Loogen, R. (1998). From (Sequential) Haskell to (Parallel) Eden: An Implementation Point of View. En *International Symposium on Programming Languages: Implementations, Logics and Programs (PLILP'98)*, volumen 1490 de *LNCS*. Springer-Verlag. También como Technical Report 96-10, Philips Universität Marburg.

Breitinger, S., Loogen, R., y Ortega Mallén, Y. (1995). Concurrency in Functional and Logic Programming. En *Fuji Intern. Workshop on Functional and Logic Programming*. World Scientific Publ. `http:// www.mathematik.uni-marburg.de/ ~loogen/paper/ fuji_ws.ps`.

Bruynooghe, M. (1982). Adding redundancy to obtain more reliable and readable Prolog programs. En Van Caneghem, M. (ed.), *Proceedings of the First International Conference on Logic Programming*, pp. 129–133, Marseille, France. ADDP-GIA.

Bundy, A. (1988). A broader interpretation of Logic in Logic Programming. En Bowen, K. y Kowalski, R. (eds.), *Proc. of fifth International Conference and Symposium on Logic Programming*, pp. 1624–1648. MIT–Press.

Burn, G. (ed.) (1991). *Lazy Functional Languages: Abstract Interpretation and Compilation*. MIT–Press, Cambridge, MA.

Burstall, R., McQueen, P., y Sannella, D. (1980). HOPE – And Experimental Applicative Language. Informe Técnico núm. CSR–62–80, Department of Computer Science, University of Edinburgh.

Burton, F. (1983). Annotations to Control Parallelism and Reduction Order un the Distributed Evaluation of Functional Programs. *ACM TOPLAS*, 3(2):329–366.

Cabeza, D., Hermenegildo, M., y Varma, S. (1996). The PiLLoW/CIAO Library for Internet/WWW Programming using Computational Logic Systems. En Tarau, P., Davison, A., DeBosschere, K., y Hermenegildo, M. (eds.), *First Workshop on Logic Programming Tools for Internet Applications*. `http://clement.info.umoncton.ca/char126lpnet/jicslp96`.

Cardelli, L. y Wegner, P. (1995). On understanding type, data abstraction and polymorphism. *ACM Computing Surveys*, 17(4):471–522.

Carlsson, M. y Widen, J. (1988). SICSTus–Prolog user's Manual. Tech. Rep. 88007B, Swedish Institute of Computer Science, Kista, Sweden.

Carriero, N. y Gelernter, D. (1989). LINDA in Context. *CACM*, 32(4):444–458.

Cauchy, A. L. (1821). Cours d'analyse. En *Oeuvres Complétes (II^e Série)*, volumen 3. École Royale Polytechnique. Reeditado en forma facsimilar por SAEM Thales (1999).

Chakravarty, M., Guo, Y., y Köhler, M. (1995). Goffin: Higher Order Functions meet Concurrent Constraints. En *First International Workshop on Concurrent Constraint Programming*.

Charatonik, W. y Podelski, A. (1998). Directional Type Inference for Logic Programs. *LNCS*, 1503:278–??

Cheong, P. y Fribourg, L. (1993). Implementing of Narrowing: The Prolog-based Approach. En Apt, K., de Bakker, J., y Rutten, J. (eds.), *Logic Programming Languages. Constraint, Functions, and Objects*, pp. 1–20. MIT Press, Cambridge, MA.

Chiu, C. (1994). Towards Practical Interval Constraint Solving in Logic Programming. Tech. report, Univ. Hong Kong.

Church, A. (1932). A set of postulates for the foundations of logic. *Annals of Mathematics*, 33:346–366.

Church, A. (1933). A set of postulates for the foundations of logic. *Annals of Mathematics*, 34:839–864.

Church, A. (1936). An Unsolvable Problem in Elementary Number Theory. *Amer. J. Math.*, 58:345–363.

Church, A. (1940). A simple theory of types. *JSL*, 5:56–68.

Church, A. (1941). *The calculi of lambda conversion*. Princeton Univ. Press.

Church, A. y Rosser, J. B. (1936). Some Properties of Conversion. *Transactions of the American Mathematical Society*, 39:472–482.

Clark, K. L. y Gregory, S. (1984). Parlog: Parallel Programming in Logic. Tech. rep., Dept. of Computing, Imperial College, Londres. También en *ACM Trans. Prog. Lang. Syst.*, 8(1): 1–49, January (1986). También en [Shapiro, 1987]:84-139.

Clinger, W. y Rees, J. (1991). The revised^4 report on the algorithmic language Scheme. *ACM Lisp Pointer IV*, 4(3). `http://www.cs.indicana.edu/scheme-repository/R4RS/r4rs_toc.html`.

Clocksin, W. y Mellish, C. (1981). *Programming in PROLOG*. Springer-Verlag. Traducido al castellano en Gustavo-Gili, Barcelona (1987).

Cohen, J. (1988). A View of The Origins and Development of Prolog. *CACM*, 31(1):26–36.

Cohen, J. (1999). Computational Molecular Biology: A Promising Application Using LP and its Extensions. En Apt, W. R., Marek, V. W., Truszczynski, M., y Warren, D. S. (eds.), *The Logic Programming Paradigm. A 25–Year Perspective*, capítulo 5, pp. 281–312. Springer.

Colmerauer, A. (1982). Prolog II Reference Manual and Theoretical Model. Internal report, GroupeIA, Unv. Aix-Marseille.

Colmerauer, A. (1987). Opening then Prolog III Universe. *Byte Magazine*, Agosto:177–182.

Colmerauer, A. (1990). An Introduction to Prolog III. *CACM*, 33(7):69–90.

Colmerauer, A., Kanoui, H., y Caneghem, M. V. (1982). Prolog, theoretical principles and current trends. *Technology and Science of Informatics*, 2(4):255–292.

Colmerauer, A., Kanoui, H., Pasero, R., y Roussel, P. (1973). Un Système de Communication Homme-Machine en Français. Rapport, Groupe d'Intelligence Artificielle, Université d'Aix-Marseille II.

Conlon, T. (1989). *Programming in PARLOG*. Addison-Wesley.

Conlon, T. y Gregory, S. (1990). *Hands On PC-Parlog 2.0*. Parallel Logic Programming Ltd., Twickenham, UK.

Constable, R. L. y Smith, S. F. (1993). Computational foundations of basic recursive function theory. *TCS*, 121(1-2):89–112.

Cousineau, G. (1997). Functional Programming and Geometry. En Hugh Glaser, P. H. y Kuchen, H. (eds.), *International Symposium on Programming Languages: Implementations, Logics and Programs (PLILP'97)*, volumen 1292 de *LNCS*, pp. 309–322. Springer-Verlag.

Curry, H. B. (1934). Functionality in Combinatory Logic. En *Proc. Nat. Acad. Science USA*, volumen 20, pp. 584–590.

Curry, H. B. (1942). The Inconsistency of Certain Formal Logics. *JSL*, 7:115–117.

Curry, H. B. (1969). Modified basic functionality in combinatory logic. *Dialectica*, 23:83–92.

Curry, H. B. (1977). *Foundations of mathematical logic*. Constable. Originalmente publicado por Dover Publications, New York (1977), y por McGraw-Hill, London.

Curry, H. B. y Feys, R. (1958). *Combinatory Logic*, volumen 1. North Holland.

Curry, H. B., Hindley, J. R., y Seldin, J. P. (1972). *Combinatory Logic*, volumen 2. North-Holland, Amsterdam.

Dahl, V. (1999). The Logic of Language. En Apt, W. R., Marek, V. W., Truszczynski, M., y Warren, D. S. (eds.), *The Logic Programming Paradigm. A 25–Year Perspective*, capítulo 9, pp. 429–456. Springer.

Dalmas, S. (1992). A polymorphic functional language applied to symbolic computation. En Wang, P. S. (ed.), *International System Symposium on Symbolic and Algebraic Computation 92*, pp. 369–375, New York,. ACM Press.

Damas, L. y Milner, R. (1982). Principal Type-Schemes for Functional Programs. En DeMillo, R. (ed.), *Conference Record of the Ninth Annual ACM Symposium on Principles of Programming Languages*, pp. 207–212, Albuquerque, NM. ACM Press.

Darlington, J. (1983). Unifying logic and functional languages. Research report, Imperial College.

Darlington, J. (1985). Program Transformation. *Byte Magazine*, Agosto:201–216.

Das, S. (1992). *Deductive Databases and Logic Programming*. Addison–Wesley.

Davie, A. (1992). *An Introduction to Functional Programming Systems using Haskell*. Cambridge University Press.

Davies, J., Weeks, R., y Revett, M. (1995). Jasper: Communicating Information Agents for WWW. En O'Reilly and Associates y Web Consortium (W3C) (eds.), *World Wide Web Journal: The Fourth International WWW Conference Proceedings*, pp. 473–482. O'Reilly & Associates, Inc.

Davis, R. (1985). Logic Programming and Prolog: A tutorial. *IEEE Transactions on Software Engineering*, 2(5):53–62. Reimpreso en [Horowitz, 1987]:493–502.

Davison, A. (1989). *Polka: A Parlog Object-Oriented Language*. Tesis Doctoral, Imperial College, University of London.

Davison, A. (1992). A survey of logic programming-based object-oriented language. Research report, University of Melbourne.

Day, N. A., Launchbury, J., y Lewis, L. (1999). Logical Abstractions in Haskell. En Meijer, E. (ed.), *Proceedings of the 1999 Haskell Workshop*. University of Utrecht (tech. rep. UU–CS–1999–28).

de Boer, F., Kok, J., Palamidessi, C., y Rutten, J. (1993). A Paradigm for Asynchronous Communication and its Application to Concurrent Constraint Programming. En Apt, W. R., de Bakker, J., y Rutten, J. (eds.), *Logic Programming Languages, Constraints, Functions and Objects*, capítulo 4, pp. 81–114. MIT Press, Cambridge, MA.

de Bruijn, N. G. (1972). Lambda Calculus Notation with Nameless Dummies: A tool for automatic formula manipulation. *Indagationes Mathematicae*, 34:381–392.

de Kergommeaux, J. C. y Codognet, P. (1994). Parallel logic programing systems. *ACM Computing Surveys*, 26(3):295–336.

Deransart, D., Ed-Dbali, A., y Cervoni, L. (1996). *Prolog: the Standard*. Springer-Verlag.

Dershowitz, N. y Jouannaud, J.-P. (1990). Rewriting Systems. En van Leeuwen, J. (ed.), *Handbook of Theoretical Computer Science*, pp. 243–320. Elsevier Publishers, Amsterdam.

Devienne, P., Lebègue, P., Parrain, A., Routier, J., y Würtz, J. (1996). Smallest Horn Clause Programs. *JLP*, 27(3):227–267.

Deville, Y. (1990). *Logic programming: systematic program development*. Addison-Wesley.

Dewdney, A. (1987a). Montañas fractales, plantas graftales y múltiples ordinográficos en Pixar. *Investigación y Ciencia*, Febrero.

Dewdney, A. (1987b). Un microscopio computarizado escudriña el objeto más complejo de la matemática (el conjunto de Mandelbrot). *Investigación y Ciencia*, Octubre.

Dewdney, A. (1988). El conjunto de Mandelbrot y una hueste de primos suyos, de apellido Julia. *Investigación y Ciencia*, Enero:94–97.

Dewdney, A. (1989). Capturas del día: biomorfos sobre teselas, con guarnición de palomitas y caracoles. *Investigación y Ciencia*, Septiembre:86–90.

Dijkstra, E. W. (1976). *A Discipline of Programming*. Prentice-Hall.

Dijkstra, E. W. y Scholten, C. S. (1990). *Predicate Calculus and Program Semantics*. Springer-Verlag, New York.

Dincbas, M., van Hentenryck, P., Simonis, H., Aggoun, A., Graf, T., y Berthier, F. (1988). The Constraint Logic Programming Language CHIP. En ICOT (ed.), *2nd Intern. Conference on Fifth Generation Computer Systems*, pp. 249–264, Institute for New Generation Computer Technology, Tokyo. Springer-Verlag.

Dzeroski, S., De Raedt, L., y Blockeel, H. (eds.) (1998). *Proc. of the 8th International Conference on Inductive Logic Programming*, volumen 1446 de *LNAI*. Springer-Verlag.

Eisenstadt, M. (1983). A User-Friendly Software Environment for the Novice Programmer. *CACM*, 27(12):1056–1064.

Eisinger, N., Gesiler, T., y Panne, S. (1997). Logic Implemented Functionally. En Hugh Glaser, P. H. y Kuchen, H. (eds.), *International Symposium on Programming Languages: Implementations, Logics and Programs (PLILP'97)*, volumen 1292 de *LNCS*, pp. 351–368. Springer-Verlag.

Elliot, C. y Hudak, P. (1997). Functional reactive animation. En *ACM International Conf. on Functional Programming*.

Felty, A. (1997). A Tutorial on Lambda Prolog and its Applications to Theorem Proving. `cm.bell-labs.com/who/felty/`.

Fernández Leiva, A. (1994). Un sistema de clases para Prolog. Proyecto de fin de carrera, dirigido por B.C. Ruiz Jiménez, Departamento de Lenguajes y Ciencias de la Computación. Universidad de Málaga.

Field, A. y Harrison, P. (1988). *Functional Programming*. Addison-Wesley.

Findler, R., Flanagan, C., Flatt, M., Krishnamurthi, S., y Felleisen, M. (1997). DrScheme: A pedagogic programming environment for Scheme. En *International Programming Symposium on Programming Language: Implementation, Logic and Programs*, volumen 1292 de *LNCS*, pp. 369–388. Springer–Verlag.

Flanagan, D. (1996). *Java in a Nutshell: A Desktop Quick Reference for Java Programmers*. Nutshell handbook. O'Reilly & Associates, Inc., California, USA.

Foster, I. y Taylor, S. (1992). Productive Parallel Programming: The PCN Approach. *Sci Prog*, 1(1):51–66.

Freeman-Benson, B. (1990). Kaleidoscope: Mixing Objects, Constraints and Imperative Programming. En ACM (ed.), *OOPSLA/ECOOP'90*, volumen 25 (10), pp. 77–88.

Fribourg, L. (1985). SLOG: A Logic Programming Language Interpreter Based on Clausal Superposition and Rewriting. En *Proceedings of the International Symposium on Logic Programming*, pp. 172–184. IEEE Computer Society, Technical Committee on Computer Languages, The Computer Society Press.

Friedman, D. y Wise, D. (1976). Cons should not evaluate its arguments. En *3rd International Colloquium on Automata, Language and Programming*, pp. 257–284, Edinburgh. University Press.

Fukunaga, K. y Hirose, S. I. (1987). An Experience with a Prolog-based Object-Oriented Language. En ACM (ed.), *Object-Oriented Programming Systems, Languages and Applications (OOPSLA'86)*, volumen 21(11) de *ACMSN*, pp. 224–231. Ver también *Vulcan: Logical Concurrent Objects*, en [Shapiro, 1987]:274-303.

Furukawa, K. (1999). From Deduction to Induction: Logical Perspective. En Apt, W. R., Marek, V. W., Truszczynski, M., y Warren, D. S. (eds.), *The Logic Programming Paradigm. A 25–Year Perspective*, capítulo 6, pp. 347–354. Springer.

Furukawa, K., Takeuchi, A., Kunifuji, S., Yasukawa, H., Ohki, M., y Ueda, K. (1986). Mandala: A logic based knowledge programming system. En Meyrowitz, N. (ed.), *International Conference on Fifth Generation Computer Systems*, pp. 613–622.

Gallardo, J. E. (1994). Un entorno para Haskell realizado en Windows. Proyecto de fin de carrera, dirigido por B.C. Ruiz Jiménez, Dpto. de Lenguajes y Ciencias de la Computación. Univ. de Málaga.

Gallardo, J. E., Guerrero, P., y Ruiz, B. (1994). Mónadas y Procesos Funcionales Comunicantes. En Alpuente, M., Barbuti, R., y Ramos, I. (eds.), *1994 Joint Conference Declarative Programming (GULP-PRODE'94)*, pp. 266–280. Universidad de Valencia, SPUPV.

Gallardo, J. E., Guerrero, P., y Ruiz, B. (1995a). Comunicación Monádica de Objetos Funcionales. En Troya, J. M. y Rodríguez, C. (eds.), *I Jornadas de Informática*, pp. 145–154. AEIA.

Gallardo, J. E., Guerrero, P., y Ruiz, B. (1995b). Mónadas para la comunicación de Objetos Funcionales. En Alpuente, M. y Sessa, M. (eds.), *1995 Joint Conference on Declarative Programming) (GULP-PRODE'95)*, pp. 471–476. Universita' degli Studi di Salerno (Italy).

Gallardo, J. E., Gutiérrez, F., y Ruiz, B. (1996). Clasificación y Comunicación de objetos en Concurrent Haskell. En Clares, B. (ed.), *II Jornadas de Informática*, pp. 555–564. AEIA.

Gallardo, J. E., Gutiérrez, F., y Ruiz, B. (1997). Inheritance and Selective Method Dispaching in Concurrent Haskell. En Hugh Glaser, P. H. y Kuchen, H. (eds.), *International Symposium on Programming Languages: Implementations, Logics and Programs (PLILP'97)*, volumen 1292 de *LNCS*, pp. 33–46. Springer-Verlag.

Gaster, B. R. y Jones, M. P. (1996). A Polymorphic Type System for Extensible Records and Variants. Informe Técnico NOTTCS-TR-96-3, Dep. of Computer Science, University of Nottingham, Languages and Programming Group, Department of Computer Science, Nottingham NG7 2RD, UK. `ftp://ftp.cs.nott.ac.uk/techreports/96/3.ps.gz`.

Gazdar, G. y Mellish, C. (1989). *Natural Language Processing in PROLOG*. Addison–Wesley.

Gehani, N. y McGettrick, A. (1988). *Concurrent Programming*. Addison-Wesley.

Gelernter, D. (1985). Generative Communication in Linda. *ACM TOPLAS*, 7(1):80–112. `http://www.cs.yale.edu/HTML/YALE/CS/Linda`.

Gelernter, D. y Carriero, N. (1992). Coordination languages and their significance. *CACM*, 35(2):96–107.

Giacalone, A., Mishra, P., y Prasad, S. (1989). Facile: A symetric integration of concurrent and functional programming. *Journal of Parallel Programming*, 18(2):121–160.

Giannesini, F., Kanoui, H., Pasero, R., y van Caneghem, M. (1985). *Prolog*. Interditions, París. El texto original es en francés y existe una versión en inglés publicada por Addison-Wesley en 1989.

Giegerich, R., Hinze, R., y Kurtz, S. (1999). Straight to the Heart of Computer Science via Functional Programming. En Felleisen, M., Hanus, M., y Thompson, S. (eds.), *Proceedings of the Workshop on Functional and Declarative Programming in Education, FDPE'99*. Technical Report of Rice University, Rice COMP TR99-346. `http://www.cs.rice.edu/~matthias/FDPE99`.

Glauert, J. R. W., Kennaway, J. R., y Sleep, M. R. (1987). DACTL: A Computational Model and Compiler Target Language Based on Graph Reduction. *ICL Technical Journal*, 5(3):509–537.

Goguen, J. y Malcolm, G. (1996). *Algebraic Semantics of Imperative Programs*. MIT–Press, Cambridge, MA.

Goguen, J. A. y Meseguer, J. (1986). Eqlog: Equality, Types, and Generic Modules for Logic Programming. En DeGroot, D. y Lindstrom, G. (eds.), *Functional and Logic Programming*, pp. 295–363. Prentice-Hall, Inc., Englewood Cliffs, N.J. También en *JLP*, 1(2):179-209, 1984.

Gold Hill Computers (1987). *Golden Common LISP 1.1 – Operating Guide*.

Green, C. (1969). Aplications of Theorem Proving to Problem Solving. En *Proceedings of IJCAI'69*, pp. 219–239.

Gregory, S. (1987). *Parallel logic programming in Parlog: the language and its implementation*. Addison-Wesley.

Gregory, S. (1996). Derivation of concurrent algorithms in Tempo. En *Proceedings of the 5th International Workshop on Logic Program Synthesis and Transformation*, volumen 1048 de *LNCS*, pp. 46–60. Utrecht, September 1995, Springer-Verlag. `http://star.cs.bris.ac.uk/papers/derive.ps.gz`.

Gregory, S. (1997). A declarative approach to concurrent programming. En *Proceedings of the 9th International Symposium on Programming Languages: Implementations, Logics, and Programs*, volumen 1292 de *LNCS*, pp. 79–93. Springer-Verlag. `http://star.cs.bris.ac.uk/papers/newtempo.html`.

Gregory, S. y Ramírez, R. (1995). Tempo: a declarative concurrent programming language. En *Proceedings of the 12th International Conference on Logic Programming*, pp. 515–529, Cambridge, Mass. MIT Press. `http://star.cs.bris.ac.uk/papers/tempo.ps.gz`.

Guezzi, C. y Jazayeri, M. (1982). *Programming Languages Concepts*. J. Wiley. Traducción al castellano en Díaz de Santos, Madrid (1986).

Hall, C., Hammond, K., Partain, W., Peyton Jones, S. L., y Wadler, P. (1992). The Glasgow Haskell compiler: a retrospective. En *Glasgow Workshop on Functional Programming*, Ayr, Scotland. Workshops in Computing, Springer-Verlag.

Hanus, M. (1990). Compiling Logic Programs with Equality. En *2nd International Symposium on Programming Language: Implementation and Logic Programming (PLILP'90)*, volumen 456 de *LNCS*, pp. 387–401. Springer Verlag.

Hanus, M. (1991a). The ALF-System. En Maluszynski, J. y Wirsing, M. (eds.), *3rd International Symposium on Programming Languages: Implementations, Logics and Programs (PLILP'91)*, volumen 528 de *LNCS*, pp. 423–424. Springer-Verlag.

Hanus, M. (1991b). Efficient Implementation of Narrowing and Rewriting. En Boley, H. y Richter, M. M. (eds.), *Proceedings of the International Workshop on Processing Declarative Knowledge (PDK'91)*, volumen 567 de *LNAI*, pp. 344–365, Kaiserslautern, FRG. Springer-Verlag.

Hanus, M. (1994). The Integration of Functions into Logic Programming: From Theory to Practice. *JLP*, 19,20:583–628. `www-i2.informatik.rwth-aachen.de/~hanus/`.

Hanus, M. (1997). Teaching Functional and Logic Programming with a Single Computation Model. En Hugh Glaser, P. H. y Kuchen, H. (eds.), *International Symposium on Programming Languages: Implementations, Logics and Programs (PLILP'97)*, volumen 1292 de *LNCS*, pp. 335–350. Springer-Verlag.

Hanus, M. (2000). Server Side Web Scripting in Curry. En Alpuente, M. (ed.), *9th International Workshop on Functional and Logic Programming (WFLP'2000)*, pp. 366–381. Dep. de Sistemas Informáticos y Computación, Universidad Politécnica de Valencia (ref. 2000.2039). September 28-30, Benicassim, Spain.

Hanus, M. y Schwab, A. (1995). ALF User's manual. Tech. report, Universität Dortmund, Germay.

Harper, R. (1999). Proof-directed Debugging: An Exercise in Reasoning About Programs. En Felleisen, M., Hanus, M., y Thompson, S. (eds.), *Proceedings of the Workshop on Functional and Declarative Programming in Education, FDPE'99*. Technical Report of Rice University, Rice COMP TR99-346. `http://www.cs.rice.edu/~matthias/FDPE99`.

Harrison, P. y Khoshnevisan, H. (1985). Functional Programming Using FP. *Byte Magazine*, Agosto:219–232.

Hayden, M. (2000). Distributed communication in ML. *JFP*, 10(1):91–120.

Henderson, P. y Morris, J. (1976). A Lazy evaluator. En *3rd Annual ACM Symposium on Principles of Programming Languages*, pp. 95–103. ACM Press.

Hennessy, M. (1990). *The Semantics of Programming Languages; An Elementary Introduction using Structural Operational Semantics*. Wiley.

Hernández, F., Peña, R., y Rubio, F. (1999). From GranSim to Paradise. En *SFP'99*. To appear in Intellect (2000).

Höhfeld, M. y Smolka, G. (1988). Definite relations over Constraint Languages. Report 53, LILOG, Stuttgart.

Hilbert, D. y Ackermann, W. (1928). *Grundzüge der theoretischen Logik*. Springer-Verlag, Berlin.

Hill, P. y Lloyd, J. (1994). *The Gödel programming language*. MIT Press.

Hindley, J. (1969). The Principal Type-Scheme of an Object in Combinatory Logic. *Transactions of the American Mathematical Society*, 146:29–60.

Hindley, J. R. (1985). Combinators and Lambda-calculus, a short outline. En Cousineau, G. (ed.), *Combinators and Functional Programming Languages - 13th Spring School of the LITP Proceedings*, volumen 242 de *LNCS*. Springer-Verlag.

Hindley, J. R. (1997). *Basic Simple Type Theory*, volumen 42 de *Cambridge Tracts in Theoretical Computer Science*. Cambridge University Press, Cambridge, UK.

Hindley, J. R. y Seldin, J. P. (eds.) (1980). *To H. B. Curry: Essays on Combinatory logic, Lambda Calculus, and Formalism*. Academic Press.

Hindley, J. R. y Seldin, J. P. (1986). *Introduction to combinators and λ-calculus*. Cambridge University Press. Reimpresión de 1988.

Hinze, R. (1999). A Generic Programming Extension for Haskell. En Meijer, E. (ed.), *Proceedings of the 1999 Haskell Workshop*. University of Utrecht (tech. rep. UU–CS–1999–28).

Hinze, R. (2000). A New Approach to Generic Functional Programming. En *Conference Record of POPL'00: The 27th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pp. 119–132, Boston, Massachusetts.

Hoare, C. (1962). Quicksort. *The Computer Journal*, 5(1):10–16.

Hoare, C. (1969). An Axiomatic Basis for Computer Programming. *CACM*, 12(10):576–580. Reimpreso en *C.ACM*, 26(1):53-56, 1983, y también en [Hoare y Jones, 1989]:45-58.

Hoare, C. (1978). Communicating Sequential Processes. *CACM*, 21(8). Reimpreso en [Gehani y McGettrick, 1988]:278-308, y también en [Horowitz, 1987]:311-322.

Hoare, C. (1985). *Communicating Sequential Processes*. Prentice–Hall, New Jersey.

Hoare, C. y Jifeng, H. (1998). *Unifying Theories of Programming*. Prentice Hall.

Hoare, C. y Jones, C. (1989). *Essays in Computing Science*. Prentice-Hall.

Hodgson, J. (1998). Validation suite for conformance with part 1 of the Standard. Informe técnico, University of Amsterdam. `www.sju.edu/~jhodgson/pun/suite.tar.gs`.

Hofstadter, D. (1979). *Gödel, Escher, Bach: an Eternal Golden Braid*. Basic Books. Traducido al castellano con el título *Gödel, Escher, Bach: un Eterno y Grácil Bucle*. Tusquets, Barcelona (1987).

Holmstrom, S. (1982). PFL: A Functional Language for Parallel Programming. En London (ed.), *Declarative Language Workshop*, pp. 114–139.

Hook, J. G. (1984). Understanding Russell: A First Attempt. En Kahn, G., MacQueen, D. B., y Plotkin, G. D. (eds.), *Semantics of Data Types*, volumen 173 de *LNCS*, pp. 69–86. Springer, Berlin.

Horn, B. (1992). Constraint Patterns as a Basis for Object-Oriented Constraint Programming. En ACM (ed.), *OOPSLA'92*, ACMSN. Septiembre.

Horowitz, E. (1984). *Fundamentals of Programming Languages*. Computer Science Press, Maryland. Esta segunda edición incluye un capítulo de LOO escrito por Tim Rentsch.

Horowitz, E. (1987). *Programming Languages. A grand Tour*. Computer Science Press. Segunda edición (la primera es de 1983).

Hudak, P. (1989). Conception, evolution, and application of Functional Programming Languages. *ACM Computing Surveys*, 21(3):359–411.

Hudak, P. (2000). *The Haskell School of Expression. Learning Functional Programming through Multimedia*. Cambridge University Press. `haskell.org/soe`.

Hudak, P. y Fasel, J. (1992). A Gentle introduction to Haskell. *ACMSN*, 27(5).

Hudak, P. y Jones, M. (1994). Haskell Vs. Ada Vs. C++ Vs. Awk Vs. ... An Experiment in Software Prototyping Productivity. Supported by the Advanced Research Project Agency and the Office of Naval Research under Arpa Order 8888, Contract N00014-92-C-0153.

Hudak, P., Peyton Jones, S., y Wadler, P. (1992). Report on the Programming Language Haskell. A Non-strict, Purely Functional Language, Version 1.2. *ACMSN*, 27(5).

Huet, G. P. (1975). A Unification Algorithm for Typed λ-Calculus. *TCS*, 1:27–57.

Hughes, J. (1990). Why Functional Programming Matters. En Turner, D. A. (ed.), *Research Topics in Functional Programming*, UT Year of Programming Series, capítulo 2, pp. 17–42. Addison-Wesley.

Hughes, J. y Sparud, J. (1995). Haskell++: An Object–Oriented Extension of Haskell. Tech. report, Dep. of Computer Sicence, Chalmers University.

Hughes, R. (1983). *The Design and Implementation of Programming Languages*. Tesis Doctoral, Oxford University.

Hullot, J.-M. (1980a). Canonical Forms and Unification. En Kowalski, R. (ed.), *Proceedings of the Fifth International Conference on Automated Deduction*, volumen 87 de *LNCS*, pp. 318–334, Berlin. Springer-Verlag.

Hullot, J.-M. (1980b). *Compilation de Formes Canoniques dans les Théories équationelles*. Thèse de Doctorat de Troisième Cycle, Université de Paris Sud, Orsay (France).

Hutton, G. y Meijer, E. (1998). Functional Pearl: Monadic parsing in Haskell. *JFP*, 8(4):437–444. `www.cs.nott.ac.uk/Department/Staff/gmh/monparsing.ps`.

Ishikawa, Y. y Tokoro, M. (1987). A Concurrent Object-Oriented Knowledge Representation Language Orient 84/K : Its Features and Implementation. En ACM (ed.), *Object-Oriented Programming Systems, Languages and Applications (OOPSLA'86)*, volumen 21(11) de *ACMSN*, pp. 232–241.

ISL (1997). *SICStus Prolog User's Manual*. Intelligent Systems Laboratory, Swedish Institute of Computer Science.

Iverson, K. (1962). *A Programming Language*. Wiley, New York.

Iverson, K. y Falkoff, A. (1973). The Design of APL. *IBM Journal of Research and Development*, pp. 324–334. Reimpreso en [Horowitz, 1987]:240–250.

Jaffar, J. (1987). Constraint Logic Programming. En ACM (ed.), *14th POPL*.

Jaffar, J. (1990). The CLP(R) Language and System. Research Report RC16292, IBM.

Jaffar, J. (1992). An Abstract Machine for CLP(R). En *Conf. on Prog. Lang. and Impl.*, pp. 128–139. ACM SIGPLAN.

Jaffar, J. y Maher, M. J. (1994). Constraint Logic Programming: A Survey. Informe Técnico núm. 19442, IBM Research Division, Thomas J.Watson Research Center, Yortown Heights, NY 10598.

Janson, S. (1992). *AKL – A Multiparadigm Programming Language*. Tesis Doctoral, Computing Science Dept., Uppsala University.

Jeffery, D., Dowd, T., y Somogyi, Z. (1999). MCORBA: A CORBA Binding for Mercury. En *First International Workshop on Practical Aspects of Declarative Languages (PADL'99)*, volumen 1551 de *LNCS*, pp. 211–227. Springer-Verlag. `http://www.cs.mu.oz.au/research/mercury/`.

Jeffery, D., Henderson, F., y Somogyi, Z. (1998). Type classes in Mercury. Technical Report 98/13, Department of Computer Science, University of Melbourne, Melbourne, Australia. `http://www.cs.mu.oz.au/research/mercury/`.

Jeuring, J. y Swierstra, D. (1999). Advanced Programming Concepts in a Course on Grammars an Parsing. En Felleisen, M., Hanus, M., y Thompson, S. (eds.), *Proceedings of the Workshop on Functional and Declarative Programming in Education, FDPE'99*. Technical Report of Rice University, Rice COMP TR99-346. `http://www.cs.rice.edu/~matthias/FDPE99`.

JFP (1993). Functional Programming in Education. *JFP*, 3(1).

Jiménez Santana, M. A. (1996). Implementación de un Intérprete Funcional del Lambda Cálculo basado en la Máquina G. Proyecto de fin de carrera, dirigido por F. Gutiérrez López, Dpto. de Lenguajes y Ciencias de la Computación. Univ. de Málaga. Existe una implementación para JAVA en `http://polaris.lcc.uma.es/~pacog/gmachine/Gmachine.html`.

Jones, M. P. (1994). Gofer 2.30a Release Notes. Research report, Dep. Computer Science, Yale University.

Jones, M. P. (1998). *Hugs User's Manual*. The University of Nottingham and Yale University.

Jones, M. P. (1999). Typing Haskell in Haskell. En Meijer, E. (ed.), *Proceedings of the 1999 Haskell Workshop*. University of Utrecht (tech. rep. UU–CS–1999–28).

Jones, M. P. y Hudak, P. (1993). Implicit and Explicit Parallel Programming in Haskell. Research Report YALE/DCS/RR-982, Dept. Comp. Science, Yale University, New Haven.

Jones, M. P. y Peyton Jones, S. L. (1999). Lightweight Extensible Records for Haskell. En Meijer, E. (ed.), *Proceedings of the 1999 Haskell Workshop*. University of Utrecht (tech. rep. UU–CS–1999–28).

Jones, N. y Nielson, F. (1995). Abstract Interpretation: a Semantics–Based Tool for Program Analysis. En Abramsky, S., Gabbay, D., y Maibaum, T. S. (eds.), *Handbook of Logic in Computer Science*, volumen IV, pp. ?? Oxford University Press.

Joosten, S. y van den Bergand G. van der Hoeven, K. (1993). Teaching functional programming to first-year students. *JFP*, 3(1):49–65.

Jung, Y. y Michaelson, G. (2000). A visualisation of polymorphic type checking. *JFP*, 10(1):57–75.

Kahn, K. (1990). Actors as a Special Case of Concurrent Constraint Programming. En ACM (ed.), *OOPSLA/ECOOP'90*, volumen 25 (10), pp. 57–66.

Kahn, K., Trible, E., Miller, M., y Bobrow, D. (1987). Objects in Concurrent Logic Programming Languages. En ACM (ed.), *Object-Oriented Programming Systems, Languages and Applications (OOPSLA'86)*, volumen 21(11) de *ACMSN*, pp. 242–257. Ver también *Vulcan: Logical Concurrent Objects*, en [Shapiro, 1987]:274-303.

Kemeny, J. G. y Kurtz, T. (1967). *Basic Programming*. Wiley, New York.

Kleene, S. C. (1935). A Theory of Positive Integers in Formal Logic. Part I. *American Journal of Mathematics*, 57:153–173.

Kleene, S. C. y Rosser, J. (1935). The inconsistency of certain formal logics. *Annals of Mathematics*, 36:630–636.

Klop, J. (1992). Term Rewriting Systems. En Abramsky, S., Gabbay, D., y Maibaum, T. S. (eds.), *Handbook of Logic in Computer Science*, capítulo 1.1, pp. 1–112. Oxford University Press.

Kowalski, R. (1974). Predicate Logic as a Programming Language. *Information Processing 74*, pp. 569–574.

Kowalski, R. (1979a). Algorithm = Logic + Control. *CACM*, 22(7):424–436. Reimpreso en [Horowitz, 1987]:480–492.

Kowalski, R. (1979b). *Logic for Problem Solving*. Elsevier Sc. Publ. Co. Traducción al castellano en Díaz de Santos, Madrid (1986), con el título *Lógica, Programación e Inteligencia Artificial*.

Kowalski, R. (1985). The Origins of Logic Programming. *Byte Magazine*, Agosto:192–199.

Kowalski, R. (1995). Logic without Model Theory. En Gabbay, D. (ed.), *What is a Logical System*. Oxford University Press.

Kowalski, R. y Kuehner, D. (1970). Resolution with selection function. *Artificial Intelligence*, 2(3):227–260.

Lalement, R. (1993). *Computation as Logic*. International Series in Computer Science. Prentice Hall, Englewood Cliffs, NJ.

Lambert, T., Lindsay, P., y Robinson, K. (1993). Using Miranda as a first programming language. *JFP*, 3(1):5–34.

Landin, P. (1963). The mechanical evaluation of expressions. *Computer Journal*, 6:308–320.

Lang, K. y Pearlmutter, B. (1987). Oaklisp: An Object-Oriented Scheme with First Class Types. En ACM (ed.), *Object-Oriented Programming Systems, Languages and Applications (OOPSLA'86)*, volumen 21(11) de *ACMSN*, pp. 30–37.

Lapidt, T., Levy, D., y Paz, T. (1999). Implementing Constructivist ideas in a Functional Programming Currículum for Secondary School Students. En Felleisen, M., Hanus, M., y Thompson, S. (eds.), *Proceedings of the Workshop on Functional and Declarative Programming in Education, FDPE'99*. Technical Report of Rice University, Rice COMP TR99-346. `http://www.cs.rice.edu/~matthias/FDPE99`.

Lassez, C. (1987). Constraint Logic Programming. *Byte Magazine*, Agosto:171–176.

Lassez, J.-L., Maher, M. J., y Marriott, K. (1988). Unification Revisited. En Boscarol, M., Aiello, L. C., y Levi, G. (eds.), *Foundations of Logic and Functional Programming, Workshop Proceedings, Trento, Italy, (Dec. 1986)*, volumen 306 de *LNCS*, pp. 67–113. Springer-Verlag.

Launchbury, J. y Peyton Jones, S. L. (1995). State in Haskell. *Lisp and Symbolic Computation*, 8(4):293–341.

Lavrac, N. y Dzeroski, S. (eds.) (1997). *Proc. of the 7th International Workshop on Inductive Logic Programming*, volumen 1297 de *LNAI*. Springer-Verlag.

Leijen, D., Meijer, E., y Hook, J. (1999). Haskell as an Automation controller. En *3rd International Summerschool on Advanced Functional Programming, Braga, Portugal*, volumen 1608 de *LNAI*. Springer-Verlag.

Leijhen, D. (1998). *Functional Components (COM components in Haskell)*. Tesis Doctoral, Univ. Amsterdan. `http://haskell.cs.yale.edu/haskellscript/doc.html`.

Leijhen, D. y Meijer, E. (2000). Domain Specific Embedded Compilers. En *Proc. of the 2nd USENIX Conference on Domain-Specific Languages*. USENIX Association.

Leler, W. (1988). *Constraint Programming Languages - Their Specification and Generation*. Addison-Wesley.

Liu, B. (1992). ConstraintLisp: An Object-Oriented Constraint Programming Language. *ACMSN*, 27(11):17–26.

Lloyd, J. (1984). *Foundations of logic programming*. Springer-Verlag. Segunda edición de 1987.

Lloyd, J. W. (1995). Declarative Programming in Escher. Informe Técnico núm. CSTR-95-013, Department of Computer Science, University of Bristol. `http://www.cs.bris.ac.uk/Tools/Reports/Abstracts/1995-lloyd.html`.

Loogen, R., López Fraguas, F., y Rodríguez Artalejo, M. (1993). A Demand Driven Computation Strategy for Lazy Narrowing. En (ACM) (ed.), *International Symposium on Programming Languages: Implementations, Logics and Programs (PLILP'93)*, volumen 714 de *LNCS*, pp. 244–247. Springer Verlag.

Lopez, G. (1994). Implementing Constraint Imperative Programming Languages: The Kaleidoscope'93 Virtual Machine. *ACMSN*, 29(10):259–271.

Loveland, D. W. (1969). A simplified format for the model elimination theorem–proving procedure. *JACM*, 16(3):349–363.

López-Fraguas, F. J. y Sanchez-Hernandez, J. (1999). TOY: A Multiparadigm Declarative System. En *RTA'99*, volumen 1631 de *LNCS*, pp. 244–258. Springer.

Maher, M. (1987). Logic semantics for a class of committed-choice programas. En Lassez, J.-L. (ed.), *Logic Programming, Proc. 4th Int. Conference*, pp. 858–876. MIT Pres, Cambridge.

Maher, M. (1999). Adding Constraints to Logic–based Formalisms. En Apt, W. R., Marek, V. W., Truszczynski, M., y Warren, D. S. (eds.), *The Logic Programming Paradigm. A 25–Year Perspective*, capítulo 5, pp. 313–331. Springer.

Maluszynski, J., Bonnier, S., Boye, J., Kluzniak, F., Kagedal, A., y Nilsson, U. (1993). Logic Programs with External Procedures. En Apt, W. R., de Bakker, J., y Rutten, J. (eds.), *Logic Programming Languages, Constraints, Functions and Objects*, capítulo 2, pp. 21–48. MIT Press, Cambridge, MA.

Mandelbrot, B. B. (1982). *The Fractal Geometry of Nature*. Freeman and Co., San Francisco.

Marriott, K. y Stuckey, P. J. (1998). *Programming with Constraints: An Introduction*. The MIT Press.

Matsuoka, S. y Yonezawa, A. (1993). Analysis of Inheritance Anomaly in Object-Oriented Concurrent Programming Languages. En Agha, E. (ed.), *Research Directions in Concurrent Object-Oriented Programming*. MIT Press.

McCabe, F. G. (1985). Lambda PROLOG. Tech. report, Department of Computing, Imperial College, London, England. También en *Logic Programming Workshop*, Lisbon (1983).

McCabe, F. G. (1993). An introduction to *L&O*. En Apt, W. R., de Bakker, J., y Rutten, J. (eds.), *Logic Programming Languages, Constraints, Functions and Objects*, capítulo 6, pp. 148–184. MIT Press, Cambridge, MA.

McCarthy, J. (1958a). An algebraic language for the manipulations of symbolic expressions. Tech. report, A.I. MEMO 1, M.I.T, RLE and MIT Computation Center, Cambridge, Massachusetts.

McCarthy, J. (1958b). Programs with Common Sense. En *Semantic Information Processing*, pp. 403–418. The MIT Press, Cambridge, MA, 1968. Reeditado como [McCarthy, 1990].

McCarthy, J. (1959). LISP: A Programming System for Symbolic Manipulations. Report, ACM Annual Meeting, Extended Abstract, Cambridge, Mass.

McCarthy, J. (1960). Recursive Function of Symbolic Expressions and their Computation by Machine. *CACM*, 3(4):184–195. Reimpreso en [Horowitz, 1987]: 203–214.

McCarthy, J. (1990). Programs With Common Sense. En Lifschitz, V. (ed.), *Formalizing Common Sense: Papers by John McCarthy*, pp. 9–20. Ablex Publishing Corporation, Norwood, New Jersey.

McCarthy, J., Abrahams, P., Edwards, D., Hart, T., y M.Levin (1965). *LISP 1.5 Programmer's Manual*. MIT Press. Reimpreso en [Horowitz, 1987]:215–239.

Meijer, E. (2000). Server Side Web Scripting in Haskell. *JFP*, 10(1):1–18.

Meijer, E., Leijen, D., y Hook, J. (1999). Client-Side Web Scripting with HaskellScript. *LNCS*, 1551:196–210.

Meijer, E. y van Velzen, D. (2000). Haskell Server Pages. Functional Programming and the Battle for the Middle Tier. Tech. report, Utrecht University. `http://www.cs.uu.nl/~erik/`.

Meseguer, J. (1992). Conditional rewriting logic as a unified model of concurrency. *TCS*, 96:73–155.

Meseguer, J. (2000). Rewriting Logic and Maude: concepts and applications. En Bachmair, L. (ed.), *Rewriting Techniques and Applications, 11th International Conference (RTA'2000)*, volumen 1833 de *LNCS*, pp. 1–26. Springer–Verlag.

Meseguer, J. y Winkler, T. (1992). Parallel Programming in Maude. *LNCS*, 574:253–293.

Meyrowitz, N. (ed.) (1986). *International Conference on Fifth Generation Computer Systems*.

Michie, D. (1968). Memo functions and machine learning. *Nature*, 218:19–22.

Miller, D. y Nadathur, G. (1986). Higher-order logic programming. En Shapiro, E. (ed.), *Proceedings of the Third International Logic Programming Conference*, pp. 448–462. MIT Press.

Milner, R. (1978). A theory of type polymorphism in programming. *Journal of Computer and System Sciences*, 17:348–375.

Milner, R. (1984). A proposal for standard ML. En *Conference Record of the ACM Symposium on LISP and Functional Languages*, pp. 184–197. ACM.

Milner, R., Tofte, M., y Harper, R. (1990). *The Definition of Standard ML*. MIT Press. Existe un revisión, publicada en 1997, con la participación adicional de D. MacQueen.

Mishra, P. (1984). Towards a Theory of Types in Prolog. En *Proceedings of the International Symposium on Logic Programming*, pp. 289–298, Atlantic City. IEEE, Computer Society Press.

Mitchell, J. C. y Plotkin, G. D. (1985). Abstract types have existential types. En *Conference Record of the Twelfth Annual ACM Symposium on Principles of Programming Languages*, pp. 37–51. ACM.

Mitchell, S. y Wellings, A. (1994). Synchronization, Concurrent Object-Oriented Programming and the Inheritance Anomaly. Tech. report, Dep. Comp. Sc., York University.

Mitchell, T. (1997). *Machine Learning*. McGraw–Hill.

Mogensen, T. Æ. (1992). Efficient Self-Interpretation in Lambda Calculus. *JFP*, 2(3):345–364.

Moon, D. (1987). Object-Oriented Programming with Flavors. En ACM (ed.), *Object-Oriented Programming Systems, Languages and Applications (OOPSLA'86)*, volumen 21(11) de *ACMSN*, pp. 1–8.

Moreno Navarro, J. y Rodríguez Artalejo, M. (1988). BABEL: A Functional and Logic Programming Language based on constructor discipline and narrowing. En *First International Workshop on Algebraic and Logic Programming, Gaussig*, pp. 14–18.

Moreno Navarro, J. y Rodríguez Artalejo, M. (1992). Logic Programming with Functions and Predicates: the language Babel. *JLP*, 12(3):191–223.

Muggleton, S. (ed.) (1991). *Proc. of the First International Workshop on Inductive Logic Programming*, Viano de Castelo, Portugal.

Muller, C. (1986). Modula-Prolog: a Software Development Tool. *IEEE Software*, 3(6):39–45.

Mycroft, A. y O'Keefe, R. (1984). A polymorphic type system for Prolog. *Artificial Intelligence*, 23:295–307.

Nadathur, G. y Jayaraman, B. (1989). Towards a WAM Model for λ-Prolog. En Lusk, Ewing L.; Overbeek, R. A. (ed.), *Proceedings of the North American Conference on Logic Programming (NACLP '89)*, pp. 1180–1200, Cleveland, Ohio. MIT Press.

Nadathur, G. y Miller, D. (1988). An Overview of λ PROLOG. En Kowalski, R. A. y Bowen, K. A. (eds.), *Proceedings of the Fifth International Conference and Symposium on Logic Programming*, pp. 810–827, Seatle. ALP, IEEE, The MIT Press.

Nadathur, G. y Miller, D. (1998). Higher-Order Programming. En Gabbay, D. (ed.), *Handbook of Logic in Artificial Intelligence and Logic Programming*, volumen 5, Logic Programming, pp. 499–590. Oxford University Press. `www.cs.uchicago.edu/~gopalan/recentpapers.html`.

Nadathur, G. y Mitchell, D. J. (1999). System Description: Teyjus-A Compiler and Abstract Machine Based Implementation of Lambda Prolog. En *Proceedings of Sixteenth Conference on Automated Deduction*. `http://www.cse.psu.edu/~dale/lProlog/`.

Nadathur, G. y Pfenning, F. (1992). The Type System of a Higher-Order Logic Programming Language. En Pfenning, F. (ed.), *Types in Logic Programming*, pp. 245–283. MIT Press.

Naish, L. (1986). *Negation and Control in Prolog*, volumen 239 de *LNCS*. Springer-Verlag.

Naughton, P. (1996). *The Java Handbook*. Osborne/McGraw-Hill, Berkeley, CA, USA. Traducido por McGraw-Hill, 1996.

Nienhuys-Cheng, S.-H. y de Wolf, R. (1997). *Foundations of Inductive Logic Programming*, volumen 1228 de *LNAI*. Springer-Verlag.

Nilsson, U. y Maluszynski, J. (1995). *Logic, Programming and Prolog.* Wiley, 2^a edición.

Nocker, E. G. J. M. H., Smetsers, J. E. W., van Eekelen, M. C. J. D., y Plasmeijer, M. J. (1991). Concurrent Clean. *LNCS*, 506:202–217.

Nordlander, J. y Carlsson, M. (1997). A Rendezvous of Functions and Reactive Objects. En *Congressus Hibernus*. `www.cs.chalmers.se/~magnus`.

Odersky, M. (1991). Objects and Subtyping in a Functional Perspective. IBM Research Report RC 16423, IBM Research, Thomas Watson Research Center.

Odersky, M. y Wadler, P. (1997). Pizza into Java: Translating theory into practice. En *24'th ACM Symposium on Principles of Programming Languages*, pp. 146–159. ACM. `www.dcs.gla.ac.uk/~wadler/topics/pizza.html`.

Odersky, M. y Wadler, P. (1998). Leftover Curry and reheated Pizza: How functional programming nourishes software reuse. En Devanbu, P. y Poulin, J. (eds.), *Fifth International Conference on Software Reuse*, pp. 2–11, Vancouver, BC. IEEE Computer Society Press.

O'Donnell, M. (1977). *Computing in Systems described by Equations*, volumen 58 de *LNCS*. Springer.

O'Donnell, M. (1985). *Equational Logic as a Programming Language*. MIT Press.

Okasaki, C. (1998). *Purely Functional Data Structures*. Cambridge University Press.

O'Keefe, R. A. (1990). *The craft of Prolog*. MIT Press.

Padawitz, P. (1988). *Computing in Horn Clause Theories*, volumen 16 de *EATCS Monographs on Theoretical Computer Science*. Springer.

Page, R. (1999). Selling Haskell for CS1. Tech. report, School of Computer Science, University of Oklahoma.

Paulson, L. C. (1991). *ML for the Working Programmer.* CUP (segunda ed. de 1996).

Pearce, D. (1997). Report on the CLN strategic planning workshop. Compulog magazine (the newsletter of the european network in computational logic), Compulog Network of Excellence. `www.cs.ucy.ac.cy/compulog/newpage5.htm`.

Peterson, J., Hudak, P., y Elliott, C. (1999). Lambda in Motion: Controlling Robots with Haskell. En Gupta, G. (ed.), *Practical Aspects of Declarative Languages (PADL'99)*, volumen 1551 de *LNCS*, pp. 91–105. Springer-Verlag.

Peyton Jones, S. L. (1987). *The Implementation of Functional Programming Languages.* Prentice-Hall.

Peyton Jones, S. L., Gordon, A., y Finne, S. (1996). Concurrent Haskell. En *Symposium on the Principles of Programming Languages*, St. Petersburg Beach, Florida. ACM.

Peyton Jones, S. L. y Hughes, J. (1999a). Report on the Programming Language Haskell 98. A Non-strict, Purely Functional Language. Research report, `http://haskell.org`.

Peyton Jones, S. L. y Hughes, J. (1999b). Standard Libraries for Haskell 98. Research report, `http://haskell.org`.

Peyton Jones, S. L., Meijer, E., y Leijen, D. (1998). Scripting COM components in Haskell. En *Fifth International Conference on Software Reuse (ICSR5).*

Pfenning, R. (ed.) (1992). *Types in Logic Programming.* MIT Press.

Pimentel, E. (1993). *$L^2||O^2$: Un lenguaje Lógico Concurrente Orientado a Objetos.* Tesis Doctoral, Facultad de Informática, Universidad de Málaga.

Pimentel, E. (1995). Programación Orientada a Objetos Concurrente. Informe técnico, Dpto. de Lenguajes y Ciencias de la Computación, Universidad de Málaga. Notas para la *IX Escuela de Verano de Informática.*

Plasmeijer, R. y van Eekelen, M. (1993). *Functional Programming and Parallel Graph Rewriting.* Addison-Wesley.

Plasmeijer, R. y van Eekelen, M. (1998). The Concurrent Clean Language Report (version 1.3). Language report, University of Nijmegen. `http://www.cs.kun.nl/ ~clean/ Manuals/manuals.html`.

Plotkin, G. (1981). A Structural approach to Operational Semantics. Tech. Report DAIMI FN-19, Comp. Science Dept., University of Aarhus.

Pollack, R. (2000). Dependently Typed Records for Representing Mathematical Structure. Tech. rep., Durham University.

Pyo, C. y Reddy, U. S. (1989). Inference of Polymorphic Types for Logic Programs. En Lusk, E. L. y Overbeek, R. A. (eds.), *Proceedings of the North American Conference on Logic Programming*, pp. 1115–1134, Cleveland, Ohio, USA.

Quintus (1997). *Quintus Prolog. User Guide and Reference Manual.* Berkhamsted, UK.

Rabhi, F. y Lapalme, G. (1999). *Algorithms. A Functional Programming Approach.* Addison-Wesley.

Raedt, L. D. (1999). A Perspective on Inductive Logic Programming. En Apt, W. R., Marek, V. W., Truszczynski, M., y Warren, D. S. (eds.), *The Logic Programming Paradigm. A 25–Year Perspective*, capítulo 6, pp. 335–346. Springer.

Reade, C. (1989). *Elements of Functional Programming*. Addison-Wesley.

Reddy, U. (1988). Notions of Polymorphism for Predicate Logic Programs. En *Logic Programming: Proceedings of the Fifth International Conference*, pp. 17–34. MIT Press.

Reddy, U. S. (1985). Narrowing as the Operational Semantics of Functional Languages. En *Proceedings of the International Symposium on Logic Programming*, pp. 138–151. IEEE Computer Society, Technical Committee on Computer Languages, The Computer Society Press.

Reppy, J. H. (1991). CML: A Higher-Order Concurrent Language. *ACMSN*, 26(6):293–305. `ftp://ftp.cs.cornell.edu/pub/`.

Revesz, G. (1988). *Lambda-Calculus, Combinators, and Functional Programming*. Cambridge University Press, Cambridge, UK. Cambridge Tracts in Theoretical Computer Science, vol. 4.

Ridoux, O. (1998). Lambda-Prolog de A à Z... ou presque. Habilitation à diriger des recherches, Université de Rennes 1. `www.irisa.fr/bibli/publi/habilitations/habilitations.html`.

Ringwood, G. A. (1988). Parlog86 and the Dining Logicians. *CACM*, 31(1):10–25.

Robinson, J. A. (1965). A machine oriented logic based on then resolution principle. *JACM*, 12(1):23–41. Reeditado en [Siekmann y Wrightson, 1983].

Robinson, J. A. (1983). Logic Programming - Past, Present and Future. *New Generation Comput. (Japan) ISSN: 0288-3635*, 1(2):107–24.

Robinson, J. A. (1984). Logic Programming and Fifth Generation Computing Systems. En *A.I.C.A. Annual Conference Proceedings*, volumen 1, pp. 3–15.

Robinson, J. A. (1988). LogLisp: combining functional and relational programming in a reduction setting. *Machine Intelligence*, 11.

Robinson, J. A. y Greene, K. J. (1987). New Generation Knowledge Processing: Final Report on the SUPER System. Case center technical report no. 8707, Syracuse University, CASE Center, Syracuse, NY.

Robinson, J. A. y Sibert, E. E. (1982). LogLisp: An alternative to Prolog. *Machine Intelligence*, 10.

Rogers, H. (1967). *Theory of Recursive Functions and Effective Computability*. McGraw-Hill, New York.

Rogerson, D. (1997). *Inside COM*. Microsoft Press.

Roselló, L. R. (1986). *LOGO. De la tortuga a la Inteligencia Artificial*. Vector, Madrid.

Rosser, J. B. (1935). A Mathematical Logic without Variables. *Annals of Mathematics*, 36(2):127–150.

Ruiz Jiménez, B., Gallardo, J., Gutiérrez, F., y Guerrero, P. (1995). *Programación Funcional con Haskell*. Universidad de Málaga.

Ruiz Jiménez, B., Gutiérrez, F., Gallardo, J. E., y Guerrero, P. (1996). Clasificación de Objetos Funcionales en Entornos Concurrentes. En Lucio, P., Martelli, M., y Navarro, M. (eds.), *1996 Joint Conference on Declarative Programming (APPIA-GULP-PRODE'96)*, pp. 581–584. Universidad del Pais Vasco.

Ruiz Jiménez, B. C. (1994). El λ cálculo. Notas para un curso, Dpto. de Lenguajes y Ciencias de la Computación. Universidad de Málaga.

Ruiz Jiménez, B. C. (1999). Condensing lemmas in Pure Type Systems with Universes. En Haeberer, A. M. (ed.), *7th International Conference on Algebraic Methodology and Software Technology (AMAST'98) Proceedings*, volumen 1548 de *LNCS*, pp. 422–437. Springer-Verlag.

Ruiz Jiménez, B. C. y Fernández Leiva, A. (1995). Un sistema de clases para Prolog. Informe técnico, Dpto de Lenguajes y Ciencias de la Computación. Univ. de Málaga.

Ruiz Jiménez, B. C. y Fernández Leiva, A. (1996). Una semántica operacional para CProlog. En Clares, B. (ed.), *II Jornadas de Informática*, pp. 21–30. AEIA.

Ruiz Jiménez, B. C., Gutiérrez López, F., Guerrero García, P., y Gallardo Ruiz, J. E. (2000). *Razonando con Haskell. Una Introducción a la Programación Funcional*. José E. Gallardo Ruiz (editor).

Russell, B. A. (1908). Mathematical Logic based on the Theory of Types. *American Journal of Mathematics*, 30:222–262. También en [van Heijenoort, 1967]:150–182.

Russell, B. A. y Whitehead, A. (1910–13). *Principia Mathematica*. Cambridge University Press. Reprinted 1960.

Sabry, A. (1999). Declarative Programming Across the Undergraduate Curriculum. En Felleisen, M., Hanus, M., y Thompson, S. (eds.), *Proceedings of the Workshop on Functional and Declarative Programming in Education, FDPE'99*. Technical Report of Rice University, Rice COMP TR99-346. `http://www.cs.rice.edu/~matthias/FDPE99`. También como Informe Técnico del Department of Computer and Information Science, University of Oregon.

Sammet, J. E. (1969). Roster of Programming Languages. *Computers & Automation*, 18(7):153–158. También en la misma revista, números 17(6),16(6).

Saram, H. D. (1985). *Programming in micro-PROLOG*. Ellis Horwood, Chichester. Traducido al castellano en Paraninfo, Madrid (1987).

Saraswat, V. (1989). *Concurrent Constraint Programming Languages*. Tesis Doctoral, Carnegie-Melon University.

Saraswat, V. (1990). Janus: A Step Towards Distributed Constraint Programming. En Debray, S. (ed.), *Logic Programming: 1990 North Am. Conference*. MIT Press.

Saraswat, V. A. (1993). *Concurrent Constraint Programming*. ACM Doctoral Dissertation Awards: Logic Programming. The MIT Press, Cambridge, MA. Actualización de la tesis doctoral [Saraswat, 1989].

Schmidt-Schauß, M. y Huber, M. (2000). A Lambda–calculus with letrec, case, constructuors and non–determinis. En Derschowits, N. y Kirchner, C. (eds.), *International Workshop on Rule–Based Programming (RULE'2000)*, Montreal, Canada. `wwww.cs.yorku.ca/pli-00`.

Schönfinkel, M. (1924). Uber die Bausteine der mathematischen logik. *Mathematische Annalen*, 92:305–316. Existe traducción inglesa con el título *On the building blocks of mathematical logic* en [van Heijenoort, 1967]:355–366.

Schreye, D. D., Bruynooghe, M., Demoen, B., Denecker, M., Marteens, B., Janssens, G., Raedt, L. D., Decorte, S., Leuschel, M., y Sagonas, K. (1997). PL+: a second generation Logic Programming language. Technical report, Department of Computer Science, K.U. Leuven.

Schreye, D. D. y Denecker, M. (1999). Assessment of Some Issues in CL-Theory and Program Development. En Apt, W. R., Marek, V. W., Truszczynski, M., y Warren, D. S. (eds.), *The Logic Programming Paradigm. A 25–Year Perspective*, capítulo 3, pp. 195–208. Springer.

Seres, S. y Spivey, M. (1999). Embedding Prolog in Haskell. En Meijer, E. (ed.), *Proceedings of the 1999 Haskell Workshop*. University of Utrecht (tech. rep. UU–CS–1999–28).

Sethi, R. (1989). *Programming Languages: Concepts and Constructs*. Addison-Wesley.

Shapiro, E. (1983). A Subset of Concurrent Prolog and Its Interpreter. Tech. report, Institute for New Generation Computer Technology, Tokyo. También en [Shapiro, 1987]:Vol.1:27-83.

Shapiro, E. (1987). *Concurrent Prolog. Collected Papers.* MIT Press, Cambridge. Dos volúmenes.

Shapiro, E. (1989). The family of concurrent logic programming languages. *ACM Computing Surveys*, 21(3):413–510.

Shapiro, E. y Takeuchi, A. (1983). Object Oriented Programming in Concurrent Prolog. *New Generation Computing*, 1(1):25–48. También en [Shapiro, 1987]:Vol.2:251-273.

Shoham, Y. (1994). *Artificial Intelligence Techniques in Prolog.* Morgan Kaufmann.

Siekmann, J. y Szabó, P. (1984). Universal Unification. En Shostak, R. (ed.), *Proceedings 7th International Conference on Automated Deduction*, volumen 170 de *LNCS*, pp. 1–42, Napa Valley (California, USA). Springer-Verlag.

Siekmann, J. y Wrightson, G. (1983). *Automation of Reasoning 1. Classical Papers on Computational Logic 1957–1966.* Springer-Verlag.

Siekmann, J. H. (1989). Unification Theory. *JSC*, 7(3–4):207–274.

Slagle, J. (1974). Automated Theorem-Proving for Theories with Simplifiers, Commutativity and Associativity. *JACM*, 28(3):622–642.

Smolka, G. (1995). The Oz Programming Model. En van Leeuwen, J. (ed.), *Computer Science Today*, volumen 1000 de *LNCS*, pp. 324–343. Springer-Verlag, Berlin.

Smolka, G. (1998). Concurrent Constraint Programming Based on Functional Programming. En Hankin, C. (ed.), *European Joint Conf. on Theory and Practice of Software (ETAPS)*, volumen 1381 de *LNCS*, pp. 1–11, Lisbon, Portugal. Springer-Verlag.

Somogyi, Z., Henderson, F., y Conway, T. (1995). Mercury: an efficient purely declarative logic programming language. En *Proceedings of the Australian Computer Science Conference*, pp. 499–512. `http://www.cs.mu.oz.au/research/mercury/`.

Sørensen, M. y Urzyczyn, P. (1998). Lectures on the Curry–Howard Isomorphism. University of Copenhagen (1998).

Steele, G. L. (1980). *The Definition and Implementation of a Computer Programming Language based on Constraints.* PhD Thesis, AI-TR 595, MIT.

Steele, G. L. (1984). *CommonLisp–The Language.* Digital Press.

Sterling, L. y Shapiro, E. (1986). *The art of Prolog: Advanced Programming Techniques.* MIT Press, Cambridge.

Sterling, L. y Shapiro, E. (1994). *The art of Prolog: Advanced Programming Techniques.* MIT Press, 2^a edición. Edición extendida y corregida de [Sterling y Shapiro, 1986].

Strachey, C. (2000). Fundamental Concepts in Programming Languages. *Higher-Order and Symbolic Computation*, 13(1–2):11–49. Primera edición del manuscrito escrito en Agosto de 1967 para la *International Summer School in Computer Programming*, celebrada en Copenhagen. `http://www.wkap.nl/oasis.htm/257993`.

Subrahmanyam, P. A. y You, J. H. (1984). Conceptual basis and evaluation strategies for integrating functional and logic programming. En *1984 International symposium on Logic Programming*, pp. 144–153. IEEE, New York.

Tarau, P. (1999). Inference and Computation Mobility with Jinni. En Apt, W. R., Marek, V. W., Truszczynski, M., y Warren, D. S. (eds.), *The Logic Programming Paradigm. A 25–Year Perspective*, capítulo 1, pp. 33–52. Springer.

Thom, J. y Zobel, J. (1986). NU-Prolog Reference Manual - Version 1.3. Technical Report 86/10, Department of Computer Science, University of Melbourne.

Thompson, S. (1995). *Miranda: The Craft of Functional Programming.* Addison-Wesley.

Thompson, S. (1996). *Haskell: The Craft of Functional Programming.* Addison-Wesley.

Thompson, S. (1999). *Haskell: The Craft of Functional Programming.* Addison-Wesley. Segunda edición corregida y ampliada de [Thompson, 1996].

Thompson, S. (2000a). A functional reactive animation of a lift using Fran. *JFP*, 10(3):245–268.

Thompson, S. (2000b). Regular Expressions and Automata using Haskell. Technical Report 5-00, Computing Laboratory, University of Kent. `http:// www.cs.ukc.ac.uk/ pubs/ 2000/958`.

Trinder, P. W., Hammond, K., Loidl, H.-W., y Peyton Jones, S. L. (1998). Algorithm + strategy = parallelism. *JFP*, 8(1):23–60.

Turner, D. A. (1976). SASL Language Manual. Tech. report, University of Kent, Canterbury, U.K.

Turner, D. A. (1985). Miranda - a non-strict functional language with polymorphic types. En *Conference on Functional Programming Languages and Computer Architecture*, volumen 201 de *LNCS*, pp. 1–16. Springer Verlag.

Ueda, K. (1985). Guarded Horn Clauses. Informe Técnico núm. 103, ICOT, Tokyo. También en [Shapiro, 1987]:(Vol.1,140-156).

Ueda, K. (1999). Concurrent Logic/Constraint Programming: The Next 10 Years. En Apt, W. R., Marek, V. W., Truszczynski, M., y Warren, D. S. (eds.), *The Logic Programming Paradigm. A 25–Year Perspective*, capítulo 1, pp. 53–72. Springer.

Ullman, J. (1988). *Principles of Database and Knowledge–base Systems.* Computer Science Press.

van den Bos, J. (1991). PROCOL - A Concurrent Object Language with Protocols, Delegation and Constraints. *Acta Informatica*, 28:511–538.

van Eekelen, M. C. J. D. y et al. (1989). Concurrent Clean. Informe Técnico núm. 89-18, University of Nijmegen, Department of Informatics, Nijmegen, the Netherlands.

van Emden, M. H. (1999). The Logic Programming Paradigm in Numerical Computation. En Apt, W. R., Marek, V. W., Truszczynski, M., y Warren, D. S. (eds.), *The Logic Programming Paradigm. A 25–Year Perspective*, capítulo 4, pp. 257–276. Springer.

van Heijenoort, J. (ed.) (1967). *From Frege to Gödel: a Source Book in Mathematical Logic, 1879–1931.* Harvard University Press, Cambridge, MA.

van Hentenryck, P. (1987). *Consistency Techniques in Logic Programming.* Tesis Doctoral, University of Namur, Namur, Belgium.

van Hentenryck, P. (1989). *Constraint Satisfaction in Logic Programming.* MIT Press, Cambridge, Massachusetts. Edición revisada de la tesis [van Hentenryck, 1987].

van Roy, P. (1992). Aquarius Prolog compiler. *IEEE Computer*, 25(1):54–68.

van Roy, P. (1999). Logic Programming in Oz with Mozart. En Schreye, D. D. (ed.), *International Conference on Logic Programming*, pp. 38–51, Las Cruces, NM, USA. MIT Press.

Voda, P. (1988). The Constraint Language Trilogy: Semantics and Computations. Tech. rep., Complete Logic Systems, Vancouver.

Voronkok, A. (1992). Logic Programming with bounded quantifiers. En Voronkov, A. (ed.), *Logic Programming and Automated Reasoning. Proc. 2nd Russian Conference on Logic Programming*, volumen 592 de *LNAI*, pp. 486–514. Springer-Verlag.

Wadler, P. (1985). An Introduction to Orwell. Tech. report, Programming Research Group, Oxford University.

Wadler, P. (1999). How Enterprises Uses Functional Languages, and Why They Don't. En Apt, W. R., Marek, V. W., Truszczynski, M., y Warren, D. S. (eds.), *The Logic Programming Paradigm. A 25–Year Perspective*, capítulo 3, pp. 209–227. Springer.

Wadsworth, C. (1971). *Semantics and Pragmatics of the Lambda–Calculus*. Tesis Doctoral, Oxford University.

Wakeling, D. (1999). Compiling lazy functional programs for the Java Virtual Machine. *JFP*, 9(6):579–603.

Walinsky, C. (1989). CLP(sigma*): Constraint Logic Programming with Regular Sets. En *Sixth Inter. Conference on Logic Programming*, pp. 181–190. MIT Press.

Warren, D. H. (1983). An abstract Prolog instruction set. Technical note 309, SRI International, Menlo Park, CA.

Wegner, P. (1976). Programming Languages- the First 25 Years. *IEEE Transactions on Computers*, pp. 1207–1225. Reimpreso en [Horowitz, 1987]:4–22.

Whittle, J., Bundy, A., y Lowe, H. (1997). An Editor for Helping Novices to Learn Standard ML. En Hugh Glaser, P. H. y Kuchen, H. (eds.), *International Symposium on Programming Languages: Implementations, Logics and Programs (PLILP'97)*, volumen 1292 de *LNCS*, pp. 389–405. Springer-Verlag.

Wilhelm, R. y Maurer, D. (1995). *Compiler Design*. Addison–Wesley.

Winskel, G. (1993). *The formal semantics of Programming Languages*. MIT.

Winston, P. H. (1992). *Artificial Intelligence*. (third edition) Addison-Wesley.

Winston, P. H. y Horn, B. K. (1981). *LISP*. Addison-Wesley.

Wirsing, M. (1990). Algebraic Specification. En van Leeuwen, J. (ed.), *Handbook of Theoretical Computer Science, Vol. B, Formal Methods and Semantics*, capítulo 13, pp. 675–788. North Holland.

Wirth, N. (1976). *Algorithms + Data Structures = Programs*. Prentice-Hall, New York. Traducción al castellano en Ed. del Castillo, Madrid, 1980.

Wolfram, S. (1999). *The Mathematica book*. Cambridge University Press and Wolfram Research, Inc., cuarta edición.

Wong, L. (2000). Kleisli, a functional query system. *JFP*, 10(1):19–56.

Yardeni, E. y Shapiro, E. (1991). A Type System for Logic Programs. *JLP*, 10(2):125–153.