Programación lógica y funcional

Álvaro Tasistro Jorge Vidart

1988

Índice general

Version: $Id: plyf.tex,v 1.3 2003/08/13 04:28:26 cballard Exp $

Introducción

Pág. 3

Todavía no hemos compuesto esta parte del libro, por favor tenga paciencia. Muchas gracias.

Version: $Id: plyf.tex,v 1.3 2003/08/13 04:28:26 cballard Exp $

Capítulo 1

Paradigmas de programación

Version: $Id: plyf.tex,v 1.3 2003/08/13 04:28:26 cballard Exp $

Todavía no hemos compuesto esta parte del libro, por favor tenga paciencia. Muchas gracias.

Parte I

Programación en lógica

Version: $Id: plyf.tex,v 1.3 2003/08/13 04:28:26 cballard Exp $

Capítulo 2

Introducción a la programación en lógica

En este capítulo se realiza una introducción a la programación en lógica. La presentación, que será informal, se basará en el concepto de relación matemática, y apelará a los conocimientos previos del lector en los aspectos de cálculo y evaluación para lenguajes tradicionales.

El ejemplo que se usará, definir relaciones de parentezco familiar, constituye el caso clásico para presentar el tema. Se definirán las relaciones por enumeración, para luego, mediante el concepto de recursión, introducir relaciones más complejas.

PÁG. 20

En la programación imperativa tradicional el usuario diseña sus algoritmos en términos de transiciones de estados de una máquina ficticia que corresponde al lenguaje que se esté utilizando. Así un programador Pascal utiliza los constructores del mismo (asignaciones, estructuras de control, etc.) para indicar a una supuesta máquina Pascal como van evolucionando sus estados a fin de alcanzar una situación final en la que puedan identificarse los resultados deseados.

Desde otro punto de vista un programa tradicional puede interpretarse como una función, que a partir de un estado inicial de la máquina ya mencionada, define los valores que corresponden al estado final de la misma luego de la evolución del algoritmo.

Por su parte un programa en lógica permite definir relaciones sobre ciertos dominios dados. Dicha definición puede realizarse por extensión, es decir enumerando todas las tuplas del producto cartesiano, o mediante una definición recursiva.

Por ejemplo:

```
PADRE ( juan,  raquel ).
PADRE ( luis,  jorge ).
PADRE ( juan,  mariana ).
PADRE ( jorge, diego ).
PADRE ( jorge, noel ).
PADRE ( luis,  ricardo ).
```

corresponde a una definición parcial de una relación llamada `PADRE` tal que `PADRE` está contenida en el producto cartesiano `NOMBRES`×`NOMBRES`, siendo `NOMBRES` un dominio que contiene nombres de personas.

En la definición presentada se conviene en que el nombre que aparece en el primer argumento corresponde al padre de la persona cuyo nombre aparece en el segundo argumento. Se podría haber convenido la otra alternativa posible. Lo que es fundamental destacar es que cualquiera sea la interpretación que se elija, la misma no aparecerá en el programa, ya que en él sólo presenta los elementos sintácticos del problema, dejando la semántica a la interpretación del usuario. Esta característica se pondrá en evidencia cuando en el capítulo siguiente se considere la ejecución de un programa en lógica como una derivación sintáctica en una teoría axiomática dada.

PÁG. 21

Considerando entonces el programa el ejemplo como la definición de la relación `PADRE`, es posible ahora operar con la misma, lo que corresponderá a interrogar la relación. Las operaciones que se pueden realizar son las clásicas: restricciones (subconjuntos), proyecciones sobre dominios, ambas para definir nuevas relaciones, y preguntas de pertenencia. La forma de exresar dichas

operaciones se realiza en programación en lógica mediante el uso de variables (que se notarán con letras mayúsculas), las que ubicadas en alguno, o varios, de los argumentos de la relación, indicarán la proyección deseada. Así por ejemplo, si se desea conocer cuáles son los hijos de Juan, dicha información corresponde a una relación unaria, que se obtiene de la relación inicial, restringiéndola para aquellas tuplas que tengan `juan` en el primer argumento, y proyectando luego por el segundo. Eso se expresa en programación lógica de la siguiente forma:

```
<- PADRE ( juan,  X ).
```

a lo que un sistema que implementa la programación en lógica respopnderá:

```
X = raquel.
X = mariana.
```

Esta respuesta corresponde a una una nueva relación, en este caso sobre un solo dominio, y lo que se presenta es la enumeración de las tuplas de la misma.

Se pudiera también estar interesado en conocer quién es el padre de Ricardo, lo que se expresará, en forma simétrica a la anterior: PÁG. 22

```
<- PADRE ( Y, ricardo ).
```

obteniéndose como respuesta:

```
Y = luis.
```

De las dos preguntas presentadas aparece una de las características más importantes de la programación en lógica. Debido a su aspecto relacional, no se introducen *direcciones*, en el sentido clásico de datos y resultados. Un mismo programa, como el formado por el conjunto de las definiciones de la relación `PADRE`, permite conocer de quién es padre una persona (primer argumento como dato y segundo como resultado), como quién es el padre de una persona dada (segundo argumento como dato y primero como resultado). En la programación tradicional un cambio de dirección dato-resultado implicaría la reformulación del programa. Este aspecto de la reversibilidad de la programación en lógica, será presentado con más detalle en el capítulo 4.

Puede observarse de los ejemplos vistos, que en cada interrogación, u operación, de la relación inicial, la presencia de constantes como argumentos indica una restricción de la relación, y la presencia de variables expresa el interés de realizar alguna proyección. Generalizando entonces, tenemos que:

```
<- PADRE ( luis, jorge ).
```

corresponde a una restricción que da como resultado una sola tupla (en este caso estamos interrogando la pertenencia de la tupla en cuestión a la relación original).

La pregunta:

```
<- PADRE ( X, Y ).
```

corresponde a la proyección de la relación en todos sus dominios (en este caso el interés es en conocer todas las tuplas de la relación).

PÁG. 23

Es posible complementar la definición de la relación `PADRE`, con la definición de una nueva relación:

```
MADRE ( raquel,  diego ).
MADRE ( lucia,   mariana ).
MADRE ( raquel,  noel ).
MADRE ( carmen,  ricardo ).
MADRE ( carmen,  jorge ).
MADRE ( lucia,   raquel ).
MADRE ( mariana, alejandra ).
MADRE ( mariana, german ).
```

Las variables, que fueron utilizadas en las interrogaciones de las relaciones, permiten a su vez la definición de nuevas relaciones, en función de relaciones existentes. Si por ejemplo se desea definir la relación `ABUELO_PATERNO`, en función de las relaciones presentadas, se puede escribir:

```
ABUELO_PATERNO(X,Y) <- PADRE(X,Z), PADRE(Z,Y).
```

Esta cláusula puede leerse de la forma siguiente:

> «una persona `X` es el abuelo paterno de una persona `Y`, si existe una tercera persona `Z` tal que `X` es el padre de esa `Z`, y esa misma persona `Z` es el padre de `Y`.»

La nueva relación `ABUELO_PATERNO`, puede ahora ser operada (o interrogada) como las relaciones definidas por extensión. De esta manera la pregunta:

```
<- ABUELO_PATERNO ( W, diego ).
```

generará como respuesta:

```
W = luis.
```

. 24 Para obtener dicho resultado el sistema debió descubrir que existe una persona Jorge tal que `PADRE(jorge,diego)` está definida, para luego determinar que `PADRE(luis,jorge)` también lo está. Ese valor de `luis` es el que resuelve el problema, y por ello el sistema lo escribe como resultado. Es de hacer notar que las variables que aparecen en cada cláusula sólo tienen validez en la misma cláusula, es decir, utilizando nociones de los lenguajes de programación imperativa, que el alcance de una variable es la cláusula donde aparece. Es por ello que la `W` de la pregunta, se conectará con la `X` de la definición de la relación, mediante un mecanismo similar al pasaje de parámetros de los lenguajes tradicionales.

Una nueva pregunta:

```
<- ABUELO_PATERNO ( W, R ).
```

generará a su vez como respuesta:

```
W = luis, R = diego.
W = luis, R = noel.
```

Esta respuesta corresponde a la enumeración de toda la relación `ABUELO_PATERNO`, en función de la información disponible; lógicamente que el agregado de nuevas tuplas a la relación `PADRE` inicial podrá modificar a `ABUELO_PATERNO`. La pregunta `ABUELO_PATERNO(W,diego)` corresponde a una restricción de la respuesta anterior.

En forma similar a lo ya visto se puede definir una nuneva relación:

```
ABUELO_MATERNO(X,Y) <- PADRE(X,Z), MADRE(Z,Y).
```

Un nuevo paso de generalización puede ser ahora dado. En base a las relaciones definidas hasta el presente es posible ahora incluir una nueva definición, que corresponda al concepto de abuelo en forma genérica. El abuelo PÁG. 25 de una persona es el abuelo paterno o el abuelo materno. Esta situación se expresa en programación lógica de la forma siguiente:

```
ABUELO(X,Y) <- ABUELO_PATERNO(X,Y).
ABUELO(X,Y) <- ABUELO_MATERNO(X,Y).
```

donde al tener dos reglas para la definición de abuelo, estamos indicando que podemos utilizar una o la otra.

Ante la pregunta:

```
<- ABUELO( X, noel ).
```

el sistema responderá:

```
X = luis.
X = juan.
```

Para elaborar dicha respuesta, el sistema ha debido determinar mediante la relación ABUELO_PATERNO el valor X=luis, y con la relación ABUELO_MATERNO el valor X=juan.

Con las herramientas ya vistas es posible continuar definiendo relaciones que correspondan a grados directos de parentesco, como por ejemplo:

```
HERMANO(X,Y) <- PADRE(Z,X), PADRE(Z,Y).
HERMANO(X,Y) <- MADRE(Z,X), MADRE(Z,Y).

TIO(X,Y) <- PADRE(Z,X), HERMANO(X,Z).
TIO(X,Y) <- MADRE(Z,X), HERMANO(X,Z).

PRIMO(X,Y) <- TIO(Z,X), PADRE(Z,Y).
PRIMO(X,Y) <- TIO(Z,X), MADRE(Z,Y).
```

PÁG. 26

Ejercicios

1. Escriba definiciones alternativas para la relación TIO.
2. Determine porqué la relación PRIMO definida anteriormente no es equivalente a la siguiente:

```
PRIMO(X,Y) <- ABUELO(Z,X), ABUELO(Z,Y).
```

En el trabajo con relaciones muy frecuentemente resulta necesario definir relaciones como clausuras transitivas de relaciones existentes. Por ejemplo, y siguiendo con las relaciones familiares que hemos planteado, puede interesar definir la relación ANCESTRO, que correspondería a la clausura de las relaciones PADRE y MADRE ya presentadas. La forma de escribirla sería la siguiente:

```
ANCESTRO(X,Y) <- PADRE(X,Y).
ANCESTRO(X,Y) <- MADRE(X,Y).

ANCESTRO(X,Y) <- PADRE(Z,Y), ANCESTRO(X,Z).
ANCESTRO(X,Y) <- MADRE(Z,Y), ANCESTRO(X,Z).
```

donde la tercera regla puede leerse como:

> «una persona `X` es el ancestro de una persona `Y`, si existe una persona `Z`, tal que `Z` es el padre de `Y` y además `X` es ancestro de ese mismo `Z`»

Puede observarse que de la descripción escrita en castellano en la frase anterior puede inducirse rápidamente la regla en programación en lógica correspondiente. Esto constituye otra de las características relevantes de la programación en lógica, que consiste en el alto nivel de expresión que ofrece. En general es posible *pasar* con cierta facilidad de la especificación de un problema expresada en lenguaje natural, a una desscripción en programación en lógica.

La novedad que aparece en la definición anterior surge de la utilización de la recursión en la definición de la relación, lo que resulta totalmente natural al recordar que estamos definiendo una cláusula transitiva. PÁG. 27

La relación `ANCESTRO` definida puede ser operada en forma totalmente similar a las otras relaciones. Así por ejemplo si escribimos:

```
<- ANCESTRO( X, diego ).
```

el sistema responderá:

```
X = luis.
X = carmen.
X = juan.
X = lucia.
```

Ejercicios

1. Demuestre que la relación `ANCESTRO` definida, es equivalente a la siguiente:

```
ANCESTRO(X,Y) <- PADRE(X,Y).
ANCESTRO(X,Y) <- MADRE(X,Y).

ANCESTRO(X,Y) <- PADRE(X,Z), ANCESTRO(Z,Y).
ANCESTRO(X,Y) <- MADRE(X,Z), ANCESTRO(Z,Y).
```

2. ¿De qué manera cree usted que se manifiestan las diferencias entre la definición original de `ANCESTRO`, y la que aparece en 1.?

En los capítulos siguientes se presentan dos formas de interpretar el significado de los programas en lógica. Una primera visión proviene de la lógica matemática, y tiene un carácter más denotacional. La segunda introduce los aspectos operacionales que son la base de las implementaciones de programación en lógica. Una fuente de referencia para la escritura de estos capítulos la constituyó el excelente libro de Christopher John Hogger [Hog84]. PÁG. 28

Capítulo 3

Interpretación lógica

Una de las características de la programación en lógica es que admite múltiples interpretaciones. Una de ellas proviene de la lógica. En ella un programa constituye una definición axiomática de una teoría, y a una invocación a la ejecución del mismo se la considera como una fórmula a demostrar en dicha teoría. PÁG. 29

Luego de una introducción general al tema, en la primera sección se introduce la definición sintáctica de los programas en lógica. Asimismo se presentan los mecanismos de estructuración de datos, los que se obtienen a partir de los símbolos funcionales que puede introducir el usuario. El significado de los programas en lógica se presenta en la segunda sección, utilizando los conceptos de lógica que se definen. A continuación se plantean los mecanismos de derivación lógica a partir de reglas de inferencia, y el concepto de unificación que es el que permite realizar dicha operación en presencia de variables. El capítulo concluye con la presentación de un modelo general para el paso inferencial y con un ejemplo de aplicación.

Version: $Id: plyf.tex,v 1.3 2003/08/13 04:28:26 cballard Exp $

PÁG. 30

En la introducción se ha visto cómo es posible definir y manipular relaciones mediante programas en lógica. Para dichos programas fue presentada una semántica intuitiva, a finde comprender las *ejecuciones* de los mismos. En lo que sigue se utilizarán nociones de la lógica para el mismo objetivo. Se tratará de no realizar un planteo excesivamente formal de tal manera que se puedan rescatar las ideas intuitivas subyacentes. El lector interesado puede recurrir al excelente libro de Lloyd [Llo84].

La idea de base de la programación en lógica radica en utilizar conceptos de la lógica para referirse a todos y cada uno de los procesos de construcción de programas y de ejecución de los mismos.

Un programa, como el formado por el conjunto de las reglas de la introducción, corresponderá a la definición de una teoría axiomática, y una ejecución, es decir la evaluación de una interrogación de alguna de las relaciones definidas, significará una demostración o prueba a partir de los axiomas. En esta perspectiva, un intérprete de programas en lógica es, en definitiva, un demostrador automático de teoremas.

Desde el punto de vista lógico una cláusula como:

```
PADRE( juan, raquel ).
```

representa la afirmación del hecho que `juan` es el `PADRE` de `raquel`. Sin embargo es importante destacar que las palabras `juan` y `raquel` son simplemente sucesiones de caracteres. La interpretación de que ambas palabras representan a ciertas personas dadas, está solamente en la intención del programador y dicha información no la tiene el sistema de evaluación lógica. Esta situación tendrá relevancia cuando se presenten los mecanismos de demostración lógica y se introduzca el concepto de satisfactibilidad.

Una cláusula como:

```
TIO(ricardo,diego) <- PADRE(jorge,diego), HERMANO(jorge,ricardo)
```

PÁG. 31

representará la afirmación condicional que si es cierto que `jorge` es el `PADRE` de `diego`, y que `jorge` es el `HERMANO` de `ricardo`, entonces se puede concluir que `ricardo` es `TIO` de `diego`.

Y si la cláusula fuera con variables:

```
TIO(X,Y) <- PADRE(Z,Y), HERMANO(X,Z)
```

representará la afirmación que para todo valor posible de `X` y `Y`, si existe un valor de `Z` tal que `Z` es el `PADRE` de `Y`, y además ese mismo `Z` es `HERMANO` de `X`, entonces `X` es `TIO` de `Y`.

Finalmente debe considerarse el caso de la invocación a la ejecución de un programa. Así la fórmula:

```
<- HERMANO(jorge, ricardo)
```

expresa que se desea determinar si el predicado que aparece en la misma, es válido a partir de la información que ofrece el programa visto anteriormente. En otras palabras se pretende demostrar que la fórmula en cuestión es una consecuencia lógica de las fórmulas que conforman el programa.

3.1. Sintaxis de programas en lógica

El lenguaje que se utiliza en la programación en lógica proviene de la lógica de predicados de primer orden.

Se dispone de:

- un conjunto de elementos simples llamados átomos.
- un vocabulario V de variables.
- un vocabulario F de símbolos funcionales.
- un vocabulario P de símbolos predicativos.

Cada símbolo funcional y predicativo tiene asociado un número entero que corresponde a su aridad. PÁG. 32

Como convención se adoptará que los átomos estarán formados por caracteres en minúsculas y números, por ejemplo:

```
a, b, raquel, 1, 3200, paris, caracas
```

Las variables se denotarán utilizando letras mayúsculas:

```
X, Y, Z, ...
```

Para los símbolos funcionales se usarán también letras minúsculas, lo que no ofrece ambigüedad con los átomos, debido a su diferenciada ubicación sintáctica. Los símbolos predicativos serán denotados usando letras mayúsculas.

Definición 3.1 (Sintaxis de programas en lógica)

- Un **término** es:
  - un átomo
  - una variable

  - un símbolo funcional seguido de una sucesión de términos, tantos como la aridad del símbolo funcional.

- Un **predicado** es un símbolo predicativo seguido de una sucesión de términos, tantos como la aridad del símbolo predicativo.

- Una **regla** o **cláusula**, es alguna de las siguientes alternativas:

$$\begin{array}{lcll} A & \leftarrow & . & \text{(I)} \\ A & \leftarrow & B_1, \ldots, B_m. & \text{(II)} \\ & \leftarrow & B_1, \ldots, B_m. & \text{(III)} \end{array}$$

  siendo $A, B_1, \ldots, B_m$ predicados.

- Un **programa lógico** es un conjunto de reglas.

PÁG. 33

Mediante el concepto de términos, y utilizando los símbolos funcionales es posible estructurar la información a ser manipulada por los programas en lógica. Por ejemplo, si se tiene:

```
empleado( jaime, 32, 1827 )
```

con `empleado` símbolo funcional, y `jaime`, `32` y `1827`, átomos, el significado de dicho término podría ser que `jaime` es un empleado que tiene `32` años, y que su número interno es `1827`. La interpretación de cada uno de los átomos presentes surge entonces de su posición en el término, lo que equivale a asignar significado a los dominios del símbolo funcional. Si se desea un mayor poder de expresión, esto puede lograrse agregando nuevos símbolos funcionales (unarios) como en el ejemplo:

```
empleado( nombre(jaime), edad(32), numero(1827) ).
```

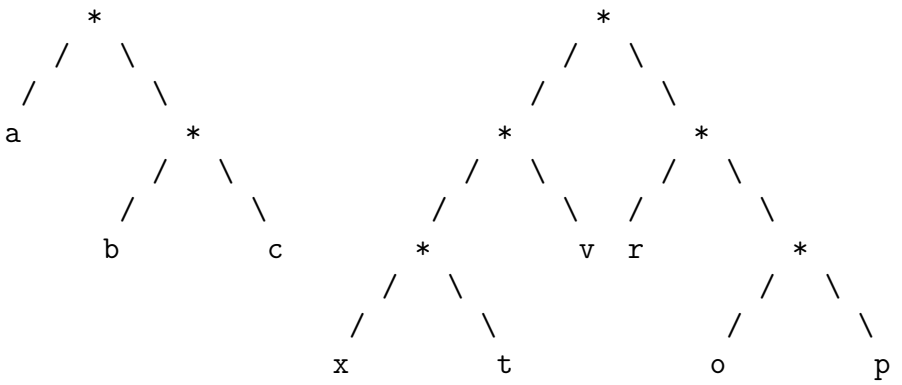
Un símbolo funcional distinguido, y que será denotado con un punto («.») permite definir expresiones simbólicas (árboles binarios). Estas estructuras, llamadas S-expresiones, fueron introducidas para los lenguajes funcionales como LISP. Su sintaxis es muy simple:

```
S-expresión ::= átomo
S-expresión ::= "[" S-expresión "." S-expresión "]"
```

Mediante el símbolo funcional «`.`», y admitiendo para él una notación infija, se obtienen las S-expresiones como términos del lenguaje. Así por ejemplo se pueden escribir los términos:

```
[a.[b.c]], [[[x.t].y].[r.[o.p]]]
```

los que *visualizados* como árboles binarios tendrían la siguiente representación:



Para la definición recursiva de los árboles es necesario considerar un elemento particular que representa el árbol *vacío* y que se denotará con «`[]`» (también se le representa como `NIL`). Con dicho elemento es posible considerar un caso particular de árboles binarios, que corresponderá al concepto de listas. Una lista, o sucesión de elementos, por ejemplo (*a b c d*), tendrá como representación al siguiente árbol binario: `[a.[b.[c.[d.[]]]]]`. PÁG. 34

En base a las estructuras de árboles recién vistas, se puede diseñar un programa lógico cuyo cometido sea el de concatenar dos listas. Este es un problema clásico para resolver en programación funcional, donde el objetivo es escribir una función que recibiendo como argumentos dos listas, como por ejemplo (*a b*) y (*c d*), evalúe como resultado la concatenación de las mismas. En el ejemplo sería: (*a b c d*).

En programación en lógica el programa es el siguiente:

```
CONCAT( [],    X, X ) <- .
CONCAT( [X.L], Y, [X.Z] ) <- CONCAT( L, Y, Z ).
```

De acuerdo a lo que se planteó en la sintaxis, este programa es un conjunto de dos reglas, que tal como se vió en la introducción definen una relación: PÁG. 35

```
CONCAT en (listas) x (listas) x (listas).
```

En realidad el problema propuesto corresponde a una función de:

```
(listas) x (listas) -> (listas).
```

si se interpreta que el tercer argumento corresponde a la concatenación del primero con el segundo, pero en la programación en lógica sólo se pueden representar relaciones.

Cabe destacar aquí que las variables que aparecen en el programa tienen como alcance cada una de las reglas. Es decir que la `X` en la primera regla es diferente a la de la segunda.

Varias pueden ser las formas de *leer* el programa `CONCAT`. La primera regla indica que la concatenación de la lista vacía con cualquier listas `X`, da como resultado la misma lista `X`. Esta regla expresa además que éste es un hecho sin condicionantes, ya que no aparecen predicados a la derecha del símbolo «`<-`». La segunda regla puede leerse de derecha a izquierda. Si adoptamos la notación de «`*`» para el operador de concatenación de listas, la regla representa:

$$\textbf{si } Z = L * Y \textbf{ entonces } \forall X \ [X.Z] = [X.L] * Y$$

lo que equivale al paso inductivo de una definición recursiva de la relación `CONCAT`.

Un primer elemento a destacar de los programas en lógica, es su carácter declarativo. Al analizar el programa `CONCAT`, se constata que el mismo se parece mucho más a una definición formal que a una descripción algorítmica. Uno de los problemas más complejos de la programación tradicional es el de estableceer mecanismos formales que permitan especificar el problema a resolver, y que además sirvan para determinar si un cierto algoritmo dado verifica las condiciones del problema. Dicho de otra manera, la dificultad consiste en vincular la especificación de un problema donde se define *qué* significa, con un algoritmo para resolverlo donde se expresa *cómo* hacerlo. La brecha entre el *qué* y el *cómo* se reduce enormemente en la programación lógica, como puede constatarse en los ejemplos presentados, aún cuando existen dificultades al introducir los componentes de control, tal como se verá en el capítulo siguiente.

PÁG. 36

Una segunda característica muy importante de la programación lógica, se basa en sus aspectos relacionales, que la diferencian de los otros paradigmas que asocian funciones a los programas. Tanto en la programación imperativa tradicional como en la programación funcional, queda perfectamente establecido la dependencia entre datos de entrada y resultados de la ejecución. El flujo de información es un elemento de diseño, y un cambio del mismo implica una modificación importante del programa. Supóngase por ejemplo que se dispone de un programa en PASCAL (o en ADA, o en cualquier lenguaje clásico), que computa la función *concatenación* en el sentido que recibe dos listas (`X` y `Y`) como datos de entrada, y devuelve como resultado la lis-

ta (`Z`) obtenida de concatenar la primera con la segunda ($Z = X * Y$). Si en un momento dado resulta necesario cambiar ligeramente las condiciones del problema, por ejemplo dadas dos listas (`X` y `Z`), se desea calcular una tercera lista (`Y`) tal que esta última concatenada a la derecha con el primer argumento dé como resultado el segundo argumento ($Z = X * Y$), entonces se debe proceder a realizar cambios sustanciales en el programa original. El aspecto relacional de la programación en lógica le da un carácter de reversibilidad a sus argumentos, y el mismo programa permite, en general, todas las combinaciones posibles de dependencias de entrada/salida.

Si se considera el programa `CONCAT` ya visto, las siguientes son utilizaciones posibles del mismo:

```
<- CONCAT ( (a b), (c d), Z ).
```

la ejecución da como resultado

```
Z = (a b c d).
```

```
<- CONCAT ( (a b), Y, (a b c d) ).
```

la ejecución da como resultado

```
Y = (c d).
```

PÁG. 37

```
<- CONCAT ( X, Y, (a b) ).
```

la ejecución da como resultados

```
X = ()      , Y = (a b).
X = (a)     , Y = (b).
X = (a b)   , Y = ().
```

donde «`()`» representa a la lista vacía (o árbol vacío «`[]`»).

La sintaxis de la programación en lógica presentada proviene de la correspondiente a la lógica de primer orden. En el lenguaje de dicha lógica se utilizan los vocabularios ya mencionados, así como conectores lógicos y los cuantificadores (universal y existencial). Con miras a unificar el tratamiento de las fórmulas bien formadas de la lógica de primer orden se han definido diversas formas normales. Entre ellas resulta de particular interés la llamada forma clausal de Skolem, cuyo formato general es el siguiente:

$$\forall X_1 \ldots X_j \; (A_1 \vee \ldots \vee A_n \vee \neg B_1 \vee \ldots \vee \neg B_m) \qquad (IV)$$

donde los A_i y los B_j son predicados en el sentido de la definición vista.

En dicho formato no aparecen los cuantificadores existenciales, y que los predicados están relacionados únicamente por el conector $\vee$. La distinción en utilizar los símbolos A y B, tiene el propósito de destacar que los segundos están prefijados por el símbolo de negación.

Se puede demostrar que para toda fórmula bien formada del cálculo de predicados, existe una fórmula en forma clausal de Skolem tal que ambas tienen las mismas propiedades de satisfactibilidad semántica.

Por propiedades bien conocidas de los conectores lógicos es fácil ver que la fórmula (IV) es equivalente a la siguiente:

$$\forall X_1 \ldots X_j \ (A_1 \vee \ldots \vee A_n \leftarrow B_1 \wedge \ldots \wedge B_m) \qquad (V)$$

PÁG. 38

Definición 3.2 (Cláusulas de Horn)
Una fórmula bien formada del cálculo de predicados de primer orden está en forma clausal de Horn si:

- está en forma clausal de Skolem
- tiene a lo sumo un predicado *positivo* (no prefijado con $\neg$)

Resulta evidente que las cláusulas de Horn conforman un subconjunto estricto del lenguaje del cálculo de predicados de primer orden.

De acuerdo a la definición son tres las posibilidades para las cláusulas de Horn:

$$\begin{array}{ll} \forall X_1 \ldots X_j \ (A \leftarrow B_1 \wedge \ldots \wedge B_m) & \text{(VI)} \\ \forall X_1 \ldots X_j \ (A) & \text{(VII)} \\ \forall X_1 \ldots X_j \ (\neg B_1 \vee \ldots \vee \neg B_m) & \text{(VIII)} \end{array}$$

que es equivalente a:

$$\nexists X_1 \ldots X_j \ (B_1 \wedge \ldots \wedge B_m) \qquad (IX)$$

Dado que en la forma clausal los únicos cuantificadores que aparecen son los universales, pueden entonces ser suprimidos, interpretando toda cláusula como clausurada universalmente. Al proceder de esta manera se puede reconocer que los tres tipos posibles de cláusulas para la programación en lógica (I, II, y III) corresponden a variaciones sintácticas de las formas VI, VII y IX. De esta forma es posible dar una interpretación a cada tipo de cláusulas. Las de tipo (I) corresponden a la afirmación del predicado A por parte del programador. Las del tipo (II) representan los hechos condicionales: si B_1 y B_2 y ... y B_m son predicados ciertos, entonces A también es cierto. Finalmente las cláusulas de tipo (III) corresponden a las invocaciones de

ejecución del programa, o si se prefiere a las interrogaciones de las relaciones definidas: ¿existen valores de las variables $X_1, \ldots, X_j$ tales que con dichos valores los predicados B_1 y B_2 y ... y B_m son ciertos?

3.2. Semántica de programas en lógica

Tal como surge de la definición sintáctica de los programas en lógica, los mismos se estructuran a partir de conceptos elementales que llamamos *átomos*. Dichos átomos tienen una estructura lexicográfica bien definida: sucesión de caracteres válidos en un vocabulario dado. Pero lo que es importante, es que carecen de todo valor semántico, y es por ello que a los lenguajes con esta concepción se les llama lenguajes de manipulación simbólica. Esta carencia de valor semántico también es aplicable a los símbolos funcionales y predicativos. Sin embargo todo usuario al escribir un programa tiene una intencionalidad semántica bien definida. Así en el ejemplo de la introducción, pocos pueden dudar que lo que se pretende con dicho programa es establecer relaciones familiares entre personas. Por otra parte, cuando se plantea una pregunta, la misma tiene también un valor semántico preciso para el usuario. PÁG. 39

Esta (aparente) falta de conexión entre los objetos sintácticos y el significado asociado a los mismos, no impide que los sistemas que procesan programas en lógica los evalúen de acuerdo a las expectativas semánticas del programa en cuestión. Desde el punto de vista de la interpretación lógica, la explicación proviene del hecho que cada invocación para la ejecución de un programa es considerada como una fórmula a ser demostrada como consecuencia lógica de la teoría formada por las reglas del programa. Esto significa que para cualquier interpretación que hace *verdaderos* los elementos del programa, hará también *verdadera* a la fórmula de la pregunta.

Definición 3.3 (Interpretación)
Sea R un programa en lógica, con sus vocabularios:

- V de átomos.
- F de símbolos funcionales.
- P de símbolos predicativos.

Sea D un conjunto (dominio) dado. PÁG. 40

Una interpretación $I(D)$ sobre el programa R asigna:

1. a cada elemento de V, un elemento de D.

II. a cada elemento f de F, una función de

$$D^n \longrightarrow D$$

siendo n la aridad del símbolo f.

III. a cada elemento p de P una función de

$$D^n \longrightarrow (\text{verdadero}, \text{falso})$$

siendo n la aridad del símbolo p.

Definición 3.4 (Satisfacción lógica)
Una interpretación $I(D)$ satisface a una fórmula f, ssi su aplicación sobre la fórmula da como resultado «verdadero».

Una interpretación $I(D)$ satisface a un conjunto de fórmulas R, ssi satisface a toda fórmula de R.

Definición 3.5 (Consecuencia lógica o deducción)
Una fórmula f es consecuencia lógica, o se deduce, de un conjunto de fórmulas R, ssi para todo dominio D, toda interpretación $I(D)$ que satisface R, satisface también a f.

Notación 1 *Si f es consecuencia lógica de R, se escribirá $R \models f$.*

El concepto de consecuencia lógica es el que permite determinar la forma en que los programas en lógica son evaluados. La aparente falta de conexión que aparecía entre los elementos sintácticos y sus significados, se resuelve al considerar una invocación de ejecución de un programa como una fórmula que debe probarse como consecuencia lógica de éste. Es decir que dicha fórmula
PÁG. 41
debe ser satisfecha para *toda* interpretación que satisfaga al programa, lo que en definitiva libera a la prueba de todo valor semántico que pudiera haberle asignado el usuario en el momento de la concepción del programa.

De acuerdo a lo que se acaba de presentar, una ejecución corresponde a una prueba de consecuencia lógica, y ésta según la definición implicaría una desmostración que involucraría a toda intrepretación posible del programa a evaluar. Existen sin embargo otros mecanismos, basados en la derivación sintáctica, que permiten realizar el proceso de la prueba en forma más simple.

3.3. Inferencia lógica

La inferencia lógica es un mecanismo de derivación sintáctica que a partir de un conjunto dado de fórmulas permite derivar nuevas fórmulas, utilizando operaciones que se denominan reglas de inferencia.

El conjunto inicial de fórmulas son sentencias válidas en un cierto lenguaje y se les llama axiomas. Los axiomas junto a las reglas de inferencia constituyen lo que Frege denomina un sistema formal.

Mediante la inferencia lógica, es posible demostrar fórmulas sin necesidad de considerar interpretación alguna. Una prueba será una derivación en el sistema formal, y la utilidad de la misma surgirá de ciertas propiedades de las reglas de inferencia, llamadas de completitud, que serán presentadas más adelante.

A los efectos de esta presentación, las fórmulas a considerar serán las que satisfagan los requerimientos sintácticos que fueron introducidos en 3.1. En una primera instancia no se considerarán los predicados con variables, para estudiar en detalle los mecanismos de demostración. La influencia de la presencia de variables será considerada en la sección 3.4.

Una de las reglas de inferencia más conocida, y que forma parte de numerosos sistemas formales es la denominada *modus ponens*. Ella se basa en el conector de implicación lógica «$\rightarrow$». PÁG. 42

Definición 3.6 (Modus ponens)
De las fórmulas A y $A \rightarrow B$, se puede inferir B, lo que usualmente se escribe de la siguiente forma:

$$\frac{\begin{array}{c} A \\ A \quad \rightarrow \quad B \end{array}}{B}$$

En lo que sigue se presentan una serie de definiciones que serán utilizadas posteriormente.

En un sistema formal, un paso de inferencia corresponde a la aplicación de una regla para inferir una nueva fórmula.

Definición 3.7 (Demostración)
Una demostración será una sucesión $F_1, F_2, \ldots, F_n$ de fórmulas del lenguaje, tal que todo F_i es un axioma, o se obtiene de las fórmulas anteriores en la sucesión por la aplicación de alguna regla de inferencia.

Definición 3.8 (Teorema)
Una fórmula F es un teorema si existe una demostración en la que F es el último término de la sucesión.

Notación 2 *Si F es un teorema se notará* $\vdash F$

PÁG. 43

Definición 3.9 (Deducción lógica)
Sea F una fórmula y R un conjunto de fórmulas de una cierta teoría, se dice que F es deducible lógicamente a partir de R, y se escribe $R \vdash F$, si existe una sucesión de fórmulas $F_1, F_2, \ldots, F_n$ tal que $F = F_n$ y cada F_i es

- un axioma de la teoría
- o una fórmula de R
- o deducible de un fórmula precedente en la sucesión.

Las fórmulas de R se llaman hipótesis.

Notación 3 *Si A es deducible lógicamente de R se notará: $R \vdash A$*

Una propiedad importante, que será utilizada más adelante, es la siguiente:

Definición 3.10 (Consistencia)
Un sistema formal, en el que existe el símbolo de la negación «$\neg$» se dice que es consistente si no existe una fórmula F en el sistema tal que pueda deducirse F y $\neg F$. Un sistema se llama inconsistente en caso contrario.

Resumiendo los conceptos presentados, se han introducido dos alternativas vinculadas con la idea de prueba de fórmulas. En primer lugar se tiene el concepto de *satisfacción lógica*, que tiene connotaciones semánticas, y que en definitiva es la que interesa cuando se intenta dar un significado a la ejecución de un programa en lógica. Sin embargo no parece evidente encontrar mecanismos automatizables para la prueba, que consideren todas las interpretaciones posibles, tal como se necesita a partir de la definición de satisfacción lógica. PÁG. 44
En segundo lugar se presentó el mecanismo de inferencia lógica, que basado en elementos sintácticos, permte definir derivaciones entre fórmulas, e introducir el concepto de *teoremas*, a partir de un conjunto de axiomas, y de las reglas de inferencia utilizadas. Esta alternativa parece ser más útil desde un punto de vista informático, si se encuentran procedimientos que implementen las reglas del sistema.

La vinculación entre ambas alternativas surge de la propiedad de *completitud* de las reglas de inferencia.

Definición 3.11 (Completitud)
Sea un programa en lógica P, y una cláusula p que corresponde a una invocación del programa P.

Sea Q una regla de inferencia.

Se dice que Q es completa si se cumple que:

p es deducible lógicamente de P, utilizando Q ssi p es consecuencia lógica de P

O sea $P \vdash p \Leftrightarrow P \models p$

Nota. 1 En la literatura se encuentra que frecuentemente se le llama robustez (*soundness*) a la parte «sólo si» de la propiedad definida, y completitud a la parte «si» de la misma.

Según la definición anterior, si se dispone de reglas de inferencia que cumplan con la propiedad de completitud, alcanza con demostrar la deducibilidad de una fórmula para demostrar que la misma es consecuencia lógica del programa.

Dentro del área de automatización de pruebas mediante el concepto de deducibilidad, un aporte importante es debido a Robinson, quien introdujo la llamada *regla de resolución*.

PÁG. 45

Definición 3.12 (Regla de Resolución)
Sean $A_1, \ldots, A_n$ y $B_1, \ldots, B_m$ símbolos predicativos, la regla dice:

$$\frac{\begin{array}{l}\neg(A_1, \ldots, A_k, \ldots, A_n) \\ A_k \leftarrow B_1, \ldots, B_m\end{array}}{\neg(A_1, \ldots, A_{k-1}, B_1, \ldots, B_m, A_{k+1}, \ldots, A_n)}$$

De la definición puede observarse que la regla de resolución resulta particularmente adaptada para los programas en lógica. Una invocación o pregunta, corresponde a fórmulas del primer tipo, y las reglas o cláusulas del programa corresponden al segundo tipo. La aplicación de las reglas introduce una nueva cláusula producto de la anulación o cancelación del predicado A_k.

Como casos particulares de a regla de resolución se pueden considerar los siguientes:

a)

$$\frac{\begin{array}{l}\neg A \\ \quad A \leftarrow B\end{array}}{\neg B}$$

llamada *modus tollens*.

b)

$$\frac{\begin{array}{l}\neg A \\ \quad A\end{array}}{\square} \qquad \text{o «} A \leftarrow \text{.» según la sintaxis.}$$

que significa que se ha concluído con una contradicción.

El aporte realmente importante de la regla de resolución surge del siguiente teorema.

PÁG. 46

Teorema 3.1 (Completitud de la resolución) *La regla de resolución es completa.*

En virtud de lo anterior es posible concebir un sistema de demostración automática que evalúe los programas en lógica, mediante la aplicación reiterada de la regla de resolución, tratando de demostrar la deducibilidad lógica de la invocación.

Una manera de realizar la prueba es la llamada prueba por contradicción o por reducción al absurdo, que se basa en el siguiente teorema.

Teorema 3.2 *Sea F una fórmula y R un conjunto de fórmulas. F es deducible lógicamente a partir de R, ssi el conjunto formado por R con $\neg F$ es inconsistente.*

O sea,

$$R \vdash F \Leftrightarrow R \cup \{\neg F\} \text{ es inconsistente}$$

Las definiciones y teoremas presentados son la base para comprender el significado de los programas en lógica, y de la ejecución de los mismos. Las demostraciones se han omitido para abreviar la exposición. El lector interesado puede encontrar una presentación formal y completa de estos temas en el libro de E. Mendelson [Men64], y un buen resumen de los mismos en el primer capítulo de la tesis de R. Caferra [Caf82].

Es posible ahora presentar un modelo sobre la evaluación de un programa en lógica. De la definición sintáctica se sabe que los tres tipos de reglas posibles en programación en lógica son:

$$\begin{array}{lcll} A & \leftarrow & . & \text{(I)} \\ A & \leftarrow & B_1, \ldots, B_m & \text{(II)} \\ & \leftarrow & B_1, \ldots, B_m & \text{(III)} \end{array}$$

PÁG. 47

Las reglas tipo (I) y (II) corresponden a las fórmulas que conforman el programa: en el sistema lógico representan los axiomas. Las fórmulas de tipo (III) son las invocaciones de ejecución, y si se recuerda su interpretación como cláusulas de Horn, representan a la negación de la conjunción de los B_i, tal como aparece en la fórmula (IX) vista anteriormente.

El modelo se presenta fácilmente mediante un ejemplo.

Ejemplo. 1
Sea el programa P:

$$\begin{array}{lcll} A & \leftarrow & . & (1) \\ B & \leftarrow & . & (2) \\ D & \leftarrow & . & (3) \\ C & \leftarrow & D, A. & (4) \\ F & \leftarrow & C, B. & (5) \end{array}$$

y la invocación dada por la fórmula p:

$$\leftarrow \quad F. \quad (6)$$

La intención es probar que

$$P \models F$$

lo que se realiza demostrando mediante la regla de resolución que:

$$P \vdash F$$

Realizando la demostración por el absurdo, el procedimiento será probar que:
el conjunto $P \cup \{\neg F\}$ es inconsistente.
Pero la fórmula p es $\neg F$, por lo que se debe demostrar es que:
el conjunto $P \cup \{p\}$ es inconsistente

Para demostrar que el conjunto 1-6 es inconsistente se va aplicando resolución para inferir nuevas cláusulas. Por ejemplo:

PÁG. 48

De (5) y (6) se infiere $\leftarrow C, B$ (7)
De (4) y (7) se infiere $\leftarrow D, A, B$ (8)
De (3) y (8) se infiere $\leftarrow A, B$ (9)
De (1) y (9) se infiere $\leftarrow B$ (10)
De (2) y (10) se infiere $\leftarrow [\,]$ (11)

El paso (11) indica que se ha inferido el conjunto vacío. Esto surge de considerar las reglas (2) y (10). La primera afirma B y la segunda afirma $\neg B$, por lo tanto el sistema es inconsistente.

De la evaluación del ejemplo surgen las siguientes consideraciones:

1. Una invocación corresponde a una cláusula de tipo (III) con un solo símbolo predicativo.

2. Las nuevas cláusulas que se van infiriendo corresponden a clásulas de tipo (III), donde cada uno de los símbolos predicativos que aparecen en ellas corresponderán a pasos de inferencia que deberán ser realizados.

3. Cuando la resolución se aplica entre una cláusula de tipo (II) y una b de tipo (III), la cláusula obtenida tiene los símbolos de b, menos el símbolo consecuente de a, que ha sido sustituído por el conjunto de sus antecedentes. Aquí la cláusula obtenida tiene más, o la misma cantidad de símbolos que b.

4. La estrategia de la demostración consiste en buscar inferir la cláusula vacía ($[\,]$).

3.4. Unificación

En la sección anterior se ha hecho hincapié en el proceso de la demostración, y para ello no se han considerado las variables que pueden aparecer en los predicados, tal como lo expresa la sintaxis de los programas en lógica.

PÁG. 49

Se asume que las variables de una cláusula están cuantificadas universalmente, y tienen como *alcance* a la propia cláusula.

En estos casos el procedimiento de demostración es similar a lo ya visto, salvo que es necesario contemplar la presencia de variables. La resolución se realizará sólo cuando es posible encontrar una instanciación de las variables que hacen iguales los predicados a anular.

Por ejemplo, sea el programa siguiente:

```
PADRE ( jorge, diego ) <- .                (1)
HERMANO ( ricardo, jorge ) <- .            (2)
TIO(X, Y) <- PADRE(Z, Y), HERMANO(X, Z).   (3)
```

La tercera cláusula, como se ha dicho, está cuantificada universalmente para las variables `X`, `Y` y `Z`.

Si se considera la invocación:

```
<- TIO ( W, diego ).                       (4)
```

La demostración se realiza de la forma siguiente:

I) La cláusula (3) está cuantificada universalmente. Por lo tanto es también válida para el caso que la variable `Y` tenga como valor `diego`, situación que puede representarse por

```
TIO(X, diego) <- PADRE(Z, diego), HERMANO(X, Z)  (5)
```

II) Como las variables tienen por alcance la cláusula donde aparecen, entonces (4) no cambia si se reemplaza `W` por `X`. Hecho esto es posible *resolver* (4) y (5), pues se ha encontrado una instanciación de (3) que lo posibilita. De esta forma se obtiene:

```
<- PADRE(Z, diego), HERMANO(X, Z)                (6)
```

PÁG. 50

III) Se busca ahora resolver (1) con (6) a través del predicado `PADRE`. Nuevamente es necesario encontrar una instanciación, esta vez en (6), obteniéndose:

```
PADRE(jorge, diego), HERMANO(X, jorge)          (7)
```

IV) Ahora es posible la resolución buscada, realizándola entre (1) y (7), lo que permite inferir:

```
HERMANO(X, jorge)                               (8)
```

V) Finalmente, e instanciando `X` en (8) con `ricardo` se puede resolver la nueva cláusula con (2), obteniéndose:

```
[ ]
```

Los pasos I) a V) constituyen la demostración de la fórmula (4). Resulta claro que al escribir dicha fórmula el usuario no estaba directamente interesado en una prueba, sino que más bien quería conocer quién es `TIO` de `diego`. La variable `W` original fue cambiada por `X` en el paso I), y esa variable fue instanciada en el paso V) para completar la prueba. El valor que le fue asignado, es decir, `ricardo`, es la respuesta que da el sistema, y que corresponde al interés inicial del usuario.

El problema que se plantea para la regla de resolución es entonces cómo *resolver* dos predicados que tengan el mismo símbolo predicativo, pero que sus argumentos no coinciden.

El mecanismo que va a permitir resolver dicho problema es el llamado de *unificación*.

PÁG. 51

Definición 3.13 (Sustitución)
Una sustitución es un conjunto de asignaciones del tipo:

$$X := t$$

donde X es una variable y t es un término, en el sentido de la definición sintáctica de los programas en lógica. En una sustitución no pueden existir más de una asignación a la misma variable.

Ejemplos

$\{X := \text{juan}, Y := \text{noe}\}$
$\{W := Z, R := \text{empleado}(T, 1200)\}$
$\{Q := [\,], X := [Y, Z]\}$

Definición 3.14 (Aplicación de sustitución)
Dada una sustitución θ, y un predicado P, la aplicación de θ a P produce un nuevo predicado, que se denota $P\theta$, y que corresponde al predicado inicial P, donde toda variable asignada en θ es cambiada por el término correspondiente, y las otras variables permanecen incambiadas.

Definición 3.15 (Unificador)
Dadas dos expresiones del lenguaje definido, por ejemplo dos predicados

$$E_1, E_2$$

se llama unificador a una sustitución θ tal que se cumple que:

$$E_1\theta = E_2\theta$$

Es decir que la aplicación de la sustitución a ambas expresiones, da la misma expresión.

PÁG. 52

Ejemplos

1. Dadas las expresiones

   `PADRE ( Z , diego)` y `PADRE ( jorge , diego)`

   la sustitución θ = { `Z := jorge` } es un unificador de las mismas.

2. Dadas las expresiones

   `TIO ( X , diego)` y `TIO ( W , diego)`

   la sustitución θ = { `X := W` } es un unificador de las mismas. La sustitución θ = { `W := X` } también es un unificador. La sustitución θ = { `X := guillermo, W := guillermo` } también es un unificador.

3. Dadas las expresiones

   $$Q(q(X,Y),X,h(4)) \quad \text{y} \quad Q(q(3,Z),W,h(Z))$$

   la sustitución $\theta = \{X := 3, Z := 4, W := 3, Y := 4\}$ es un unificador de las mismas.

4. Sean las expresiones

   `R ( [X . Y] )` y `R( [[a.[b.[]]].[c.[d.[]]]] )`

Debe recordarse que el término `[[a.[b.[]]].[c.[d.[]]]]` corresponde a la lista $((a\ b)\ c\ d)$.

La sustitución θ = { `X:= [a.[b.[]]], Y := [c.[d.[]]]`} es un unificador de las mismas.

En notación de listas: $\theta = \{X := (a\ b), Y := (c\ d)\}$.

Si se utiliza la notación de listas, lo que frecuentemente resulta muy cómodo, la expresión `[X.Y]` representa una lista, donde, según otros lenguajes, `X` corresponde al «`CAR`», «*head*» o «*cabeza*» de la lista; `Y` corresponde al «`CDR`», «*tail*», o «*resto*» de la misma; «`.`» corresponde al constructor «`CONS`».

5. Dadas las expresiones

   `R ( [X . Y] )` y `R( [] )`

   No existe un unificador para ambas.

De la definición de unificador y de los ejemplos presentados surge que existen expresiones para las cuales no existe un unificador, mientras que en otros casos es posible encontrar más de un unificador. PÁG. 53

Para el procedimiento de demostración, y en caso de existir más de un unificador entre dos predicados que permita aplicar la regla de resolución, va a interesar aquel que sea *más general*, en el sentido que necesite asignaciones menos específicas de términos a variables. En el ejemplo 2. visto anteriormente, los dos primeros unificadores, que en definitiva son el mismo —a menos de un renombramiento— son más generales que el tercero.

Definición 3.16 (Unificador más general)
Dadas dos sustituciones θ_1 y θ_2 y que ambas son unificadores de las expresiones E_1 y E_2, se dice que θ_1 es más general que θ_2 si existe una sustitución θ_3 tal que:

$$E_1\theta_1\theta_3 = E_2\theta_2$$

La relación «más general que» es un preorden. Al mínimo del preorden «más general que» se le llama *unificador más general*.

Existen múltiples propuestas de algoritmos para calcular el unificador más general, y ello se debe a que dicho algoritmo tiene una importancia capital en el proceso de la demostración.

3.5. Evaluación de programas en lógica

En las secciones precedentes se ha visto la síntesis de los programas en lógica, así como las bases lógicas que permiten dar un significado a la ejecución de los mismos.

En lo que sigue se presentarán las acciones a cumplir para ejecutar un paso de inferencia usando resolución.

PÁG. 54 Se dispone de un programa P formado por las cláusulas de tipo I) y II) ya vistas, y una cláusula de tipo III)

$$\leftarrow Q_1, \ldots, Q_m$$

que corresponde a la invocación de ejecución.

Al conjunto $Q = \{Q_1, \ldots, Q_m\}$ se le denomina conjunto de objetivos.

LA evaluación del programa implica la aplicación de la regla de recolución. Para cada paso de inferencia se procede siguiendo los pasos siguientes:

1. Se elige un predicado del conjunto Q tal que su símbolo predicativo sea igual al símbolo del predicado consecuente de una regla del programa. Si esto no es posible fracasa el paso inferencial.

2. Se calcula el unificador «más general» entre el predicado escogido y el consecuente de la regla determinada en el paso anterior. Si no existe tal unificador, entonces fracasa el paso de inferencia.

3. Se reemplaza en el conjunto Q el predicado escogido, por el conjunto de predicados que aparecen como antecedentes de la regla determinada; si la regla es de tipo I), entonces se elimina el predicado de Q. En este paso se modifica el conjunto Q, eventualmente convirtiéndolo en el conjunto vacío.

4. Se aplica el unificador más general a todos lo predicados de Q.

Ejemplo Sea el programa en lógica para la concatenación de listas que ya fue presentado:

```
CONCAT(    [], X,    X  ) <- .                    (1)
CONCAT( [X.L], Y, [X.Z] ) <- CONCAT( L, Y, Z ).   (2)
```

PÁG. 55 y la invocación:

```
<- CONCAT ( [a.[]], [b.[]], W ).
```

El conjunto de objetivos a ser demostrado es:

$$Q = \{\texttt{CONCAT}([\texttt{a}.[]], [\texttt{b}.[]], \texttt{W})\}$$

I La regla (1) no puede resolverse con el único predicado de Q. Dicha regla necesita que su primer argumento sea la lista vacía, mientras que el predicado tiene en ese argumento una lista no vacía.

II Se resuelve la regla (2) con el objetivo, mediante la sustitución

$$\theta_1 = \{\texttt{X} := \texttt{a}, \texttt{L} := [], \texttt{Y} := [\texttt{b}.[]], \texttt{W} := [\texttt{a}.\texttt{Z}]\}$$

III Se sustituye el predicado de Q por el antecedente de la regla, y se le aplica la sustitución anterior quedando

$$Q = \{\texttt{CONCAT}([], [\texttt{b}.[]], \texttt{Z})\}$$

IV El nuevo y único predicado de Q no puede resolverse con la regla (2) debido al primer argumento de `CONCAT`. El de la regla es `[X.L]`, y ello indica, por la existencia del símbolo funcional «`.`», que se necesita una lista con al menos un elemento, mientras que el argumento del predicado de Q es precisamente la lista vacía.

V Se resuelve la regla (1) con el objetivo, mediante la sustitución

$$\theta_2 = \{\texttt{X} := [\texttt{b}.[]], \texttt{Z} := [\texttt{b}.[]]\}$$

VI Como la regla aplicada es del tipo I), se elimina el predicado del conjunto Q, quedando

$$Q = \{\ \}$$

por lo que se termina la demostración, concluyéndose que la invocación es consecuencia lógica del programa.

Es que lo que el usuario necesita es el valor de `W`. PÁG. 56

VII En el paso ii) la primera sustitución produjo que

```
W := [a.Z]
```

VIII En el paso v) la segunda sustitución profujo que

```
Z := [b.[]]
```

IX De vii) y viii) se concluye que el valor de `W` que permitió la demostración es:

```
W := [a.[b.[]]]
```

es decir la lista $(a\ b)$

Problemas

1. Definir el predicado `MEMBER(X,L)` que determina si el elemento `X` pertenece a la lista `L`.

2. Definir los siguientes predicados para listas:

   a) `SUFIX(L1,L2)` donde `L1` es sufijo de `L2`.

   b) `PREFIX(L1,L2)` donde `L1` es prefijo de `L2`.

   c) `SUBLIST(L1,L2)` donde `L1` es una sublista de `L2`.

3. Escribir un programa en lógica que ordene listas, usando la idea de *quicksort*.

4. Definir el predicado `ISOTREE(T1,T2)` que será verdadero si `T1` y `T2` son árboles binarios isomorfos (es decir, iguales a menos de los valores asociados a los nodos).

Capítulo 4

Interpretación algorítmica

Pág. 57

La interpretación algorítmica de la programación en lógica, induce una semántica operacional, que sirve de base a las implementaciones existentes. Un programa en lógica se interpreta como un conjunto de definiciones de procedimientos. Sin embargo el modelo de activación y ejecución de los procedimientos aquí presentados, no corresponde al tradicional de los lenguajes imperativos. Por un lado los parámetros servirán para determinar cuál procedimiento activar. La ejecución corresponderá al recorrido de un espacio de computaciones posibles, ya que el modelo general de evaluación será no determinista. En este capítulo se analizarán todos estos aspectos y se culminará presentando las estrategias usuales de evaluación.

Version: $Id: plyf.tex,v 1.3 2003/08/13 04:28:26 cballard Exp $

PÁG. 58

En el capítulo anterior se presentó una interpretación lógica que permite entender el significado de los programas en lógica. Un programa representa la definición axiomática de una teoría formal, y una invocación a la ejecución del mismo se interpreta como una fórmula a ser demostrada como válida en dicha teoría. Dicha demostración se puede realizar mediante los mecanismos de derivación sintáctica, utilizando reglas de derivación que sean completas. En particular se presentó la regla de resolución de Robinson, que cumple con dicha propiedad, y se culminó el capítulo con un modelo para la aplicación de un paso inferencial. Sin embargo queda sin explicar una estrategia general de la prueba, que contemple los casos de imposibilidad de unificación, así como la selección de predicados y reglas del programa. En definitiva, es necesario definir un control global que especifique en cada punto de decisión cuál es el camino a seguir.

La componente de control es una de las características más claras de la programación imperativa tradicional. La semántica operacional de los lenguajes de programación clásicos se basan en una máquina de estados, y cada constructor se define en términos de de la transición de estados que la ejecución del mismo produce. La evolución de la máquina de estados es la base del concepto de algoritmo. La dificultad que surge para utilizar el mismo enfoque en la programación en lógica proviene de que la máquina que naturalmente se necesitaría, sería no determinística, en el sentido en que se aplica en teoría de autómatas.

La ejecución de un programa en lógica implica la existencia de una derivación sintáctica. El problema que se presenta es cómo hallar dicha derivación. De la metodología presentada en el capítulo anterior, surge que en el proceso de la prueba existen varios puntos donde es necesario realizar una decisión. En los ejemplos presentados las elecciones de objetivos y reglas se han hecho escogiendo la *buena* alternativa. No se planteó, sin embargo, qué hacer en caso contrario. Lo que está faltando, entonces, es definir una componente de

PÁG. 59

control a la programación en lógica, que determine cómo proceder para la ejecución total de un programa.

Para expresar una visión de la programación que se fue imponiendo hace unos años, Wirth escribió un libro con un título muy sugestivo:

$$\text{Algoritmos} + \text{Estructuras de datos} = \text{Programas}$$

donde se pretende poner en evidencia que un programa se obtiene a partir de la información estructurada a través del concepto de *tipo* (que constituye una componente estática), combinada con un algortimo, donde aparece el control.

Kowalski[Kow79] por su parte escribió un artículo cuyo título, emulando al de Wirth, fue:

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

En este caso la componente estática proviene de la lógica, y si a ella se le agrega un mecanismo de control, entonces se obtienen los algoritmos. Es de hacer notar que en la parte que representa la lógica en la ecuación se incluyen tanto los datos como la especificación del problema a resolver. Este elemento es el que le da el carácter denotacional a la programación en lógica, y utilizando una terminología que ya ha sido presentada, es posible reformular la ecuación de Kowalski escribiendo:

$$\text{Algoritmos} = \textit{Qué} + \textit{Cómo}$$

La interpretación algorítmica que se verá en este capítulo se concentrará en cómo iintroducir una componente de control a la programación en lógica. En esta interpretación los programas serán considerados como definiciones de *procedimientos*, y la ejecución de los mismos se realizará siguiendo el modelo de invocación a rutinas de un lenguaje a la Pascal. Es por dicha razón que a la interpretación que aquí se llama algorítmica, se la llama también procedimental, que pretende ser una adaptación al castellano de la palabra inglesa «*procedural*».

El auge actual de la programación en lógica, se inicia con los trabajos de PÁG. 60
A. Colmerauer y Ph. Roussel en la Universidad de Aix-Marsella II. Ellos crearon en 1973 el lenguaje PROLOG, inicialmente concebido para el tratamiento del lenguaje natural[Col73]. El nombre PROLOG proviene de *PROGgrammation en LOGique*, y una primera presentación del mismo apareció en[Rou75]. Luego de implementaciones internas realizadas por Ph. Roussel, el primer intérprete que tuvo difusión fue escrito por Battani y Meloni[Bat73]. Dicho intérprete fue escrito en lenguaje FORTRAN, y ese hecho facilitó su instalación en diversas universidades y centros de investigación; ¿quién no disponía de un compilador FORTRAN en aquella época? (¡y aún hoy día!). Enormes paquetes de tarjetas perforadas con el intérprete se trasladaron por Europa, llegando a Inglaterra, Escocia, Hungría, Portugal, entre otros países. En 1976, uno de los autores de este trabajo, J. Vidart, consiguió una copia que instaló en la Universidad Simón Bolívar, en Venezuela. Se formó así el primer grupo en Latinoamérica trabajando en PROLOG, y ese grupo se reforzó con la presencia durante cuatro años de Ph. Roussel.

PROLOG fue durante bastante tiempo un lenguaje de investigación, considerado por muchos como un juguete académico. Su utilización estaba poco generalizada, e incluso era muy poco conocido en los Estado Unidos de Norteamérica, donde se pensaba que para las aplicaciones de Inteligencia Artificial alcanzaba con el lenguaje LISP. Sin embargo, se habían producido

avances muy importantes en la implementación de PROLOG, y el excelente trabajo de Warren[War79] constituyó un aporte de capital importancia, al abandonar el terreno de los prototipos y disponer de eficiencia en el trabajo con el lenguaje. A nivel de la divulgación de PROLOG, el libro de Clocksin y Mellish[Clo81] significó a su vez, un aporte relevante.

No fue, sin embargo, hasta la aparición del proyecto de 5.ª generación de computadoras anunciado por el Japón, que PROLOG adquirió relevancia internacional, y pasó a ocupar el primer plano de la investigación y desarrollo de lenguajes para la inteligencia artificial. El proyecto japonés representó un
PÁG. 61
verdadero impacto en el desarrollo de la inteligencia artificial, y de la informática en general, tal vez no tanto por los objetivos planteados, sino por el cambio que manifestaba en la priorización de las herramientas informáticas. La programación de los años ´90 sería simbólica y no numérica: las nuevas máquinas deben ser capaces de inferir, más que de calcular; la capacidad de esas máquinas se deberá medir en LIPS (*logical inferences per second*) y no en MIPS como actualmente. Desde el punto de vista de diseño, la máquina a que apunta el proyecto, tendría como lenguaje de máquina a alguna versión de PROLOG.

A partir del anuncio japonés, PROLOG tuvo un desarrollo impresionante. Se multiplicaron las implementaciones, y aparecieron compiladores muy eficientes. Las desventajas iniciales respecto a las implementaciones de lenguajes como LISP, se han ido reduciendo, y actualmente la decisión de elegir entre PROLOG y LISP para una aplicación de inteligencia artificial, no pasa necesariamente por la eficiencia de los programas que se construyen. No sólo se han desarrollado implementaciones para máquinas de tamaño importante, sino que existen múltiples versiones para micro computadores. La aparición del TURBO-PROLOG (MR) de Borland a un precio muy accesible, contribuyó a la difución masiva del lenguaje, si bien introduce algunas modificaciones a lo que puede llamarse el PROLOG tradicional.

No solamente el desarrollo consistió en implementaciones cada vez más eficientes, sino que además se ha ido generando una serie de lenguajes derivados del PROLOG inicial, donde se ha ido incorporando mejoras. Esencialmente las modificaciones conciernen a las primitivas de control, y a mecanismos de modularización para facilitar la construcción de sistemas de tamaño importante.

Para poder utilizar alguna versión de PROLOG como lenguaje de base para el proyecto de 5.ª generación, resultaba imprescindible disponer de primitivas explícitas para el manejo de la concurrencia. Los lenguajes Concurrent PROLOG y GHC (*Guarded Horn Clauses*) proveen mecanismos para expresar que ciertas operaciones pueden ejecutarse en paralelo.

4.1. Procedimientos y programación en lógica

La interpretación algorítmica de la programación en lógica se basa en el concepto de procedimiento de los lenguajes imperativos tradicionales. Un programa será la definición de un conjunto de procedimientos, y una invocación de ejecución será precisamente la invocación de un procedimiento. El control general corresponderá, entonces, a la ejecución de procedimientos, que serán activados mediante una invocación, y que al terminar devolverán el control al procedimiento invocante. El pasaje de parámetros adquirirá un papel más importante que en el sentido tradicional, ya que de la adecuación de los parámetros reales y formales (que es lo que corresponde al concepto de unificación ya visto) dependerá si un cierto procedimiento es susceptible de ser invocado o no. PÁG. 62

La diferencia esencial entre el concepto de procedimientos que serán presentados en lo que sigue, y el símil de los lenguajes imperativos, radica en el aspecto no determinístico de la programación en lógica.

En un lenguaje como Pascal, el orden en que se van a ejecutar las instrucciones del programa está dado explícitamente por el usuario. Una invocación a un procedimiento implica que cuando deba ejecutarse la misma, el control será transferido al procedimiento, que se identifica sin ambigüedades en la misma invocación. En cambio en programación en lógica, una invocación puede provocar la activación de más de un procedimiento. Cada uno de ellos implica un camino de ejecución posible. Habiendo tomado una decisión de cuál camino escoger es posible que el camino seleccionado no produzca una solución, en el sentido del fracaso del proceso inferencial que se vio en el capítulo anterior. Dicho problema no implica que no exista una solución en los otros caminos que fueron descartados, por lo que se hace necesario volver al punto de decisión y escoger otra alternativa. Por otro lado, podría suceder que se obtenga una solución en la primera decisión, pero al estar evaluando relaciones, es posible que existan más soluciones por los otros caminos. También en este caso es necesario volver al punto de decisión y explorar todas las alternativas posibles. Las implementaciones de programación en lógica deben contemplar las alternativas expuestas, y utilizan el mecanismo de *backtracking* para resolverlo. PÁG. 63

4.2. Definición de procedimientos

Un programa en lógica está formado por un conjunto de reglas, o cláusulas, de acuerdo a la sintaxis vista en el capítulo anterior. Dichas reglas son

de dos tipos:

$$\begin{array}{lcll} A & \leftarrow & . & \text{(I)} \\ A & \leftarrow & B_1, \ldots, B_m & \text{(II)} \end{array}$$

Cada regla de un programa se interpretará como la definición de un procedimiento. De acuerdo a la tradición de la programación imperativa toda definición está compuesta de un encabezamiento y un cuerpo. El predicado que aparece a la izquierda del símbolo «←» será el encabezamiento, y los predicados que aparecen a la derecha de dicho símbolo constituirán el cuerpo del procedimiento. Los argumentos que pueda contener el predicado del encabezamiento serán los parámetros formales. Los predicados que aparecen en el cuerpo representan invocaciones a otros procedimientos, que serán ejecutadas durante la evaluación del procedimiento. El orden en que aparecen los predicados en el cuerpo no necesariamente indica la secuencia en la que se harán las invocaciones. Si la regla es de tipo (I), entonces el procedimiento tiene cuerpo vacío, y la evaluación del mismo se reduce a un retorno de control.

Si se considera nuevamente el programa para la concatenación de listas:

```
CONCAT (   [] , X,   X   ) <- .
CONCAT ( [X.L], Y, [X.Z] ) <- CONCAT ( L, Y, X ).
```

el mismo está formado por la definición de dos procedimientos. El primero tiene cuerpo vacío, mientras que el segundo al ser activado producirá una invocación recursiva.

4.3. Invocación de procedimientos

Pág. 64

Tal como se planteó en la sintaxis una invocación de ejecución de un programa tendrá la forma:

```
<- A.
```

En la interpretación algorítmica dicha cláusula representa una invocación de procedimiento.

Un requerimiento de evaluación del programa del ejemplo sería:

```
<- CONCAT ( [a.[]] , [b.[]] , Z ).
```

La semántica de la activación de un procedimiento en un lenguaje estilo ALGOL, indica que una invocación es sustituída por el cuerpo del procedimiento, luego de haber realizado la transferencia de parámetros. En el caso de la programación en lógica el proceso es similar, salvo en lo que concierne a los parámetros. Éstos, debido al proceso de unificación visto en el capítulo anterior, forman parte del mecanismo de selección del procedimiento a activar. Aún cuando haya coincidencia de nombres entre el predicado de la invocación y el de la cabeza de una definición, si no existe unificación de argumentos, entonces el procedimiento es descartado. En caso que la unificación tenga éxito, se procede al reemplazo del predicado de la invocación por el cuerpo del predicado, y se aplica dicho unificador al conjunto de predicados que representan invocaciones pendientes.

Ejemplo Sea el programa:

```
P <- R, S, T.          (1)
R <- M, N.             (2)
S <- M.                (3)
T <- .                 (4)
M <- .                 (5)
N <- .                 (6)
```

y la cláusula de invocación: PÁG. 65

```
<- P.
```

que representa una invocación al procedimiento `P`. A los efectos de mostrar la evaluación del programa se considerará al conjunto Q que contendrá los nombres de los procedimientos a ser invocados. En la terminología de la interpretación lógica vista en el capítulo anterior, Q constituye el conjunto de objetivos a ser demostrados. Inicialmente:

$$Q = \{P\}$$

La evolución de los valores de Q a lo largo de la ejecución sería:

$$\begin{array}{lcll} Q & = & \{P\} & \\ Q & = & \{R, S, T\} & \text{por invocación al procedimiento (1)} \\ Q & = & \{M, N, S, T\} & \text{por invocación al procedimiento (2)} \\ Q & = & \{N, S, T\} & \text{por invocación al procedimiento (5)} \\ Q & = & \{S, T\} & \text{por invocación al procedimiento (6)} \\ Q & = & \{M, T\} & \text{por invocación al procedimiento (3)} \\ Q & = & \{T\} & \text{por invocación al procedimiento (5)} \\ Q & = & \{\,\} & \text{por invocación al procedimiento (4)} \end{array}$$

Al llegar a una situación donde el conjunto Q sea vacío, no habrá más procedimientos para invocar, y concluye la ejecución.

Ya se ha mencionado que el orden en que aparecen los predicados en el cuerpo de un procedimiento no debe significar, en principio, que con dicho orden los procedimientos serán invocados. Ello implica que al ir construyendo el conjunto Q del ejemplo, en cada paso cuaquiera de los predicados que aparecen son suceptibles de ser invocados. En el caso del ejemplo, cualquiera que sea la elección en cada paso, se obtiene la finalización satisfactoria de la ejecución.

Supóngase ahora que se modifica el programa anterior, agregándose la
PÁG. 66
cláusula:

```
N <- F.          (7)
```

El programa sigue siendo válido, y la ejecución presentada continúa siendo un camino posible en la medida que al invocar el procedimiento `N` se opte por la definición (6). Sin embargo, si la decisión hubiera recaído en la definición (7), la ejecución hubiese podido continuar, pero se llegaría a un estado representado por:

$$Q = \{F\}$$

de terminación anormal, al no disponerse de definiciones del procedimiento `F`.

De lo presentado en el ejemplo puede concluirse lo siguiente:

1. Pueden existir múltiples caminos que conduzcan a una terminación exitosa de la ejecución de un programa.

2. Pueden existir caminos que conduzcan a una terminación anormal de la ejecución.

Además, dado un programa y una invocación, puede suceder que tanto 1. como 2. sean ciertos. Esto significa que la decisión de escoger un predicado para ser activado puede tener capital importancia para obtener una ejecución que culmine exitosamente.

Habiendo presentado en forma general el problema de la invocación de procedimientos se analizará a continuación el papel que juegan las variables en la selección del procedimiento a activar. Se considera nuevamente el programa para la concatenación de listas:

```
CONCAT (  [ ] , X,   X   ) <- .                          (1)
CONCAT ( [X.L], Y, [X.Z] ) <- CONCAT ( L, Y, X ).        (2)
```

La invocación:

```
CONCAT ( [a.[]], [b.[]], W ).
```

producirá una ejecución que se presenta como evolución del conjunto Q: PÁG. 67

$$
\begin{array}{lcll}
Q & = & \{\texttt{CONCAT}([\texttt{a}.[]], [\texttt{b}.[]], \texttt{W})\} & \\
Q & = & \{\texttt{CONCAT}([], [\texttt{b}.[]], \texttt{Z})\} & \text{por proc. (2) con unif.} = \texttt{W:=[a.Z]} \\
Q & = & \{\,\} & \text{por proc. (1) con unif.} = \texttt{Z:=[b.[]]}
\end{array}
$$

Es decir que la ejecución culmina exitosamente, y en el transcurso de la misma se han definido unificadores que van precisando el valor pedido de `W`. El resultado obtenido es:

```
W = [a.[b.[]]]
```

En el desarrollo de la ejecución no han habido, en este caso, puntos de decisión. Si bien el programa tiene tiene dos procedimientos con el mismo nombre, `CONCAT`, ha sido el proceso de unificación de parámetros el que ha descartado alternativas durante la ejecución.

4.4. Intérprete no determinista

A partir de lo planteado en la sección anterior, puede concluirse que un programa en lógica implica un no determinismo en su evaluación, que se evidencia por los puntos de decisión que aparecen en el transcurso de una ejecución. Parece natural que para explicar el comportamiento de un evaluador de programas en lógica, se utilice un lenguaje que tenga primitivas de no determinismo. En esta sección se presentarán dichas primitivas, y se describirá una máquina no determinista como semántica operacional para describir la evaluación de programas en lógica.

Sea P el conjunto de procedimientos que conforman un programa. Se dispone además de las funciones:

```
cabeza : procedimientos -> predicados
```

cuya semántica es que dado un procedimiento devuelve el predicado del encabezamiento del mismo, y PÁG. 68

```
cuerpo : procedimientos -> conjunto de predicados
```

que devuelve los predicados del cuerpo del procedimiento que recibe como argumento.

Sea Q el conjunto de invocaciones de procedimientos. Inicialmente contendrá el (los) predicado(s) dado(s) por el usuario al solicitar una ejecución.

Se dispone de un procedimiento:

```
unif ( P1, P2, éxito, θ )
```

que intenta resolver los predicados `P1` y `P2`. Si lo consigue devuelve en θ el unificador más general, y asigna el valor verdadero a `éxito`. Si la unificación no es posible, `éxito` toma el valor falso.

Finalmente se tiene la función `sust`:

```
sust : conjunto de predicados  × unificador ->conjunto de predicados
```

cuya función es aplicar el unificador al conjunto de predicados y devolver el conjunto así obtenido.

El constructor no determinista que se usará es:

```
elegir : conjunto => elemento del conjunto
```

y cuya función es escoger arbitrariamente un elemento de un conjunto. Se utiliza el símbolo «`=>`» para expresar el no determinismo de la «función».

Una forma de entender el comportamiento de `elegir` es escribir su significado mediante primitivas más básicas de no determinismo. Supóngase que a los conjuntos los representamos como sucesiones, y se dispone del constructor de programas «`O`». Si `P1` y `P2` son dos programas, entonces `P1 O P2` es un nuevo programa cuya ejecución será la ejecución de `P1`, o la ejecución de `P2`, siendo la decisión entre ambas alternativas arbitraria. Con estos elementos se puede definir la función `elegir`:

Pág. 69

```
elegir ( sucesión ) :=
        si la sucesión tiene un solo elemento
           entonces devolver ese elemento
        sino ( devolver primer elemento de la sucesión
                 O
               elegir ( sucesión sin el primer elemento )
             )
```

Con todo lo definido, es posible presentar la máquina no determinista para la evaluación de programas en lógica.

ITUIR LOS @ POR θ

Máquina no determinista ***

```
sea Q un conjunto de predicados,
    P un conjunto de procedimientos;
sea q un predicado, p un procedimiento,
    éxito un lógico, @ una sustitución;

comienzo
        Q := { predicado dado por el usuario };

        mientras Q no sea vacío hacer
                q := elegir(Q);
                p := elegir(P);
                unif( cabeza(p), q, éxito, @ );
                si éxito
                        entonces Q := sust( Q-q+cuerpo(p), @ );
        fin
fin
```

4.5. Estrategias de evaluación

Una descripción no determinista como la presentada, representa una especificación formal de la evaluación de programas en lógica, interpretada como invocaciones de procedimientos. PÁG. 70

Cada punto de no determinismo de la descripción determina un conjunto de posibles caminos de computación. Una implementación de programación en lógica, que será necesariamente determinista, deberá recorrer cada uno de esos caminos, y realizar las evaluaciones corresspondientes. De esta forma se podrá encontrar la o las soluciones a la invocación del programa.

El siguiente ejemplo, extraído de[Hog84] ilustra lo anterior. Sea el programa:

```
CONSEC ( X, Y, [X.[Y.Z]] ) <- .                  (1)
CONSEC ( X, Y, [W.Z] ) <- CONSEC ( X, Y, Z ).    (2)
```

El primero y el segundo de los argumentos, son elementos que deben ser consecutivos en la lista que aparece en el tercero. Como se trata de listas se puede utilizar la notación para las mismas que ya fue presentada y cuya estructura [1.[2.[5.[]]]] se escribe (1 2 5).

Sea la invocación:

```
<- CONSEC ( 2, Z, (2 3 2 4) ).
```

En este caso la invocación puede activar tanto el procedimiento (1) como el (2). Si se escoge el primero, se inicia una computación que culmina con un resultado de:

```
Z := 3
```

Si por el contrario la decisión hubiera recaído en el procedimiento (2), entonces otra computación se hubiera realizado y el nuevo conjunto de objetivos
PÁG. 71
sería:

$$Q = \{\texttt{CONSEC (2, Z, (3 2 4))}\}$$

Sólo el procedimiento (2) satisface este objetivo, por lo que se obtiene:

$$Q = \{\texttt{CONSEC (2, Z, (2 4))}\}$$

Se presenta una nueva disyuntiva ya que ambos procedimientos pueden ser activados. Si se opta por el (1), la computación finaliza satisfactoriamente dando como resultado:

```
Z := 4
```

En cambio si se escoge el procedimiento (2), la computación sería:

$$\begin{aligned} Q &= \{\texttt{CONSEC (2, Z, (4))}\} \\ Q &= \{\texttt{CONSEC (2, Z, ())}\} \end{aligned}$$

El último predicado no representa ninguna invocación posible. La misma no puede ser satisfecha por los procedimientos del programa. Se ha llegado a una situación de terminación no satisfactoria sin resultados:

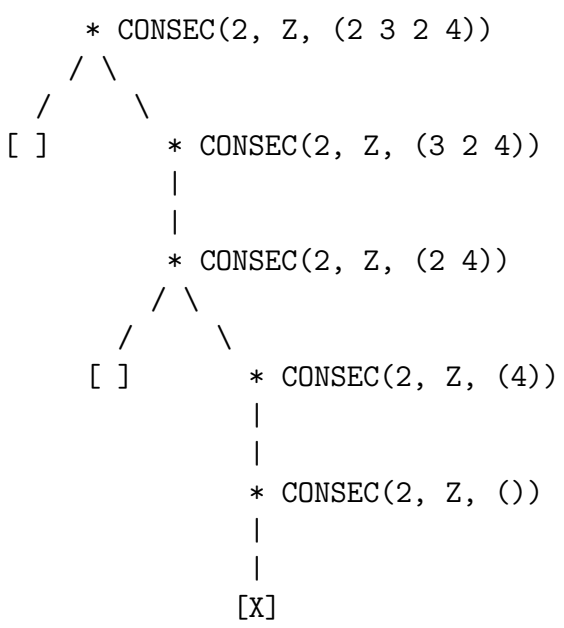
```
[ ]
```

En el ejemplo se evidencian los distintos caminos de computación posibles, y la forma en que pueden obtenerse las diferentes soluciones de la ejecución de un programa.

Al conjunto de todos los caminos de evaluación se le llama espacio de computación. Un intérprete de programas en lógica debe considerar dicho espacio, y recorrerlo mediante una estrategia definida. Una forma de representar ese espacio es mediante un árbol cuyos nodos son los valores del conjunto Q de invocaciones a ser ejecutadas. Los descendientes de un nodo representan las distintas alternativas de evaluación del mismo. Las hojas del árbol que

correspondan al valor *vacío* para Q significarán una culminación exitosa de una evaluación, y se representarán por «[ ]». En caso que se termine sin posibilidad de resolver una invocación, o sea en la culminación insatisfactoria, se denotará con«[X]».

Si se considera el ejemplo anterior y la invocación considerada, el espacio de computación puede representarse de la siguiente manera:



Todas las computaciones que han sido presentadas son finitas en el sentido que culminan, satisfactoriamente o no. Pueden existir, sin embargo, casos de caminos de computación infinitos, como sería el caso si al programa del ejemplo se lo invocara con:

```
<- CONSEC ( 1, 2, Z ).
```

Un espacio con computaciones infinitas implica un árbol de profundidad infinita.

En el ejemplo presentado cada nodo contenía un solo predicado. Sin embargo, en el caso general pueden ser más de uno, y entonces la decisión a tomar no es solamente cuál procedimiento activar, sino también cuál invocación elegir. Este hecho no hace sino poner en evidencia los dos puntos de no determinismo de la máquina presentada en la sección anterior.

Un intérprete de programación en lógica tendrá definido internamente el orden en que irá recorriendo el espacio de computación, es decir el orden en que irá escogiendo invocaciones y procedimientos. Esto implica que una

vez escogido un camino, será necesario retornar al punto de la decisión para buscar una nueva alternativa. Cabe señalar que todos los caminos deben ser iniciados con el mismo estado de las variables. El retorno no solamente necesita de un posicionamiento de control en el punto de decisión, sino que además se deben restaurar los valores que tenían las variables al iniciar la alternativa anterior. Al mecanismo para realizar tal proceso se le denomina *backtracking*.

Las implementaciones de PROLOG y de sus lenguajes derivados poseen una estrategia generalmente aceptada de decisión, que se basa en el orden establecido por el usuario al escribir sus programas.

Elección de invocaciones. Se considera al conjunto Q de invocaciones como una secuencia y el mecanismo de elección implica el recorrido de la misma. Esta estrategia produce un tipo de recorrido del árbol que se conoce como *depth first* (primero en profundidad). En cada nodo se analiza primero, totalmente, el subárbol engendrado por un descendiente antes de considerar a los hermanos del mismo. En conocimiento de este ordenamiento, el usuario puede guiar la evaluación de sus programas, determinando el orden con que escribe los predicados en el cuerpo de los procedimientos.

PÁG. 74

Elección de procedimientos. Se utiliza el orden en que el usuario escribió las definiciones de procedimientos. En este sentido se sigue la tradición de los lenguajes imperativos. Bajo esta estrategia la evaluación del programa `CONSEC`, visto anteriormente, recorrería el árbol de izquierda a derecha, y las soluciones se las daría al usuario en el orden `Z := 3` y `Z := 4`. Si en cambio se intercambiaran las definiciones de los procedimientos, la misma invocación produciría un recorrido del mismo árbol de derecha a izquierda, encontrando primero la culminación insatisfactoria para luego dar las soluciones en el orden `Z := 4` y `Z := 3`.

Con el objeto de ofrecer al usuario la posibilidad de alterar el árbol del espacio de computación, y así obtener mayor eficiencia en la ejecución de sus programas, el lenguaje PROLOG dispone de un operador de control explícito. Normalmente se le escribee con el símbolo «`/`» y se le denomina **cut**. Sintácticamente cumple la función de un predicado sin parámetros que sólo puede aparecer en el cuerpo de un predicado. Su función es *podar* el árbol de computación, eliminando todas las alternativas pendientes desde que el procedimiento que contiene al «`/`» fue invocado. Si la primera regla del programa `CONSEC` se escribiera:

```
CONSEC ( X, Y, [X.[Y.Z]] ) <- / .
```

la ejecución terminaría luego de encontrar la solución `Z := 3`.

El operador «`/`» no tiene interpretación lógica y su explicación algorítmica no es simple. Es por ello que a su utilización en programación lógica se la compara con el uso del `GO TO` en la programación tradicional. Es indudable que permite construir programas más eficientes (como el `GO TO`), pero oscurece la comprensión de los programas y es fuente de errores durante la modificación de los mismos.

4.6. Tratamiento de la negación

Durante el diseño de un programa en lógica, algunas veces resulta necesario expresar la negación de algunos hechos. A pesar de que la negación existe en la lógica tradicional, ella no aparece en la programación en lógica. Esta carencia puede ser remediada de diferentes formas. PÁG. 75

Dado un predicado `P` puede definirse un nuevo predicado, llamado por ejemplo `NO_P`, el que deberá ser verdadero cada vez que `P` sea falso. Para construir el nuevo predicado se utilizarán predicados predefinidos del sistema, que expresan la negación a un nivel básico, como por ejemplo la desigualdad de átomos.

Otra alternativa surge de la utilización del operador «`/`» interpretando un fallo para derivar un predicado, como la negación del mismo. Así por ejemplo, si se desea definir un predicado para negar a `P`, y se dispone de un predicado predefinido `FAIL` que fracasa siempre, se puede escribir:

```
NO_P <- P , / , FAIL.
NO_P <- .
```

Si la activación de `P` tiene éxito, entonces se *atraviesa* el operador «`/`», y la invocación a `FAIL` produce un fallo de `NO_P`. Por eel contrario, si la activación de `P` falla, entonces se activa el segundo procedimiento, que es una aserción, y `NO_P` tiene éxito. Este mecanismo, si bien puede ser de utilidad, no representa la implementación general de la negación. Si los predicados tienen variables, el fracaso de `NO_P` en el primer caso se produce para una cierta instanciación de las variables, mientras que el éxito de `NO_P` resultante de utilizar el segundo procedimiento, se obtiene para cualquier valor de las mismas.

Con el objeto de clarificar el uso de la negación, ciertos lenguajes como IC-PROLOG, proveen la facilidad de un operaador de *cuasi-negación* que se implementa interpretando la negación como un fallo, de acuerdo a la llamada asunción del mundo cerrado. PÁG. 76

Problemas

1. Evaluar `PREFIX([a,b,c], X)` con el programa construído para el problema 3.2***.

REFERENCIA AL PROBLEMA

2. Definir un predicado `HANOI(N, A, B, C, Movim)` que resuelva el problema de las torres de Hanoi, con `N` discos, pasándolos de `A` a `B` usando `C` como auxiliar y siendo `Movim` la secuencia de movimientos realizada.

3. Escribir un programa para resolver el siguiente *rompecabezas lógico*:

   - Hay cinco casas, habitadas por hombres de diferentes nacionalidades, con diferentes mascotas, que suelen tomar diferentes bebidas y fuman distintas marcas de cigarrillos.
   - El inglés vive en la casa roja.
   - El español es dueño del perro.
   - En la casa verde se toma café.
   - El ucraniano toma té.
   - La casa verde está a la derecha de la casa marfil.
   - El que fuma Winston tiene ardillas.
   - En la casa amarilla se fuma Camel.
   - En la casa del medio se toma leche.
   - El noruego vive en la casa que está más a la izquierda.
   - El que fuma Chesterfield vive en la casa de al lado del dueño del zorro.
   - Al lado de donde está el caballo se fuma Camel.
   - El que fuma Lucky Strike toma jugo de naranja.
   - El japonés fuma Parliament.
   - El noruego vive al lado de la casa azul.

   ¿Quién es dueño de la cebra? ¿Quién toma agua?

PÁG. 77

4. Escribir un programa lógico para resolver el problema de las *ocho reinas*. Esto es determinar las formas en que pueden colocarse ocho reinas en un tablero de ajedrez de modo que no se amenacen mutuamente.

5. Escribir un programa en lógica para resolver el siguiente problema. Tres misioneros y tres caníbales están en la orilla izquierda de un río. Hay un bote para cruzar el río, con capacidad a lo sumo para dos personas. Todos deben cruzar el río. Si en algún momento en alguna orilla quedan más misioneros que caníbales, éstos serán convertidos por aquellos. Encontrar modos de transportar los misioneros y caníbales a través del río sin exponer ningún caníbal al peligro de la conversión.

Parte II

Programación funcional

Version: $Id: plyf.tex,v 1.3 2003/08/13 04:28:26 cballard Exp $

Capítulo 5

Un caso de diseño con lenguaje funcional

Pág. 81

La idea central de este capítulo es introducir algunas características de la programación funcional, vinculando ésta a las técnicas de diseño modular. El punto tiene interés por cuanto luego se verá que las propiedades más importantes de los lenguajes funcionales están relacionadas con la facilidad que proveen para expresar la aplicación de dichas técnicas.

Se presenta un caso de estudio (tomado de [Hen80] y un lenguaje funcional, (inspirado en un similar usado en [Gla84] que no se define formalmente. A pesar de este hecho, puede verse como su estructura insinúa las características esenciales de los lenguajes funcionales las que se tratarán, con mayor rigor, a partir del capítulo 7.

Version: $Id: plyf.tex,v 1.3 2003/08/13 04:28:26 cballard Exp $

5.1. Funciones, problemas y diseño modular

PÁG. 82

El primer presupuesto del paradigma funcional es caracterizar un problema como una función entre conjuntos, a ser definida.

Tal función podría definirse por extensión, si todos los pares de correspondientes fueran conocidos, o bien, en el caso más general, por una expresión que, convenientemente manipulada, determine la imagen de cada objeto del dominio.

Un ejemplo, de aspecto seguramente familiar, de una tal definición es:

$$f(x) = 2 * x + 1$$

En definiciones como la anterior, se llama a x un argumento nominal de f, y será, en todos los casos, una variable que denota un valor genérico del dominio de la función.

A la derecha del símbolo de igualdad hay una expresión que denota la imagen del valor del argumento nominal en la función que está siendo definida. Es útil analizar esta expresión con detalle:

En ella aparecen los símbolos $*$ y $+$, que (es de suponer) denotan las funciones producto y suma, aplicadas a constantes y al argumento x. De este modo se indica que, para calcular la imagen de x en f deben usarse otras funciones, aplicándolas a x y a ciertos valores convenientes. De modo que, en definitiva, la definición de f se muestra como una combinación adecuada de otras funciones.

En general, podría requerirse que las funciones así usadas tuvieran que ser, a su vez, definidas. Esto conduciría, en el caso anterior, a complementar la definición dada con un par de expresiones, de la misma forma, para $*$ y $+$, en las cuales habrán de introducirse, a su vez, nuevas funciones. De esta manera:

(a). La estructura de definiciones sucesivas se organiza, a grandes rasgos, en forma jerárquica, según como cada función aparece en la definición de otra «más compleja».

PÁG. 83

(b). En cada definición, las funciones utilizadas a la derecha de la igualdad se referencian simplemente por un nombre. El significado preciso de éste puede conocerse si se lo reemplaza por su propia definición.

(c). Debe convenirse un modo de terminar esta sucesión de definiciones. La convención consiste en establecer que ciertos nombres no necesitan ser definidos, aceptándose que tienen un significado «sobreentendido».

Este esquema evoca uno de los aspectos más importantes de las técnicas de diseño modular, en particular, la idea de descomposición de un problema complejo en componentes más simples (módulos) que se combinan convenientemente.

Cada uno de ellos representa, nuevamente, un problema que, o bien tiene una solución sobreentendida, o bien es, a su vez, descompuesto de igual forma.

Como en el caso de $*$ y $+$, los módulos pueden ser referenciados apenas por un nombre. Esta facilidad es importante, puesto que hace posible que el empleo de un módulo no involucre el conocimiento de su estructura interna, que podría ser compleja, sino simplemente su afecto. Luego, tal estructura puede desarrollarse por separado, asociándola al nombre utilizado a través de otra definición.

Algunos de estos nombres no necesitan tener sus propias definiciones, conviniendose que sus significados son sobreentendidos: corresponderán a modulos *primitivos*. Los que no sean primitivos, se llamarán *abstractos*.

Nótese que unos y otros no se distinguen desde el punto de vista de su forma de uso. En ambos casos, ésta consiste, en principio, en citar su nombre.

De hecho, los módulos abstractos pueden pensarse, en virtud de esta característica, como «primitivas potentes». La idea caracteriza el diseño por refinamientos sucesivos:

La estructura de un problema complejo puede diseñarse en términos de componentes pensados como «ya resueltos», que además, pueden efectivamente usarse como tales. Tales componentes se definirán en un siguiente nivel de detalle, si no son primitivos.

Sin embargo, la elección de estos componentes para un problema arbitrario dado no es, en modo alguno, trivial. El enfoque anterior se complementa, en este sentido, con la idea de elegir la descomposición de modo que puedan emplearse, en todo lo posible, soluciones ya desarrolladas, con un mínimo costo de adaptación. PÁG. 84

En términos de los lenguajes de programación, ésto hace interesante el estudio de las facilidades que ellos proveen para definir y emplear módulos de aplicación general. Como se verá más adelante, dichas facilidades constituyen uno de los principales atributos de los lenguajes funcionales.

Algunas de estas ideas comenzarán a ser aplicadas enseguida.

5.2. El caso de estudio

Se trata de un problema clásico de las ciencias físicas: el análisis dimensional de fórmulas.

En estas ciencias, las fórmulas usadas (para denotar ciertos fenómenos)

son formadas por operadores aplicados a variables o constantes, las cuales tienen asociadas dimensiones. Estas son, a su vez, expresiones formadas a partir de ciertas magnitudes fundamentales, por ejemplo: masa (M), longitud (L) y tiempo (T). En este caso, las dimensiones asociadas a las variables o constantes de estas fórmulas serán expresiones definidas sobre las constantes M, L, T. Algunos ejemplos se muestran a continuación:

Una variable o constante que representa:

velocidad tiene dimensión LT^{-1}

por ser cociente de una cantidad de longitud por una cantidad de tiempo.

área tiene dimensión L^2

por ser producto de dos longitudes.

PÁG. 85

fuerza tiene dimensión MLT

por ser producto de una cantidad de masa por otra de aceleración que, a su vez, es cociente entre una cantidad de velocidad y una cantidad de tiempo.

Ahora, a partir de las dimensiones de las variables y constantes, pueden deducirse las dimensiones asociadas a las fórmulas definidas sobre ellas.

Para esto es necesario establecer:

I. cuáles son los operadores que pueden ser empleados en la construcción de fórmulas.

II. para cada operador, cuál es la dimensión del resultado, en función de las dimensiones de los operandos.

A efectos de I se considerarán solamente los operadores aritméticos usuales: +, -, *, /.

Para establecer II es necesario adoptar algunos supuestos:

- Se asumirá que existen tres magnitudes fundamentales y que
- Las dimensiones de las constantes y variables tienen, en todos los casos, la forma de un producto de esas magnitudes, cada una afectada por un exponente entero.

Así, si las magnitudes fundamentales fueran masa, longitud y tiempo, una dimensión genérica sería:

mlt
MLT con m, l y t enteros

y para una cantidad de velocidad, en particular, valdría:

$$m = 0,\ l = 1\ ,\ t = -1$$

En este punto, el lector puede ensayar una primera versión de las reglas que caracterizan la dimensión del resultado de cada operador artimético a partir de las dimensiones de sus operandos.

5.3. Desarrollo del diseño

Como se ha dicho, el enfoque funcional caracteriza un problema como una función a ser definida. Desde el punto de vista de la programación interesan, fundamentalmente, las definiciones de funciones que son dadas en la forma de reglas que definen la construcción del correspondiente de cualquier objeto del dominio. PÁG. 86

Una forma (en principio, simplificada) de una tal definición es:

«**f**» es la función que a cada valor de **x** hace corresponder «**exp**»

donde:

«**f**» es el nombre de la función a definir,

x es una variable que representa un objeto genérico del dominio,

«**exp**» es la expresión que define el correspondiente de **x** en **f**. Como ya se vió, consistirá en general, de aplicaciones de otras funciones.

Se usan las comillas para denotar objetos cuya forma concreta se determinará como consecuencia de decisiones tomadas durante el proceso de diseño. Este estará terminado cuando todos ellos hayan sido sustituidos por construcciones de significado convenido.

La frase usada: «es la función que a cada valor de x hace corresponder» cumple dos roles: por una parte asocia el nombre de la función a su definición. Por otra, indica cuál es el argumento nominal de esta definición. Para abreviar, se usará en su lugar: **:- (x)**, donde la asociación *nombre - definición*

es representada por **:-** y la indicación de argumento nominal, por su nombre entre paréntesis.

Puede parecer que la notación es extraña y, más aún, invertida en relación a la usanza corriente. Sin embargo, se verá luego que es más razonable, incluso, que la tenida por normal.

PÁG. 87

El caso más general parecería exigir la posibilidad de definir funciones de más de un argumento. De otro modo, funciones elementales, como la simple suma de enteros, no podrían ser expresadas. Aunque más adelante esta tesis será rebatida, se aceptará de momento y, de hecho, será aplicada al caso de estudio, como surge de las siguientes consideraciones:

El argumento obvio de la función a definir es una fórmula, cuya dimensión debe ser construída como resultado. Sin embargo, éste depende también de las dimensiones de los componentes más primitivos de la fórmula, es decir, de las constantes y variables a partir de las cuales ésta se construye.

Por explicación adicional de lo anterior véase que

$$ma - f$$

por ejemplo, tendrá distintas dimensiones según si m representa una cantidad de masa o de velocidad, e idénticamente ocurre para las demás variables.

De modo que las dimensiones de las variables y constantes intervinientes en la fórmula deben constituir un segundo argumento de esta función, y así aparecerá en su definición, que será de la forma:

dimension :- (f,datom) «exp»

donde

f representa una fórmula,

datom representa la asociación de dimensiones a los componentes atómicos (variables, constantes) de **f**,

«**exp**» es una expresión que define el correspondiente de todo par (f, datom) posible en la función dimension.

Es ya inevitable iniciar la discusión acerca de la forma de «**exp**». Para ésto es necesario definir el conjunto de fórmulas con precisión. Una técnica general que puede usarse con este próposito es la de inducción. Como se verá, constituye una poderosa herramienta de diseño, que será aplicada reiteradamente. Para el caso de las fórmulas pueden usarse las siguientes ideas:

PÁG. 88

(Base) Las fórmulas primitivas son las que consisten de apenas una variable o una constante. Se reunirán éstas en una clase, que llamaremos de fórmulas atómicas.

(Inducción) El resto de las fórmulas puede construirse usando cuatro funciones constructoras: *Suma, Resta, Producto, Cociente*. La idea será denotar dichas fórmulas como expresiones que sean aplicaciones de estas funciones.

Entonces, si f y g son fórmulas ya construidas,

```
  Suma     (f, g)
  Resta    (f, g)
  Producto (f, g)
y Cociente (f, g)
```

son expresiones cada una de las cuales denota una nueva fórmula, respectivamente la suma, resta, producto y cociente de las denotadas por las f y g originales.

Se exigen propiedades de los constructores:

Para cualesquiera constructores K y K' y fórmulas $f1$, $f2$, $g1$, $g2$ vale:

- Si f es una fórmula atómica, entonces $f \neq K(f1, g1)$
- $K(f1, g1) = K'(f2, g2)$ implica K idéntico a K', $f1 = f2$ y $g1 = g2$

Es decir, las fórmulas construidas por aplicación de constructores son todas diferentes entre sí y diferentes de las atómicas.

En adelante se llamará fórmula a toda expresión formada de acuerdo con las reglas anteriores. Estas permiten establecer que una fórmula cumple una (y sólo una) de las condiciones siguientes:

- es atómica (variable o constante)
- es Suma aplicada a dos fórmulas
- es Resta aplicada a dos fórmulas
- es Producto aplicado a dos fórmulas
- es Cociente aplicado a dos fórmulas

De este modo, se clasifican las fórmulas en cinco clases, según como han sido construídas. PÁG. 89

Ahora bien, volviendo al problema, dada una fórmula genérica deberá ser posible distinguir entre sus cinco posibles formas y definir las respectivas

dimensiones asociadas por separado. En base a esta idea puede resolverse al problema en cuestión.

Algunas funciones, con significado conveniente, deben ser usadas para expresar lo anterior. En principio, puede pensarse en las siguientes:

I. Una función asociada a cada clase de las arriba definidas, teniendo como dominio el conjunto de fórmulas y como codominio el conjunto de valores de verdad (Verdadero, Falso). Que una tal función aplicada a una fórmula f dé como resultado *Verdadero* significará que f está en la clase asociada a dicha función.

II. Una función cuyo resultado pueda escogerse de entre varios en base a la verificación de cierta condición.

El candidato para II está sugerido en una construcción conocida desde los lenguajes imperativos; se usará:

```
if b then v1 else v2
```

con el significado habitual. Es interesante enfatizar que esta construcción denota una función aunque esta idea sea ajena a su pariente imperativo. El carácter funcional podría quedar más de manifiesto si se usara la notación *prefija*:

```
if-then-else (b, v1, v2)
```

más familiar cuando se habla de funciones, en lugar de la empleada, llamada *infija*.

PÁG. 90

Cualquiera sea la notación usada, el significado de if-then-else se establece como sigue:

- b puede tomar uno de los dos valores: Verdadero o Falso.
- si b vale Verdadero, entonces if-then-else toma el valor de $v1$ y, en caso contrario, el de $v2$.

A poco de razonar sobre el problema puede verse que los requisitos exigidos en I pueden relajarse. En efecto, es necesario usar funciones de la forma citada para distinguir todos los posibles comportamientos de una fórmula genérica en relación a su dimensión asociada, pero también es cierto que los comportamientos diferentes no coinciden con las cinco clases de construcciones definidas. El lector que haya pensado en la relación entre los operadores

aritméticos y las dimensiones de sus resultados habrá detectado que la suma y la resta se comportan idénticamente (si no fue así, he aquí una nueva oportunidad para pensarlo).

De modo, pues, que sólo cuatro casos deben ser distinguidos. Uno de ellos puede decidirse por el simple descarte de los otros, si se asume que el argumento f es siempre una fórmula correctamente formada.

Estas consideraciones conducen a la siguiente versión de la función:

```
dimension :- (f,datom)

          if Atómica? (f)       then «exp_atom»
     else if Suma-o-Resta? (f) then «exp_+_-»
     else if Producto? (f)     then «exp_*»
     else                           «exp_/»
```

El proceso de descomposición comienza a insinuarse. La función dimensión se expresa en términos de otras funciones, en el caso:

if-then-else, Atómica?, Suma-o-Resta?, Producto? y de las que se usen para especificar las aún desconocidas expresiones denotadas entre comillas, que serán precisamente las que construyan los resultados para cada caso.

Nótese que, a la manera de lo anunciado en la sección anterior, las funciones son usadas sólo citando su nombre y no su propia estructura, como «módulos abstractos» de significado convenido. Obviamente deberán continuar el diseño de dimensión. PÁG. 91

Este debe ocuparse ahora de las «expresiones» todavía no especificadas.

Entonces:

I. si la fórmula es atómica, el argumento **datom** define su dimensión asociada, como ya ha sido dicho; **datom** mismo puede verse, de hecho, como una función (dada) cuyo dominio es el conjunto de las fómulas atómicas y cuyo codominio es el conjunto de las dimensiones posibles. No importando como estén definidas las dimensiones, lo anterior establece que si **f** es atómica, entonces **datom (f)** es su dimensión. Esto resuelve la especificación de «exp_atom».
   Nótese el uso de una función como argumento de otra. Esto es permitido ya en algunos lenguajes imperativos, pero como se verá, alcanza toda su generalidad dentro del esquema funcional: las funciones podrán ser, sin restricciones, argumentos y resultados de otras funciones.

II. si la fórmula no es atómica, entonces es el resultado de aplicar un constructor a dos operandos que son fórmulas y su dimensión puede expresarse como la aplicación de una transformación conveniente a las dimensiones de los operandos, siguiendo un esquema recursivo.
Para ésto será necesario disponer de funciones que, aplicadas a una fórmula no atómica, devuelvan las fómulas operandos a partir de las cuales aquella fue construida. Tales funciones son, usualmente, denominadas **selectores**. En el caso, los selectores de fórmulas serán las funciones *PrimerOperando* y *SegundoOperando*.
Hay tres casos interesantes de fórmulas no atómicas, como ya se ha visto. Estas definen tres formas de combinar las dimensiones de los operandos para dar la de la fómula compuesta.
Se llamarán

```
DimAd   (adición de dimensiones),
DimProd (producto de dimensiones) y
DimCoc  (cociente de dimensiones)
```

PÁG. 92

Lo analizado justifica la nueva versión:

```
dimension :- (f, datom)

        if Atómica? (f)      then datom (f)

    else if Suma-o-Resta? (f) then
            DimAd ( dimension (PrimerOperando  (f), datom),
                    dimension (segundoOperando (f), datom) )

    else if Producto? (f)     then
            DimProd ( dimension (PrimerOperando  (f), datom),
                      dimension (segundoOperando (f), datom) )

    else    DimCoc  ( dimension (PrimerOperando  (f), datom),
                      dimension (segundoOperando (f), datom) )
```

Esto completa un nivel del diseño. En efecto, todas las construcciones «entrecomilladas» han sido sustituídas por expresiones cuyo significado, a este nivel, se da por conocido, siguiendo la idea de la sección 5.1.

Estas expresiones son siempre de la forma:

```
«nombre de función» («lista de argumentos»)
```

denotando aplicaciones de funciones sobre argumentos convenientes.

De este modo, el significado de la función dimensión, se resuelve en términos de los significados de otras funciones, cada uno de los cuales debería, en principio, ser definido en niveles inferiores más detallados.

Se dará un nuevo paso en esa dirección, definiendo las funciones de adición, producto y cociente de dimensiones.

Estas son funciones sobre el conjunto de las dimensiones (más precisamente sobre el conjunto de pares de dimensiones). Como en el caso de las fórmulas, debe definirse este conjunto.

En la sección 5.2 se sugirió caracterizar las dimensiones por el producto de PÁG. 93
tres mangitudes fundamentales afectadas por exponentes enteros. Con esta aproximación, construir una dimensión requiere proveer los valores de esos exponentes, es decir, una terna de dimensiones, que se llamará **DimCons**. El conjunto de todas las dimensiones es el conjunto de las aplicaciones de **DimCons** a toda posible terna de enteros.

Inversamente, dada una dimensión genérica **b**, será necesario obtener los valores de sus exponentes. Para esto se usarán tres funciones: $PrimerExp$, $SegundoExp$, $TercerExp$, a la manera de **selectores**.

También se usará una función que detecta si dos dimensiones dadas son idénticas. Su codominio es, obviamente, (Verdadero, Falso) y su nombre **DimEq**. Su comportamiento, el esperable.

A partir de estos supuestos, se desarrollan las tres funciones requeridas:

I. La adición de dos dimensiones está definida sólo para el caso en que éstas sean idénticas, y el resultado es la misma dimensión.
   La restricción impuesta es la manera de expresar el hecho de que no se puedan sumar (ni restar) velocidades con fuerzas, naranjas con libros, o cualquier otro par de cantidades que no hayan sido expresadas idénticamente en relación a las magnitudes fundamentales que se manejen.
   En el lenguaje de funciones:

```
DimAd :- (d1, d2)
            if DimEq (d1, d2)
               then d1
               else ERROR
```

   ERROR juega aquí como un valor especial distinguido, empleado para denotar casos de excepción como el anotado. En particular representa un

caso de dimensión especial (la de las fómulas mal formadas, que intentan combinar operandos «incompatibles»).

Pág. 94

II. El producto de dos dimensiones es, en todos los casos, otra dimensión cuyos exponentes son, cada uno, la suma de los respectivos exponentes de los operandos:

```
DimProd :- (d1, d2)
               DimCons
                   (PrimerExp  (d1) + PrimerExp  (d2),
                    SegundoExp (d1) + SegundoExp (d2),
                    TercerExp  (d1) + TercerExp  (d2))
```

III. El cociente de dos dimensiones es otra dimensión, cuyos exponentes son, cada uno, la diferencia de los respectivos exponentes de los operandos:

```
DimProd :- (d1, d2)
               DimCons
                   (PrimerExp  (d1) - PrimerExp  (d2),
                    SegundoExp (d1) - SegundoExp (d2),
                    TercerExp  (d1) - TercerExp  (d2))
```

Aquí han aparecido ya funciones que el lector puede, sin duda, aceptar como «primitivas»: son las denotadas por los conocidos símbolos + y − y que representan la suma y resta de enteros. Seguramente no extrañará que no se den definiciones sobre dimensiones, por ejemplo.

A la vez, durante el desarrollo se ha hecho referencia a otras funciones que parecerían merecer una definición detallada, como las llamadas **Atómica?**, **Suma-o-Resta?**, etc.

En definitiva, debe convenirse acerca de cuál es el conjunto de funciones primitivas. Tal convención debe establecerse en forma precisa y definitiva para todos los problemas que quieran ser resueltos y determina el nivel de explicación al que debe llegarse, en un lenguaje dado, para que una solución se considere suficientemente descripta.

En este caso, el lenguaje no ha sido definido con precisión de antemano, lo que autoriza a detener el proceso de diseño en el momento en que las explicaciones dadas se consideren suficientes para ilustrar la técnica de descomposición modular. Ese momento ya ha llegado.

Pág. 95

En efecto, se ha mostrado la utilización del concepto de «módulo abstracto» en su forma funcional y definiciones de algunos de éstos han sido

desarrolladas con independencia de la del módulo de mayor jerarquía que los utilizaba. En capítulos siguientes habrá ocasión de someterse a reglas rigurosas que obliguen a llegar a niveles de refinamiento precisamente establecidos.

También se ha mostrado el empleo de algunas técnicas de diseño generales, como las de definiciones inductivas, y algunas características novedosas de los lenguajes funcionales han comenzado a insinuarse, en especial la posibilidad de tratar a las funciones como argumentos y resultados de otras funciones.

Estos puntos serán especialmente considerados en el siguiente capítulo.

Problemas.

5.1 Escribir la definición de la función que calcula el máximo común divisor de dos naturales dados.

5.2 Usando selectores y funciones auxiliares apropiadas (que no se definirán) escribir la definición de una función que sume todos los elementos de un árbol binario.

5.3 Escribir la definición de una función que, aplicada a una lista de enteros, devuelve un árbol binario «ordenado».

5.4 Escribir definiciones de funciones que, aplicada a un árbol binario, devuelvan la lista de sus elementos, en:

a) pre orden

b) orden central

c) post orden

5.5 Usando 3 y 4 escribir la definición de una función que ordene una lista dada.

Capítulo 6

Lenguajes funcionales

Pág. 97

La idea central del capítulo es desarrollar, a partir del lenguaje usado en el caso de estudio, una estructura sintáctica y una interpretación informal de los lenguajes funcionales, identificando las facilidades de modularización que proveen.

Se presta especial atención a aquellas que no existen en los tradicionales lenguajes imperativos, (en particular, el concepto de función de orden superior), señalándose que constituyen la principal diferencia entre ambos esquemas y, por lo tanto, un aporte relevante del paradigma funcional al desarrollo de los lenguajes de programación.

En este análisis se siguen ideas de [**?**], [**?**] y [**?**].

Version: $Id: plyf.tex,v 1.3 2003/08/13 04:28:26 cballard Exp $

6.1. La estructura de los lenguajes funcionales

PÁG. 98

Véase la expresión que define la función `dimension`:

```
dimension :- (f, datom)

  if Atómica?(f)   then datom(f)
  else if Suma-o-Resta?(f) then
    DimAd( dimension( PrimerOperando(f), datom ),
           dimension( SegundoOperando(f), datom ) )
  else if Producto?(f) then
    DimProd( dimension( PrimerOperando(f), datom ),
             dimension( SegundoOperando(f), datom ) )
  else
    DimCoc( dimension( PrimerOperando(f), datom ),
            dimension( SegundoOperando(f), datom ) )
```

Interesa examinar la forma de las expresiones que ocurren a la derecha del símbolo `:-`. Véase que:

- La expresión principal denota la definición de una función y tiene la siguiente estructura:

  ```
  ( "variable", "variable", ... , "variable" ) "expresión"
  ```

  donde las variables son los argumentos nominales y la expresión definen la construcción de la imagen de éstas en la función.

- Aparecen aplicaciones de funciones definidas, en el ejemplo, bajo dos formas:

  ```
  "nombre" ("expresión", "expresión", ... , "expresión" )
  "variable" ("expresión", "expresión", ... , "expresión" )
  ```

  donde las expresiones entre paréntesis se llamarán argumentos efectivos de la aplicación y se hace notar que pueden ser, como en algunos casos del ejemplo, a su vez expresiones complejas y no necesariamente sólo variables.

Lo anterior sugiere cierta similitud en el comportamiento de los nombres y las variables. En particular, ambos pueden denotar funciones que son aplicables a un conjunto de argumentos.

Esta similitud es aún más amplia, puesto que ambos pueden denotar también resultados de aplicaciones de funciones. Esto es claro para las variables, como en el caso de `f`, que, en general denota el resultado de la aplicación de un constructor de fórmulas.

Sucede lo mismo para los nombres. Aunque no hayan sido usados con ese sentido en nuestro caso de estudio, podrían darse, por ejemplo, definiciones como:

```
1 :- sucesor(cero)
2 :- sucesor(sucesor(cero))
...
```

que asociarían la representación corriente (con dígitos) a las expresiones dadas por la definición inductiva de los naturales con los constructores `cero` y `sucesor`.

De hecho, el uso de nombres es el mecanismo a través del cual una expresión arbitrariamente compleja puede ser referenciada sin necesidad de mostrar su estructura interna. Es, por tanto, una facilidad de abstracción en la formulación de expresiones.

La diferencia entre nombre y variable reside en que éste representa una expresión sobre cuya forma no se hacen hipótesis, es decir, «una expresión cualquiera». Cada nombre está asociado, por el contrario, a una expresión específica.

Estas consideraciones iluminan aspectos parciales de la estructura y el sentido de las expresiones usadas (a la derecha de `:-`):

- El propósito de estas es, claramente, expresar definiciones y aplicaciones de funciones.

- Dos casos particulares que sirven a ese objetivo ya han sido identificados. Éstos autorizan a escribir:

```
<expresión> ::= <variable> |
                <nombre>
```

que se lee: *expresión* es una *variable* o bien un *nombre*.

A partir de estas expresiones primitivas pueden formarse otras más complejas, que, como éstas, se interpretan como definiciones y aplicaciones de funciones.

La definición del conjunto de estas expresiones, se completa usando inducción de una manera informal:

```
<expresión> ::= <variable> |
                <nombre>   |
                ( <variable>,...,<variable> ) <expresión> |
                <expresión> ( <expresión>,...,<expresión> )
```

Se definen expresiones atómicas (*variables* y *nombres*) y otras compuestas que se llamarán *abstracciones funcionales* y *aplicaciones*.

Nótese que la denominación *abstracción funcional* se asocia a las expresiones que definen funciones. Esto no esun mero refinamiento de la terminología.

Efectivamente, considérese una expresión sencilla como:

```
4 + 2
```

Usando la abstracción funcional puede construirse una expresión de la cual la anterior sea un *caso particular*, haciendo, por ejemplo:

```
(x) 4 + x
```

La nueva expresión da una forma más general, de la cual pueden obtenerse, por aplicaciones convenientes, la expresión original además de otras. El cambio de constantes por argumentos es una técnica de generalización corriente también en la práctica de la programación.

Este concepto resultará de utilidad más adelante.

PÁG. 101

Es interesante analizar con más cuidado la sintaxis dada, tratando de determinar el tipo de construcciones que ella autoriza. En particular, considérese el caso en que se desea definir una función. Corresponde a una expresión de la forma de *abstracción funcional*:

```
( <variable>, ... , <variable> ) <expresión>
```

En particular, es, obviamente, admisible que contenga un único argumento nominal, como en:

```
(g) <expresión>
```

Ahora *¡expresión¿* puede ser reemplazada por cualquiera de sus formas válidas. Su significado será, de acuerdo a lo dicho, denotar el correspondiente de `g` en la función que se está definiendo.

Una de las posibilidades admitidas es que *¡expresión¿* sea reemplazada por otra abstracción funcional. Esto debe interpretarse como que el resultado de una función puede, a su vez, ser otra función.

Por ejemplo:

```
(g) (x) <expresión>
```

Ahora está dicho que la imagen de `g` será una cierta función de `x`. Está por darse, todavía, la definición de esta última.

En particular, `g` también puede ser considerada como una función, a su vez. Esto no es una novedad, puesto que una variable denota en principio una expresión genérica y, de hecho, así se usó `datom` en `dimensión`.

Entonces puede entenderse:

```
(g) (x) g (g (x))
```

La función definida sobre `g` le hace corresponder otra función que dado otro argumento `x`, aplica `g` a éste dos veces (*twice*, en inglés).

Ahora es posible asociar un nombre a esta función: PÁG. 102

```
twice :- (g) (x).g (g (x))
```

y calcular la imagen en `twice` de una función conocida, por ejemplo la raíz cuadrada (`sqrt`):

```
twice (sqrt) :- (x) sqrt (sqrt (x))
```

donde aparece claro que la aplicación de `twice` a una función es otra función. Ésta puede aplicarse, a su vez, a un cierto valor, como en:

```
twice (sqrt) (16)
```

cuyo resultado, salvo error u omisión, debería ser `2`.

Varias conclusiones deben obtenerse:

- Los argumentos y resultados de funciones son, en general, otras funciones, las cuales pueden aplicarse, a su vez, a otros argumentos. Así ocurre con `g`, argumento de `twice`, que aparece aplicada en la expresión que define a ésta y también con `twice(g)` que, a su vez, se aplica a otro argumento numérico.

  El tipo de función que este lenguaje maneja es llamado «*de orden superior*» (en inglés *high order functions*) para denotar el hecho de que sus argumentos y resultados son, a la vez, funciones.

- Nótese la forma de la expresión:

  ```
  twice (sqrt) (16)
  ```

  que denota la composición de dos aplicaciones, que deben entenderse realizadas de izquierda a derecha. La primera devuelve una función, como ya se vió, la cual es aplicada al argumento efectivo `16`.

  Esto es diferente, aunque similar, a:

  ```
  twice (sqrt, 16)
  ```

PÁG. 103

  En el caso, la última expresión denotaría la aplicación de `twice` a dos argumentos. Sin embargo, su definición sólo admite uno. Para hacer consistente tal aplicación, la definición de `twice` debió haber sido:

  ```
  twice :- (g, x) g (g (x))
  ```

  que también es válida.

  Esto muestra que toda función de más de un argumento puede expresarse en términos de funciones de orden superior de un argumento.

  La transformación es muy sencilla:

  ```
  ( x1, ... , xN ) <expresión>
  (definición con N argumentos)
  ```

  puede reescribirse como:

  ```
  (x1), (x2), ... , (xN) <expresión>
  (función de orden superior de 1 argumento)
  ```

  Para mayor ejemplo, la suma de enteros podría verse como:

  ```
  suma :- (x) (y) x + y
  ```

  es decir, una función de un argumento, que asocia a éste otra función de un argumento, que, a su vez, asocia a este último su suma con el primero.

  Para fijar ideas, piénsese en la expresión `suma (x)` como la función que aplicada a cualquier valor, le suma a éste la cantidad `x`.

Lo anterior permite simplificar la estructura de las expresiones del lenguaje: en la formulación de la forma general de las abstracciones funcionales y aplicaciones no es necesario denotar una cantidad indeterminada de argumentos. Apenas uno es suficiente para todos los casos:

```
<expresión> :- <variable> |
               <nombre>   |
               ( <variable> ) <expresión> |
               <expresión> ( <expresión> )
```

Estas expresiones pueden interpretarse usando un único concepto: el de función. Éstas pueden aplicarse entre sí y servir para definir a otras con plena libertad. Así lo autoriza la sintaxis. PÁG. 104

El lector atento podrá argumentar que esta libertad es excesiva y posiblemente la crítica sea aceptable.

En efecto, hay expresiones permitidas cuyo sentido es oscuro, si no existente. Véase por ejemplo:

```
5 (8)
```

que es un caso de aplicación donde función y argumento efectivo son nombres, pero que carece de sentido puesto que el primero no admite ser aplicado a ningún argumento, y también:

```
sucesor (sucesor)
```

donde el argumento efectivo no es un natural, como debiera esperarse.

El problema es importante, y constituye la principal motivación para la introducción del concepto de *tipo* en los lenguajes.

Por supuesto, existen lenguajes funcionales con tipos, aunque no serán tratados en este desarrollo. Como consecuencia, la formación de expresiones con sentido exigirá la aplicación de una disciplina (no formalizada) adicional a las reglas de sintaxis de los lenguajes que se definan.

Sin embargo, la generalidad de la construcción de expresiones del lenguaje tiene importantes ventajas. En particular, si se reinterpreta el término *función* como *programa* se tendrá entonces que, en estos lenguajes, los programas pueden ser, con toda generalidad, argumentos de otros programas y pueden producir, también, otros programas como resultado.

Esta caracterización es, efectivamente, adecuada y se justificará plenamente en la siguiente sección, donde quedará de manifiesto su utilidad.

6.2. La importancia de los lenguajes funcionales

PÁG. 105

Se considerará un caso en que las ideas dadas al final de la sección anterior pueden aplicarse con resultados apreciables. El caso se desarrolla en [Hug84]:

Se trata del problema de realizar la suma de todos los elementos de una lista. También las listas, como las fórmulas y los lenguajes, pueden definirse inductivamente:

- Hay una lista primitiva, llamada `Vacía`. Intuitivamente, representa una lista sin elementos.
- Hay un constructor de listas, que llamaremos `Cons`. Aplicado a un elemento y una lista, devuelve otra lista.

Intuitivamente, si `l` es la lista:

```
< x1, x2, ... , xN >
```

`Cons ( x0, l )` es la lista

```
< x0, x1, x2, ... , xN >
```

De este modo, las listas no vacías pueden pensarse como compuestas de un elemento (su cabeza) y otra lista (su cola).

Se supondrá que pueden aplicarse sobre listas las funciones:

- `Vacía?`: que denota `Verdadero` si es aplicada a la lista `Vacía`, y `Falso` en caso contrario.
- `Cabeza` y `Resto`: que son selectores para listas no vacías, devolviendo, respectivamente, cabeza y cola.

Ahora puede definirse la función en cuestión, siguiendo un esquema recursivo:

```
List_suma :- (l) if Vacía? (l)
                    then 0
                    else Cabeza (l) + List_suma (Resto (l) )
```

. 106 es decir, se define la imagen de la lista primitiva (`Vacía`) y luego se enuncia la regla para calcular la imagen de una lista compuesta en función de su cabeza y de la imagen de su cola.

El esquema sería idéntico si se quisiera definir el producto de todos los objetos de la lista, en lugar de la suma. Las modificaciones a introducir se marcan en la siguiente figura:

```
                                    1
List_suma :- (l) if Vacía? (l)    /
                    then [0] <---+
                    else Cabeza (l) [+] List_suma (Resto (l) )
                                        ^
                                        |
                                        *
```

Esto muestra un patrón recursivo, general para el tratamiento de listas, que está siendo aplicado a las partes entre corchetes.Con este enfoque se está sugiriendo una partición del problema que produciría un módulo de aplicación general, representando el patrón citado. Este módulo puede definirse como una función:

```
reduce :- (f) (b) (l) if Vacía? (l)
                         then b
                         else
                              f (Cabeza (l))
                                (reduce (f) (b) (Resto (l)))
```

donde se ha hecho abstracción, a partir de la expresión de `List_suma`, de:

- la función a aplicar (representada ahora por el argumento nominal `f`)
- el resultado para el caso base de la inducción (representado ahora por `b`)

En términos de la jerg clásica podría decirse que se ha construído un PÁG. 107 módulo que representa una *forma general de tratamiento de listas* que, combinada con funciones adecuadas, construye una variedad de *programas* concretos. Efectivamente, los anteriores *programas* de suma y producto de elementos de una lista son ahora resultados de aplicaciones convenientes de este módulo más general:

```
List_suma :- reduce (+) (0)
List_product :- reduce (*) (1)
```

que, como puede verse, a partir de la definición de `reduce`, son, como antes, funciones, cada una de un argumento, que se espera represente una lista.

De igual modo podrán construirse módulos similares para tratar otras clases de objetos, lo que equivale a decir que pueden definirse constructores de programas adaptados a cada aplicación particular.

Para esto es básico el concepto de *orden superior*, que autoriza a realizar abstracciones sobre funciones, es decir a que éstas sean representadas por variables que son argumentos de otras funciones.

Esta propiedad es un atributo esencial de los lenguajes funcionales.

En el caso general de los lenguajes imperativos hay, por ejemplo, una sustancial diferencia conceptual entre los programas y sus argumentos. Estos últimos (los llamados *datos*) denotan direcciones de una memoria con valores asociados mientras que los programas representan una transformación de, precisamente, tal asociación entre direcciones y valores. En los lenguajes en que esta heterogeneidad de significado entre los programas y sus argumentos se mantiene estrictamente, el concepto de orden superior no puede ser incorporado: una variable que sea argumento de un programa sólo puede representar una dirección de memoria que aquel eventualmente habrá de afectar y, por lo tanto, no otro programa.

PÁG. 108

Como consecuencia, las formas en que los programas pueden combinarse en estos lenguajes quedan restringidas a la aplicación de un conjunto preestablecido de constructores (las estructuras de control) sobre programas fijos (ya sea referenciados por un nombre o por una estructura expresa), lo cual limita la posibilidad de desarrollar módulos de aplicación general. Las *formas generales de tratamiento de objetos* como `reduce` pueden tratarse, en todo caso, como *esquemas de programas*, pero tal *esquema* no es un elemento del lenguaje y cada programa debe reproducir, en su construcción, la estructura del mismo.

Esta concepción, debida a la semántica particular de programas y datos, es tomada del modelo de cálculo en que estos lenguajes se fundamentan que corresponde al *cálculo por efecto* de los computadores Von Neumann.

En el esquema funcional, por el contrario, todas las expresiones denotan un valor, representando tanto el concepto de *dato* como el de función o *programa*. La diferencia es apenas una cuestión de interpretación:

- Lo que usualmente se llama *dato* es una expresión que no admite ser aplicada a ningún argumento, es decir, está *completamente evaluada*.

- Las funciones, por su parte, esperan ser aplicadas a una cierta cantidad de argumentos para ser *completamente evaluadas*.

Según esta interpretación, no hay más diferencias entre un entero y la función `+` (de dos argumentos) que entre ésta y, por ejemplo, `List_suma` (que admite un solo argumento).

Todos son casos particulares del concepto sintáctico de expresión, interpretado como valor y los constructores de expresiones (aplicación y abstracción funcional) deberían interpretarse simplemente como operadores cuyo resultado es una expresión *más evaluada* (en el caso de la aplicación) o *menos evaluada* (en el de la abstracción funcional) que sus operandos.

Otras consecuencias interesantes de esta uniformidad conceptual entre *datos* y *programas* pueden ilustrarse con otro ejemplo.

Considérese el problema de definir funciones para representar las operaciones clásicas sobre conjuntos: unión, intersección, pertenencia de un elemento a un conjunto. PÁG. 109

Se supondrá, además, que los conjuntos a manejar son definidos por comprensión, es decir, por una propiedad que deben cumplir sus elementos, los cuales supondremos, para fijar ideas, que son, en todos los casos, números naturales.

Con esta idea, tales conjuntos pueden representarse por medio de funciones que, aplicadas a un elemento, devuelven `Verdadero` si éste tiene la propiedad que define al conjunto, y `Falso` en caso contrario.

Por ejemplo, la función

```
Pares :- (x) (x / 2) = 0
```

define el conjunto de los pares, si se asume que `=` representa el predicado de igualdad en los naturales y, análogamente puede interpretarse:

```
Menores_que_10 :- (x) x < 10
```

De este modo, los *datos* del problema (conjuntos) están siendo representados por funciones que pueden ser calculadas aplicándolas a argumentos (los que se llamarían *programas* en la concepción tradicional) y no como estructuras localizadas en una memoria.

Ahora las operaciones clásicas sobre estos conjuntos pueden recibir representaciones muy sencillas:

- La pertenencia de un elemento a un conjunto se resuelve, eimplemente, aplicando la función que representa a este último al elemento en cuestión:

  ```
  Member :- (x,c) c(x)
  ```

donde `c` representa un conjunto y `x` el elemento cuya pertenencia a `c` se quiere verificar. La forma de esta función se justifica, simplemente, por la representación elegida para los conjuntos.

PÁG. 110

- La unión de dos conjuntos `c1` y `c2` es otro conjunto cuyos elementos se caracterizan por satisfacer la propiedad que define a `c1` o la que define a `c2`. De aquí, entonces, la definición:

```
Unión :- (c1, c2)
                  (x)
                  c1 (x) or c2 (x)
```

  donde `or` representa la disyunción lógica.

  Nótese que `Unión` se define como función de dos argumentos (`c1` y `c2`)

TODAVÍA NO HEMOS COMPUESTO ESTA PARTE DEL LIBRO, POR FAVOR TENGA PACIENCIA. MUCHAS GRACIAS.

Capítulo 7

El cálculo Lambda

Version: $Id: plyf.tex,v 1.3 2003/08/13 04:28:26 cballard Exp $

Todavía no hemos compuesto esta parte del libro, por favor tenga paciencia. Muchas gracias.

Capítulo 8

El lenguaje de programación SCHEME

Version: $Id: plyf.tex,v 1.3 2003/08/13 04:28:26 cballard Exp $

Todavía no hemos compuesto esta parte del libro, por favor tenga paciencia. Muchas gracias.

Version: $Id: plyf.tex,v 1.3 2003/08/13 04:28:26 cballard Exp $

Capítulo 9

Anexo: Conceptos básicos

Version: $Id: plyf.tex,v 1.3 2003/08/13 04:28:26 cballard Exp $

Todavía no hemos compuesto esta parte del libro, por favor tenga paciencia. Muchas gracias.

Version: $Id: plyf.tex,v 1.3 2003/08/13 04:28:26 cballard Exp $

Bibliografía

[Abe85] Abelson H., Sussman G. with Sussman J.
Structure and Interpretation of Computer Programs
MIT Press - McGraw Hill, 1985.

[Bac78] Backus J.
Can Programming Be Liberated From Von Neumann Style?
Communications ACM 21(8), 613–641, 1978.

[Bar84] Barendregt H.
The Lambda Calculus: its syntax and semantics
North Holland, 1984

[Bat73] Battani, G. Meloni, H.
Interpreteur du language de programmation PROLOG
Research Report, Univ. of Aix-Marseille, 1973

[Caf82] Caferra R.
Tesis doctoral
Univ. Scientifique et Medicale de Grenoble, 1982

[Clo81] Clocksin W., Mellish C.
Programming in PROLOG
Springer-Verlag, 1981

[Col73] Colmerauer, A. Kanoui H., Roussel Ph., Pasero, R.
Un Systeme de Communication Homme-Machine en Francais
Research Report, Univ. of Aix-Marseille, 1973

[Cur58] Curry H., Feys R.
Combinatory Logic, Vol. I
North Holland, 1958

[Dij72] Dijkstra E.
Notes on Structured Programming

en *Structured Programming*, Dahl O., Dijkstra E., Hoare C.
Academic Press, 1976

[Gla84] Glaser H., Hankin C., Till D.
Principles of Functional Programming
Prentice Hall, 1984

[Hen80] Henderson, P.
Functional Programming: Application and Implementation
Prentice Hall, 1980

[Hin86] Hindley R., Seldin J.
An Introduction to Combinators and th Lambda Calculus
Cambridge University Press, 1986

[Hog84] Hogger C.
Introduction to Logic Programming
Academic Press, 1984

[Hug84] Hughes J.
Why Functional Programming Matters
University of Goteborg, Programming Methodology Group, 1984

[Knu68] Knuth D.
The Art of Computer Programming, Vol. 1–3
Addison Wesley, 1968

[Kow79] Kowalski R.
Algorithm = Logic + Control
Communications ACM 22, 424–431, 1979

[Llo84] Lloyd J.
Foundations of Logic Programming
Springer-Verlag, 1984

[Loe84] Loeckx J., Sieber K.
Foundations of Program Verification
John Wiley & Sons, 1984

[Men64] Mendelson E.
Introduction to Mathematical Logic
Van Nostrand, 1964

[Rou75] Rousell Ph.
PROLOG: manuel de reference et d'utilisation
Research Report, Univ. of Aix-Marseille, 1975

[Wan80] Wand M.
Induction, Recursion and Programming
North Holland, 1980

[War79] Warren D.
PROLOG on the DEC System-10
en *Expert Systems in the Microelectronic Age (Michie D. (ed.))*
Edinburgh University Press, 1979

[Wir71] Wirth N.
Program Development by Stepwise Refinement
Communications ACM 14, 4, 221–227, 1971