TESIS

UNA APROXIMACION A LA PROGRAMACION LOGICA CON FUNCIONES INDETERMINISTAS

Antonio Sarmiento Escalona

A Mila, Oriol y Elena

TESIS

UNA APROXIMACION A LA PROGRAMACION LOGICA CON FUNCIONES INDETERMINISTAS

Antonio Sarmiento Escalona

UNA APROXIMACION A LA PROGRAMACION LOGICA

CON FUNCIONES INDETERMINISTAS

TESIS presentada por ANTONIO SARMIENTO ESCALONA
Licenciado en Ciencias Matemáticas por la Universidad de Santiago en la FACULTAD DE INFORMATICA de la Universidad de La Coruña para la obtención del GRADO DE DOCTOR EN INFORMATICA.

Director de la Tésis:
D. Mario Rodriguez Artalejo
Universidad Complutense de Madrid.

Ponente de la Tésis:
D. José Luis Freire Nistal
Universidad de La Coruña.

TRIBUNAL CALIFICADOR:

Presidente:	D. Isidro Ramos Salavert.
Vocales:	D. Fernando Orejas Valdés.
	D. José María Troya Linero.
	D. José María Barja Gómez.
Secretario:	D. Antonio Blanco Ferro.

La Coruña, abril 1992.

AGRADECIMIENTOS

Mi contacto inicial con Mario Rodriguez Artalejo fué algo complicado. Comencé a trabajar en 1986 en reescritura con un profesor de la Facultad de Informática de Madrid llamado Miguel Muñoz. Desgraciadamente, Miguel falleció repentinamente. En nuestras conversaciones me habló de Mario como la persona más indicada para orientarme en programación lógica. Así que, en unos momentos un tanto difíciles, me puse en contacto con Mario. Nunca podré agradecer suficientemente la comprensión que Mario tuvo en aquellos momentos y la ayuda, amabilidad y paciencia que ha tenido para conducirme por los temas aquí tratados. Lo que sigue es fruto de estos años de colaboración y nunca mejor dicho aquello de que sin él este trabajo no sería posible. En estos momentos, mi mayor deseo sería tener que darle de nuevo las gracias porque querría decir que habría hecho un nuevo trabajo con él.

Debo mencionar a los profesores del Departamento de Informática de la Universidad Complutense de Madrid por su cariñosa acogida siempre que estuve por allí y por no enfadarse por el tiempo que les arrebaté la atención de Mario.

También quiero agradecer a los profesores de la Facultad de Informática de La Coruña, y especialmente a José Luis Freire, por las facilidades dadas para resolver todos los problemas burocráticos.

Finalmente, hay que mencionar la paciencia de Mila, Oriol y Elena para aguantarme tanto tiempo enfrascado en estos temas.

INDICE

INTRODUCCION

En los últimos años se han hecho múltiples intentos para diseñar lenguajes que integren la programación lógica y funcional. [Bellia & Levi 86] [DeGroot & Lindstrom 86]. Los lenguajes lógicos, de primer orden, sin tipos, sin direccionalidad, con variables lógicas y datos incompletos, usando evaluación con búsqueda (resolución), ... son potentes pero díficiles de controlar. Los lenguajes funcionales, por su parte, de orden superior, con tipos y polimorfismo, con direccionalidad y datos completos, usando evaluación sin búsqueda (reescritura), ... son elegantes y claros pero menos potentes. Con la integración de ambos tipos de lenguajes en unos nuevos lenguajes lógico-aplicativos se trata de sumar las ventajas de ambos minimizando los problemas que dicha integración puede crear (por ejemplo, la implementación). Un resultado en esta integración que ha influido en este trabajo es el lenguaje BABEL [Moreno & Rodriguez 89].

En las páginas que siguen se aborda desde el punto de vista de la integración de la programación lógica y funcional el problema del **indeterminismo**. La palabra indeterminismo se usa como un concepto técnico con el que se quiere expresar ciertos modelos de computación abstracta. Intuitivamente, un algoritmo se dice indeterminista si durante su ejecución se puede elegir entre un número de alternativas libre e impredeciblemente. Muchos programadores creen que el indeterminismo debe tenerse en cuenta desde el momento que es un componente esencial de los sistemas de tiempo real tales como sistemas

operativos y controladores de dispositivos. Aunque una máquina indeterminista verdadera no existe si es posible simular el indeterminismo. En nuestro trabajo, lo hemos simulado definiendo unas operaciones indeterministas en el ámbito de un lenguaje (del cual BABEL puede considerarse un antecedente), que integra la programación lógica y funcional.

El lenguaje que aquí presentamos se puede considerar como un lenguaje funcional con disciplina de constructores en el que los predicados son tratados como funciones booleanas. El indeterminismo ha sido simulado definiendo unas así llamadas **funciones indeterministas** que permiten aumentar la potencia y expresividad del lenguaje juntando las buenas cualidades de ambos estilos de programación. De otro modo, podemos describirlo como un sistema de reescritura de términos no confluentes en cuyas reglas las **partes izquierdas son lineales** y se permiten **variables libres**.

La semántica declarativa del lenguaje es necesario estudiarla en el marco de los dominios de Scott porque queremos permitir la existencia de datos y estructuras infinitas. Al no tener funciones en sentido estricto hemos desarrollado el estudio de los grafos correspondientes a la relación definida por las funciones indeterministas. Entonces se ha construido un **dominio** (de Scott) **de funciones indeterministas**. Algunas de las funciones serán deterministas por lo que el dominio citado debe contener como subdominio al de las funciones continuas (deterministas). Estos resultados se pueden comparar también con otros más clásicos de la teoría de dominios: los **dominios potencia**. Finalmente, se ha definido una semántica de punto fijo, como en PROLOG [van Emden & Kowalski 76]

[Apt & van Emden 82], que permite la construcción de un modelo mínimo de Herbrand, necesario para demostrar la completitud del lenguaje.

Ya es conocido [Reddy 85] que el uso de **estrechamiento** como semántica operacional en los lenguajes lógico-aplicativos permite simular al mismo tiempo SLD-resolución y reescritura. Sin embargo, el uso de estrechamiento no "innermost" puede provocar incorrección en algunos casos. El problema surge de la necesidad de tener en cuenta en todo el cómputo la compartición de la información en la partes derechas de las reglas. Hacer lineales las partes derechas de reglas reduciría la expresividad del lenguaje. Limitar el estrechamiento al caso "innermost" restringiría la posibilidad de hacerle perezoso. La solución propuesta es **representar las expresiones del lenguaje como grafos dirigidos acíclicos** (gda's) y usar estrechamiento en ellos como mecanismo de cómputo. Previamente se ha construido un **algoritmo de unificación lineal en gda's** que devuelve como salida EXITO si una expresión y un lado izquierdo de regla son unificables, FALLO si no lo son, o SUSPENDIDA si se requiere más evaluación antes de que la unificación pueda realizarse.

Agradablemente se prueba que esta semántica operacional de estrechamiento en gda's es correcta y completa respecto a la semántica declarativa citada.

La forma de utilización de este lenguaje, aún bastante primitivo, es la preocupación final de nuestro trabajo. Como muestra de sus posibilidades, se ha desarrollado un compilador para un lenguaje ideal que es un subconjunto de PASCAL.

El trabajo está organizado en capítulos. Cada capítulo está subdividido en apartados o secciones que se numeran añadiendo un

dígito al del capítulo. Las definiciones y teoremas se numeran añadiendo un tercer dígito con lo que en el texto se pueden encontrar fácilmente en el capítulo y sección correpondiente. El final de una definición o teorema viene marcado por el signo ■.

El contenido de cada capítulo es como sigue: en el capítulo 1 se indican las cuestiones más relevantes para nuestro trabajo de los programas lógicos, la programación funcional y los lenguajes lógico aplicativos; en el capítulo 2 se introduce el lenguaje y se exponen unos ejemplos que muestran su potencia y expresividad, y otro ejemplo que motiva buena parte del trabajo; en el capítulo 3 se describen los nuevos dominios necesarios para la semántica declarativa; ésta se describe en el capítulo 4; en el capítulo 5, se estudian los gda's y la semántica operacional del lenguaje; en el 6 su corrección y completitud; y finalmente en el capítulo 7 se muestran las posibilidades de uso de nuestro lenguaje y se construye el compilador citado.

1. HACIA LA INTEGRACION DE LA PROGRAMACION LOGICA Y FUNCIONAL

En este capítulo vamos a hacer una descripción breve de algunos fundamentos teóricos de la programación lógica (apartado 1), de la programación funcional (apartado 2), y de la integración de ambos paradigmas (apartado 3). Necesitamos tenerlos en cuenta antes de diseñar el lenguaje deseado. También analizaremos las respectivas semánticas y los mecanismos de cómputo más usuales en cada caso: SLD-resolución, reescritura y estrechamiento.

1.1. PROGRAMACION LOGICA Y SLD-RESOLUCION

La programación lógica fué introducida por Colmerauer y su grupo a partir de sus trabajos sobre "Procesamiento del lenguaje natural" [Colmerauer & al 73]. La idea clave que propició su nacimiento, compartida con Kowalski [Kowalski 74], es que: "la lógica puede ser usada como un lenguaje de programación". El interés por ella creció al ser la base teórica del PROLOG, lenguaje de repentino y "popular" éxito al ser incluido en el proyecto japonés de la quinta generación. Razones de eficiencia hacen que el PROLOG sea diferente de la programación lógica: se suprime un importante test en el algoritmo de unificación ("occur-check") y se añade una facilidad de control ("cut") que afecta a la completitud del lenguaje. Esto dificulta

aplicar a su estudio los resultados teóricos de la programación lógica. Los fundamentos de esta última pueden verse por ejemplo en [Lloyd 87] y en [Apt 90].

1.1.1. PROGRAMAS LOGICOS

La potencia de la programación lógica está basada en su sencillez formal y en estar fundamentada en la lógica matemática que suministra métodos de trabajo y una rigurosa base matemática.

Los programas lógicos son simplemente un conjunto de ciertas fórmulas de un lenguaje de primer orden. Así, si A y B_i para i = 1, ..., n son **átomos** (fórmulas de la forma $p(t_1,\ldots,t_k)$ donde p es una relación k-aria y los $t_1,\ldots,t_k$ son términos)

$$A \leftarrow B_1,\ldots,B_n \qquad n \geq 0$$

es una **cláusula definida**, dónde suponemos que las variables están cuantificadas universalmente. Consideramos, fórmulas como

$$\leftarrow B_1,\ldots,B_k \qquad k > 0$$

que llamaremos **cláusulas objetivo** y también □, la **cláusula vacía**.
Un **programa lógico** es un conjunto finito no vacío de cláusulas definidas.

Hay dos maneras de interpretar una cláusula definida:
1) " Para resolver A hay que resolver B_i para i = 1,...,n ". Esta interpretación es precisamente la que marca la diferencia entre lo que es la programación lógica y la lógica de primer orden. Se suele llamar **interpretación procedural** y explica **cuál** es el mecanismo computacional que sirve de base para la ejecución de un programa. Tiene que ver con la semántica operacional de los lenguajes de programación.
2) La otra interpretación llamada **interpretación declarativa** analiza el significado de un programa basado en la semántica de la teoría de

modelos de la lógica de primer orden, y explica qué puede ser computado por el programa. Las claúsulas definidas del programa son los axiomas de la teoría y la lógica de primer orden suministra métodos para deducir los teoremas de la teoría, que son las consecuencias lógicas de los axiomas. Tiene que ver con la semántica denotacional de los lenguajes de programación.

1.1.2. SLD-RESOLUCION

En programación lógica se asignan valores a las variables por medio de un tipo especial de sustituciones: los **unificadores más generales** (u.m.g.). Existe un algoritmo de unificación [Robinson 65] que para dos átomos cualesquiera produce un u.m.g. si son unificables y devuelve fallo en otro caso.

Los programas lógicos computan mediante una combinación de dos mecanismos -reemplazamiento y unificación- usando un refinamiento de la primitiva regla de inferencia de Robinson llamada **SLD-resolución** (Linear resolution with Selection rule for Definite Clauses). Así, si $N \equiv \leftarrow A_1,\ldots,A_n$ es un objetivo y $C \equiv A \leftarrow B_1,\ldots,B_k$ una cláusula **variante** (con variable distintas a las de N) de un programa Π y si para algún A_i con $1 \leq i \leq n$ seleccionado via una regla R de selección A_i y A **unifican** con u.m.g. θ obtenemos

$$N' \equiv \leftarrow (A_1,\ldots,A_{i-1},B_1,\ldots,B_k,A_{i+1},\ldots,A_n)\theta$$

que llamamos un **resolvente** de N y C.

Una **SLD-derivación** de un programa Π y una cláusula objetivo N .es una sucesión máximal N_0, N_1,... de cláusulas objetivo junto con una sucesión C_0, C_1,... de variantes de cláusulas de P y una sucesión θ_0, θ_1,... de sustituciones. Cuando uno de los resolventes N_i es $\square$ entonces es el último objetivo de la derivación y hablamos de una

SLD-refutación.

Es importante hacer notar que los programas lógicos pueden usarse no sólo para refutar sino también, y es el punto de vista más importante, para **computar** a través de las ligaduras entre variables y términos obtenidas por los unificadores θ_i.

La existencia de una SLD-refutación para $\Pi \cup \{ N \}$ puede verse como una **contradicción**. O sea, como si hubiesemos probado la negación de N. Como N es lo mismo que $\forall x_1 \ldots \forall x_s (\neg A_1 \vee \ldots \vee \neg A_k)$, su negación sería semánticamente equivalente a $\exists x_1 \ldots \exists x_s (A_1 \wedge \ldots \wedge A_k)$. Las ligaduras obtenidas por las θ_i nos llevan entonces a interpretar una SLD-refutación como una prueba de la fórmula $[(A_1 \wedge \ldots \wedge A_k)\theta_0 \ldots \theta_m]^{\forall}$. La restricción de $\theta_0 \ldots \theta_m$ a las variables de N se llama una **sustitución respuesta computada** para $\Pi \cup \{ N \}$.

Se puede establecer un resultado de corrección para SLD-resolución: Si existe una SLD-refutación de $\Pi \cup \{ N \}$ entonces $\Pi \cup \{ N \}$ es inconsistente. El inverso de este resultado es una primera versión de la completitud de la SLD-resolución: si $P \cup \{ N \}$ es inconsistente existe una SLD-refutación de $P \cup \{ N \}$ [Hill 74].

El siguiente resultado de corrección es debido a Clark: sea Π un programa y $N = \leftarrow A_1, \ldots, A_k$ un objetivo. Si existe una SLD-refutación de $\Pi \cup \{ N \}$ con sustituciones $\theta_0, \ldots, \theta_n$ entonces $[(A_1 \wedge \ldots \wedge A_k)\theta_0 \ldots \theta_n]^{\forall}$ es una consecuencia semántica de Π [Clark 79].

El concepto de **sustitución respuesta correcta** es la contrapartida denotacional al concepto de sustitución respuesta computada. Dado un programa Π y un objetivo $N = \leftarrow A_1, \ldots, A_n$; θ es una sustitución respuesta correcta para $\Pi \cup \{ N \}$ si θ está restringida a las variables de N y $[(A_1 \wedge \ldots \wedge A_n)\theta]^{\forall}$ es consecuencia

semántica de Π. Así la corrección de la SLD-resolución nos asegura que si existe una SLD-refutación la sustitución que se obtiene es una sustitución correcta.

Reciprocamente, el teorema de completitud afirma que para toda sustitución respuesta correcta θ para Π ∪ { N } existe una sustitución respuesta computada para Π ∪ { N } que es más general que θ [Clark 79].

Se puede obtener una versión fuerte del teorema de completitud teniendo en cuenta la regla de selección. Para ello se construye el **SLD-árbol** para Π ∪ { N } via R agrupando todas las SLD-derivaciones de Π ∪ { N } via R. SLD-árboles para Π ∪ { N } pueden diferir en tamaño y forma (incluso pueden ser infinitos). Un SLD-árbol que contiene la cláusula vacía se llama SLD-árbol con **éxito**. Un resultado de Hill asegura que si Π ∪ { N } es inconsistente entonces todo SLD-árbol con N en la raíz es un SLD-árbol con éxito [Hill 74].

Podemos con lo visto hasta ahora formalizar una semántica operacional para un lenguaje de programación lógica: identificamos cada objetivo N con una entrada para un programa Π; éste calcula mediante SLD-resolución una sustitución θ que afecta a las variables de N y una computación tiene éxito cuando se logra una SLD-refutación de Π ∪ { N } dando como salida θ. De este modo hacemos corresponder a esta semántica un procedimiento de deducción formal en la lógica de primer orden que puede ser utilizado como intérprete de programas lógicos en virtud de la correción y completitud, que muestra la equivalencia con la semántica declarativa: el significado de los programas lógicos como sentencias de primer orden.

1.1.3. MODELO MINIMO DE LOS PROGRAMAS LOGICOS

Se puede introducir una nueva semántica, llamada **semántica de punto fijo** [Van Emden & Kowalski 76] y [Apt & Van Emden 82], que es un caso particular de la semántica de modelos de la lógica de primer orden. Y se puede ver su equivalencia con las dos anteriores a través de resultados de corrección y completitud [Lloyd 87].

Para el estudio de la semántica punto fijo es necesario construir el **operador consecuencia inmediata** T_π. Para ello hay que observar que para cada programa Π hay un lenguaje de primer orden asociado. Definimos U_π, **universo de Herbrand** del programa, como el conjunto de todos los términos básicos (i.e. sin variables) del lenguaje asociado. B_π, **base de Herbrand** del programa, es el conjunto de todos los átomos básicos. Una **interpretación de Herbrand** para Π se puede identificar con un subconjunto de B_π, y un **modelo de Herbrand** para Π es una interpretación de Herbrand que es modelo para Π. Los modelos de Herbrand son suficientes para la lógica de cláusulas, ya que si Π tiene un modelo tiene un modelo de Herbrand.

Definimos T_π como una aplicación que a cada interpretación de Herbrand I de Π le hace corresponder una interpretación $T_\pi(I)$ tal que:

$A \in T_\pi(I) \iff$ para algunos átomos básicos $B_1, \ldots, B_n$

$A \leftarrow B_1, \ldots, B_n$ es instancia básica de

alguna cláusula de Π e $I \models B_1 \wedge \ldots \wedge B_n$.

T_π es un operador continuo (y en consecuencia monótono) del retículo completo de interpretaciones de Herbrand [Lloyd 87] y puede aplicarse la teoría punto fijo de Tarski considerando las potencias ordinales de T_π [Tarski 55].

En el modelo mínimo Herbrand I_π confluyen todas las ideas anteriores [Lloyd 87] [Apt 90]:

I_π = intersección de todos los modelos de Herbrand de Π.

= $\{ A \in B_\pi \mid \Pi \vdash_{SLD} A \}$ la totalidad de información que cabe extraer de Π mediante SLD-resolución.

= $\{ A \in B_\pi \mid \Pi \models A \}$ toda la información contenida en Π: las instancias verdaderas de todas las relaciones.

= el menor pre-punto-fijo de T_π.

= el menor punto-fijo de T_π.

= $T_\pi \omega = \bigcup_n T_\pi n$.

El hecho fundamental que permite la existencia del modelo mínimo es que pueden encontrarse resultados de completitud basándonos en que si $A \in I_\pi$ tenemos que $A \in T_\pi^N(\varnothing)$ para algún N; lo que quiere decir que una "prueba" de A, **en un número finito de etapas**, puede encontrarse de acuerdo con un método de deducción [Apt & Van Emden 82].

1.1.4. OTRAS CUESTIONES QUE NOS INTERESAN DE LOS PROGRAMAS LOGICOS

Algunos comentarios finales muy breves sobre algunos aspectos de la programación lógica:

1) COMPUTABILIDAD. Se puede afirmar que toda función **computable** puede ser computada por un programa lógico apropiado. Hay varias maneras de abordar este problema según la definición de "computable" que se tome. Puede consultarse por ejemplo [Andreka & Nemeti 78].

2) NEGACION. SLD-resolución es una forma muy restrictiva de razonamiento a causa de que sólo hechos positivos pueden deducirse de ella. Esto es debido a que la base de Herbrand es un modelo de cualquier programa Π pero no lo es de $\neg A$, para A átomo básico; luego

sólo información positiva puede ser una consecuencia lógica del programa. No puede deducirse información negativa sin afectar la corrección y/o la monotonía. El tratamiento de la negación en los programas lógicos se hace introduciendo reglas como la "**negation as finite failure**" [Clark 77] y estudiando las nuevas semánticas que origina.

3) INDETERMINISMO. Se habla de **indeterminismo** en un programa lógico en relación con cuál de los objetivos del resolvente se reduce. Pero como ya hemos dicho en la introducción máquinas indeterministas sólo pueden simularse; por tanto, el intérprete PROLOG soluciona el problema escogiendo el objetivo más a la izquierda y "reemplazando" la elección indeterminista por búsqueda secuencial y "backtracking".

También se cita [Sterling & Shapiro 86] el indeterminismo como una "forma de pensar" en el diseño de programas lógicos; por ejemplo del tipo *Generate-and-Test*:

$$find(X) \leftarrow generate(X), test(X).$$

El generador busca una solución dentro de las posibles y el "chequeador" verifica que es correcta.

Dos formas de indeterminismo se suele indicar que existen en programación lógica [Sterling & Shapiro 86]. Difieren en la naturaleza de la elección entre alternativas que debe ser hecha: Para *don't-care nondeterminism* la elección puede ser arbitraria como en el programa:

$$minimum(X,Y,X) \leftarrow X \leq Y.$$

$$minimum(X,Y,Y) \leftarrow Y \leq X.$$

Cualquier reducción lleva a una solución y no importa que solución se encuentre. Para *don't-know nondeterminism* la elección importa pero la correcta no se conoce en el momento de la elección. Por ejemplo, cada

una de las siguientes claúsulas es correcta pero no sabemos cual de ellas hay que usar para probar el isomorfismo de dos árboles binarios:

```
isotree(empty,empty).
isotree(tree(X,L1,R1),tree(X,L2,R2)) ←
    isotree(L1,R1),isotree(L2,R2).
isotree(tree(X,L1,R1),tree(X,L2,R2)) ←
    isotree(L1,R2),isotree(L2,R1).
```

4) TENDENCIAS. Hay dos tendencias actuales en la construcción de lenguajes de programación lógica [Lloyd 87]: En la primera, referida a los llamados lenguajes "sistema", se hace hincapie en el paralelismo AND, en el indeterminismo del tipo "no importa" (don't-care) y en los programas definidos; es decir, sin negación. Ejemplos de esta tendencia son PARLOG [Clark & Gregory 86] y PROLOG concurrente [Shapiro 83]. En la segunda, referida a los lenguajes "aplicación", se busca unos lenguajes de proposito general y se hace hincapie en el paralelismo OR, en el indeterminismo del tipo "no conozco" (don't-know) y en los programas generales. Ejemplos son micro-PROLOG [Clark & McCabe 84] y Quintus-PROLOG [Bowen & al. 85].

1.2. PROGRAMACION FUNCIONAL ECUACIONAL Y REESCRITURA

Los lenguajes funcionales pueden considerarse un paso "natural" en la evolución histórica de los lenguajes de programación. Frente a los lenguajes **imperativos** dependientes de la arquitectura Von Neumann, con variables no referencialmente transparentes, y con asignaciones y efectos, los lenguajes **funcionales** presentan la

posibilidad de paralelismo implícito, usan variables referencialmente transparentes, funciones en vez de asignaciones, y no tienen efectos. [Hudak 89]

1.2.1. EVOLUCION HISTORICA DE LOS LENGUAJES FUNCIONALES

El desarrollo de los lenguajes funcionales ha estado influido por muchas teorías pero sus fundamentos matemáticos hay que buscarlos en el **λ-cálculo** [Church 41] [Barendreght 84], la **lógica ecuacional** y los **sistemas de reescritura de términos** [Dershowitz & Jouannaud 90] [Dershowitz & Okada 90] [Klop 90].

El trabajo de Church fué motivado por el deseo de crear un cálculo (es decir: una sintaxis para términos y un conjunto de reglas de reescritura para transformarlos) que capturase el comportamiento computacional de las funciones. Este cálculo tiene propiedades muy interesantes: por ejemplo, las funciones pueden aplicarse a ellas mismas; y el efecto de recursión se puede conseguir, mediante el **Y-combinator**, sin escribir explícitamente una función recursiva. Esta habilidad para simular recursión es la que da su potencia al λ-cálculo y la que le ha permitido permanecer como un modelo útil de computación.

En paralelo con el desarrollo del λ-cálculo Schönfinkel y Curry crearon las bases de la **lógica combinatoria**. [Curry & Feys 58]. Aunque la infuencia de ésta sobre los lenguajes funcionales no es comparable con la ejercida por el λ-cálculo, el cálculo combinatorio ha jugado un importante papel en la implementación de estos lenguajes. (Ver [Turner 79] y [Peyton Jones 87]).

Un hecho notable del λ-cálculo es su restricción a funciones de un argumento. Esta restricción será explotada por Curry y Feys que

usarán la notación (f x y); conocida característica sintáctica de los lenguajes funcionales modernos.

Se suele citar el LISP [McCarthy 60] como el ejemplo más clásico de lenguaje funcional aunque no lo sea en sentido estricto. Lo que es indudable es la influencia del LISP en el desarrollo de estos lenguajes. Por otra parte, ha sido exagerada la influencia del λ-cálculo en el LISP. Lo que toma el LISP del λ-cálculo es la representación de funciones anónimamente. En cambio, el papel que juega el "Y-combinator" se sustituye por la existencia de **expresiones condicionales,** con lo que las funciones recursivas se pueden definir explícitamente. Otras novedades aportadas por el LISP son: el uso de **listas** y operaciones de orden superior sobre ellas como **mapcar**; y la idea de un constructor de listas como **cons**.

Se puede citar el trabajo de Landin en el diseño del lenguaje ISWIN [Landin 66] por algunas importantes contribuciones al desarrollo de los lenguajes funcionales: Sintácticamente, la introducción de **let** y **where** y la noción de definiciones mutuamente recursivas. Semánticamente, por hacer hincapie en el razonamiento ecuacional ("la habilidad para sustituir iguales por iguales") y por el diseño de la **SECD machine** que permite mecanizar la evaluación de expresiones mediante una máquina abstracta.

APL [Iverson 62] aunque no es un lenguaje puramente funcional se puede mencionar porque su parte funcional es un ejemplo de como conseguir programar funcionalmente sin hacer uso de expresiones lambda. El lenguje FP [Backus 78] tiene mucho en común con APL: la diferencia principal es que la estructura de datos fundamental en FP es la **sequence** mientras que en APL es el **array**. Sin embargo, se

suele citar la autoridad de Backus (que había intervenido en el diseño de FORTRAN y ALGOL) como polémico defensor de los lenguajes funcionales frente a los lenguajes imperativos ligados a la máquina von Neumann e incapaces, según él, de adaptarse a la moderna demanda de "software".

A mediados de los 70 trabajando con un sistema de deducción automática llamado LCF para funciones recursivas surge el lenguaje ML [Gordon & al. 79] cuyo nombre viene de servir como MetaLenguaje para el LCF. Posteriormente, se ha hecho un esfuerzo para "normalizar" el lenguaje diseñando el Standard ML ó SML [Milner 87] tomando ideas de otros lenguajes como HOPE. Lo más señalable de SML es el sistema de tipos: Es un lenguaje fuertemente tipado, que usa un sistema de inferencia de tipos en vez de declararlos explicitamente; permite funciones polimorfas y tiene tipos de datos abstractos y concretos definidos por el usuario. Otros lenguajes funcionales posteriores han incorporado el sistema de tipos de ML como MIRANDA y HASKELL.

Al mismo tiempo que se desarrollaban ML y FP se realizaban los trabajos de Turner en torno al SASL [Turner 76] KRC [Turner 81] y MIRANDA [Turner 85]. Turner defiende el valor de la evaluación perezosa, las funciones de orden superior y el uso de ecuaciones recursivas como ayuda sintáctica para el λ-cálculo. En KRC se introducen las abstracciones de conjuntos y en MIRANDA el sistema de tipos presente en ML.

Finalmente, se pueden citar el HOPE [Burstall & al. 80] por su popularidad y sencillez y el HASKELL [Hudak 89] un intento de hacer un "lenguaje funcional normalizado" ante la excesiva proliferación de lenguajes funcionales.

1.2.2. SISTEMAS DE REESCRITURA DE TERMINOS

La teoría de los **Sistemas de Reescritura de Términos** (SRT) constituye una alternativa para extender las técnicas de automatización del razonamiento a los dominios de naturaleza esencialmente algebraica.

La Reescritura ha sido adoptada como un procedimiento para extender los métodos de la Programación Lógica sobre aquellos problemas cuya especificación sea esencialmente funcional. Suministra, por tanto, un punto de vista diferente y va a ser más útil que el λ-cálculo en nuestro trabajo.

De otro modo, la teoría de los SRT puede ser entendida como un modelo abstracto de computabilidad obtenido como resultado de analizar los procedimientos automáticos de demostración para la lógica ecuacional [Hoffmann & O'Donnell 82].

Las técnicas de reescritura encontraron sus primeras aplicaciones en la Informática en relación con los problemas de verificación y ejecución simbólica de las especificaciones formales de Tipos Abstractos Algebraicos. Los SRT se estudian en el marco de las especificaciones algebraicas clásicas desarrolladas para el estudio de los tipos abstractos de datos: las **álgebras heterogeneas** [Goguen & Thatcher & Wagner 78].

Para nuestro trabajo, un lenguaje de primer orden, podemos limitarnos a álgebras con un género. Un desarrollo completo para más géneros puede verse en [Goguen & al 77] y [Huet & Oppen 80].

Una **signatura** es un conjunto $\Sigma = \bigcup_n \Sigma_n$ donde cada Σ_n para $n \in \omega = \{ 0, 1, 2, \ldots \}$ es un conjunto de símbolos.

$f \in \Sigma_n$ se dice un símbolo de operación de **aridad** n.

Un símbolo $e \in \Sigma_0$ se dice una **constante** (aridad 0).

Una **Σ-álgebra** A es un par <A, I> donde: 1) A es un conjunto que se dice el **soporte** del álgebra. 2) I es una función interpretación que asigna a cada símbolo $f \in \Sigma_n$ una operación $f_A : A^n \longrightarrow A$. Si $e \in \Sigma_0$ entonces e_A es una constante del soporte A.

Un **Σ-morfismo** h de una Σ-álgebra A en una Σ-álgebra B es una aplicación $h: A \longrightarrow B$ que conserva las operaciones:

1) si $e \in \Sigma_0$ entonces $h(e_A) = e_B$

2) si $f \in \Sigma_n$ entonces $h(f_A(a_1,\ldots,a_n)) = f_B(h(a_1),\ldots,h(a_n))$.

Σ-álgebras con los Σ-morfismos definidos constituyen una **categoría** que representamos por Alg(Σ). Hay un elemento **inicial** en esta categoría que representamos por T(Σ), que tiene la bien conocida propiedad de que para cada álgebra $A \in Alg(\Sigma)$ existe un único morfismo de T(Σ) en A. T(Σ) puede verse como el conjunto de términos (árboles) "básicos" construidos a partir de las operaciones de Σ. Coincide con el universo Herbrand de Σ siempre que es no vacío.

Sea X un conjunto de **variables**. Representamos por $\Sigma \cup X$ la signatura formada añadiendo a Σ el conjunto X considerado como un conjunto de constantes. $T(\Sigma \cup X)$ es un conjunto de términos con variables: la Σ-álgebra **libre** generada por X.

Un **Σ-término** t es una función parcial del monoide libre $\mathbb{N}_+^*$ en $\Sigma \cup X$. Su dominio , que representamos por **O(t)**, es el conjunto de ocurrencias o **posiciones** de t.

Para $m \in O(t)$, t(m) se llama la **etiqueta** de m en t.

O(t) debe incluir:

(1) la posición vacía para el nodo raiz de t;

(2) los elementos m.i para i = 1,...,n representando los n nodos

hijos de m, cuando la posición m en el término t tiene etiqueta t(m) de aridad n.

Las posiciones de t están parcialmente ordenadas por $u < v$ si u es un prefijo de $v \iff \exists i$ tal que $v = u.i$

var(t) representa el conjunto de variables de t.

t/m representa el subtérmino de t en la posición m para $m \in O(t)$.

t[m←t'] representa el término obtenido reemplazando t/m por t' en t.

$O^+(t)$ representa el conjunto de posiciones no variables en un término.

Una **sustitución** σ es una aplicación tal que $\sigma(Z) = Z$ para casi todas las variables Z (es decir, todas menos un número finito). Puede extenderse de manera natural a un Σ-endomorfismo de $T(\Sigma \cup X)$. Escribimos $t\sigma$ en vez de $\sigma(t)$ para $t \in T(\Sigma \cup X)$. Definimos el **dominio** y el **rango** de una sustitución σ como sigue:

$$dom(\sigma) = \{X_1, \ldots, X_n\} = \{X \in VS \mid X\sigma \neq X\}$$

$$ran(\sigma) = \bigcup_{X \in dom(\sigma)} var(X\sigma).$$

Decimos que dos términos t y $t' \in T(\Sigma \cup X)$ son **unificables** si existe una sustitución σ tal que $t\sigma = t'\sigma$. Si dos términos son unificables existe el mínimo unificador. Este elemento es único salvo renombramiento de variables [Robinson 65].

Si A es un álgebra una relación $\approx$ sobre los elementos de A se llama una **congruencia** si y sólo si para todo símbolo de función f y para todo $a_1, b_1, \ldots, a_n, b_n \in A$ se tiene

$$a_1 \approx b_1, \ldots, a_n \approx b_n \Longrightarrow f(a_1, \ldots, a_n) \approx f(b_1, \ldots, b_n).$$

Una **ecuación** es cualquier par de términos $t = t'$ donde $t, t' \in T(\Sigma \cup S)$ y las variables están universalmente cuantificadas.

Sea A una Σ-álgebra, A es un **modelo** para $t = t'$ si y sólo si para cada valoración $\rho : X \to A$ tenemos $t\rho = t'\rho$ y escribimos $A \models t = t'$.

La igualdad $=_{\mathcal{E}}$ generada por un conjunto de ecuaciones $\mathcal{E}$ es la más fina congruencia sobre $T(\Sigma \cup X)$ que contiene todos los pares $(t\sigma, t'\sigma)$ para $(t = t') \in \mathcal{E}$ y σ sustitución arbitaria.

$=_{\mathcal{E}}$ es una **teoría ecuacional** y $\mathcal{E}$ son los **axiomas** de la teoría.

Tenemos el resultado fundamental de Birkhoff:

$$Mod(\mathcal{E}) \models t = t' \iff t =_{\mathcal{E}} t'.$$

Una ecuación es **verdadera** para todos los modelos de $\mathcal{E}$ si y sólo si puede obtenerse a partir de las ecuaciones de $\mathcal{E}$ en un número finito de etapas de prueba. Esto da un procedimiento **semicomputable** si $\mathcal{E}$ es un conjunto recursivo de ecuaciones o axiomas.

Se puede construir un **modelo inicial** de una teoría ecuacional $I(\mathcal{E})$. Se define como el cociente de $T(\Sigma)$ por la congruencia $=_{\mathcal{E}}$ restringida a términos básicos. Y, efectivamente, hay un mórfismo único de $I(\mathcal{E})$ a cualquier álgebra de $Mod(\mathcal{E})$ que es precisamente el que define la interpretación de los términos representantes en cada álgebra.

En este marco teórico se pueden definir los conceptos fundamentales para la **semántica operacional** de los lenguajes funcionales **de primer orden**. Un **Sistema de Reescritura de Términos** es un conjunto dirigido de ecuaciones:

$$R = \{ l_i \rightarrow r_i \mid i \in I \}$$

tal que para todo $l_i \rightarrow r_i \in R$ se tiene que $var(r_i) \subseteq var(l_i)$.

La relación de reescritura $\rightarrow_R$ asociada con R se define como sigue:

$t \rightarrow_R t' \iff$ existe $m \in O^*(t)$,

existe una regla $l \rightarrow r \in R$,

existe una sustitución σ con $t/m = l\sigma$

tal que $t' = t[m \leftarrow r\sigma]$.

En este caso, t/m se llama un **redex** en t en la posición (redex) m bajo

la regla de reescritura $l \longrightarrow r$. Como hemos visto las únicas sustituciones requeridas para reescritura son la llamadas sustituciones en una dirección ("**one way matching**").

Los dos problemas clave en un SRT son la confluencia y la terminación.

R es **confluente** si y sólo si para todo r, s y t, $r \overset{*}{\longrightarrow} t$ y $s \overset{*}{\longrightarrow} t$ implica que hay algún t' tal que $r \overset{*}{\longrightarrow} t'$ y $s \overset{*}{\longrightarrow} t'$; donde $\overset{*}{\longrightarrow}$ significa la clausura reflexiva y transitiva de $\longrightarrow_R$.

R es **noetheriano** ó de terminación finita si y sólo si para ningún t hay una cadena de reducciones infinita $t = t_1 \longrightarrow t_2 \longrightarrow \ldots$

También es importante saber cuando un término está en forma **normal**. t' se dice una forma normal de t si y sólo si $t \overset{*}{\longrightarrow} t'$ y no hay ningún t" tal que $t' \longrightarrow t''$. El problema de las formas normales es que existan y sean únicas. Un SRT se dice **canónico** cuando todas las formas normales de cada término son idénticas.

Se prueba fácilmente que si hay confluencia las formas normales están univocamente determinadas. Y si hay confluencia y terminación las formas normales existen y son alcanzables con cualquier cadena de reducción.

Cuando R es un SRT finito que es confluente y noetheriano (termina) la teoría ecuacional $=_R$ es decidible ya que $s =_R t$ si y sólo si s' = t' (formas normales). La propiedad de confluencia es, en general, indecidible. El procedimiento de compleción de Knuth-Bendix [Knuth & Bendix 70] puede usarse para decidir si un sistema de reescritura noetheriano y finito es completo ó canónico. Se ha encontrado un algoritmo de unificación que usa "estrechamiento" cuando la teoría puede ser descrita mediante un SRT canónico asegurando terminación para una subclase de los sistemas de reescritura canónicos [Fay 79]

[Hullot 80]. Existen resultados de confluencia para SRT donde no se cumple la condición de terminación. [Huet 77] describe uno para términos lineales (un término t es lineal si ninguna variable ocurre en t más de una vez).

Un conjunto de ecuaciones puede verse como un sistema de reescritura de términos donde cada ecuación sirve como una regla de reescritura si un tal sistema de reescritura tiene la propiedad de confluencia. En [O'Donnell 85] se define un lenguaje ecuacional o lenguaje de reescritura de términos donde introduce una disciplina de constructores y plantea las ecuaciones como SRT (ecuaciones dirigidas) añadiendoles unas condiciones: 1) Las partes izquierdas son lineales. 2) Si dos partes izquierdas diferentes unifican la misma expresión entonces la parte derecha debe ser la misma. 3) Cuando dos partes izquierdas unifican con dos partes diferentes de la misma expresión, las dos partes no deben "solaparse". Estas restricciones son equivalentes a las planteadas por Huet y Levy para los sistemas lineales de reescritura de términos [Huet & Levy 79]. Con estas tres restricciones se demuestra que para cada término t_0 hay una forma normal como máximo t_n que puede obtenerse reduciendo t_0; y cualquier estrategia para elegir un redex que garantice que cada redex más exterior ("outermost redex") en una expresión es reducido, produce una forma normal siempre que exista. Ejemplos en esta refencia muestran que las tres condiciones anteriores no son arbitrarias y están conectadas con problemas de computación en paralelo. Las ecuaciones consideradas de esta manera con una semántica "consecuencia lógica" semejante a la de la programación lógica pueden utilizarse para escribir programas como conjuntos de ecuaciones, para diseñar

intérpretes de lenguajes (como LISP), para producir implementaciones correctas de tipos de datos, para diseñar "pruebas" de teoremas, ...

1.2.3. EL INDETERMINISMO EN LENGUAJES FUNCIONALES

Las especificaciones algebraicas pueden generalizarse en el sentido de abarcar **operaciones indeterministas**. El indeterminismo ha sido estudiado simulándolo dentro de las especificaciones algebraicas clásicas [Subrahmanyam 81] y [Broy & Wirsing 81]. Otros trabajos más recientes [Kaplan 88] han intentado estudiar el indeterminismo cambiando los conceptos básicos de las especificaciones algebraicas introduciendo el concepto de multi-álgebra y usando un lenguaje de especificación indeterminista.. Vamos a seguir en pocas lineas la aproximación al problema que hace [Hussmann 88].

Como hemos visto, el éxito de las especificaciones algebraicas ecuacionales se basa en gran medida en la existencia de intérpretes que usan técnicas de reescritura de términos para asignar una semántica operacional al conjunto de axiomas. El principal requisito para un tal tratamiento de las ecuaciones es la condición de confluencia. La propuesta de Hussmann es utilizar **sistemas de reescritura de términos no confluentes** como un lenguaje de especificación para especificaciones algebraicas indeterministas.

El primer problema es como incluir la indeterminación en el concepto clásico de álgebra. Así, una operación en este álgebra A sería:

$$f_A : A \longrightarrow P^+ (A)$$

dónde $P^+(M) = \{ N \mid N \subseteq M \wedge N \neq \emptyset \}$ es el conjunto de las partes no vacías de M. Que el **input** sea un dato mientras que el **output** es un conjunto de datos tiene importantes consecuencias para un lenguaje de

especificación apropiado. Por ejemplo las variables lógicas pueden tomar valores básicos deterministas y no expresiones indeterministas.

Una **especificación algebraica indeterminista** debe estar basada en el concepto de multi-álgebra. Una **Σ-multiálgebra** A es un par <A,F> dónde A es el soporte no vacío; y dónde F es una familia de funciones:

$$F = (f_A)_{f \in F} \text{ con } f_A: A^n \longrightarrow P^+(A) \text{ para } f \in F \text{ de aridad n.}$$

La clase de todas las Σ-multiálgebras se llama MAlg(Σ).

La interpretación de términos en una Σ-multiálgebra se define por una extensión aditiva de las semánticas para los símbolos de función; mientras que las variables, como ya dijimos se interpretan como objetos del conjunto soporte.

Sea A una multiálgebra, sea $\beta : X \longrightarrow A$ una valoración. Una **interpretación** I_A^β es una aplicación de $T(\Sigma \cup X) \longrightarrow P^+(A)$ dónde I_A^β se define inductivamente como sigue:

(1) Si $t = x \in X$ entonces $I_A^\beta [t] = \{ \beta(x) \}$.

(2) Si $t = f(t_1, \ldots, t_n)$ dónde $f \in F$ de aridad n:

$$I_A^\beta [t] = \{ y \in f_A(x_1, \ldots, x_n) \mid x_i \in I_A^\beta [t_i] \ 1 \leq i \leq n \}.$$

Una regla $t_1 \longrightarrow t_2$ es **válida** en A ($A \models t_1 \longrightarrow t_2$) si y sólo si para todas las valoraciones β se tiene que:

$$I_A^\beta [t_1] \supseteq I_A^\beta [t_2]$$

Una **especificación algebraica indeterminista** es un par T = (Σ, R) dónde Σ es una signatura y R es un conjunto finito de reglas.

Una multiálgebra A es un **modelo** de T si y sólo si para todas las reglas $\Phi \in R$, $A \models \Phi$.

Representamos por Mod(R) la clase de todos los modelos de R.

Pueden usarse las técnicas de los sistemas de reescritura para ver si un conjunto confluente de reglas de reescritura se deducen de los

axiomas de la teoría. Es decir, planteamos si se cumple el teorema de Birkhoff:

$$t_1 \rightarrow_R^* t_2 \iff Mod(R) \models (t_1 \rightarrow t_2)$$

Sin embargo, la corrección no vale (ver el ejemplo en [Hussmann 88]) sin restricciones sintácticas demasiado fuertes.

Hussmann soluciona el problema introduciendo un predicado "**determinicidad**" y añadiendo nuevas reglas al programa de la forma DET(t) para $t \in T(\Sigma \cup X)$ de modo que la regla sea válida en la multiálgebra si para todas las valoraciones la interpretación de t es única (el conjunto tiene exactamente un elemento). Esto obliga a definir unas nuevas reglas de derivación y nuevas especificaciones que complican notablemente los resultados de corrección y completitud débil. Ver [Hussmann 88] para los detalles.

1.3. LENGUAJES LOGICO-APLICATIVOS Y ESTRECHAMIENTO

1.3.1. JUSTIFICACION DE LA INTEGRACION

Podemos detenernos brevemente en lo que a nivel elemental es la principal diferencia entre ambos lenguajes [Reddy 86]: la direccionalidad input/output que está ligada al mecanismo de cómputo de cada uno de ambos tipos de lenguajes. Consideremos el ejemplo de la concatenación de listas. Un programa lógico es:

```
append([] ,Y ,Y).
append([A | X]) ,Y, [A | Z]) :-append(X ,Y ,Z).
```

Un programa funcional formulado ecuacionalmente es:

```
apd([] ,Y) = Y.
```

apd([A : X] ,Y) = [A : apd(X ,Y)].

En ambos casos podemos obtener el mismo resultado para *append([1], [2,3], L)* via resolución y para *apd([1] ,[2,3])* via reescritura. Sin embargo, si el objetivo propuesto fuese *append(L, M, [1,2,3])* sería necesario definir otro programa funcional para conseguir todas las descomposiciones de la lista [1,2,3]. El programa sería (por ejemplo en HOPE con abstracciones de conjuntos):

split(Z) = split1(Z) ∪ split2(Z).

split1(Y) = {([], Y)}.

split2([A : Z]) = {([A : X], Y) | (X, Y) ← split(Z)}.

La última igualdad puede leerse como "el conjunto de todos los pares ([A : X], Y) tal que (X, Y) es un miembro de split(Z)". Vemos que un sólo programa lógico corresponde a varios programas funcionales que tienen la misma información declarativa.

Consideremos el objetivo *append([X], [Y,Z], [1,2,3])*. Al aplicar resolución se producen ligaduras adecuadas con las variables de los patrones del objetivo. La correspondiente formulación del programa funcional llevaría a conseguir el mismo efecto con el generador:

([X], [Y, Z]) ← split ([1,2,3]).

Entonces mientras resolución utiliza la información implícita en los patrones para cortar con éxito la búsqueda, reescritura genera el conjunto entero lo que puede llevar a sobre-computación. Los problemas de sobre-computación pueden reducirse con evaluación perezosa pero pueden llegar a plantear problemas de terminación. Sin embargo, a veces ocurre lo contrario y resolución puede llevar a sobre-computación comparado con reducción. (Esto ocurre, desde luego, si no se escriben en el orden adecuado las claúsulas de los programas

en las actuales implementaciones del PROLOG).

Si consideramos el objetivo *append([1], L, M)* resolución produce la ligadura [1 : L]/M y deja L no acotada. Este efecto indefinido no puede conseguirse por un programa funcional a causa de que un programa funcional no puede producir variables libres como resultados. Este tratamiento de las variables lógicas permite resultados muy interesantes como el programa lógico "serialize" [Warren 77] o la "difference lists" [Clark & Tarnlund 77]. Unicamente extendiendo los programas funcionales permitiendo términos no básicos puede conseguirse el efecto pero esto lleva a nuevos planteamientos que alteran la direccionalidad de los programas funcionales. Se suele argumentar que la evaluación perezosa cuando se detiene tiene el mismo efecto de indefinición pero en relidad en este caso la información existe y está en suspenso mientras en el caso del programa lógico hay una falta de información.

Si nos fijamos en el mecanismo de cómputo, vemos que los lenguajes funcionales emplean **ajuste de patrones** ("pattern matching") mientras que los lenguajes lógicos emplean unificación. Como unificación **subsume** ajuste de patrones esto da a los programas lógicos capacidades que no poseen los lenguajes funcionales. Cuando se produce una unificación entre el objetivo y la cabeza de la claúsula se pueden establecer dos sustituciones: una sustitución "input" que liga las variables de la cabeza de la claúsula con términos del objetivo, y una sustitución "output" que liga las variables del objetivo con términos. Con "pattern matching" sólo se pueden obtener sustituciones "input".

Es bastante natural por lo visto antes que las principales características de los lenguajes de programación lógica que se

pretendan añadir a los lenguajes funcionales sean el **comportamiento de las variables lógicas**, la **unificación** y el **comportamiento indeterminista**.

Por otra parte, otro argumento usado es que los lenguajes funcionales son muy adecuados para implementaciones en **paralelo**. La programación lógica tiene problemas para implementar el paralelismo debido a su no direccionalidad. Se han hecho intentos en el sentido de dotar de paralelismo a la programación lógica con el diseño de lenguajes de programación lógica concurrente [Shapiro 83]. Los problemas de paralelismo y sincronización se resuelven imponiendo condiciones al lenguaje que disminuyen el indeterminismo y aumentan el carácter funcional. La sincronización se consigue por medio de la introducción de anotaciones de las variables con posibles modificaciones en el algoritmo de unificación.

Resumimos los argumentos que se han usado para justificar la integración comparando ambas familias de lenguajes:

1.- Los lenguajes lógicos permiten la definición de **relaciones**. Sin embargo, algunos problemas pueden describirse de forma más natural usando **funciones**. Un lenguaje potente debe permitir la definición (y composición) de relaciones y funciones.

2.- Un programa funcional contiene mucho más **control** que el correspondiente programa lógico.

3.- Algunos lenguajes funcionales tienen técnicas muy poderosas de compilación e implementación. Estas técnicas pueden trasladarse a un lenguaje integrado.

4.- Los propiedades clásicas (funciones de orden superior, evaluación perezosa, tipos, polimorfismo, ...) de los lenguaje funcionales

desarrollados son dificiles de conseguir en lenguajes lógicos y pueden ser fáciles de trasladar a un lenguaje lógico-funcional.

5.- Existen buenos entornos de programación (por ejemplo LISP) que podrían aprovecharse en una integración adecuada.

La situación ideal sería un lenguaje que sumara las mejores propiedades de los dos estilos. Con el fin de mantener las ventajas de la programación declarativa debería poseer una semántica matemática (denotacional) clara y rigurosa basada en la lógica y una semántica operacional que se pueda implementar eficientemente y que se relacione con la semántica matemática a través de resultados de corrección y completitud. Dicho de otro modo, debería ser posible interpretar los programas como teorías y los cómputos como deducciones. Además de estas características fundamentales debería tener todas las facilidades de cómputo que se han mostrado eficaces en los recientes diseños de lenguajes: usar una disciplina de constructores (functores en PROLOG), poder utilizar con comodidad la igualdad (que sea eficientemente computable), permitir el uso de la negación lógica, permitir el uso de tipos, funciones de orden superior, y objetos infinitos (gracias a la posibilidad de evaluación perezosa), posibilidad de explotación del paralelismo y creación de entornos de programación, etc...

1.3.2. ESTRECHAMIENTO COMO SEMANTICA OPERACIONAL

La semántica operacional de un lenguaje lógico funcional tiene que símular los dos mecanismos de cómputo: SLD-resolución y reescritura. Esto se puede lograr, por ejemplo, integrándolos en un nuevo mecanismo llamado **estrechamiento** ("narrowing"). En el enriquecimiento de la programación lógica con funciones y ecuaciones

se ha usado estrechamiento como un modo de resolver ecuaciones [Fribourg 85] [Goguen & Meseguer 86] [Subrahmanyam & You 86]. El uso de estrechamiento para resolver ecuaciones ha dado origen a algoritmos de unificación extendidos en el contexto de la prueba de teoremas con igualdad [Fay 79] [Hullot 80]. Como dice Hullot, **estrechar una expresión es aplicarle la mínima sustitución tal que la expresión resultante es reducible y entonces reducirla.**

Consideremos un ejemplo propuesto en [Reddy 85]:

ap([], Y) = Y

ap([K ; A], Y) = [K ; ap(A, Y)].

Si consideramos la expresión ap(A, []), vemos que no se puede reducir (está en forma normal). Por otra parte, no puede considerarse un "**resultado**" ya que tiene todavía un símbolo de función. Para poderla reducir, hacemos una sustitución $\{A \longrightarrow []\}$ que hace que la expresión sea reducible; entonces obtenemos [] como resultado de la reducción. Observemos que otra sustitución estrechamiento $\{A \longrightarrow [K_1 ; A_1]\}$ sería posible y que, de forma más general, podríamos obtener muchas reducciones por estrechamiento:

$$ap(A,[]) \Longrightarrow_{\{A \to []\}} [];$$

$$ap(A,[]) \Longrightarrow_{\{A \to [K_1 ; A_1]\}} [K_1 ; ap(A_1,[])] \Longrightarrow_{\{A_1 \to []\}} [K_1];$$

$$ap(A,[]) \Longrightarrow_{\{A \to [K_1 ; A_1]\}} [K_1 ; ap(A_1,[])]$$

$$\Longrightarrow_{\{A_1 \to [K_2 ; A_2]\}} [K_1,K_2 ; ap(A_2,[])] \Longrightarrow_{\{A_2 \to []\}} [K_1,K_2];$$

.....

Hay que observar que aunque un estrechamiento individual no conserva el significado de una expresión el conjunto de todos los estrechamientos lo hace. El poder expresivo del mecanismo de estrechamiento es similar al de SLD-resolución: permite que las

funciones sean invertidas; es decir, permite resolver ecuaciones como *ap(A, B) = [1, 2, 3]*; y permite conseguir el mismo efecto que los programas lógicos sobre las "variables lógicas".

Definamos lo anterior de una manera más formal:

Sea $R = \{ l_i \longrightarrow r_i \mid i \in I \}$ un SRT. Sea t un término y V un conjunto finito de variables $\supseteq$ var(t). Supongamos que existe un subtérmino no variable de t que es unificable con la parte izquierda de una regla. O sea, $\exists u \in O^+(t)$ y $\exists\ l_i \longrightarrow r_i \in R$ y t/u es unificable con l_i; donde suponemos renombrada $l_i \longrightarrow r_i$ si es preciso de modo que $var(l_i) \cap V = \emptyset$. Sea σ el mínimo unificador de t/u y l_i. Decimos que σ es una **sustitución estrechamiento** de t respecto V. Sea:

$$t' = t\sigma\ [u \leftarrow r_i\sigma] = (\ t[u \leftarrow r_i]\ )\sigma$$

decimos que t se **estrecha** a t' en la ocurrencia u usando $l_i \longrightarrow r_i$.

1.3.3. ALGUNOS RESULTADOS DE LA INTEGRACION

Podemos señalar tres grandes estrategias desde las que se ha intentado resolver el problema de la integración:

1.3.3.1. Suma de lenguajes.

Lo que parece más sencillo y que por tanto primero se intentó es juntar dos lenguajes (uno lógico y otro funcional) mediante un "interfaz" de modo que uno de ellos sea el "huésped" del otro. Los dos lenguajes pueden comunicarse a través de las estructuras de datos del "anfitrión". Para el usuario el sistema se comporta como un todo, aprovechando todas las facilidades del entorno de programación del anfitrión.

El lenguaje más conocido en esta linea es LOGLISP [Robinson & Sibert 82]. En este caso, el lenguaje lógico se implementa en LISP. A través de un sistema añadido a LISP llamado LOGIC se ofrecen

funciones LISP especificas para actualizar y manejar la base de datos del programa y consultar objetivos al programa lógico. Las expresiones normales de LISP pueden contener referencias a estas otras funciones. Un ejemplo de expresión LISP para consultar el programa lógico es:

$$(\text{ALL } (x_1 \ldots x_m)\ C_1 \ \ldots\ C_n)$$

donde $C_1,\ldots,C_n$ es una representación LISP de un objetivo y $x_1,\ldots,x_m$ son las variables de este. La función ALL devuelve la lista de todas las tuplas de variables (como lista) que son solución del objetivo con el programa lógico. Por tanto, el resultado de una llamada al programa lógico es una estructura de datos de LISP, que puede ser procesado por otras funciones LISP. Además es posible usar el programa LISP dentro del programa lógico, ya que cualquier expresión LISP puede aparecer en un objetivo o en el cuerpo de una claúsula.

El lenguaje resultante puede ser visto como una combinación de lenguaje y metalenguaje donde el programa lógico es el lenguaje y LISP el metalenguaje. El lenguaje resultante es muy poderoso ya que permite extender a través de funciones del metalenguaje las reglas de inferencia del lenguaje. Sin embargo, el paso de un, lenguaje a otro requiere interfaces de comunicación, ya que sólo las estructuras de datos de LISP son las que pueden usarse para la comunicación.

Sin embargo como crítica del sistema, Robinson señala en posteriores trabajos que no es deseable tener dos sistemas lógicos diferentes y que es preferible tener alguna forma de programación lógica pura.

Una propuesta similar a la anterior donde el lenguaje anfitrión es PROLOG es APPLOG [Cohen 86]. En este caso se tiene un lenguaje funcional puro sobre PROLOG y un doble interfaz: "goal" para

ejecutar objetivos PROLOG y "eval" para evaluar expresiones funcionales.

1.3.3.2. Enriquecimiento de un lenguaje con las caracteristicas que le faltan del otro.

En esta linea está diseñado HOPE con abstracciones de conjunto [Darlington & al. 86] del cual es un ejemplo el programa "split" anterior (1.3.1). La idea de añadir indeterminismo en el cómputo de funciones puede llevar a considerar funciones y predicados multivaluados como funciones deterministas que devuelven un conjunto como resultado. La definición de estas abstracciones de conjuntos se hace intensionalmente como conjunto de soluciones de un cierto sistema de restricciones. La unión de conjuntos está permitida como operación.

Este modelo se desarrolla dentro de HOPE con lo que se preservan nociones como evaluación perezosa, objetos infinitos y orden superior. La estrategia de evaluación usa esencialmente estrechamiento. Un trabajo posterior [Darlington & Guo 89] aporta una base matemática a este modelo definiendo claramente una semántica declarativa y una semántica operacional, estableciendo un teorema de completitud que relaciona ambas.

Por su parte Reddy [Reddy 85] [Reddy 86] propone usar estrechamiento (y estrechamiento perezoso) como semántica operacional para conseguir que un lenguaje funcional asuma los beneficios de la programción lógica. El lenguaje de Reddy está basado en el λ-cálculo y utiliza constructores. También desarrolla una semántica declarativa basada en retículos.

Otro lenguaje en esta linea es HASL [Abramson 86] un descendiente del lenguaje funcional SASL [Turner 76]. Introduce una

expresión de ligadura condicional basada en unificación en un sólo sentido. Esta expresión es un operador explícito para un uso limitado de la unificación, que admite un uso condicional. El lenguaje admite objetos infinitos (mediante evaluación perezosa) y orden superior mediante aplicación parcial de las funciones.

El lenguaje funcional MIRANDA [Turner 85] incorpora una construcción de listas, las llamadas ZF-expresiones que son en cierto modo una forma simplificada de abstracciones de conjunto.

1.3.3.3. Integracion de lenguajes.

Esta estrategia trata de hacer la integración de la programación lógica y funcional buscando un modelo teórico que comprenda ambos paradigmas como casos particulares. En realidad es la única estrategia que asegura una plena integración.

El marco teórico para la integración es la lógica ecuacional; o sea la lógica con igualdad. El problema es la carencia de igualdad en la programación lógica. Los primeros intentos se hicieron en la linea de dotar de igualdad a la programación lógica mediante su implementación [Kornfeld 86]. Sin embargo, el problema con la igualdad es la complejidad del procedimiento de refutación. Parecen necesarias algunas restricciones a la igualdad que se computa para retener las propiedades computacionales de los programas lógicos. Ya vimos que los SRT permiten entender los lenguajes funcionales como lenguajes lógicos basados en la lógica ecuacional.

La idea de EQLOG [Goguen & Meseguer 86] es construir un lenguaje que unifique la programación relacional (de las claúsulas Horn) con la programación funcional (basada en la igualdad). Esto se hace construyendo la más pequeña lógica que comprende ambas cosas:

claúsulas Horn e igualdad. Utiliza en la semántica operacional unificación extendida y estrechamiento. Considerando separadamente el sistema de reescritura que define la parte funcional del programa y las claúsulas Horn para los predicados el mecanismo de cómputo es SLD-resolución pero usando unificación extendida en vez de unificación normal para decidir la claúsula que se utiliza. Además la igualdad se puede usar como un predicado más.

Esta lógica se combina con un mecanismo de módulos para conseguir un tratamiento conveniente de las abstracciones de datos como en OBJ [Goguen & Tardo 79]. También admite un tratamiento de subgéneros con mecanismos de herencia. La semántica declarativa está basada en modelos iniciales. Sin embargo, EQLOG no tiene una semántica operacional completa ni está implementado, por lo que se dice que es más un lenguaje de especificación que de programación.

Otra forma de trabajar que se ha intentado es la utilización de sistemas de reescritura condicional. El lenguaje de [Dershowitz & Plaisted 85] es uno de los trabajos originales en utilizar reglas ecuacionales condicionales. No aporta una semántica declarativa pero aporta una semántica operacional: el mecanismo de cómputo utiliza "estrechamiento" (uso de una regla con unificación) para dar un paso de reducción y "simplificación" (uso de una regla con reescritura). Los pasos de "estrechamiento" admiten "backtracking" pero los de simplificación no.

Otros resultados interesantes en la integración de la programación lógica y funcional son: FUNLOG [Subrahmanyam & You 86], LEAF [Barbuti & al. 86], K-LEAF [Levi & al. 87] y BABEL [Moreno & Rodriguez 89].

1.3.4. REVISION DE UN EJEMPLO: BABEL

El lenguaje BABEL diseñado por Mario Rodriguez Artalejo y Juan José Moreno Navarro se podría definiir como un lenguaje funcional sin tipos basado en una disciplina de constructores que usa estrechamiento perezoso (para simular SLD-resolución) como mecanismo de evaluación (En un trabajo reciente se han incorporado tipos polimórficos y funciones de orden superior [Kuchen & al. 90]). Identifica los predicados con funciones booleanas lo que suministra dos valores veritativos que permiten un uso flexible de las conectivas lógicas y una mayor aproximación a la negación que en PROLOG (no hay negación clásica ya que se admite el valor booleano indefinido). Como se admiten datos infinitos la igualdad no es computable aunque se define una aproximación computable (similar a la "igualdad fuerte" de K-LEAF [Levi & al. 87]). Se define una semántica declarativa basada en los dominios de Scott y una semántica operacional basada en una versión perezosa de estrechamiento. Se han obtenido resultados de corrección y completitud relacionando ambas semánticas [Moreno & Rodriguez 89].

En el BABEL de primer orden, que es el que vamos a exponer aquí, se admiten dos tipos: *data* y *boolean*. En su sintáxis se utilizan cinco conjuntos disjuntos de símbolos: variables de datos DV, variables booleanas BV, constructores CS, símbolos de función FS, y símbolos de predicado PS. Esto permite distinguir entre términos de datos y términos booleanos y entre expresiones de datos y expresiones booleanas. Una regla BABEL es de la forma:

$$k\ (t_1, \ldots, t_2)\ :=\ \{\ C \longrightarrow\ \}\ E.$$

dónde k es un símbolo de función o un símbolo de predicado; C es una

expresión booleana dónde la existencia de { } significa que la guarda es opcional; y E es una expresión de datos o una expresión booleana según que k sea un símbolo de función o de predicado. Una regla se escribe abreviadamente L := R.

Una regla debe satisfacer dos restricciones: ser lineal por la izquierda (ninguna variable puede tener ocurrencias múltiples en la parte izquierda de una regla) y tener variables locales (si es el caso) sólo en las guardas (una variable se dice local si ocurre en la parte derecha de la regla y no en la izquierda). Si $X_1,\ldots,X_r$ son las variable en la parte izquierda e $Y_1,\ldots,Y_s$ son las variables locales, el significado lógico de una regla es:

$$\forall X_1 \ldots \forall X_r \forall Y_1 \ldots \forall Y_s \; (\{ C \Longrightarrow \} \, k(t_1,\ldots,t_n) \equiv E \;)$$

ó, equivalentemente:

$$\forall X_1 \ldots \forall X_r \; (\exists Y_1 \ldots \exists Y_s \; \{ C \Longrightarrow \} \; k(t_1,\ldots,t_n) \equiv E \;)$$

dónde ≡ significa la idéntidad semántica.

Un programa BABEL Π es cualquier conjunto recursivamente enumerable de reglas BABEL que satisface una restricción de no-ambigüedad: Dadas dos reglas cualesquiera en Π para el mismo símbolo k

$$k(t_1,\ldots,t_n) := \{ B \longrightarrow \} \; E.$$

$$k(s_1,\ldots,s_n) := \{ C \longrightarrow \} \; F.$$

alguna de las siguientes condiciones se debe satisfacer:

(a) No superposición: $k(t_1,\ldots,t_n)$ y $k(s_1,\ldots,s_n)$ no son unificables.

(b) Fusión de cuerpos: $k(t_1,\ldots,t_n)$ y $k(s_1,\ldots,s_n)$ tienen un u.m.g. θ tal que Eθ y Fθ se identifican.

(c) Incompatibilidad de guardas: $k(t_1,\ldots,t_n)$ y $k(s_1,\ldots,s_n)$ tienen un u.m.g. θ tal que (B, C)θ es proposicionalmente insatisfacible.

Estas condiciones tienen que ver (aunque son más liberales) con las citadas en el apartado anterior para los programas ecuacionales de O'Donnell [O'Donnell 85] y con las consideradas por Huet y Levy [Huet & Levy 79] para los sistemas de reescritura de términos lineales no-ambiguos; y permiten obtener confluencia en ausencia de terminación.

El mecanismo de cómputo de BABEL está basado en **estrechamiento perezoso**. Para computar una expresión se distingue dentro de las ocurrencias redex las ocurrencias redex perezosas. La perezosidad se consigue seleccionando ocurrencias redex más exteriores y yendo al interior de la expresión sólo cuando es demandado por las partes izquierdas de las reglas pendientes.

Sea Π un programa y E una expresión. Si u es una ocurrencia redex para una variante de una regla L := R en Π tal que M/u unifica con éxito con L con u.m.g. $\sigma = \sigma_m \cup \sigma_t$, decimos que M estrecha en una etapa a la nueva expresión $M[u \leftarrow R]\sigma$ y escribimos:

$$M \overset{N}{\longrightarrow}_{\sigma_m} M[u \leftarrow R]\sigma.$$

Si σ_m es la sustitución identidad escribimos:

$$M \longrightarrow M[u \leftarrow R]\sigma_t.$$

y decimos que M reescribe en una etapa a $M[u \leftarrow R]\sigma_t$.

Si u es una ocurrencia redex perezosa escribimos:

$$M \overset{N\text{-}l}{\longrightarrow}_{\sigma_m} M[u \leftarrow R]\sigma.$$

y $$M \overset{-l}{\longrightarrow}_{\sigma_m} M[u \leftarrow R]\sigma.$$

respectivamente.

Definiciones para reducciones generales por estrechamiento y estrechamiento perezoso, y para reescritura y reescritura perezosa son generalizaciones obvias.

Sea Π un programa. Para cualquier derivación por estrechamiento (no necesariamente perezoso)

$$M \xrightarrow[\sigma_m]{N}{}^{*} N$$

definimos la salida como el par $(|N|, \sigma_m)$, dónde $|N|$, la cáscara de N, es el resultado y σ_m la respuesta. El resultado puede estar parcialmente definido si alguna ocurrencia de $|N|$ es $\perp_{BOOL}$ ó $\perp$. Dos tipos de salidas son importantes considerar: salidas funcionales, cuando $|N|$ es un término básico y σ_m es la sustitución identidad; y salidas tipo PROLOG cuando $|N|$ es *true*.

Se puede construir una semántica declarativa para BABEL basada en los dominios de Scott. Como dominio booleano se usa el conocido c.p.o. plano ("flat") BOOL. Se puede definir el dominio de Herbrand H que está formado por todos los árboles (finitos o infinitos) con nodos etiquetados por los símbolos del conjunto $CS \cup \{ \perp \}$, dónde $\perp$ es un nuevo símbolo de aridad 0. Una interpretación de Herbrand tiene como dominio el dominio de Herbrand e interpreta los constructores como árboles "libres".

Se puede declarar una semántica punto fijo definiendo un operador consecuencia inmediata T_π como en programación lógica. La corrección de la semántica operacional se obtiene probando que: cualquier reducción por estrechamiento (no necesariamente perezoso) computa una salida correcta. El resultado recíproco (completitud) afirma que cualquier salida correcta es subsumida por otra salida que puede ser computada via estrechamiento perezoso.

El como programar un mismo problema (la concatenación de listas) en HOPE, PROLOG y BABEL puede mostrarnos las diferencias de planteamiento señaladas en los tres apartados anteriores.

Un programa HOPE es el siguiente:

typevar alpha

data Lista(alpha) == nil ++ cons(alpha # Lista (alpha))

dec append : (lista (alpha) # Lista (alpha)) → Lista (alpha)

--- append (nil, L) <= L ;

--- append (cons (X, L_1), L_2) <= cons (X, append(L_1, L_2)) ;

"alpha" es un tipo variable, un identificador que representa algún tipo. "Lista(alpha)" es la descripción recusiva de una lista genérica: "Una lista de objetos (alpha) es una estructura de datos que es vacía ó consta de un objeto (la cabeza de la lista) y otra lista de objetos (la cola de la lista)". Esto permite definir la función append sobre cualquier tipo de objetos. Hay comprobación de tipos. El programa computa mediante reescritura de manera unidireccional respondiendo a objetivos de la forma:

append([1, 2], [3, 4, 5]).

Un programa PROLOG es:

append (nil, L, L).

append ([X |L_1], L_2, [X | L_3]) :- append (L_1, L_2, L_3).

No hay tipos y las listas pueden contener cualquier elemento. Se pueden plantear objetivos de la forma:

append(X, Y, [1, 2, 3])

para dividir la lista [1, 2, 3] en dos partes. Esto no es posible en programación funcional: como vimos sería necesario escribir otro programa distinto.

Un programa en BABEL es:

append [] Ys Zs := (Zs = Ys) → true.

append [X|Xs] Ys [Z|Zs] := (Z=X ∧ (append Xs Ys Zs)) → true.

Este programa admite como en PROLOG usos multiples: se puede usar para concatenar dos listas ó para dividir una lista en dos. Los cómputos se realizan utilizando estrechamiento perezoso (el papel de las guardas es esencial para lograr la simulación de SLD-resolución). Un objetivo como *append Xs Ys [1, 2, 3]* da como salida $(true, \sigma_m)$ dónde:

$$\sigma_m = \{Xs \rightarrow [1],\ Ys \rightarrow [2,\ 3]\}.$$

2. UN FORMALISMO PARA LA PROGRAMACION LOGICA CON FUNCIONES INDETERMINISTAS. SISTEMAS DE REESCRITURA REGULARES.

En este capítulo, vamos a definir un formalismo para el estudio de la programación lógica con funciones indeterministas. En esencia, puede considerarse un sistema de reglas de reescritura con disciplina de constructoras y variables libres. Además, tenemos que prescindir de la condición de confluencia debido a la existencia de funciones indeterministas.

Considerado como lenguaje de programación nuestro formalismo se encuentra en el ámbito de la integración de la programación lógica y funcional. Es un pariente de BABEL, lenguaje que integra ambos paradigmas y que ya hemos hemos descrito en 1.3.4. Las restricciones que se imponen a las reglas de los programas BABEL no son necesarias en nuestro lenguaje al no tener que asegurar la confluencia. También se ha prescindido de las reglas predefinidas, existentes en aquél, para las conectivas lógicas, los condicionales y la "igualdad fuerte". En nuestro caso, las definiremos según las necesidades del programa. (Ver los ejemplos del final del capítulo)

El estilo de definición de los programas es esencialmente funcional y los predicados son tratados como funciones booleanas. Las características más potentes de la programación lógica, por ejemplo el comportamiento de las variables lógicas, se consiguen merced a la

existencia de funciones indeterministas y variable libres.

Para empezar consideramos un **vocabulario** con los siguientes tres conjuntos disjuntos de símbolos:

variables	X, Y, Z,... $\in$ VAR
constructores	c, d,..... $\in$ CS con la **aridad** correspondiente.
operadores	f, g,..... $\in$ FS con la **aridad** correspondiente.

VAR es un conjunto **numerable**.

La distinción entre constructores y operadores es bien conocida en los lenguajes de programación funcional. Esta distinción permite, por ejemplo, caracterizar teorías ecuacionales definidas por programas funcionales:

Por ejemplo, si *nil* y *cons* son constructores, las ecuaciones:

append(nil,X) = X

append(cons(X,Y),Z) = cons(X,append(Y,Z))

cuyas partes izquierdas no contienen anidamiento de operadores son definiciones de una función; mientras que una ecuación como

append(append(X,Y),Z) = append(X,append(Y,Z))

es una propiedad del programa.

Como es sabido, los constructores representan **funciones libres** y son análogos a los functores de PROLOG.

$\Sigma = CS_\Sigma \cup FS_\Sigma$ es la **signatura** que varía con el programa considerado. En particular, CS^n_Σ (FS^n_Σ) $n \geq 0$, representa los constructores (operadores) de aridad n. Se verifica:

$$CS_\Sigma = \bigcup_n CS^n_\Sigma; \qquad FS_\Sigma = \bigcup_n FS^n_\Sigma.$$

2.1.-DEFINICION:

Definimos los **términos** del lenguaje por recursión:

t ::= X % variable

| c % constante (constructor de aridad 0)

| $c(t_1,\ldots,t_n)$ % construcción.

$Term_\Sigma$ representa el conjunto de Σ-términos.

Definimos las **expresiones** del lenguaje también por recursión:

e ::= t % término

| $c(e_1,\ldots,e_n)$ % aplicación de constructor

| $f(e_1,\ldots,e_n)$ % aplicación de operador

Exp_Σ representa el conjunto de Σ-expresiones.

Una expresión se dice **lineal** si ninguna variable aparece en ella más de una vez.

Una expresión se dice **cerrada** si no incluye variables ■

Por convenio, suponemos que $CS^0_\Sigma \neq \emptyset$. O sea, el lenguaje posee al menos una constante: $c_0 \in CS^0_\Sigma$.

2.2.-DEFINICION:

Una **regla** de lenguaje es de la forma:

$$f(t_1,\ldots,t_n) \longrightarrow e$$

donde $f \in FS^n_\Sigma$, $n \geq 0$, $t_i \in Term_\Sigma$ para $1 \leq i \leq n$, y $e \in Exp_\Sigma$.

Llamaremos a esta clase de reglas **reglas de reescritura regulares**.

Una regla tiene que cumplir la condición de **linealidad por la izquierda**: una variable no puede aparecer más de una vez en la parte izquierda de la misma. O sea, el lado izquierdo de una regla es lineal.

La linealidad por la izquierda es necesaria para excluir reglas como la siguiente:

$$eq(X, X) \longrightarrow true$$

que define una función no computable: "la identidad entre objetos posiblemente infinitos".

Podemos abreviar una regla escribiendo solamente:

$$l \longrightarrow r$$

para distinguir las partes izquierda l y derecha r de la misma.

En la parte derecha admitimos la existencia de variables que no aparezcan en la parte izquierda: puede ocurrir que $var(r) \setminus var(l) \neq \emptyset$. Las variables $Y \in var(r) \setminus var(l)$, si las hay, se llaman **variables libres**. Las variables libres no existen en sistemas de reglas de reescritura ordinarios. Su introducción en reglas de reescritura regulares aumenta la potencia y expresividad del lenguaje y permite simular el comportamiento de los programas lógicos.

Un **programa** Π es un sistema de reglas de reescritura regulares. ■

Los siguientes ejemplos muestran las posibilidades de nuestro lenguaje. Como ya comentamos en el capítulo 1 (ver 1.3.2) la semántica operacional más adecuada para los lenguajes que integran los paradigmas funcional y lógico es **estrechamiento**; el cuál permite simular al mismo tiempo SLD-resolución y reescritura. Este es el mecanismo de cómputo que aplicamos, aún de manera informal, y que da las soluciones que citamos en los ejemplos.

EJEMPLO 1:

En nuestro lenguaje podríamos considerar un programa para **ordenar una lista de números naturales** como el siguiente:

La "idea" es que para ordenar una lista, se genera una permutación suya y se comprueba que está ordenada. La generación de una permutación es "indeterminista". Son necesarias algunas reglas adicionales para definir las conectivas lógicas que utilizamos y la "igualdad". En lo que sigue, se escriben, para hacerlo más legible, algunas funciones en forma infija.

constructors:

[] / 0

[. : .] / 2

0 / 0

succ / 1

true / 0

false / 0

operators:

sort / 1

permutation / 1

ordered / 1

insert / 2

if..then / 2

$\wedge$ / 2

== / 2

< / 2

rules:

sort(Xs) $\longrightarrow$ if (Ys == permutation(Xs)) $\wedge$ ordered(Ys)
then Ys

permutation([]) $\longrightarrow$ []

permutation([X : Xs]) $\longrightarrow$ insert(X, permutation(Xs))

insert(X, Ys) $\longrightarrow$ [X : Ys]

insert(X, [Y : Ys]) $\longrightarrow$ [Y : insert(X, Ys)]

ordered([]) $\longrightarrow$ true

ordered([Y]) $\longrightarrow$ true

ordered([Y_1, Y_2 : Ys]) $\longrightarrow$ (Y_1 < Y_2) $\wedge$ ordered([Y_2 : Ys])

(if true then Z) $\longrightarrow$ Z

(false $\wedge$ Y) $\longrightarrow$ false

(true $\wedge$ Y) $\longrightarrow$ Y

(0 == 0) $\longrightarrow$ true

(succ(X) == succ(Y)) $\longrightarrow$ (X == Y)

([] == []) $\longrightarrow$ true

([X : Xs] == [Y : Ys]) $\longrightarrow$ (X == Y) $\wedge$ (Xs == Ys)

0 < succ(Y) $\longrightarrow$ true

succ(X) < succ(Y) ⟶ X < Y

Este es un programa del tipo *Generate-and-Test* bastante habituales en programación lógica (ver 1.1.3).

Se pueden plantear distintos objetivos. Por ejemplo, si el objetivo es *sort([3,5,2,6])* se obtiene como resultado la lista ordenada *[2,3,5,6]* después de generar de manera indeterminista listas, que son permutaciones distintas de la lista inicial *[3,5,2,6]*, y comprobar si están ordenadas.

Si planteamos un objetivo como *sort([3,5,2,6]) == Xs* devuelve como resultado *true* obteniendo en el transcurso de la computación la sustitución respuesta $\{Xs \rightarrow [2,3,5,6]\}$.

Por otra parte, un objetivo como *sort(Xs) == [2,3,5,6]* tiene multiples posibilidades: devuelve como resultado *true* y genera distintas sustituciones respuesta; por ejemplo una de ellas $\{Xs \rightarrow [3,5,2,6]\}$.

La definición de igualdad que basta para nuestro programa puede ser ineficiente en algunos casos como mostramos en un ejemplo del capítulo 7. En ese mismo capítulo se indica una posible solución al problema.

EJEMPLO 2:

Un segundo ejemplo, quizás más interesante, muestra las posibilidades del lenguaje para extraer toda la potencia que permite el uso conjunto de predicados, funciones, variables lógicas, ... En él puede verse como la existencia de funciones indeterministas aumenta notablemente la **expresividad** del lenguaje.

El programa permite obtener, dada una lista de números naturales, la lista formada por el *rango* de cada uno de ellos; donde el rango de un elemento es el número de elementos menores que él, que ocurren en la lista.

Por ejemplo *ranks_of([3,5,1,3,7,5,4]) = [1,3,0,1,4,3,2].*

constructors:

empty / 0

mkdic / 4

[] / 0

[. : .] / 2

0 / 0

succ / 1

true / 0

operators:

ranks_of / 1

ranks / 2

complete / 1

allocated / 3

lookup / 2

== / 2

if..then / 2

∧ / 2

< / 2

rules:

```
ranks_of(Xs) ⟶ ranks(Dict,Xs)

ranks(Dict,[]) ⟶ if complete(Dict) then []

ranks(Dict,[X : Xs]) ⟶ [lookup(Dict,X) : ranks(Dict,Xs)]

complete(Dict) ⟶ allocated(Dict,0,N)

allocated(empty,M,N) ⟶ N == M

allocated(mkdic(Key,Value,Left,Right),M,N) ⟶
          ( allocated(Left,M,Value) ∧
          allocated(Right,succ(Value),N) )

lookup(mkdic(Key,Value,Left,Right),K) ⟶
          if Key == K then Value

lookup(mkdic(Key,Value,Left,Right),K) ⟶
          if K < Key then lookup(Left,K)

lookup(mkdic(Key,Value,Left,Right),K) ⟶
          if Key < K then lookup(Right,K)

(if true then X) ⟶ X
```

(true ∧ X) ⟶ X

(0 == 0) ⟶ true

(succ(X) == succ(Y)) ⟶ (X == Y)

([] == []) ⟶ true

([X ¦ Xs] == [Y ¦ Ys]) ⟶ (X == Y) ∧ (Xs == Ys)

(0 < succ(X)) ⟶ true

(succ(X) < succ(Y)) ⟶ (X < Y)

El programa construye la lista Ys "atravesando" la lista Xs una vez y consultando los "ranks" asociados a los miembros de Xs en un diccionario Dict. El diccionario es inicialmente desconocido y debe por tanto ser creado, lo que se consigue asignándole una variable lógica que crea aquél como un árbol binario ordenado en el que se van insertando los elementos de la lista y los "ranks" de cada elemento que son, a su vez, nuevas variables lógicas que se llenan de contenido al final gracias al operador (predicado) *allocated*.

EJEMPLO 3:

El tercer ejemplo se refiere a la sucesión de Fibonacci y se propone para mostrar las posibilidades del lenguaje para calcular con estructuras de datos infinitas. Las ventajas de la computación con este tipo de estructuras es bien conocida (ver como ejemplo [Hudak 89]).

Como ya se sabe el problema que resuelve esta sucesión es el de saber cuántas parejas (por ejemplo de conejos) se obtienen a partir de un momento determinado suponiendo que cada pareja produzca una nueva pareja cada mes, que a su vez puede reproducirse a partir del segundo mes. Se obtiene así la sucesión 1, 1, 2, 3, 5, 8, ... La obtención del n-simo término de la sucesión de Fibonacci es un típico ejemplo de

programación imperativa. Aquí planteamos una versión sencilla utilizando estructuras infinitas de datos que permite hacer distintos cómputos haciendo uso de evaluación perezosa:

constructors:

- [] / 0
- [. : .] / 2
- 0 / 0
- succ / 1
- true / 0
- false / 0

operators:

- fib_nbs / 0
- fib_nbs_from / 2
- fib_nb / 1
- member / 2
- == / 2
- add / 2
- ∨ / 2
- ∧ / 2
- < / 2

rules:

- fib_nbs ⟶ fib_nbs_from(succ(0),succ(0))
- fib_nbs_from(X,Y) ⟶ [X : fib_nbs_from(Y,add(X,Y))]
- fib_nb(X) ⟶ member(X,fib_nbs)
- member(X,[]) ⟶ false
- member(X,[Y : Ys]) ⟶ (X == Y) ∨ (member(X,Ys))
- add(X,0) ⟶ X
- add(X,succ(Y)) ⟶ succ(add(X,Y))
- (false ∨ X) ⟶ X
- (true ∨ X) ⟶ true
- (0 == 0) ⟶ true
- (succ(X) == succ(Y)) ⟶ (X == Y)
- (false ∧ X) ⟶ false
- (true ∧ X) ⟶ X

(0 < succ(X)) ⟶ true

(succ(X) < succ(Y)) ⟶ (X < Y)

Se puede plantear un objetivo como *fib_nb(5)* que da como resultado *true*. Por otra parte un objetivo como *fib_nb(X)* ∧ *(X < 6)* que da como resultado *true* y distintas posibles sustituciones respuesta para un cómputo que termina si se evalúa **perezosamente**.

EJEMPLO 4:

El ejemplo siguiente extraido de [Hussmann 88] muestra las dificultades que surgen al usar funciones indeterministas.

constructors:

zero / 0

succ / 1

operators:

add / 2

double / 1

or / 2

rules:

add(zero,X) ⟶ X

add(succ(X),Y) ⟶ succ(add(X,Y))

double(X) ⟶ add(X,X)

or(X,Y) ⟶ X

or(X,Y) ⟶ Y

Con reescritura "**outermost**" podemos obtener:

double(or(zero, succ(zero))) ⟹

add(or(zero, succ(zero)), or(zero, succ(zero))) ⟹

add(zero, or(zero, succ(zero))) ⟹

or(zero, succ(zero)) ⟹

succ(zero).

Consideremos informalmente una interpretación sobre los números naturales ℕ. Entonces, como *or* es una operación indeterminista:

$\|or(zero, succ(zero))\|_{\mathbb{N}}$

puede admitir como valores 0 y 1. Por tanto:

$\|double(or(zero, succ(zero)))\|_{\mathbb{N}}$

admite como valores totales 0 y 2. Vemos que es necesario estudiar los **grafos** de las correspondencias, al no ser las operaciones funciones (continuas). O también, de otro modo, se podría hacer correponder a una operación una función de $\mathbb{N}$ en $P^{+}(\mathbb{N})$ conjunto de partes no vacías de $\mathbb{N}$ (Ver 1.2.3). Estas reflexiones nos sugieren cual puede ser la semántica **denotacional** adecuada a nuestro lenguaje.

Por otra parte, la expresión:

$\|succ(zero)\|_{\mathbb{N}}$

sólo admite el valor total 1 ya que *succ* es un constructor, una función libre, y por tanto puede interpretarse como una función continua.

La existencia de esta incorrección usando reescritura "outermost" es debida a la necesidad de "compartir la información" en las partes derechas de las reglas. En el ejemplo el problema es la regla:

$double(X) \longrightarrow add(X, X).$

Para solucionar el problema podríamos limitarnos a reescritura "**innermost**"; pero esto limitaría las posibilidades de perezosidad y por tanto la computación con estructuras de datos infinitas.

Otra forma de solucionarlo es obligar a que las partes derechas fuesen lineales; pero esto haría el lenguaje demasiado restrictivo ya que, por ejemplo, las reglas clásicas del producto:

$product(zero, Y) \longrightarrow zero.$

$product(succ(X), Y) \longrightarrow add(product(X, Y), Y).$

no serían válidas.

Los problemas anteriores motivan la necesidad de usar **estrechamiento en grafos dirigidos acíclicos** (gda's) como semántica **operacional** ya que en ellos puede expresarse sin dificultad la compartición.

Volveremos a retomar este ejemplo en los capítulos 5 y 6.

3. ELEMENTOS DE LA TEORIA DE DOMINIOS.

Una vez definido el lenguaje nuestro primer paso es dotarlo de una semántica declarativa. En principio, como base para la semántica matemática del mismo podríamos usar el universo Herbrand de la programación lógica [Lloyd 87]. Debido a la existencia de objetos infinitos el universo de Herbrand no es adecuado y es necesario introducir los c.p.o.'s (ordenes parciales completos) para definir dominios semánticos que permitan reflejar el significado que buscamos. En concreto, usaremos dominios de Scott [Scott 82].

Este capítulo tiene por objeto establecer la base matemática necesaria para poder desarrollar en el próximo la semántica declarativa precisa para nuestro lenguaje. En primer lugar, apartado 1, recordamos brevemente algunas definiciones (y resultados relacionados con ellas) conocidas de la teoría de dominios que vamos a utilizar con algunas variaciones respecto a las habituales. Una exposición detallada de las mismas y, en general, de la teoría de dominios, puede encontrarse en [Scott & Strachey 71] [Scott 81] [Scott 82] [Bermudez 85] y [Mulmuney 86]. A continuación, en el segundo apartado, construimos detalladamente una clase de dominios adecuados para la semántica de funciones indeterministas que denominaremos dominios de funciones indeterministas o de **grafos**. Finalmente, en el tercer apartado, comparamos el dominio construido con algunos trabajos clásicos sobre semántica denotacional de lenguajes indeterministas y

demostramos que el dominio construido está estrechamente relacionado con los **dominios potencia**, especialmente con el de Hoare.

3.1 PRIMERAS DEFINICIONES Y RESULTADOS

Recordamos algunas definiciones y resultados conocidos.

Consideraremos un conjunto D y una relación de orden parcial $\ll_D$. De manera informal, cada elemento de D es un conjunto de "informaciones", y se puede dar un significado intuitivo a la relación $\ll_D$: si $x \ll_D y$ entonces todas las informaciones que son verdad de x lo son también de y.

3.1.1.-DEFINICION: Conjuntos Dirigidos

Sea D un conjunto y $\ll_D$ un orden parcial en D. Decimos que un subconjunto $S \subseteq D$ es **dirigido** si para todo $x, y \in S$ existe $z \in S$ tal que $x \ll_D z$ e $y \ll_D z$. ■

3.1.2.-DEFINICION: CPO's

$D = \langle D, \ll_D, \perp_D \rangle$ es un **orden parcial completo (cpo)** si y sólo si:

(i) Existe un elemento mínimo de D para $\ll_D$ que escribimos $\perp_D$.

(ii) Todo subconjunto dirigido $S \subseteq D$ tiene supremo $\bigsqcup_D S$ en D. ■

3.1.3.-DEFINICION: Elementos Finitos y Elementos Totales

Sea D un cpo. Un elemento $u \in D$ es **finito** en D si para todo subconjunto dirigido S de D tal que $u \ll_D \bigsqcup_D S$ existe algún $x \in S$ que cumple $u \ll_D x$.

Representaremos por Fin_D el conjunto de elementos finitos de D.

Un elemento $x \in D$ es **total** en D si y sólo si es la única cota superior de $\{x\}$; es decir, para cualquier y tal que $x \ll_D y$ se cumple $x = y$. ■

Podemos pensar en un elemento finito como en una cantidad de información que un proceso de cómputo usa o produce en un tiempo finito.

3.1.4.-LEMA: Propiedad de los Elementos Finitos

Sean $u_1, u_2, \ldots, u_n \in Fin_D$. Si existe el supremo de $\{u_1, u_2, \ldots, u_n\}$ entonces éste es finito: $\bigsqcup_D \{u_1, u_2, \ldots, u_n\} \in Fin_D$. ■

3.1.5.-DEFINICION: CPO ω-algebraico

Sea D un cpo. D es **algebraico** si para todo $x \in D$ el conjunto:

$$B_x = \{u \in Fin_D \mid u \ll_D x\}$$

es dirigido y además $x = \bigsqcup_D B_x$.

D es **ω-algebraico** si D es algebraico y Fin_D es numerable. ■

3.1.6.-DEFINICION: CPO Consistentemente Completo

Sea D un cpo.

Un subconjunto L de D es **consistente** si todo subconjunto finito de L está acotado en D.

D es **consistentemente completo** si todo subconjunto consistente de D tiene supremo en D. ■

3.1.7.-DEFINICION: CPO Acotadamente Completo

Sea D un cpo.

Un subconjunto M de D es **acotado** si existe algún elemento $d \in D$ tal que para todo $m \in M$, $m \ll_D d$.

D es **acotadamente completo** si todo subconjunto acotado de D tiene supremo en D. ■

3.1.8.-LEMA:

Un cpo es acotadamente completo si y sólo si es consistentemente completo. ■

3.1.9.-DEFINICION: Retículo Completo

Un conjunto parcialmete ordenado L es un **retículo completo** si existe el supremo e ínfimo de S para todo subconjunto S de L. ■

3.1.10.-LEMA:

Todo retículo completo es un cpo consistentemente completo. ■

3.1.11.-DEFINICION: Dominio

Un **dominio** es un cpo algebraico consistentemente completo. ■

El concepto de monotonía es esencial para la teoría de dominios. Una función monótona f es tal que no hay perdida de información en el sentido de que si x poseee menos información que y lo mismo ocurre entre f(x) y f(y). Una función continua es una función monótona que es consistente con el concepto de "supremo".

3.1.12.-DEFINICION: Funciones Monótonas y Continuas

Sean D y D' dominios. Sea $f : D \longrightarrow D'$ una función.

f es **monótona** si y sólo si

$$(x \ll_D y \Longrightarrow f(x) \ll_{D'} f(y) \text{ para todo } x,y \in D).$$

f es **continua** si y sólo si

para todo S dirigido, $S \subseteq D$, se cumple que f(S) es dirigido en D' y $f(\bigsqcup_D S) = \bigsqcup_{D'} f(S)$, donde $f(S) = \{f(x) \mid x \in S\}$.

Es inmediato que f continua $\Longrightarrow$ f monótona. ■

También son importantes conceptos relacionados con la presentación de los dominios.

3.1.13.-DEFINICION: Dominios Efectivamente Presentados

Sea D un dominio, $\mathbb{N}$ conjunto de números naturales. Una **enumeración** de Fin_D es una función sobreyectiva $u : \mathbb{N} \longrightarrow Fin_D$ tal que:

$$u_0 = \perp_D \qquad Fin_D = \{ u_i \mid i \in \mathbb{N} \}$$

donde escribimos u_i por u(i).

Sea F_i el conjunto codificado por $i \in \mathbb{N}$ en una biyección de $\mathbb{N}$ en el conjunto de las partes finitas de $\mathbb{N}$ (por ejemplo puede servir esta aplicación: F_i es el conjunto que cumple que $i = \Sigma_{k \in F_i} 2^k$). Llamamos S_i al subconjunto de Fin_D codificado por F_i; es decir,

$$S_i = \{u_j \mid j \in F_i\}.$$

Una enumeración u es una **enumeración efectiva** de D si los predicados en los naturales:

$$acot(i) \quad :\Longleftrightarrow S_i \text{ está acotado en D}$$

$$sup(n,i) :\Longleftrightarrow \bigsqcup_D S_i = u_n$$

son decidibles.

Un dominio D es **efectivamente presentado** si existe una enumeración efectiva u de D. ■

3.1.14.-LEMA:

Si D es efectivamente presentado entonces son decidibles los siguientes predicados entre elementos finitos de D, a partir de los índices:

(1) $x \ll_D y$

(2) $\{x, y\}$ es consistente

(3) $x = y$. ■

Se pueden construir nuevos cpo's a partir de dos de ellos ya definidos. Dos construcciones inmediatas son:

3.1.15.-DEFINICION: CPO Producto

Sean $C = \langle C, \ll_C, \perp_C \rangle$ y $D = \langle D, \ll_D, \perp_D \rangle$ dos cpo's.

Se define el **cpo producto** $C \times D = \langle D_\times, \ll_\times, \perp_\times \rangle$ donde:

$$D_\times := C \times D$$

$$(x, y) \ll_\times (x, y) :\Longleftrightarrow x \ll_C x' \wedge y \ll_D y'$$

$$\perp_\times := (\perp_C, \perp_D)$$ ■

3.1.16.-LEMA:

C × D es un dominio efectivamente presentado si C y D lo son. Además:

$$Fin_{C\times D} = Fin_C \times Fin_D. \blacksquare$$

3.1.17.-DEFINICION: CPO Potencia

Sean $C = \langle C, \ll_C, \perp_C\rangle$ y $D = \langle D, \ll_D, \perp_D\rangle$ dos cpo's.

Se define el **cpo potencia** $[C\to D] = \langle D_\to, \ll_\to, \perp_\to\rangle$ donde:

$D_\to := \{ f : C \longrightarrow D \mid f \text{ continua} \}$

$f \ll_\to g :\Longleftrightarrow \forall x \in C,\ f(x) \ll_D g(x)$

$\perp_\to$ es la función definida por $\perp_\to(x) = \perp_D$ para todo $x \in D$. $\blacksquare$

3.1.18.-LEMA:

$[C\to D]$ es un dominio efectivamente presentado si C y D lo son. Además, $Fin_{[C\to D]}$ es el conjunto de funciones continuas $f : C \longrightarrow D$ de la forma $f = [u_1,\ldots,u_n;v_1,\ldots,v_n]$ donde:

(1) $u_1,\ldots,u_n \in Fin_C$; $v_1,\ldots,v_n \in Fin_D$

(2) para cada subconjunto $I \subseteq \{1,\ldots,n\}$ se cumple:

$\{u_i \mid i \in I\}$ consistente en $C \Longrightarrow \{v_i \mid i \in I\}$ consistente en D.

Los valores de f se definen:

$$f(x) = \bigsqcup_D \{v_i \mid u_i \ll_C x\}. \blacksquare$$

En relación con la computabilidad tenemos lo siguiente:

3.1.19.-DEFINICION: Funciones y Elementos Computables

Sean C y D dominios efectivamente presentados via u y u' (resp.), y sea $f: C \longrightarrow D$. **f es computable** si y sólo si su grafo, definido como el predicado sobre $\mathbb{N}$

$$grafo_f(i, j) :\Longleftrightarrow u'_j \ll f(u_i)$$

es recursivamente enumerable.

Sea D un dominio efectivamente presentado via u. Un elemento $x \in D$ **es computable** si el conjunto de números naturales

$$G_x = \{i \in \mathbb{N} \mid u_i \ll x\}$$

es recursivamente enumerable. ■

3.1.20.-LEMA:

Los elementos finitos de un dominio efectivamente presentado son computables.

Los elementos computables del dominio C × D (ambos efectivamente presentados) son los pares (x, y) con x computable en C e y computable en D.

Los elementos computables del dominio [C⟶D], donde ambos C y D son efectivamente presentados, son las funciones computables de C en D. Además, cualquier función computable aplica elementos computables en elementos computables. ■

Un resultado muy conocido ([Tarski 55] y [Lloyd 87]), que utilizaremos en los apartados siguientes, y que nos limitamos a enunciar aquí es el:

3.1.21.-TEOREMA: (de Punto Fijo)

Sea D un dominio. Si f : D ⟶ D es una función continua sobre el dominio D entonces:

(1) f tiene un menor punto fijo d ∈ D.

(2) El menor punto fijo de f se puede obtener como $d = \bigsqcup_{n\geq 0} f^n(\perp_D)$, cumpliéndose que $\perp_D \ll f(\perp_D) \ll \ldots \ll f^n(\perp_D) \ll \ldots$ ■

3.2 CONSTRUCCION DE DOMINIOS PARA LAS FUNCIONES INDETERMINISTAS

Surgen dificultades obvias para dotar de una semántica, dentro de los dominios de Scott, a operadores como éste:

zero_or_one $\longrightarrow$ 0.

zero_or_one $\longrightarrow$ succ(0).

posibles en nuestro lenguaje. Ante la ausencia de una relación funcional fijamos, como en teoría de conjuntos, nuestra atención en el grafo de la correspondencia definido sobre los elementos finitos. En particular puede ocurrir que la correspondencia definida por el grafo en algunos casos sea una función. Entonces, estas funciones para ser adecuadas a la teoría de dominios deben ser continuas.

3.2.1.-DEFINICION: $[C \xrightarrow[n]{} D]$

Sean C y D dominios. $f \subseteq Fin_C \times Fin_D$ es un **grafo** si y sólo si:

(1) $\langle \perp_C, \perp_D \rangle \in f$

(2) $\forall u_C, u'_C \in Fin_C$; $\forall u_D, u'_D \in Fin_D$

$$\langle u_C, u_D \rangle \in f \wedge u_C \ll_C u'_C \wedge u'_D \ll_D u_D \Longrightarrow \langle u'_C, u'_D \rangle \in f.$$

Definimos el **conjunto de funciones indeterministas**:

$$[C \xrightarrow[n]{} D] := \{f \subseteq Fin_C \times Fin_D \mid f \text{ es un grafo}\}. \blacksquare$$

La idea de la definición es que por la propiedad (1) un grafo posee la información mínima ("bottom") y por (2) al aumentar la información por el canal de entrada el grafo mantiene toda la información del canal de salida que ya poseía.

La siguiente definición introduce una relación de pertenencia "$\ni$" que nos permite una mayor brevedad y claridad expositiva.

3.2.2.-DEFINICION:

Sean C, D dominios y $f \in [C \xrightarrow[n]{} D]$. Dados $x \in C$ e $y \in D$ decimos que f(x) **admite** y, escrito $f(x) \ni y$, si y sólo si

$$\forall u_D \ll_D y, \exists u_C \ll_C x \text{ tal que } \langle u_C, u_D \rangle \in f$$

siendo $u_C \in Fin_C$, $u_D \in Fin_D$. $\blacksquare$

3.2.3.-LEMA:

Sea $f \in [C \xrightarrow[n]{} D]$. Para todo $u_C \in Fin_C$, $u_D \in Fin_D$:

$$f(u_C) \ni u_D \iff \langle u_C, u_D \rangle \in f.$$

dem:

"$\Longrightarrow$" Sea $f(u_C) \ni u_D$. Entonces para todo $u'_D \ll_D u_D$ existe un $u'_C \ll_C u_C$ tal que $\langle u'_C, u'_D \rangle \in f$. Tomando $u'_D = u_D$ se deduce que

$$\langle u'_C, u_D \rangle \in f \wedge u'_C \ll_C u_C \wedge f \text{ grafo}, \Longrightarrow \langle u_C, u_D \rangle \in f.$$

"$\Longleftarrow$" Sea $u'_D \ll_D u_D$ donde u'_D es cualquier elemento de Fin_D. De f grafo $\wedge$ $\langle u_C, u_D \rangle \in f \wedge u'_D \ll_D u_D$ se deduce que $\langle u_C, u'_D \rangle \in f$. Tenemos que para cualquier $u'_D \ll_D u_D$ existe $u_C \ll_C u_C$ tal que

$$\langle u_C, u'_D \rangle \in f \Longrightarrow f(u_C) \ni u_D. \blacksquare$$

3.2.4.-LEMA:

Sea $f \in [C \xrightarrow[n]{} D]$. $f(x) \ni y \wedge z \ll_D y \Longrightarrow f(x) \ni z$.

dem:

Sea u_D un elemento finito cualquiera $\ll z$. Entonces $u_D \ll_D y$.

Como $f(x) \ni y$; para este u_D existe un $u_C \ll_C x$ tal que $\langle u_C, u_D \rangle \in f$.

Esto quiere decir que $f(x) \ni z$. $\blacksquare$

Para profundizar en la estructura de $[C \xrightarrow[n]{} D]$ necesitamos algunas definiciones más:

3.2.5.-DEFINICION: Pregrafo

Sean C y D dominios. Un **pregrafo** es cualquier conjunto $\gamma \subseteq Fin_C \times Fin_D$ tal que $\langle \perp_C, \perp_D \rangle \in \gamma$.

Si γ es un conjunto finito se dice un **pregrafo finito** $\blacksquare$

3.2.6.-DEFINICION: Cierre de un Pregrafo

El **cierre** $\overline{\gamma}$ de un pregrafo γ es el grafo mínimo generado por γ :

$$\overline{\gamma} = \{\langle u'_C, u'_D \rangle \in Fin_C \times Fin_D \mid \exists \langle u_C, u_D \rangle \in \gamma \text{ tal que } u_C \ll_C u'_C \wedge u'_D \ll_D u_D\}.$$

Se prueba de inmediato que $\overline{\gamma}$ es un grafo:

(1) $\langle \perp_C, \perp_D \rangle \in \overline{\gamma}$.

(2) $\langle u'_C, u'_D \rangle \in \overline{\gamma} \wedge u'_C \ll_C u''_C \wedge u''_D \ll_D u'_D \Longrightarrow$

$\exists \langle u_C, u_D \rangle \in \gamma$ tal que $u_C \ll_C u'_C \wedge u'_D \ll_D u_D \wedge u'_C \ll_C u''_C \wedge u''_D \ll_D u'_D \Longrightarrow$

$\exists \langle u_C, u_D \rangle \in \gamma$ tal que $u_C \ll_C u''_C \wedge u''_D \ll_D u_D \Longrightarrow$

$\langle u''_C, u''_D \rangle \in \overline{\gamma}$.

Además, $\overline{\gamma}$ es, en efecto, el menor grafo que contiene a γ. ■

3.2.7.-TEOREMA:

Sean C y D dominios. $[C \xrightarrow[n]{} D]$ es un retículo completo ω-algebraico con el orden parcial siguiente: si $f, g \in [C \xrightarrow[n]{} D]$ entonces

$$f \ll g :\Longleftrightarrow \forall u_C \in Fin_C,\ \forall u_D \in Fin_D\ (\ f(u_C) \ni u_D \Longrightarrow g(u_C) \ni u_D\ ).$$

$[C \xrightarrow[n]{} D]$ es, en particular, un dominio cuyos elementos finitos son los cierres de los pregrafos finitos.

dem:

▷ Sean $f, g \in [C \xrightarrow[n]{} D]$ y "$\ll$" la siguiente relación:

$f \ll g :\Longleftrightarrow \forall u_C \in Fin_C,\ \forall u_D \in Fin_D\ (\ f(u_C) \ni u_D \Longrightarrow g(u_C) \ni u_D\ )$.

"$\ll$" es un orden parcial ya que $f \ll g \Longleftrightarrow f \subseteq g$ (ver lema 3.2.3).

▷ El elemento mínimo para este orden parcial es:

$$\perp_{[C \xrightarrow[n]{} D]} = \{\langle u_C, \perp_D \rangle \in Fin_C \times Fin_D\}.$$

Es un grafo. Y para todo $f \in [C \xrightarrow[n]{} D]$ se tiene $\perp_{[C \xrightarrow[n]{} D]} \ll f$ ya que

$$\langle \perp_C, \perp_D \rangle \in f \wedge \perp_C \ll_C u_C \wedge f \text{ grafo} \Longrightarrow \langle u_C, \perp_D \rangle \in f.$$

▷ Todo subconjunto de $[C \xrightarrow[n]{} D]$ tiene supremo en $[C \xrightarrow[n]{} D]$:

Sea $S \subseteq [C \xrightarrow[n]{} D]$ cualquiera:

si $S = \emptyset$ entonces $\bigsqcup S = \perp_{[C \xrightarrow[n]{} D]}$.

si $S \neq \emptyset$ entonces $\bigsqcup S = \bigcup S$ (unión de conjuntos).

Podemos englobar los dos casos escribiendo para cualquier S:

$$\bigsqcup S = \bigcup S \cup \perp_{[C \xrightarrow[n]{} D]}.$$

Si $S \neq \emptyset$ podemos escribir:

$$\bigsqcup S = \{\langle u_C, u_D\rangle \in Fin_C \times Fin_D \mid \exists f \in S \text{ tal que } \langle u_C, u_D\rangle \in f\} = \bigcup S.$$

Es inmediato que $\bigsqcup S$ es un grafo.

En consecuencia, $[C \xrightarrow[n]{} D]$ es un retículo completo.

▷ $[C \xrightarrow[n]{} D]$ es ω-algebraico:

Vamos a definir, para ello, los elementos finitos. decimos que un **grafo f es finito** si y sólo si f es el cierre de algún pregrafo finito:

$$f \text{ finito} \iff_{def} f = \overline{\gamma} \wedge \gamma \text{ pregrafo finito.}$$

▷▷ f es un elemento finito en el sentido de la teoría de dominios:

En efecto. Sea S un conjunto dirigido de grafos tal que $f \ll \bigsqcup S$. Tenemos que probar que existe un $g \in S$ tal que $f \ll g$. Si f es finito en el sentido de la definición anterior $f = \overline{\gamma}$ con γ pregrafo finito. Tenemos que $\gamma \subseteq \bigcup S$ como conjuntos ya que al ser S dirigido $S \neq \emptyset$. Sea $\gamma = \{\gamma_1, \ldots, \gamma_n\}$ donde $\gamma_1 \in f_1 \in S$, ... , $\gamma_n \in f_n \in S$. Como S es dirigido existe $g \in S$ tal que $f_i \ll g$ para $(1 \leq i \leq n)$. Por tanto, $\gamma \ll g$ y como f es el menor grafo que contiene a γ se tiene $f \ll g$.

▷▷ Por otra parte, si $f \in [C \xrightarrow[n]{} D]$ es finito en el sentido de la teoría de dominios, entonces $f = \overline{\gamma}$ para algún pregrafo finito γ:

Esto se demostrará si probamos que el conjunto

$$B_f = \{\ \overline{\gamma} \mid \gamma \text{ pregrafo finito} \wedge \gamma \subseteq f\ \}$$

es dirigido y $f = \bigsqcup B_f$. Ya que, entonces, según la teoría de dominios existe $\overline{\gamma} \in B_f$ tal que $f \ll \overline{\gamma}$; y como por otra parte $\gamma \subseteq f$ se tiene

$$f = \overline{\gamma}.$$

Vamos a probar un resultado más general: Si $f \in [C \xrightarrow[n]{} D]$ (finito o no) y $B_f = \{\overline{\gamma} \mid \gamma \text{ pregrafo finito} \wedge \gamma \subseteq f\}$ entonces $f = \bigsqcup B_f$.

- B_f es dirigido:

$B_f \neq \emptyset$ pues $\{\overline{\langle \perp_C, \perp_D \rangle}\} \in B_f$.

Sean γ_1, γ_2 pregrafos finitos contenidos ambos en f. Entonces $\gamma_1 \cup \gamma_2$ es un pregrafo finito contenido en f. Y se cumple que:

$$\overline{\gamma}_1 \ll \overline{\gamma_1 \cup \gamma_2} = \overline{\gamma}_1 \cup \overline{\gamma}_2.$$

$$\overline{\gamma}_2 \ll \overline{\gamma_1 \cup \gamma_2} = \overline{\gamma}_1 \cup \overline{\gamma}_2.$$

- $\bigsqcup B_f \ll f$:

Para todo $\overline{\gamma} \in B_f$ se tiene $\gamma \subseteq f \Longrightarrow \overline{\gamma} \ll f \Longrightarrow \bigsqcup B_f \ll f$.

- $f \ll \bigsqcup B_f$:

Sea $\langle u_C, u_D \rangle \in f$. Representamos por $\overline{\langle u_C, u_D \rangle}$ el grafo mínimo generado por $\langle u_C, u_D \rangle$; o sea, el cierre de $\{\langle u_C, u_D \rangle, \langle \perp_C, \perp_D \rangle\}$.

Tenemos:

$\langle u_C, u_D \rangle \in f \Longrightarrow \overline{\langle u_C, u_D \rangle} \ll f \Longrightarrow \overline{\langle u_C, u_D \rangle} \in B_f$ (esto ocurre por ser finito $\{\langle u_C, u_D \rangle, \langle \perp_C, \perp_D \rangle\}$) $\Longrightarrow \overline{\langle u_C, u_D \rangle} \ll \bigsqcup B_f \Longrightarrow \langle u_C, u_D \rangle \in \bigsqcup B_f$ (ya que el supremo es la unión).

El resultado obtenido nos ha permitido demostrar, de paso, que $[C \xrightarrow[n]{} D]$ es algebraico.

▷▷ Finalmente, el conjunto de grafos finitos es numerable con tal que Fin_C y Fin_D sean numerables. ■

En $[C \xrightarrow[n]{} D]$ distinguimos una parte importante: el subconjunto de funciones deterministas $[C \xrightarrow[d]{} D]$. La idea de la definición es obvia.

3.2.8.-DEFINICION: $[C \xrightarrow[d]{} D]$

Sean C y D dominios. $f \in [C \xrightarrow[n]{} D]$ es **determinista** si y sólo si

$$\forall u_C \in Fin_C;\ \{\ u_D \in Fin_D \mid \langle u_C, u_D \rangle \in f\ \} \text{ es consistente en D.}$$

Definimos el **conjunto de funciones deterministas**:

$$[C \xrightarrow[d]{} D] := \{\ f \in [C \xrightarrow[n]{} D] \mid f \text{ determinista}\ \}. \blacksquare$$

Hay varias caracterizaciones equivalentes para las funciones deterministas:

3.2.9.-PROPOSICION:

Sean C y D dominios y sea $f \in [C \xrightarrow[n]{} D]$. Las tres afirmaciones siguientes son equivalentes:

(a) f es determinista.

(b) Para todo F finito y acotado, $F \subseteq Fin_C$

$$\{u_D \in Fin_D \mid \exists u_C \in F,\ \langle u_C, u_D\rangle \in f\} \text{ es consistente.}$$

(c) Para todo F consistente, $F \subseteq Fin_C$

$$\{u_D \in Fin_D \mid \exists u_C \in F,\ \langle u_C, u_D\rangle \in f\} \text{ es consistente.}$$

dem:

(a) $\Longrightarrow$ (b). Sea f determinista y F finito y acotado, $F \subseteq Fin_C$. Sea

$$G = \{u_D \in Fin_D \mid \exists u_C \in F,\ \langle u_C, u_D\rangle \in f\}.$$

$\bigsqcup F$ existe por ser F acotado en el dominio C, acotadamente completo. Por el lema 3.1.4, $\bigsqcup F \in Fin_C$. Entonces:

$$G' = \{u_D \in Fin_D \mid \langle \bigsqcup F, u_D\rangle \in f\}$$

es consistente por ser f determinista. Además $G \subseteq G'$:

$u_D \in G \Longrightarrow \exists u_C \in F$ con $\langle u_C, u_D\rangle \in f$. Como $u_C \ll_C \bigsqcup F \wedge \langle u_C, u_D\rangle \in f \wedge f$ es grafo se obtiene que $\langle \bigsqcup F, u_D\rangle \in f \Longrightarrow u_D \in G'$.

G es consistente al estar contenido en un conjunto consistente.

(b) $\Longrightarrow$ (a). Basta considerar $F = \{u_C\}$ para todo $u_C \in Fin_C$. F es finito y acotado; luego $\{u_D \in Fin_D \mid \langle u_C, u_D\rangle \in f\}$ es consistente.

(b) $\Longrightarrow$ (c) Sea F consistente, $F \subseteq Fin_C$. Sea

$$G = \{u_D \in Fin_D \mid \exists u_C \in F,\ \langle u_C, u_D\rangle \in f\}.$$

Sea G_0 un subconjunto finito de G, $G_0 = \{u_D^1, \ldots, u_D^n\}$. Por la definición de G para cada $u_D^i \in G_0$ existe un $u_C^i \in F$ tal que $\langle u_C^1,\ u_D^1\rangle \in f$ con $(1 \le i \le n)$. Sea $F_0 = \{u_C^1, \ldots, u_C^n\}$.

F consistente $\Longrightarrow F_0$ acotado.

F_0 finito y acotado $\Longrightarrow$

$G'' = \{u_D \in Fin_D \mid \exists u_C \in F_0,\ \langle u_C, u_D\rangle \in f\}$ es consistente $\Longrightarrow$

G_0 acotado (G_0 subconjunto finito de G'' consistente) $\Longrightarrow$

G consistente.

(c) $\Longrightarrow$ (b). Sea $F \subseteq Fin_C$, F finito y acotado. Entonces F es consistente y por (c)

$$G = \{u_D \in Fin_D \mid \exists u_C \in F,\ \langle u_C, u_D\rangle \in f\}$$

es consistente. ■

Para que las funciones deterministas sean adecuadas a la teoría de dominios han de ser continuas. Un isomorfismo natural surge entre $[C \xrightarrow[d]{} D]$ y $[C \to D]$ el dominio de las funciones continuas de C en D. Necesitamos un resultado previo:

3.2.10.-LEMA:

Sean C y D dominios. Si f es determinista, entonces la aplicación

$\hat{f} : C \longrightarrow D$ definida por:

$\hat{f}(x) = \bigsqcup \{y \in D \mid f(x) \ni y\}$

$= \bigsqcup \{u_D \in Fin_D \mid \exists u_C \in Fin_C \text{ tal que } u_C \ll_C x \wedge \langle u_C, u_D\rangle \in f\}$

es continua y cumple:

$$f(x) \ni y \Longleftrightarrow y \ll_D \hat{f}(x).$$

dem:

▷ $\hat{f}$ está bien definida:

Fijamos $x \in C$; y sea:

$$M = \{u_D \in Fin_D \mid \exists u_C \in Fin_C \text{ tal que } u_C \ll_C x \wedge \langle u_C, u_D\rangle \in f\}.$$

Consideramos cualquier $y \in D$ tal que $f(x) \ni y$; y cualquier $u_D \in Fin_D$ tal que $u_D \ll_D y$. Entonces, por la definición de $f(x) \ni y$, existe un $u_C \ll_C x$ tal que $\langle u_C, u_D\rangle \in f$. Luego $u_D \in M$.

Por tanto, para cualquier y tal que $f(x) \ni y$

$$\{u_D \in Fin_D \mid u_D \ll_D y\} \subseteq M.$$

Tomando supremos, y $\ll_D \bigsqcup M$ siempre que $\bigsqcup M$ exista.

Pero M es consistente, ya que $F = \{u_C \in Fin_C \mid u_C \ll_C x\}$ es consistente y por la proposición 3.2.9

$$M = \{u_D \in Fin_D \mid \exists u_C \in F \wedge \langle u_C, u_D \rangle \in f\}$$

es consistente.

Es inmediato que $\{y \in D \mid f(x) \ni y\}$ es consistente y además que

$$\bigsqcup \{y \in D \mid f(x) \ni y\} \ll_D \bigsqcup M.$$

Finalmente, es obvio que $\bigsqcup M \ll_D \bigsqcup \{y \in D \mid f(x) \ni y\}$.

▷ $\hat{f}$ es monótona:

Sea $x \ll_C x'$, entonces $f(x) \ni y \Longrightarrow f(x') \ni y$:

$f(x) \ni y \Longleftrightarrow \forall u_D \ll_D y,\ \exists u_C \ll_C x$ tal que $\langle u_C, u_D \rangle \in f$

$\Longrightarrow \forall u_D \ll_D y,\ \exists u_C \ll_C x \ll_C x'$ tal que $\langle u_C, u_D \rangle \in f$

$\Longleftrightarrow f(x') \ni y.$

Luego $x \ll_C x' \Longrightarrow \{y \in D \mid f(x) \ni y\}$

$\Longrightarrow \{y \in D \mid f(x') \ni y\}$

$\Longrightarrow \hat{f}(x) \ll_D \hat{f}(x').$

▷ $\hat{f}$ es continua:

Sea S un subconjunto dirigido de C y sea $\bigsqcup S$ el supremo de S.

$\bigsqcup \hat{f}(S) \ll_D \hat{f}(\bigsqcup S)$ se sigue de la monotonía.

Veamos que $\hat{f}(\bigsqcup S) \ll_D \bigsqcup \hat{f}(S)$:

$$\hat{f}(\bigsqcup S) = \bigsqcup \{y \in D \mid f(\bigsqcup S) \ni y\}$$

entonces $f(\bigsqcup S) \ni y \Longleftrightarrow \forall u_D \ll_D y,\ \exists u_C \ll_C \bigsqcup S$ tal que $\langle u_C, u_D \rangle \in f$.

Pero al ser S dirigido y u_C finito tenemos

$$u_C \ll_C \bigsqcup S \Longrightarrow \exists x \in S \text{ tal que } u_C \ll_C x \Longrightarrow f(x) \ni y \text{ para un } x \in S.$$

Luego:

$$\{y \in D \mid f(\bigsqcup S) \ni y\} \subseteq \bigcup \{\{y \in D \mid f(x) \ni y\} \mid x \in S\}$$

y obtenemos lo deseado tomando supremos.

▷ $f(x) \ni y \iff y \ll_D \hat{f}(x)$:

"$\Longrightarrow$" Inmediato.

"$\Longleftarrow$" Supongamos que

$y \ll \hat{f}(x) = \bigsqcup \{u'_D \in Fin_D \mid \exists u'_C \in Fin_C,\ u'_C \ll_C x,\ \langle u'_C, u'_D\rangle \in f\} = \bigsqcup N.$

Para todo $u_D \ll_D y$ existe, al ser finito, un u'_D tal que $u_D \ll_D u'_D$ y existe un $u'_C \ll_C x$ con $\langle u'_C, u'_D\rangle \in f$ ya que u'_D es de N. Como f es grafo, $\langle u'_C, u_D\rangle \in f$, de donde obtenemos que $f(x) \ni y$. ■

Una aplicación natural se establece de inmediato:

$$\Phi : [C \xrightarrow[d]{} D] \longrightarrow [C \to D]$$
$$f \longmapsto \hat{f}$$

▷ Φ es inyectiva:

Sean $f, g \in [C \xrightarrow[d]{} D]$ tal que $\Phi(f) = \Phi(g) \iff \hat{f} = \hat{g} \iff \hat{f}(x) = \hat{g}(x)$ para todo $x \in C$. Vamos a probar que, entonces, $f = g$.

$\forall u_C \in Fin_C$, $\forall u_D \in Fin_D$ tenemos $\langle u_C, u_D\rangle \in f \iff f(u_C) \ni u_D$ (por el lema 3.2.3) $\iff u_D \ll_D \hat{f}(u_C)$ (por el lema 3.2.10) $\iff u_D \ll_D \hat{g}(u_C) \iff g(u_C) \ni u_D \iff \langle u_C, u_D\rangle \in g$.

▷ Φ es sobreyectiva:

Sea $h \in [C \to D]$. Definimos el conjunto h_1 como sigue:

$$h_1 = \{\langle u_C, u_D\rangle \in Fin_C \times Fin_D \mid u_D \ll_D h(u_C)\}$$

h_1 es un grafo:

(1) $\langle \perp_C, \perp_D\rangle \in h_1$

(2) $\langle u_C, u_D\rangle \in h_1 \wedge u_C \ll_C u'_C \wedge u'_D \ll_D u_D \Longrightarrow u'_D \ll h(u_C) \Longrightarrow u'_D \ll h(u'_C)$ por ser h monótona $\Longrightarrow \langle u'_C, u'_D\rangle \in h_1$.

Evidentemente, para todo $u_C \in Fin_C$ el conjunto

$$\{u_D \in Fin_D \mid \langle u_C, u_D\rangle \in h_1\}$$

es consistente, pues $\langle u_C, u_D\rangle \in h_1 \iff u_D \ll_D h(u_C)$. Luego $h_1 \in [C \xrightarrow[d]{} D]$. Queda ver que $\hat{h}_1 = h$.

Fijamos un $x \in C$, entonces:

$$\hat{h}_1(x) = \bigsqcup \{u_D \in Fin_D \mid \exists u_C \in Fin_C,\ u_C \ll_C x,\ \langle u_C, u_D\rangle \in h_1\}$$
$$= \bigsqcup \{u_D \in Fin_D \mid \exists u_C \in Fin_C,\ u_C \ll_C x,\ u_D \ll_D h(u_C)\}$$
$$= \bigsqcup \{u_D \in Fin_D \mid u_D \ll_D h(x)\}$$
$$= h(x).$$

▷ Φ conserva el orden de aproximación: $f \ll g \iff \hat{f} \ll_{\to} \hat{g}$:

"$\Longrightarrow$"

$$\hat{f}(x) = \bigsqcup \{u_D \in Fin_D \mid \exists u_C \in Fin_C,\ u_C \ll_C x,\ \langle u_C, u_D\rangle \in f\}$$
$$\ll \bigsqcup \{u_D \in Fin_D \mid \exists u_C \in Fin_C,\ u_C \ll_C x,\ \langle u_C, u_D\rangle \in g\}$$
$$= \hat{g}(x).$$

"$\Longleftarrow$"

$$f(x) \ni y \iff y \ll_D \hat{f}(x) \Longrightarrow y \ll_D \hat{g}(x) \iff g(x) \ni y.$$

Finalmente, recordemos [Mulmuney 86] que los cpo's como objetos y las funciones continuas sobre ellos como morfismos constituyen una categoría: **la categoría de cpo's**. Se sabe, además que la categoría de dominios y funciones continuas es una subcategoría de la de cpo's. Representamos por **A** $\approx$ **B** cuando dos objetos en una categoría (referenciada en el contexto) son **isomorfos**.

Con el razonamiento anterior hemos probado el resultado siguiente:

3.2.11.-TEOREMA:

Sean C y D dominios.

$$[C \xrightarrow[d]{} D] \approx [C \to D]. \ \blacksquare$$

Para acabar esta sección, estudiamos la computabilidad del dominio de funciones indeterministas que hemos construido. Podemos ver, directamente, en primer lugar que:

3.2.12.-TEOREMA:

$[C \xrightarrow[n]{} D]$ es un dominio efectivamente presentable si lo son los dominios C y D.

dem:

Tenemos que ver lo primero que $Fin_{[C \xrightarrow[n]{} D]}$ es efectivamente enumerable si lo son C y D. Para ello, sea u una enumeración de C y u' una enumeración de D. Recordemos que los elementos finitos de $[C \xrightarrow[n]{} D]$ son los cierres de los pregrafos finitos. Basta, pues, enumerar éstos: los $\gamma \subseteq Fin_C \times Fin_D$ tal que $\langle \perp_C, \perp_D \rangle \in \gamma \wedge \gamma$ finito. Para resolver nuestro problema, en conclusión, tenemos que enumerar las partes finitas de $Fin_C \times Fin_D$ que contienen a $\langle \perp_C, \perp_D \rangle$.

Procedemos como sigue:

Supongamos que F_m es el subconjunto finito de $\mathbb{N}$ que cumple

$$m = \Sigma_{k \in F_m} 2^k.$$

Sea u una enumeración efectiva de C, con $u_0 = \perp_C$.

Sea u' una enumeración efectiva de D con $u'_0 = \perp_D$.

Supongamos una biyección $\pi : \mathbb{N} \times \mathbb{N} \longrightarrow \mathbb{N}$. Y sean π_1, π_2 las proyecciones de π^{-1}. Entonces se define:

$$U_i =_{def} \{\langle u_{\pi_1(j)}, u'_{\pi_2(j)} \rangle \mid j \in F_i\} \cup \{\langle \perp_C, \perp_D \rangle\}.$$

U es la enumeración efectiva que buscabamos para las partes finitas de $Fin_C \times Fin_D$ que contienen a $\langle \perp_C, \perp_D \rangle$.

Entonces, sean los predicados:

(a) $acot(i) :\Longleftrightarrow \{U_j \mid j \in F_i\}$ acotado en $[C \xrightarrow[n]{} D]$.

(b) $sup(n,j) :\Longleftrightarrow \bigsqcup \{U_i \mid i \in F_n\} = U_j$.

Vamos a ver que son decidibles:

(a) $\{U_j \mid j \in F_i\}$ acotado en $[C \xrightarrow[n]{} D]$ es cierto pues cualquier conjunto finito de grafos finitos tiene cota superior.

(b) $U_j = \bigsqcup \{U_1,\ldots,U_k\} \iff U_j \ll \bigsqcup \{U_1,\ldots,U_k\} \wedge \bigsqcup \{U_1,\ldots,U_k\} \ll U_j$.

Ya hemos visto que $\bigsqcup \{\bar{\gamma}_1,\ldots,\bar{\gamma}_n\} = \overline{\gamma_1 \cup \ldots \cup \gamma_n}$. Por tanto, ambas afirmaciones de la derecha de (b) pueden decidirse si es decidible que el cierre de un pregrafo finito está contenido en el cierre de otro:

$$\bar{\gamma}_1 \ll \bar{\gamma}_2.$$

Pero esto equivale a ver si es decidible que

$$\gamma_1 \ll \bar{\gamma}_2$$

que equivale a

$$\forall \langle u_C, u_D \rangle \in \gamma_1 \text{ se tiene que } \langle u_C, u_D \rangle \in \bar{\gamma}_2.$$

Pero, por definición de $\bar{\gamma}_2$, esto será decidible si lo es

$$\exists \langle u'_C, u'_D \rangle \in \gamma_2 \ (\ u'_C \ll u_C \wedge u_D \ll u'_D \ )$$

que es por el lema 3.1.14 una condición decidible.

El algoritmo podría formalizarse usando la presentación efectiva de $[C \xrightarrow[n]{} D]$ (tésis de Church). ■

¿ Cuáles son los elementos computables de $[C \xrightarrow[n]{} D]$?. Podemos enunciar: $f \in [C \xrightarrow[n]{} D]$ es computable $\iff$ es recursivamente enumerable como grafo; más formalmente el conjunto

$$\{\langle i,j \rangle \mid \langle u(i), u'(j) \rangle \in f\}$$

debe ser recursivamente enumerable siendo u y u' las enumeraciones de C y D respectivamente. En el apartado siguiente encontramos el isomorfismo preciso para responder a la pregunta con mayor claridad.

2.3 DOMINIOS DE FUNCIONES INDETERMINISTAS VERSUS DOMINIOS POTENCIA

El dominio $[C \xrightarrow[n]{} D]$ de grafos o funciones indeterministas que hemos construido corresponde a un concepto más clásico de **dominio**

potencia. Los dominios potencia se han utilizado, en la literatura sobre el tema, para especificar la semántica denotacional de lenguajes indeterministas [Plotkin 76] [Smyth 78] [Scott 82]. Pueden entenderse como análogos a los "conjuntos potencia" con elementos que representan los "conjuntos" de los diferentes caminos que una computación indeterminista puede seguir. Los dominios potencia fueron introducidos en primer lugar independientemente por Egli y Milner pero sólo trabajaban correctamente con dominios planos [Winskel 83a]. Plotkin construyó un dominio potencia aplicable a una clase más amplia de dominios [Plotkin 76]. En [Smyth 78] se muestran construcciones sencillas de los tres dominios potencia más conocidos: el de Plotkin, el de Hoare, y el suyo propio. Una construcción común se consigue preordenando los conjuntos bifinitos (conjuntos finitos de elementos finitos) de un cpo ω-algebraico de tres maneras diferentes y construyendo sendos cpo's ω-algebraicos usando el llamado método de complección por ideales. Además, para el dominio potencia de Hoare hay una construcción muy sencilla: es el conjunto de todos los conjuntos de elementos finitos no vacíos y cerrados inferiormente, ordenados por inclusión. Para el desarrollo de estas ideas veánse [Winskel 83a] [Winskel 83b] [Heckmann 90].

3.3.1.-DEFINICION: Dominio Potencia de Hoare

Sea D un dominio y Fin_D el conjunto de sus elementos finitos. Un conjunto $A \subseteq Fin_D$ se dice **cerrado hacia abajo** si y sólo si para todo $a \in A$ y para todo $u \in Fin_D$ tal que $u \ll a$ se tiene que $u \in A$.

El **dominio potencia inferior** o de **Hoare** P(D) es el conjunto de todos los subconjuntos no vacíos cerrados hacia abajo de Fin_D, ordenados por $\subseteq$. ■

3.3.2.-PROPOSICION:

Sea P(D) el dominio potencia de Hoare.

P(D) es un retículo completo ω-algebraico con respecto al orden parcial "$\subseteq$" siendo el supremo y el ínfimo dados por la unión (añadiendo $\perp$ si ésta es $\emptyset$) e intersección respectivamente. Los elementos finitos son los conjuntos

$$c(A) = \{u \in Fin_D \mid \exists a \in A \text{ tal que } u \ll a\}$$

donde A es un conjunto (no vacío) finito de Fin_D.

(Una demostración puede verse en [Heckmann 90]) ■

Podemos relacionar el dominio potencia P(D) con el dominio de grafos $[\{\perp\} \xrightarrow[n]{} D]$ donde por $\{\perp\}$ representamos el dominio trivial.

3.3.3.-TEOREMA:

Sea D un dominio y sea P(D) el dominio potencia correspondiente en el sentido de Hoare. Entonces:

$$P(D) \approx [\{\perp\} \xrightarrow[n]{} D].$$

dem:

Definimos una aplicación:

$$\Psi : P(D) \longrightarrow [\{\perp\} \xrightarrow[n]{} D]$$

$$I \longmapsto f_I = \{\langle \perp, v\rangle \mid v \in I\}$$

donde I es un conjunto no vacío cerrado hacia abajo de Fin_D.

▷ Ψ está bien definida pues f_I es un grafo:

(1) $\langle \perp, \perp_D\rangle \in f_I$ ya que $\perp_D \in I \neq \emptyset$.

(2) $\langle \perp, v_1\rangle \in f_I \wedge v_2 \ll v_1 \Longrightarrow$

$\langle \perp, v_2\rangle \in f_I$ ya que $v_1 \in I \wedge v_2 \ll v_1 \Longrightarrow v_2 \in I$.

▷ Ψ es inyectiva:

$\Psi(I) = \Psi(J) \Longrightarrow f_I = f_J \Longrightarrow I = J$.

▷ Ψ es sobreyectiva:

Sea $g \in [\{\perp\} \xrightarrow[n]{} D]$. Definimos

$$I_g = \{u \in Fin_D \mid \langle \perp, u \rangle \in g\}$$

que es trivialmente un conjunto no vacío cerrado hacia abajo de D.

Además, la imagen de I_g por Ψ es precisamente g.

▷ $\Psi(\{\perp_D\}) = \{\langle \perp, \perp_D \rangle\}$.

▷ Ψ es continua trivialmente pues la unión de conjuntos cerrados hacia abajo se corresponde con la unión de grafos del dominio $[\{\perp\} \xrightarrow[n]{} D]$. ■

Finalmente, es posible una caracterización del dominio de funciones indeterministas como un dominio de funciones continuas con valores en un dominio potencia de Hoare.

3.3.4.-TEOREMA:

Sean C y D dominios. Entonces:

$$[C \xrightarrow[n]{} D] \approx [C \to P(D)]$$

dem:

Definimos:

$$\Psi : [C \xrightarrow[n]{} D] \longrightarrow [C \to P(D)]$$

$$f \longmapsto f'$$

donde $\forall x \in C$, $f'(x) = \{u_D \in Fin_D \mid f(x) \ni u_D\}$

▷ Ψ está bien definida:

Para ello hay que ver lo primero que para un $x \in C$ cualquiera $f'(x)$ es un subconjunto no vacío cerrado hacia abajo de Fin_D:

$f'(x) \neq \emptyset$ pues $\perp_D \in f'(x)$ ya que $f(x) \ni \perp_D$ debido al hecho que $\langle \perp_C, \perp_D \rangle \in f$ para todo grafo f. Además, $u'_D \ll u_D \in f'(x) \Longrightarrow f(x) \ni u_D \Longrightarrow f(x) \ni u'_D$ (lema 3.2.4) $\Longrightarrow u'_D \in f'(x)$.

También tenemos que ver que f' es continua:

En primer lugar, f' es monótona ya que $x \ll y \Longrightarrow f'(x) \ll f'(y)$:

$u \in f'(x) \Longrightarrow f(x) \ni u \Longrightarrow f(y) \ni u \Longrightarrow u \in f'(y)$.

Sea S un subconjunto dirigido de C y sea $\bigsqcup S$ su supremo. Por la monotonía, $\bigsqcup f'(S) \ll f'(\bigsqcup S)$. Queda ver que $f'(\bigsqcup S) \leq \bigsqcup f'(S)$:

En efecto. Sea un elemento cualquiera $u \in f'(\bigsqcup S) \iff f(\bigsqcup S) \ni u \implies$ al ser S dirigido, existe un $s \in S$ tal que $f(s) \ni u \implies u \in f'(s) \implies f'(\bigsqcup S) \leq \bigsqcup f'(S)$.

▷ Ψ es inyectiva:

$\Psi(f) = \Psi(g) \implies f' = g' \implies \forall x \in C,\ f'(x) = g'(x)$. Sea $\langle u_C, u_D \rangle \in f \iff f(u_C) \ni u_D \iff u_D \in f'(u_C) \iff u_D \in g'(u_C) \iff \langle u_C, u_D \rangle \in g$. Entonces los grafos f y g son iguales.

▷ Ψ es sobreyectiva:

Sea $g \in [C \to P(D)]$. Definimos un grafo h como sigue:

$$h = \{\langle u_C, u_D \rangle \in Fin_C \times Fin_D \mid u_D \in g(u_C)\}$$

h es, efectivamente, un grafo:

(1) $\langle \bot_C, \bot_D \rangle \in h$ ya que $g(\bot_C) \neq \emptyset$ y $\bot_D \in g(\bot_C)$.

(2) Sea $\langle u_C, u_D \rangle \in h \wedge u_C \ll_C u'_C \wedge u'_D \ll_D u_D$. Entonces $\langle u_C, u_D \rangle \in h \implies u_D \in g(u_C) \implies u_D \in g(u'_C)$ (ya que al ser g continua $u_C \ll_C u'_C$ implica que $g(u_C) \subseteq g(u'_C)$) $\implies u'_D \in g(u'_C)$ (por ser $g(u'_C)$ un conjunto cerrado hacia abajo y $u'_D \ll_D u_D$) $\implies \langle u'_C, u'_D \rangle \in h$.

Nos falta ver que $h' = g$. Sea cualquier $x \in C$, $h'(x) = g(x)$:

Sea $u \in h'(x) \iff h(x) \ni u$

$\iff \forall u_D \ll_D u,\ \exists u_C \ll_C x$ tal que $\langle u_C, u_D \rangle \in h$

$\iff \forall u_D \ll_D u,\ \exists u_C \ll_C x$ tal que $u_D \in g(u_C)$

$\iff \forall u_D \ll_D u,\ u_D \in g(x)$ (por la continuidad de g)

$\iff u \in g(x)$ (tomando $u_D = u$)

▷ Ψ es continua:

Que Ψ es monótona es sencillo de demostrar. En efecto, sean $f, g \in [C \xrightarrow[n]{} D]$ con $f \leq g$. Sea x un elemento cualquiera de C Entonces: $u_D \in$

$f'(x) \iff f(x) \ni u_D \Longrightarrow g(x) \ni u_D$ (debido a que el grafo f está contenido en el g) $\iff u_D \in g'(x)$.

De aquí, $f'(x) \subseteq g'(x)$ para todo $x \in C$ y en consecuencia $f' \ll g'$, siendo ambas, funciones continuas de $[C \to P(D)]$.

Teniendo en cuenta la monotonía; para la continuidad basta con probar que $\Psi(\bigcup S) \ll \bigsqcup \Psi(S)$ para S conjunto dirigido de grafos de $[C \xrightarrow[n]{} D]$.

Sea x un elemento cualquiera de C, entonces:

$u \in \Psi(\bigcup S)(x) \iff u \in (\bigcup S)'(x) \iff (\bigcup S)(x) \ni u \Longrightarrow \exists f \in S$ tal que $f(x) \ni u$ al ser S dirigido $\iff u \in f'(x)$ con $f \in S \iff u \in \Psi(f)(x) \Longrightarrow u \in \bigsqcup \Psi(S)\ (x)$. ■

La presentación efectiva de P(D) puede realizarse debido al teorema 3.2.12 usando la presentación efectiva de $[\{\perp\} \xrightarrow[n]{} D]$. Podemos, además, responder de manera más precisa a la pregunta final del apartado anterior (3.2): El isomorfismo $[C \xrightarrow[n]{} D] \approx [C \to P(D)]$ nos permite determinar que las funciones indeterministas computables se corresponden (son isomorfas) con los elementos computables de $[C \to P(D)]$, que son precisamente las funciones continuas computables de C en P(D). (3.1.21).

NOTA: En lo sucesivo, se usarán dominios de funciones indeterministas o dominios potencia según se requiera.

4. SEMANTICA DECLARATIVA

Consideramos en este capítulo una semántica declarativa que asocia a cada expresión sintácticamente correcta un valor en algún dominio matemático bien definido de los que hemos considerado en el capítulo anterior. La semántica que vamos a estudiar es declarativa en el sentido de que interpreta cada regla "l $\longrightarrow$ r" como una aserción "l » r".

El concepto de interpretación se puede enfocar desde los dos puntos de vista isomorfos allí precisados. Un operador de aridad n puede considerarse como una función continua de un dominio producto de orden n en el dominio potencia o como un grafo (función indeterminista) que es simplemente un subconjunto, del producto de orden n+1 de los elementos finitos del dominio, con unas propiedades definidas (ver 3.2.1). Por otra parte, un constructor va a interpretarse como una función continua determinista.

Análogamente al caso de los programas lógicos de Horn obtenemos el concepto de modelo de un programa. Centramos la atención en las interpretaciones y modelos de Herbrand (interpretaciones y modelos "standard"), para dar paso al resultado más importante: la construcción de un modelo mínimo de Herbrand que se corresponde con una semántica punto fijo similar a la de los programas lógicos de Horn [Van Emden & Kowalski 76] y [Apt & Van Emden 83].

Además, extraemos la idea propuesta en K-LEAF [Levi & al 87]

de usar dominios de Scott para poder considerar funciones parciales y estructuras de datos infinitas y parciales.

4.1. INTERPRETACIONES

Para describir una interpretación necesitamos un dominio y un "significado" para los constructores y operadores del programa.

4.1.1.-DEFINICION: Σ-Interpretación

Sea un programa Π y sea Σ la signatura del mismo formada por CS ∪ FS. Una **Σ-interpretación** es un álgebra

$$I = < D_I, (c_I)_{c \in CS}, (f_I)_{f \in FS} >$$

donde:

1) D_I es un dominio de Scott.

2) ▷ $c_I \in D_I$ para $c \in CS^0_\Sigma$.

c_I es, simplemente, un elemento de D_I.

▷ $c_I \in [D_I^k \xrightarrow[d]{} D_I] \approx [D_I^k \longrightarrow D_I]$ para $c \in CS^0_k$, $k>0$.

c_I se interpreta como una función continua determinista.

En virtud del isomorfismo c_I puede interpretarse de manera indeterminista como un grafo determinista: un elemento de $[D_I^k \xrightarrow[d]{} D_I]$.

Si representamos por c_I un elemento de $[D_I^k \xrightarrow[d]{} D_I]$ y por $\hat{c}_I$ el elemento isomorfo de $[D_I^k \longrightarrow D_I]$ se cumple:

$$c_I(x_1,\ldots,x_k) \ni x \iff x \ll \hat{c}_I(x_1,\ldots,x_k)$$

(Recordar 3.1.10 y 3.1.11).

En lo que sigue identificaremos c_I y $\hat{c}_I$.

3) f_I se interpreta como un grafo o función indeterminista.

▷ Para k = 0, interpretamos $D_I^0 = \{\perp\}$ con lo que

$$f_I \in [\{\perp\} \xrightarrow[n]{} D_I] \approx P(D_I) \text{ para } f \in FS^0_\Sigma.$$

O sea, f_I está formado por pares $\langle \perp, u \rangle$ con $u \in Fin_{D_I}$ tales que:

$$\langle \perp, u \rangle \in f_I \wedge u' \ll u \Longrightarrow \langle \perp, u' \rangle \in f_I.$$

En consecuencia podemos interpretar f_I, para $f \in FS^0_\Sigma$ como un elemento de $P(D_I)$: un conjunto no vacio cerrado hacia abajo de elementos finitos de D_I.

▷ $f_I \in [D_I^k \xrightarrow[n]{} D_I] \approx [D_I^k \longrightarrow P(D_I)]$ para $f \in FS^k_\Sigma$, $k>0$. ■

Sea Π un programa con signatura $\Sigma = CS \cup FS$. Sea W un conjunto de variables. Por $\Sigma(W)$ entendemos la signatura formada por W $\cup$ CS $\cup$ FS. Una $\Sigma(W)$-interpretación es como en 4.1.1 un álgebra donde los elementos de W se interpretan como constructores de aridad 0 (o sea elementos del dominio).

Vamos, a continuación, con el importante concepto de interpretación de Herbrand. Para ello, vemos en primer lugar lo que se entiende por dominio de Herbrand.

Sea Π un programa. Sea Σ la signatura de Π con CS como conjunto de constructores. El **dominio de Herbrand** H_Σ para Π consiste en el cpo generado libremente por CS $\cup \{\perp_H\}$ en la categoría de las álgebras continuas con morfismos continuos y estrictos y donde $\perp_H$ se puede interpretar como una constante [Goguen & al 77]. Es un **objeto inicial** en dicha categoría lo que permite considerar las interpretaciones de Herbrand como interpretaciones "standard" de un programa.

Los elementos de H no tienen variables y podemos representarlos como árboles (finitos o infinitos) con nodos (de aridad finita) etiquetados por constructores y $\perp_H$ (de aridad 0). Las hojas de los árboles pueden ser constantes ó $\perp_H$.

Definimos un orden parcial en H como sigue:

▷ $\perp_H \ll_H t$, para todo $t \in H_\Sigma$

▷ $c(t_1,\ldots,t_n) \ll_H c(s_1,\ldots,s_n) \quad :\Longleftrightarrow \quad t_1 \ll_H s_1$

$\ldots\ldots\ldots$

$t_n \ll_H s_n$.

Esto es equivalente a decir (y es una definición más adecuada al considerar árboles infinitos):

$t \ll_H s \Longleftrightarrow_{def}$ s puede obtenerse reemplazando en t algunas hojas marcadas con $\perp_H$ por elementos (árboles) de H_Σ.

4.1.2.-TEOREMA:

El dominio de Herbrand $\mathbf{H_\Sigma} = \langle H_\Sigma, \ll_H, \perp_H, Fin_H \rangle$ es un dominio efectivamente presentado.

(Una demostración puede encontrarse en [Moreno 89]). ■

Podemos considerar también el **dominio de Herbrand con variables**. Sea Π un programa y $\Sigma(W)$ una signatura extensión de $\Sigma = CS \cup FS$. Definimos el dominio de Herbrand con conjunto de variables W:

$$H_\Sigma(W) =_{def} H_{\Sigma(W)}.$$

$H_\Sigma(W)$ es el dominio que se obtiene considerando las variables de W como nuevos constructores de aridad 0 (constantes).

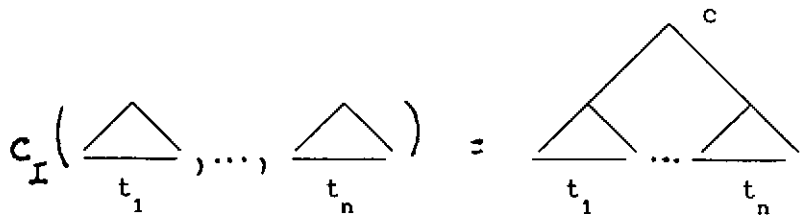
4.1.3.-DEFINICION: Interpretación de Herbrand

I es una **Σ-interpretación de Herbrand** si y sólo si:

(1) $D_I = H_\Sigma$.

(2) $\forall c \in CS^0_\Sigma$, c_I es el árbol que tiene c en la raiz.

$\forall c \in CS^n_\Sigma$, $n>0$, $t_1,\ldots,t_n \in H_\Sigma$, c_I se identifica con la función continua que es un **constructor libre** del árbol con c en la raiz y tal que de él salen n ramas de cada una de las cuales cuelga el árbol correspondiente a t_i para $(1 \le i \le n)$.



(3) La interpretación de los símbolos de FS como grafos (es decir, como funciones indeterministas) es arbitraria.

Análogamente se puede definir **Σ(W)-interpretación de Herbrand.** ■

4.1.4.-DEFINICION: Interpretación Computable

I es una **Σ-interpretación computable** si y sólo si:

(1) D_I es efectivamente presentable.

(2) c_I, f_I son funciones computables. ■

Dada una Σ-interpretación I, una **Σ-valoración** ρ es una aplicación:

$$\rho : VS \longrightarrow D_I.$$

Análogamente se pueden considerar Σ(W)-valoraciones.

Un concepto de evaluación indeterminista de una expresión viene dado por la siguiente definición:

4.1.5.-DEFINICION: Σ-evaluación de una Expresión

Sea I una Σ-interpretación y sea $\rho : VS \longrightarrow D_I$ una Σ-valoración.

$\|e\|_I(\rho)$ la **Σ-evaluación** de una expresion e bajo la Σ-valoración ρ es un elemento del conjunto potencia $P(D_I)$; o sea, un conjunto de elementos finitos de D_I cerrado por abajo, que se define recursivamente como sigue:

▷ Si e ::= X ∈ VS

$$\|X\|_I(\rho) = \{y \in Fin_{D_I} \mid y \ll \rho(X)\}.$$

▷ Si $e ::= k \in CS^0_\Sigma \cup FS^0_\Sigma$

$$\|k\|_I(\rho) = \{y \in Fin_{D_I} \mid k_I \ni y\}.$$

▷ Si $e ::= k(e_1,\ldots,e_n)$, $k \in CS^n_\Sigma \cup FS^n_\Sigma$; $n>0$. Para $y \in Fin_{D_I}$

$$\|k(e_1,\ldots,e_n)\|_I(\rho) \ni y$$

$$\Longleftrightarrow \exists x_i \in Fin_{D_I},\ (1\le i\le n),\ \text{tal que } \|e_i\|_I(\rho) \ni x_i,$$

y tal que $k_I(x_1,\ldots,x_n) \ni y$.

Análogamente, para W conjunto de variables, se puede definir la $\Sigma(W)$-evaluación de una expresión e. ■

Necesitaremos, además una noción de evaluación determinista para los términos lineales. Esto es posible porque los constructores los hemos interpretado como funciones continuas deterministas.

4.1.6.-DEFINICION: Σ-evaluación de un Término Lineal

Sea I una Σ-interpretación y sea $\rho : VS \longrightarrow D_I$ una Σ-valoración.

La **Σ-evaluación determinista de un término lineal t** se puede definir recursivamente:

▷ $t ::= X \in VS \Longrightarrow \|X\|^{det}_I(\rho) = \rho(X)$

▷ $t ::= c \in CS^0_\Sigma \Longrightarrow \|c\|^{det}_I(\rho) = \hat{c}_I$

▷ $t ::= c(t_1,\ldots,t_n)$, $c \in CS^n_\Sigma$, $n>0 \Longrightarrow$

$$\|c(t_1,\ldots,t_n)\|^{det}_I(\rho) = \hat{c}_I(\|t_1\|^{det}_I(\rho),\ldots,\|t_n\|^{det}_I(\rho)).$$

Se puede considerar también la $\Sigma(W)$-evaluación determinista de un término lineal. ■

El siguiente resultado es bastante evidente.

4.1.7.-PROPOSICION:

Para todo $x \in Fin_{D_I}$ se verifica:

$$\|t\|_I(\rho) \ni x \Longleftrightarrow x \ll \|t\|^{det}_I(\rho)\ .$$

dem:

Sea $x \in Fin_{D_I}$. Usamos la definición recursiva de t:

▷ $t ::= X \in VS$

$\|t\|_I(\rho) \ni x \iff x \ll \rho(X) \iff x \ll \|X\|_I^{det}(\rho)$.

▷ $t ::= c \in CS_\Sigma^0$

$\|c\|_I(\rho) \ni x \iff c_I \ni x \iff x \ll \hat{c}_I$ (ver 4.1.1) $\iff x \ll \|c\|_I^{det}(\rho)$.

▷ $t ::= c(t_1,\ldots,t_n)$, $c \in CS_\Sigma^n$, $n>0$

$$\|c(t_1,\ldots,t_n)\|_I(\rho) \ni x \iff \exists x_i \in Fin_{D_I}, (1\le i\le n),\ \|t_i\|_I(\rho) \ni x_i$$
$$\text{tal que } c_I(x_1,\ldots,x_n) \ni x$$
$$\iff \exists x_i \in Fin_{D_I}, (1\le i\le n), \text{ con } x_i \ll \|t_i\|_I^{det}(\rho)$$
$$\text{tal que } c_I(x_1,\ldots,x_n) \ni x$$

(ya que por hip. de ind. $\|t_i\|_I(\rho) \ni x_i \iff x_i \ll \|t_i\|_I^{det}(\rho)$ $(1\le i\le n)$)

$$\iff x \ll \hat{c}_I(\|t_1\|_I^{det}(\rho),\ldots,\|t_n\|_I^{det}(\rho))$$

(ya que c_I es monótona y continua)

$$\iff x \ll \|c(t_1,\ldots,t_n)\|_I^{det}(\rho))$$

(por la definición 4.1.6). ■

Observar que la definición anterior sugiere que se puede expresar la evaluación de un lado izquierdo de regla desde otro punto de vista.

4.1.8.-PROPOSICION:

Sea $l = f(t_1,\ldots,t_n)$ un lado izquierdo de regla, sea I una Σ-interpretación y sea ρ una Σ-valoración. Entonces:

$$\|f(t_1,\ldots,t_n)\|_I(\rho) = f_I(\ \|t_1\|_I^{det}(\rho),\ldots,\|t_n\|_I^{det}(\rho)\ ).$$

dem:

"⊆"

Sea $y \in Fin_{D_I}$ tal que $y \in \|f(t_1,\ldots,t_n)\|_I(\rho)$. Por la definición 4.1.5

existen x_i elementos finitos de D_I con $\|t_i\|_I(\rho) \ni x_i$ para todo i, $(1 \le i \le n)$, tal que $y \in f_I(x_1, \ldots, x_n)$. Por 4.1.7 se cumple que $\|t_i\|_I(\rho) \ni x_i \iff x_i \ll \|t_i\|_I^{det}(\rho)$ para todo i, $(1 \le i \le n)$. Entonces del hecho que $y \in f_I(x_1, \ldots, x_n)$ y la monotonía de f_I se tiene que

$$y \in f_I(\|t_1\|_I^{det}(\rho), \ldots, \|t_n\|_I^{det}(\rho)).$$

"$\supseteq$"

Sea $y \in Fin_{D_I}$ con $y \in f_I(\|t_1\|_I^{det}(\rho), \ldots, \|t_n\|_I^{det}(\rho))$. Consideremos:

$$C = \{(x_1, \ldots, x_n) \in Fin_{D_I} \times \ldots \times Fin_{D_I} \mid x_i \ll \|t_i\|_I^{det}(\rho) \ (1 \le i \le n)\}$$

Entonces $\bigsqcup C = (\|t_1\|_I^{det}(\rho), \ldots, \|t_n\|_I^{det}(\rho))$. Por la continuidad de f_I y el hecho de ser y finito se verifica que existen elementos finitos $x_i \ll \|t_i\|_I^{det}(\rho)$ para todo i $(1 \le i \le n)$ con $y \in f_I(x_1, \ldots, x_n)$. Entonces por 4.1.7, $x_i \ll \|t_i\|_I^{det}(\rho) \iff \|t_i\|_I(\rho) \ni x_i$ para todo i $(1 \le i \le n)$. Como $y \in f_I(x_1, \ldots, x_n)$ y $\|t_i\|_I(\rho) \ni x_i$ para todo i $(1 \le i \le n)$ se tiene por 4.1.5 que $y \in \|f(t_1, \ldots, t_n)\|_I(\rho)$. ■

4.2 MODELOS

Comenzamos definiendo el concepto de modelo de un programa con objeto de llegar a construir un modelo particular muy importante en nuestro estudio: el modelo mínimo de Herbrand del programa.

4.2.1.-DEFINICION:

Sea Π un programa y sea I una Σ-interpretación.

I es un **Σ-modelo de Π** (escrito $I \models \Pi$) si y sólo si I es un Σ-modelo de $l \longrightarrow r$ para toda regla $l \longrightarrow r$ de π.

I **es un Σ-modelo de** $l \longrightarrow r$ (escrito $I \models l \longrightarrow r$) si y sólo si para toda Σ-valoración $\rho : VS \longrightarrow D_I$ se tiene

$$\|l\|_I(\rho) \gg \|r\|_I(\rho).$$

Al ser ambos elementos del dominio potencia, cuyo orden parcial es $\subseteq$, el conjunto del lado izquierdo contiene al conjunto del lado derecho, con lo que podemos sustituir "$\gg$" por "$\supseteq$".

$\|l\|_I(\rho) \supseteq \|r\|_I(\rho)$ si y sólo si para todo $x \in Fin_{D_I}$

$$\|r\|_I(\rho) \ni x \Longrightarrow \|l\|_I(\rho) \ni x. \blacksquare$$

Todo lo anterior vale igualmente para signatura $\Sigma = \Sigma(W)$.

Como ya hemos dicho el resultado más importante que queremos obtener es la **existencia**, construyéndolo efectivamente, de un **modelo mínimo de Herbrand para un programa** Π. La idea es que el modelo mínimo es el modelo más "austero" posible: sus únicos objetos son los que proporcionan los constructores y los operadores definen sólo lo que piden las reglas del programa.

Para construir el modelo mínimo, consideramos las Σ-interpretaciones de Herbrand de Π y observamos que cualquier Σ-interpretación de Herbrand I puede ser identificada por la tupla de Σ-interpretaciones de sus operadores $\langle (f_I)_{f \in FS} \rangle$. Si representamos por H-INT la colección de Σ-interpretaciones de Herbrand para el programa Π, entonces:

$$H\text{-}INT = \langle H\text{-}INT, \ll_{H\text{-}INT}, \bot_{H\text{-}INT}, Fin_{H\text{-}INT} \rangle$$

donde:

▷ el orden parcial $\ll_{H\text{-}INT}$ está definido por:

$$I \ll_{H\text{-}INT} J \Longleftrightarrow f_I \ll f_J \text{ para toda } f \in FS.$$

▷ $\bot_{H\text{-}INT}$ es la interpretación que cada símbolo $f \in [H^k \xrightarrow[n]{} H]$ lo entiende como el grafo

$$\bot_{[H^k \xrightarrow[n]{} H]} = \{\langle u_1, \ldots, u_k, \bot_H \rangle \mid u_i \in Fin_H, \ (1 \le i \le k)\}.$$

▷ $Fin_{H\text{-}INT} = \{I \in H\text{-}INT \mid f_I$ finito como elemento del correspondiente dominio de grafos para todo $f \in FS\}$.

Podemos considerar la colección de Σ-interpretaciones de Herbrand como el dominio producto de dominios:

$$H\text{-}INT = [H^{k1} \xrightarrow[n]{} H] \times [H^{k2} \xrightarrow[n]{} H] \times \ldots \times [H^{kr} \xrightarrow[n]{} H]$$

que es efectivamente presentado con elemento mínimo y elementos finitos como los descritos anteriormente. (Ver resultados de la sección 3.1)

Definimos un operador, similar al operador consecuencia inmediata de los programas lógicos, como sigue:

$$T_\pi : H\text{-}INT \longrightarrow H\text{-}INT$$

$$I \longmapsto J$$

tal que $T_\pi(I) = J$ es la interpretación de Herbrand definida para cada $f \in FS$ por:

$$f_J = \{ \langle u_1, \ldots, u_k, u\rangle \mid \exists\, f(t_1, \ldots, t_k) \longrightarrow r \in \Pi$$
$$\exists\, \rho : VS \longrightarrow Fin_H$$
$$f(t_1, \ldots, t_k)\rho = f(u_1, \ldots, u_k)$$
$$\|r\|_I(\rho) \ni u \} \quad \cup \quad \perp_{[H^k \xrightarrow[n]{} H]} \qquad (1)$$

donde $f_J \in [H^k \xrightarrow[n]{} H]$.

La inclusión de $\perp_{[H^k \xrightarrow[n]{} H]}$ viene justificada por el hecho de asegurar la pertenencia a f_J de $\langle \perp, \ldots, \perp\rangle$ que no es cierta directamente en todos los casos como muestra el siguiente ejemplo:

Sea el programa Π: $f(0) \longrightarrow 0$.

la interpretación : $I = \perp_{H\text{-}INT}$

la interpretación : $J = T_\pi(I)$. ■

Por otra parte, el isomorfismo descrito en el capítulo anterior entre $[H^k \xrightarrow[n]{} H]$ y $[H^k \longrightarrow P(H)]$ nos permite una definición

equivalente para $J = T_\pi(I)$.

Tenemos el isomorfismo:

$$[H^k \xrightarrow[n]{} H] \longrightarrow [H^k \longrightarrow P(H)]$$

$$f_J \longmapsto f'_J$$

definido por $f'_J(z_1, \ldots, z_k) = \{u \in Fin_H \mid f_J(z_1, \ldots, z_k) \ni u\}$.

(ver 3.3.4)

Recordemos (3.2.2) que $f_J(z_1, \ldots, z_k) \ni u \iff$ existen $u_1, \ldots, u_k \in Fin_H$ con $(u_1, \ldots, u_k) \ll (z_1, \ldots, z_k)$ y tal que $\langle u_1, \ldots, u_k, u\rangle \in f_J$ al ser u finito.

$$f'_J(z_1, \ldots, z_k) = \{u \in Fin_H \mid \exists\, u_1, \ldots, u_k \in Fin_H \text{ tal que}$$
$$(u_1, \ldots, u_k) \ll (z_1, \ldots, z_k) \wedge$$
$$\langle u_1, \ldots, u_k, u\rangle \in f_J\}$$

Si definimos **instancia básica** de una regla como el resultado de reemplazar en la regla todas las variables por elementos finitos del dominio de Herbrand tenemos que $J = T_\pi(I)$ es la interpretación de Herbrand que para cada $f \in FS$, le hace corresponder el grafo f_J o lo que es lo mismo la función:

$$f'_J(z_1, \ldots, z_k) = \bigsqcup \{\|r'\|_I \mid \exists\, u_1, \ldots, u_k \in Fin_H$$
$$(u_1, \ldots, u_k) \ll (z_1, \ldots, z_k)$$
$$f(u_1, \ldots, u_k) \longrightarrow r'$$
$$\text{es instancia básica de una regla de } \Pi\}$$

$f'_J \in [H^k \longrightarrow P(H)]$.

En particular, se comprueba fácilmente que, para $u_1, \ldots, u_k \in Fin_H$

$$f'_J(u_1, \ldots, u_k) = \bigsqcup \{\|r'\|_I \mid f(u_1, \ldots, u_k) \longrightarrow r'$$
$$\text{es instancia básica de una regla de } \Pi\} \qquad (2)$$

Estudiamos las propiedades de T_π. Primeramente, demostramos un lema sencillo :

4.2.2.-LEMA: Monotonía y Continuidad de la Evaluación

Sea D un dominio.

Sea D-INT el dominio de todas las Σ-interpretaciones sobre D con el orden parcial natural. ($I \ll J \iff_{def} f_I \ll f_J$ para toda $f \in FS_\Sigma \wedge c_I \ll c_J$ para toda $c \in CS_\Sigma$)

Sea VAL_D el dominio de todas las Σ-valoraciones $\rho : VS \longrightarrow D$ también con el orden parcial natural. ($\rho \ll \rho' \iff_{def} \rho(X) \ll \rho'(X)$ para toda $X \in VS$)

Sea e una expresión fija. La aplicación:

$$D\text{-}INT \times VAL_D \longrightarrow P(D)$$

$$(I , \rho) \longmapsto \|e\|_I(\rho)$$

es monótona y continua.

dem:

1) La aplicación es monótona. Tenemos que probar que:

$$\text{si } (I_1,\rho) \ll (I_2,\rho') \text{ entonces } \|e\|_{I_1}(\rho) \subseteq \|e\|_{I_2}(\rho').$$

La demostración puede hacerse por inducción estructural sobre e.

▷ Si $e ::= X \in VS$

$\|X\|_{I_1}(\rho) \ni u \iff u \ll \rho(X)$ (por 4.1.5) $\Longrightarrow u \ll \rho'(X)$ (ya que $\rho \ll \rho'$) $\iff \|X\|_{I_1}(\rho') \ni u. \iff \|X\|_{I_2}(\rho') \ni u.$

▷ Si $e ::= k \in CS^0_\Sigma \cup FS^0_\Sigma$

$\|k\|_{I_1}(\rho) \ni u \iff k_{I_1} \ni u \Longrightarrow k_{I_2} \ni u$ (ya que $I_1 \ll I_2$) $\iff$

$\|k\|_{I_2}(\rho') \ni u.$

▷ Si $e ::= k(e_1,\ldots,e_n)$ con $k \in CS^n_\Sigma \cup FS^n_\Sigma$ y $n>0$

$\|k(e_1,\ldots,e_n)\|_{I_1}(\rho) \ni u \iff$

$\exists x_i \in Fin_D$ tal que $\|e_i\|_{I_1}(\rho) \ni x_i$ $(1\le i\le n) \wedge k_{I_1}(x_1,\ldots,x_n) \ni u \Longrightarrow$

$\exists x_i \in Fin_D$ tal que $\|e_i\|_{I_2}(\rho') \ni x_i$ $(1 \le i \le n)$ (por hipótesis de inducción) $\wedge\ k_{I_1}(x_1,\ldots,x_n) \ni u \Longrightarrow$

$\|k(e_1,\ldots,e_n)\|_{I_2}(\rho') \ni u$ (ya que $I_1 \ll I_2$).

2) La aplicación es continua. Al ser monótona tenemos solo que probar que si (I_α, ρ_α) para todo $\alpha \in A$ es un conjunto dirigido de D-INT x VAL_D cuyo supremo es $(\bigsqcup I_\alpha, \bigsqcup \rho_\alpha)$ entonces $\|e\|_{\bigsqcup I_\alpha}(\bigsqcup \rho_\alpha) \subseteq \bigsqcup \|e\|_{I_\alpha}(\rho_\alpha)$.

Nuevamente utilizamos la inducción estructural sobre e.

▷ Si e ::= X ∈ VS

$\|X\|_{\bigsqcup I_\alpha}(\bigsqcup \rho_\alpha) \ni u \Longleftrightarrow u \ll (\bigsqcup \rho_\alpha)(X) = \bigsqcup \rho_\alpha(X) \Longrightarrow$ al ser u finito existe un $\alpha' \in A$ tal que $u \ll \rho_{\alpha'}(X) \Longleftrightarrow \|X\|_{I_{\alpha'}}(\rho_{\alpha'}) \ni u \Longrightarrow \bigsqcup \|X\|_{I_\alpha}(\rho_\alpha) \ni u$.

▷ Si e ::= $k \in CS^0_\Sigma \cup FS^0_\Sigma$

$\|k\|_{\bigsqcup I_\alpha}(\bigsqcup \rho_\alpha) \ni u \Longleftrightarrow k_{\bigsqcup I_\alpha} \ni u \Longrightarrow$ por ser u finito y $k_{\bigsqcup I_\alpha} = \bigsqcup k_{I_\alpha}$ existe un $\alpha' \in A$ tal que $k_{I_{\alpha'}} \ni u \Longleftrightarrow \|k\|_{I_{\alpha'}}(\rho_{\alpha'}) \ni u \Longrightarrow$

$$\bigsqcup \|k\|_{I_\alpha}(\rho_\alpha) \ni u.$$

▷ Si e ::= $k(e_1,\ldots,e_n)$ con $k \in CS^n_\Sigma \cup FS^n_\Sigma$ y $n>0$

$\|k(e_1,\ldots,e_n)\|_{\bigsqcup I_\alpha}(\bigsqcup \rho_\alpha) \ni u \Longleftrightarrow$

$\exists x_i \in Fin_D$ tal que $\|e_i\|_{\bigsqcup I_\alpha}(\bigsqcup \rho_\alpha) \ni x_i$ con $1 \le i \le n \wedge k_{\bigsqcup I_\alpha}(x_1,\ldots,x_n) \ni u$.

Por hipótesis de inducción $\|e_i\|_{\bigsqcup I_\alpha}(\bigsqcup \rho_\alpha) \ni x_i$ con $1 \le i \le n \Longrightarrow$

$\bigsqcup \|e_i\|_{I_\alpha}(\rho_\alpha) \ni x_i$ con $1 \le i \le n$.

Al ser los x_i finitos podemos encontrar entonces unos índices α_i tal que $\|e_i\|_{I_{\alpha_i}}(\rho_{\alpha_i}) \ni x_i$ con $1 \le i \le n$.

Por otra parte, de $k_{\bigsqcup I_\alpha}(x_1,\ldots,x_n) \ni u$ y el hecho de ser u finito podemos deducir que existe un $\alpha^* \in A$ tal que $k_{I_{\alpha^*}}(x_1,\ldots,x_n) \ni u$.

Como los (I_α, ρ_α) con $\alpha \in A$, forman un conjunto dirigido existe un α' $\in A$ tal que $(I_{\alpha_1}, \rho_{\alpha_1}) \ll (I_{\alpha'}, \rho_{\alpha'})$; $(I_{\alpha_2}, \rho_{\alpha_2}) \ll (I_{\alpha'}, \rho_{\alpha'})$; ... ; $(I_{\alpha_n}, \rho_{\alpha_n}) \ll (I_{\alpha'}, \rho_{\alpha'})$; y además $(I_{\alpha^*}, \rho_{\alpha^*}) \ll (I_{\alpha'}, \rho_{\alpha'})$.

En conclusión, $\exists x_i \in Fin_D$ tal que $\|e_i\|_{I_{\alpha'}}(\rho_{\alpha'}) \ni x_i$ con $1 \le i \le n \wedge$ $k_{I_{\alpha'}}(x_1,\ldots,x_n) \ni u \iff \exists \alpha' \in A$ tal que $\|k(e_1,\ldots,e_n)\|_{I_{\alpha'}}(\rho_{\alpha'}) \ni u \iff$ $\bigsqcup \|k(e_1,\ldots,e_n)\|_{I_\alpha}(\rho_\alpha) \ni u$. ■

Probada la monotonía y continuidad de la evaluación el siguiente teorema es inmediato.

4.2.3.-TEOREMA:

T_π es una función de H-INT $\longrightarrow$ H-INT que está bien definida, es continua y computable.

dem:

▷ T_π está bien definida:

Para ello hay que probar que f_J es un grafo:

(1) $\langle u_1, u_2, \ldots, u_k, \perp \rangle \in f_J$ por definición.

(2) Sean $\langle u_1,\ldots,u_k,u\rangle \in f_J$, $\langle u_1,\ldots,u_k\rangle \ll \langle u'_1,\ldots,u'_k\rangle$ y $u' \ll u$. Hay que probar que $\langle u'_1,\ldots,u'_k,u'\rangle \in f_J$. Podemos restringirnos al caso en que $u,\ u' \neq \perp$.

Por definición:

$$\langle u_1,\ldots,u_k,u\rangle \in f_J \Longrightarrow \exists\ l \longrightarrow r \in \Pi$$
$$\exists\ \rho : VS \longrightarrow Fin_H$$
$$l\rho = f(u_1,\ldots,u_k)$$
$$\|r\|_I(\rho) \ni u\ .$$

Dado $l\rho = f(t_1,\ldots,t_k)\rho = f(u_1,\ldots,u_k)$ vamos a ver que $\exists \rho'$: VS $\longrightarrow$ Fin_H tal que

$$l\rho' = f(t_1,\ldots,t_k)\rho' = f(u'_1,\ldots,u'_k)$$

con $u_i \ll u'_i$ $\forall i = 1,\ldots,n$, y $\rho \ll \rho'$.

Como las partes izquierdas de las reglas son lineales podemos reducir el problema a ver que con las siguientes hipótesis:

t lineal $\wedge$ u finito $\wedge$ u instancia de t (i.e.: $t\rho = u$) $\wedge$ $u \ll u'$ se cumple que u' es instancia de t (i.e.: $t\rho' = u'$) para alguna ρ' tal que $\rho \ll \rho'$.

La prueba podemos hacerla por inducción estructural sobre el árbol t:

→ t = X. Entonces $\rho = \{X \rightarrow u\}$ hace que $t\rho = u$. Basta tomar $\rho' = \{X \rightarrow u'\}$ para obtener $t\rho' = u'$. Además, $\rho \ll \rho'$ ya que $u \ll u'$.

→ t = c. Entonces ha de ser $u = u' = c$ y es trivial.

→ $t = c(t_1,\ldots,t_n)$. Entonces forzosamente u ha de ser de la forma $c(u_1,\ldots,u_n)$ con u_i instancia de t_i para $i = 1,\ldots,n$; $t_i\rho_i = u_i$, y por hipótesis de inducción existen $\rho'_i \gg \rho_i$ con $t_i\rho'_i = u'_i$ para $u_i \ll u'_i$. Sea $\rho' = \rho'_1 \cup \ldots \cup \rho'_n$ (unión disjunta ya que podemos suponer que los ρ'_i tienen dominios disjuntos) y sea $\rho = \rho_1 \cup \ldots \cup \rho_n$, tenemos que $\rho \ll \rho'$.

Nos queda ver que si $u' \ll u$ y $\rho \ll \rho'$

$$\|r\|_I(\rho) \ni u \Longrightarrow \|r\|_I(\rho') \ni u'.$$

Como $u' \ll u$ tenemos:

$$\|r\|_I(\rho') \ni u \Longrightarrow \|r\|_I(\rho') \ni u'.$$

Basta, entonces, ver que

$$\rho \ll \rho' \Longrightarrow (\|r\|_I(\rho) \ni u \Longrightarrow \|r\|_I(\rho') \ni u).$$

Esto es inmediato del lema de monotonía y continuidad de la evaluación (4.2.2).

▷ T_π es monótona:

Supongamos interpretaciones de Herbrand $I_1 \ll I_2$ (o sea, $f_{I_1} \ll f_{I_2}$ para toda $f \in FS$). Tenemos que probar $J_1 = T_\pi(I_1) \ll T_\pi(I_2) = J_2$. Sea $f \in FS$ entonces $f_{J_1} \ll f_{J_2}$ será cierto si

$$I_1 \ll I_2 \Longrightarrow (\|r\|_{I_1}(\rho) \ni u \Longrightarrow \|r\|_{I_2}(\rho) \ni u).$$

que es inmediato por 4.2.2.

▷ T_π es continua:

Sea $(I_\alpha)_{\alpha\in A}$ un conjunto dirigido de interpretaciones Herbrand. Tenemos que probar que $T_\pi(\bigsqcup I_\alpha) = \bigsqcup T_\pi(I_\alpha)$ para $\alpha\in A$. Por la monotonía $\bigsqcup T_\pi(I_\alpha) \ll T_\pi(\bigsqcup I_\alpha)$.

Queda ver que $T_\pi(\bigsqcup I_\alpha) \ll \bigsqcup T_\pi(I_\alpha)$:

Sea $J = T_\pi(\bigsqcup I_\alpha)$ y sean $J_\alpha = T_\pi(I_\alpha)$ para cada $\alpha \in A$. $\{J_\alpha\}_{\alpha\in A}$ es un conjunto dirigido y existe el supremo $\bigsqcup J_\alpha$ para $\alpha \in A$. Tenemos que probar que $J \ll \bigsqcup J_\alpha$. O lo que es lo mismo, para toda $f \in FS$ tenemos que $f_J \ll f_{\bigsqcup J_\alpha}$. Esto se deduce inmediatamente, teniendo en cuenta la definición del operador T_π, si previamente demostramos que:

$$\|r\|_{\bigsqcup I_\alpha}(\rho) \ni u \Longrightarrow \bigsqcup \|r\|_{I_\alpha}(\rho) \ni u$$

que sale de nuevo de 4.2.2.

▷ T_π es computable:

Probar la computabilidad de T_π puede hacerse apoyándonos en la tésis de Church de manera informal si probamos que el conjunto:

$$\text{grafo}(T_\pi) = \{ (I,J) \mid I,J \text{ finitas} \wedge J \ll_{\text{H-INT}} T_\pi(I) \}$$

es semidecidible.

Para $I,J \in$ H-INT y finitas, tenemos:

$J \ll_{\text{H-INT}} T_\pi(I) \Longleftrightarrow$

$\forall f \in FS,\ f_J \ll f_{T_\pi(I)} \Longleftrightarrow$

$(\forall f \in FS\ \forall \langle u_1,\ldots,u_k,u\rangle \in f_J,\ u\neq\perp \Longrightarrow \langle u_1,\ldots,u_k,u\rangle \in f_{T_\pi(I)}) \Longleftrightarrow$

$\forall f \in FS;\ \forall \langle u_1,\ldots,u_k,u\rangle \in f_J,\ u\neq\perp$ hay alguna instancia básica $f(u_1,\ldots,u_k) \longrightarrow r'$ de una regla tal que $\|r'\|_I \ni u$.

Teniendo en cuenta que " $\|r'\|_I \ni u$ " es decidible para I finita, lo

anterior proporciona un procedimiento de semidecisión. ■

Vamos a obtener el modelo mínimo del programa Π como el mínimo punto fijo del operador asociado T_π. Aplicando el teorema de punto fijo (3.1.21) obtenemos que T_π tiene un mínimo punto fijo μT_π que se calcula:

$$\mu T_\pi = \bigsqcup_{i<\omega} T_\pi\ i$$

donde

$$T_\pi\ 0 = \bot_{H\text{-}INT}$$
$$T_\pi\ (i+1) = T_\pi(T_\pi\ i).$$

Antes de comprobar la existencia del modelo mínimo de Herbrand necesitamos el siguiente resultado:

4.2.4.-TEOREMA:

Para cualquier Σ-interpretación Herbrand I tenemos:

$$I \models \Pi \text{ si y sólo si } T_\pi(I) \ll I.$$

dem:

"$\Longrightarrow$"

Sea $I \models \Pi$. Entonces I es un modelo para toda regla $l \longrightarrow r$ de Π y se verifica para toda Σ-valoración ρ: $VS \longrightarrow Fin_H$ y para todo $x \in Fin_H$:

$$\|r\|_I(\rho) \ni x \Longrightarrow \|l\|_I(\rho) \ni x.$$

Tenemos que probar que $T_\pi(I) = J \ll I$. O sea, para toda $f \in FS$, $f_J \ll f_I$.

Sea $\langle u_1,\ldots,u_k,u\rangle \in f_J$. Sin pérdida de generalidad podemos suponer que $u \neq \bot$ y entonces $\exists \rho$ valoración

$$\exists\ l \longrightarrow r \text{ regla}$$
$$l\rho = f(u_1,\ldots,u_k)$$
$$\|r\|_I(\rho) \ni u.$$

Supongamos que la parte izquierda de la regla l, que es lineal, es

$f(t_1,\ldots,t_k)$. Entonces por la proposición 4.1.8:

$\|f(t_1,\ldots,t_k)\|_I(\rho) = f_I(\|t_1\|_I^{det}(\rho),\ldots,\|t_n\|_I^{det}(\rho)) = f_I(u_1,\ldots,u_k)$.

Por lo cual, usando además que $I \models l \longrightarrow r$, se tiene:

$\|r\|_I(\rho) \ni u \Longrightarrow \|l\|_I(\rho) \ni u \Longrightarrow f_I(u_1,\ldots,u_k) \ni u \Longrightarrow$

$\langle u_1,\ldots,u_k,u\rangle \in f_I$ como queríamos demostrar.

"$\Longleftarrow$"

Supongamos $T_\pi(I) = J \ll I$ y sea $l \longrightarrow r$ una regla cualquiera.

Tenemos que probar que $\|r\|_I(\rho) \subseteq \|l\|_I(\rho)$ para cualquier valoración ρ.

Por continuidad de la evaluación (lema 4.2.2) basta tratar el caso en que ρ sea finita. Supongamos que $\|r\|_I(\rho) \ni u$.

Sea $l\rho = f(u_1,\ldots,u_k)$ y $\|r\|_I(\rho) \ni u$; entonces, por definición $\langle u_1,\ldots,u_k,u\rangle \in f_{T_\pi(I)}$. Como $T_\pi(I) \ll I$ tenemos $\langle u_1,\ldots,u_k,u\rangle \in f_I$ o lo que es lo mismo $f_I(u_1,\ldots,u_k) \ni u$.

Es cuestión de repetir el razonamiento de la parte anterior para probar que $\|l\|_I(\rho) \ni u$. ■

Ahora seguimos el razonamiento de [Lloyd 87] teniendo en cuenta los siguientes hechos:

1) H-INT es un retículo completo;

2) T_π es monótona y continua;

3) $I \models \Pi$ si y sólo si $T_\pi(I) \ll I$.

En consecuencia, tenemos las siguientes equivalencias para el modelo mínimo de Herbrand (ver 1.1.2):

$$\begin{aligned} I_\pi &= \mu T_\pi \\ &= \prod \{I \mid T_\pi(I) = I\} \\ &= \prod \{I \mid T_\pi(I) \ll I\} \\ &= \bigsqcup T_{\pi\uparrow i} \\ &= T_{\pi\uparrow\omega}. \end{aligned}$$

Además, $T_{\pi}(I_{\pi}) = I_{\pi}$.

Todos los razonamientos desarrollados a lo largo del capítulo son válidos considerando como signatura $\Sigma(W)$ por lo que podemos afirmar que, para cualquier conjunto W de variables, existe un $\Sigma(W)$-modelo mínimo de Herbrand del programa Π, que representaremos en lo sucesivo por $I_{\pi}(W)$.

5. SEMANTICA OPERACIONAL.

Tenemos ahora que dotar a nuestro lenguaje de una semántica operacional adecuada que permita simular, al mismo tiempo, los mecanismos de cómputo de los lenguajes funcionales y de los lenguajes lógicos; es decir reescritura y SLD-resolución respectivamente. Como ya hemos indicado en el capítulo 1, el estrechamiento ("narrowing") es una técnica de reducción de expresiones adecuada para conseguir nuestro objetivo. Sin embargo, en el ejemplo 4 del capítulo 2 veíamos que el uso de reescritura y estrechamiento no "innermost" como semántica operacional produce resultados incorrectos en algunos casos.

Vamos a estudiar el ejemplo citado más formalmente. Recordemos el programa:

$$add(zero,X) \longrightarrow X$$

$$add(succ(X),Y) \longrightarrow succ(add(X,Y))$$

$$double(X) \longrightarrow add(X,X)$$

$$or(X,Y) \longrightarrow X$$

$$or(X,Y) \longrightarrow Y$$

Habíamos visto que con reescritura "outermost" se podía obtener:

$$double(or(zero,\ succ(zero))) \overset{*}{\Longrightarrow} succ(zero).$$

Vamos a analizar el ejemplo aplicando las definiciones dadas para la semántica declarativa en el capítulo anterior. Podríamos formar el siguiente dominio de Herbrand:

$$H = \{\ \perp,\ zero,\ succ(\perp),\ succ(zero),\ succ(succ(\perp)),\ \ldots\ \}$$

Definiendo el operador T_π como en 4.2 obtenemos las interpretaciones de Herbrand $I_0 = \perp_{H\text{-}INT}$; $I_1 = T_\pi(I_0)$; $I_2 = T_\pi(I_1)$; ... que nos permiten hallar el menor punto fijo del operador: I_π que es precisamente el modelo mínimo de Herbrand del programa. Si escribimos f_i para representar la interpretación del símbolo f en I_i, tendríamos como valores posibles de los operadores para las primeras potencias de T_π:

$add_1(t, zero) \ni t;$

$add_1(t, succ(t')) \ni succ(\perp);$

$double_1(t) \ni \perp;...$

$add_2(t, zero) \ni t;$

$add_2(t, succ(zero)) \ni succ(t);$

$add_2(t, succ^{n+2}(t')) \ni succ(succ(\perp));...$

$double_2(zero) \ni zero;$

$double_2(succ(t)) \ni succ(\perp);...$

$double_3(zero) \ni zero;$

$double_3(succ(zero)) \ni succ(succ(zero));$

$double_3(succ(succ(t))) \ni succ(succ(\perp));...$

Finalmente, en el modelo mínimo $I_\pi = \bigsqcup_{i<\omega} I_i$ se verifica:

$zero \in double_{I_\pi}(or_{I_\pi}(zero, succ(zero)));$

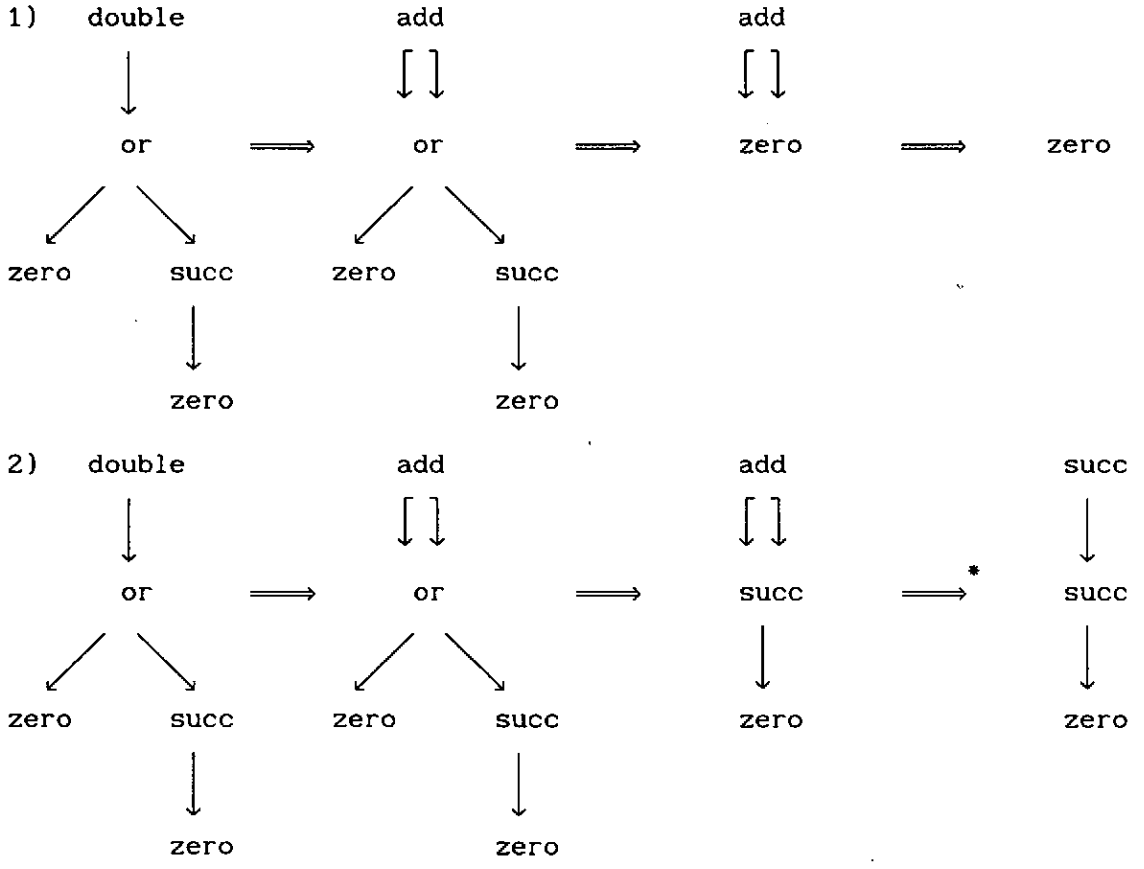
$succ(succ(zero)) \in double_{I_\pi}(or_{I_\pi}(zero, succ(zero)));$

$succ(zero) \notin double_{I_\pi}(or_{I_\pi}(zero, succ(zero))).$

Ya vimos en el ejemplo citado del capítulo 2 que la existencia de esta incorrección usando reescritura "no innermost" es debida a la necesidad de compartir la información en las partes derechas de las reglas e indicamos algunas soluciones "malas" para el problema. La solución "buena" nos lleva a utilizar GRAFOS DIRIGIDOS ACICLICOS

(G.D.A.'S) para representar las expresiones del lenguaje y a construir un mecanismo de cómputo basado en estrechamiento de los mismos. Ligado con lo anterior surge la necesidad de construir un algoritmo que permita la unificación de expresiones y reglas en el marco establecido de los grafos dirigidos acíclicos.

Así, en el ejemplo anterior tendríamos dos posibilidades:



que nos dan resultados correctos.

En lenguajes de programación funcional la reducción de expresiones mediante grafos es conocida y se ha usado como técnica de implementación de lenguajes perezosos [Turner 79] [Cardelli 83] [Peyton Jones 87]. También, se han usado lenguajes definidos por los propios grafos como lenguajes "intermedios" para sistemas de reescritura de términos [Barendregt & al 87] y para descripciones

semánticas [Holm 90]. Incluso, en algunos casos, se han estudiado grafos con ciclos para representar objetos infinitos y para implementar eficientemente ciertas reglas de reescritura tales como la regla para el "combinador de punto fijo Y" [Farmer & Watro 91].

La representación de los cómputos por medio de la reducción de grafos tiene ventajas inmediatas. En primer lugar, la forma de mostrar la **compartición** (problema que como hemos visto surge en nuestro trabajo) es fácil de expresar ya que no se requiere ninguna estructura adicional para **almacenar** las subexpresiones ligadas compartidas. Es posible, por medio de **arcos del grafo**, referirnos a subgrafos el número de veces que lo necesitemos.

En la primera sección de este capítulo se define, convenientemente adaptado para nuestro trabajo, el concepto de grafo dirigido acíclico (g.d.a.). En la segunda, vemos como se pueden representar y manejar expresiones del lenguaje que hemos definido por medio de grafos. En la tercera mostramos como se puede conseguir la unificación lineal de lados izquierdos de reglas y expresiones representadas por medio de grafos, construyendo un algoritmo de unificación sobre gda's con tres posibles salidas: éxito, fallo y suspendida. Finalmente en la cuarta describimos, utilizando un grafo conveniente, el mecanismo de cómputo (paso de estrechamiento) de nuestro lenguaje definiendo posterioremente el estrechamiento, en general, y los importantes conceptos de respuesta, resultado, y salida de una computación.

5.1. GRAFOS DIRIGIDOS ACICLICOS.

Ya hemos justificado la necesidad de utilizar grafos para representar las expresiones de nuestro lenguaje. En esta sección resumimos las definiciones básicas relativas al concepto de grafo dirigido acíclico etiquetado ("labeled directed acyclic graph"), fijando la terminología empleada en el resto del trabajo, adaptándola convenientemente a nuestras necesidades.

5.1.1.-DEFINICION:

Sea $\mathfrak{N}$ un conjunto no vacío de **nodos** que representaremos por letras u, v, w, ... Sea $\longrightarrow \subseteq \mathfrak{N} \times \mathbb{N}_+ \times \mathfrak{N}$ una relación que llamaremos la relación de **paternidad.**

Si $(u,i,v) \in \longrightarrow$ escribimos $u \underset{i}{\longrightarrow} v$ y decimos que "**v es el i-ésimo hijo de u**".

Decimos que cada elemento de $\longrightarrow$ es un **arco**: u es el **origen** y v el **destino** del arco $u \underset{i}{\longrightarrow} v$.

Escribiremos abreviadamente:

$$u \underset{i}{\longrightarrow} \quad :\Longleftrightarrow \quad \exists v\ (u \underset{i}{\longrightarrow} v).$$

$$u \underset{i}{-\!\!\not\rightarrow} \quad :\Longleftrightarrow \quad \neg\, \exists v\ (u \underset{i}{\longrightarrow} v).$$

$$u \longrightarrow v \quad :\Longleftrightarrow \quad \exists i\ (u \underset{i}{\longrightarrow} v).$$

$$u -\!\!\not\rightarrow \ldots :\Longleftrightarrow \quad \neg\, \exists i\ (u \underset{i}{\longrightarrow}).$$

$$u \overset{*}{\longrightarrow} v \quad :\Longleftrightarrow \quad \exists u = u_0 \underset{i_1}{\longrightarrow} u_1 \underset{i_2}{\longrightarrow} \ldots \underset{i_n}{\longrightarrow} u_n = v.$$

Como es habitual, diremos que un nodo $u \in \mathfrak{N}$ es una **hoja** si $u -\!\!\not\rightarrow$; es decir, u no tiene descendientes.

Un **grafo dirigido acíclico (gda)** es un par $(\mathfrak{N}, \longrightarrow)$, donde $\mathfrak{N}$ es un conjunto de nodos y $\longrightarrow$ una relación de paternidad, que cumple las siguientes condiciones:

(1) Si un nodo tiene un hijo en la posición i, tiene hijos en todos los enteros positivos anteriores a i

$$\forall u \in \mathfrak{N},\ \forall i,j \in \mathbb{N}_+\ (u \xrightarrow[i]{} \wedge\ j < i \Longrightarrow u \xrightarrow[j]{}\ ).$$

(2) Un nodo no puede tener infinitos hijos

$$\forall u \in \mathfrak{N},\ \exists i \in \mathbb{N}_+\ (u \not\xrightarrow[i]{}\ v\ ).$$

(3) Para todo nodo u y todo entero positivo i existe como máximo un hijo en esa posición

$$\forall u \in \mathfrak{N},\ \forall i \in \mathbb{N}_+\ \exists^{\leq 1} v \in \mathfrak{N}\ (u \xrightarrow[i]{} v).$$

(4) No existen cadenas

$$u_0 \xrightarrow[i_1]{} u_1 \xrightarrow[i_2]{} u_2 \longrightarrow \ldots \xrightarrow[i_n]{} u_n$$

con $u_0 = u_n$ (es decir, no hay ciclos). ■

Los gda's en la terminología clásica de grafos son **multigrafos**: pueden tener más de un arco entre cada par de nodos.

5.1.2.-DEFINICION:

La **aridad** de un nodo en un gda es el número de hijos del nodo; o, de otro modo, el número de arcos que salen del nodo:

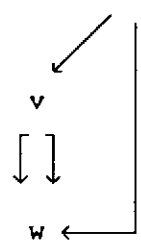
$$ar(u) :\Longleftrightarrow \min\{n \in \mathbb{N}_+ \mid (u \not\xrightarrow[n+1]{}\ )\}.$$

El **rango** de un nodo en un gda se define recursivamente:

$rg(u) = 0$, si $u \not\rightarrow$

$rg(u) = \max\{rg(v) + 1 \mid u \xrightarrow[i]{} v\}$, en otro caso. ■

Ejemplo:



$ar(w) = 0$ $\quad rg(w) = 0$

$ar(v) = 2$ $\quad rg(v) = 1$

$ar(u) = 2$ $\quad rg(u) = 2$

5.1.3.-DEFINICION:

Un gda con raíz, es un gda $(\mathcal{N}, \longrightarrow)$ que cumple además:

(5) Hay un nodo raiz; o, de otro modo, existe un único nodo que no es hijo de ninguno

$$\exists!\ \varepsilon \in \mathcal{N} \text{ tal que } \neg\ \exists u, i\ (u \xrightarrow[i]{} \varepsilon) \text{ con } u \in \mathcal{N} \text{ e } i \in \mathbb{N}_+.$$

(6) Todo nodo $u \in \mathcal{N}$ es accesible desde el nodo raiz

$$\forall u \in \mathcal{N},\ (\varepsilon \xrightarrow{*} u).$$

ε es accesible desde él mismo considerando la sucesión vacía. ■

5.1.4.-DEFINICION:

Un gda etiquetado $(\mathcal{N}, \longrightarrow, \gamma)$ es un gda $(\mathcal{N}, \longrightarrow)$ y una aplicación

$$\gamma:\ \mathcal{N} \longrightarrow VS \cup CS \cup FS$$

donde $\gamma[u]$ representa la imagen de u, tal que:

$$\forall u \in \mathcal{N},\ ar(u) = ar(\gamma[u])$$

con $ar(X) = 0$ para $X \in VS$.

Y que, además, cumple la siguiente condición: las variables en un gda etiquetado no pueden ocurrir más de una vez.

$$\forall X \in VS,\ \exists^{\leq 1} u \in \mathcal{N} \text{ tal que } \gamma[u] = X.$$ ■

Habitualmente escribiremos G[u] en vez de $\gamma[u]$ en un gda etiquetado $G = (\mathcal{N}, \longrightarrow, \gamma)$.

En lo que sigue representaremos por O(G), que denominaremos **posiciones** del gda G, el conjunto de nodos $\mathcal{N}$.

Podemos clasificar de manera obvia los nodos etiquetados de un grafo. Un nodo $u \in O(G)$ se denomina **nodo de variable** de un gda etiquetado si $G[u] \in VS$. Análogamente se define **nodo de constructora** y **nodo de operador**.

5.1.5.-DEFINICION:

Sea G un gda etiquetado, y sea $u \in O(G)$. Definimos G/u, **subgda**

etiquetado en la posición u, como el gda formado por el nodo u y sus descendientes.

Formalmente:

- $O(G/u) = \{v \in O(G) \mid u \longrightarrow^{*} v\}$
- Para todo $v,w \in O(G/u)$ $v \xrightarrow[i]{G} w \Longrightarrow v \xrightarrow[i]{G/u} w$
- $G/u[v] = G[v]$ para todo $v \in O(G/u)$
- u es el nodo raiz de G/u.

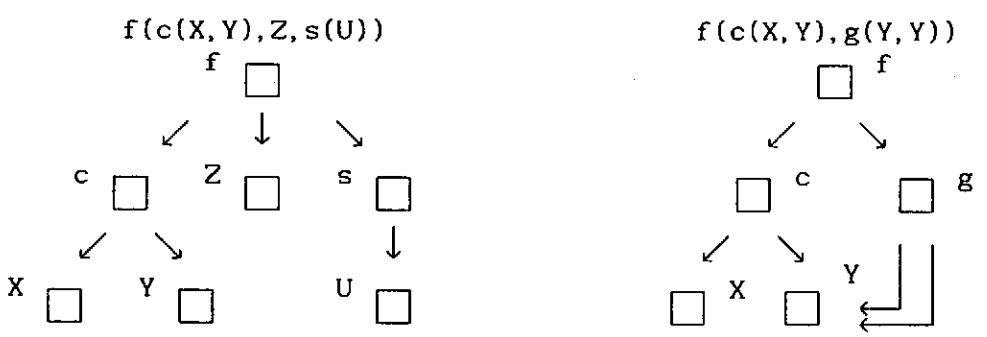
G/u es un gda etiquetado con raiz. ■

5.2. REPRESENTACION POR GDA'S

Es necesario para nuestro estudio representar las expresiones y las reglas de nuestro lenguaje como grafos dirigidos acíclicos.

Así, podemos asociar:

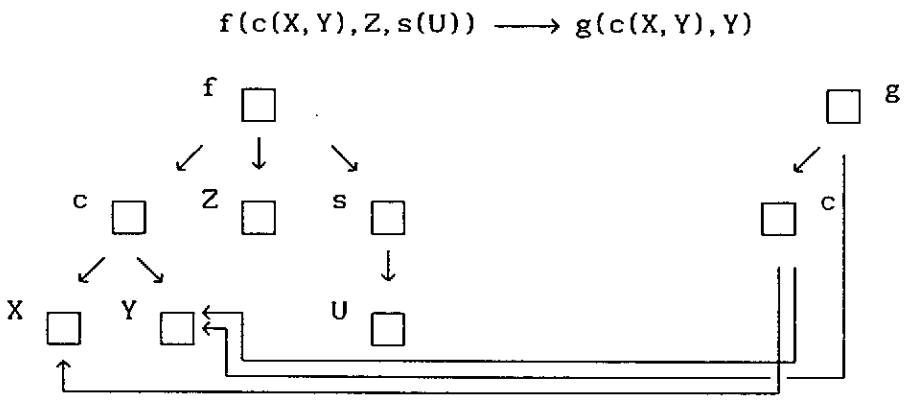
1.- **A una expresión e, en general, un gda etiquetado G_e con raíz en el cuál sólo están compartidas las variables.** Por ejemplo:



Cuando la expresión es un término lineal ó una parte izquierda de regla el gda etiquetado con raiz se convierte en un árbol etiquetado

(debido a que por la linealidad por la izquierda no hay variables repetidas).

2.- **A una regla l ⟶ r un gda etiquetado donde las variables comunes de l y r están compartidas.** La compartición de variables es necesaria por la definición 5.1.4. Por ejemplo:



Para abreviar, en lo sucesivo, llamaremos **expresión-gda** a un gda etiquetado con raiz. En particular, como vimos en el apartado anterior, dado un gda cualquiera G y una posición u ∈ O(G), G/u es un gda etiquetado con raiz u; y, por tanto, una expresión-gda.

El nombre viene justificado por el hecho de que podemos asociar al gda etiquetado G y a la posición u de G una expresión, que representamos **expr(G,u)** y que definimos por recursión sobre el rango de u:

Para todo u ∈ O(G):

- ▷ $G[u] = X \in VS \Longrightarrow expr(G,u) = X$
- ▷ $G[u] = k \in CS^0_\Sigma \cup FS^0_\Sigma \Longrightarrow expr(G,u) = k$
- ▷ $G[u] = k \in CS^n_\Sigma \cup FS^n_\Sigma,\ n>0 \wedge u \xrightarrow[i]{G} u_i,\ 1 \le i \le n \Longrightarrow$
  $expr(G,u) = k(expr(G,u_1),\ldots,expr(G,u_n))$.

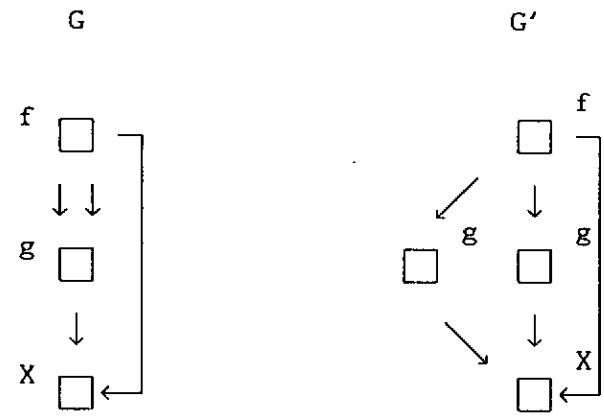
Hay que observar que expresiones-gda distintas pueden dar origen a la misma expresión. O, de otro modo, si la expresión **e** se representa como la expresión-gda G_e entonces G_e no es la única expresión-gda que "realiza" **e**. Llamaremos a G_e el **gda canónico** correpondiente a e.

5.2.1.-DEFINICION:

Sea G un gda etiquetado, $u \in O(G)$. Sea $e \in Exp_\Sigma$. Entonces:

G/u **realiza** e $:\Longleftrightarrow$ expr(G,u) = e. ■

Por ejemplo, las dos expresiones-gda G y G' siguientes:



realizan la misma expresión e = f(g(X),g(X),X). Sin embargo, G' ≡ G_e es la expresión-gda canónica que realiza e.

Lo primero que necesitamos en nuestra representación de las expresiones como gda's es un procedimiento para poder reflejar el efecto de las sustituciones.

5.2.2.-DEFINICION:

Sea G un gda etiquetado y sean u, v dos nodos de G distintos y tal que no hay un camino que vaya de v a u (es decir, $\neg(v \longrightarrow^{*} u)$). Definimos el procedimiento **reemplaza** asociando a G un gda G' como sigue:

G' = reemplaza(G,u,v)

es el gda etiquetado siguiente:

▷ $O(G') = O(G)$

▷ Para todo $u', v' \in O(G')$

$$u' \xrightarrow[i]{G} u \Longrightarrow u' \xrightarrow[i]{G'} v$$

$$u' \xrightarrow[i]{G} v' \wedge v' \neq u \Longrightarrow u' \xrightarrow[i]{G'} v'$$

Todos los arcos que van en G a u van en G' a v y los demás permanecen igual.

▷ $G'[w] = G[w]$ para todo $w \in O(G')$. ■

5.2.3.-PROPOSICION:

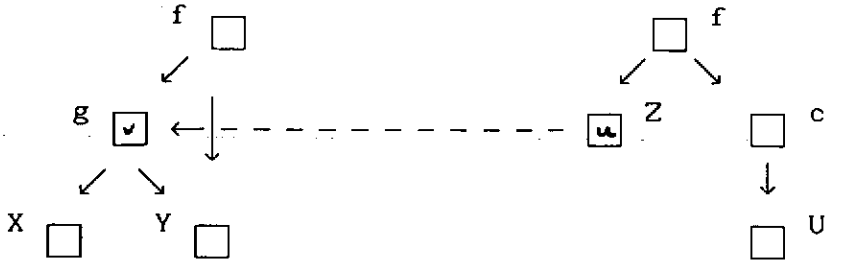
Sea G un gda etiquetado y sean u,v dos nodos de G distintos tal que $\neg(v \overset{*}{\longrightarrow} u)$. Entonces G' = reemplaza(G,u,v) es un gda etiquetado.

dem:

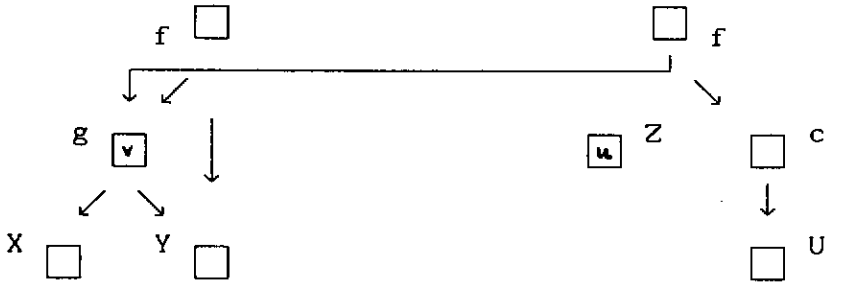
Es inmediato que G' es un gda. etiquetado comprobando las condiciones de la definición 5.1.1. En particular, la aciclidad se deduce de la condición impuesta a u y v. (Ver también 5.2.5). ■

El procedimiento reemplaza se puede imaginar "implementado" construyendo un **puntero que va del nodo u al nodo v**. Por ejemplo, sea G el gda etiquetado siguiente:

Con un "puntero" unimos los nodos u y v:



que representa el gda etiquetado G'= reemplaza(G,u,v)



Observar que si en el grafo G, u es un nodo de variable tal que G[u] = Z y σ es la sustitución $\{Z \longrightarrow expr(G,v)\}$ entonces para todo nodo $w \in O(G)$ se tiene:

$$expr(reemplaza(G,u,v),w) = \underline{if}\ (w = u)\ \underline{then}\ Z\ \underline{else}\ expr(G,w)\sigma.$$

La siguiente definición es simplemente una generalización del procedimiento reemplaza.

5.2.4.-DEFINICION:

Sea G un gda etiquetado.

Sean $u_1,\ldots,u_n$ n nodos diferentes de G, $1 \le i \le n$.

Sean $v_1,\ldots,v_n$ n nodos de G no necesariamente diferentes.

Supongamos, para todo i, j, $1 \le i, j \le n$, $\neg(v_i \longrightarrow^* u_j)$.

Definimos:

$$G' = G[u_1 \longrightarrow v_1, \ldots, u_n \longrightarrow v_n]$$

como sigue:

▷ $O(G') = O(G)$.

▷ Para todo $u', v' \in O(G')$

$$u' \xrightarrow[i]{G} u_j \Longrightarrow u' \xrightarrow[i]{G'} v_j \text{ para todo } j,\ 1 \leq j \leq n.$$

$$u' \xrightarrow[i]{G} v' \wedge v' \neq u_j \text{ para todo } j,\ 1 \leq j \leq n \Longrightarrow u' \xrightarrow[i]{G'} v'.$$

Todos los arcos que van en G a u_j van en G' a v_j y los demás permanecen igual.

▷ $G'[w] = G[w]$ para todo $w \in O(G')$. ■

5.2.5.-PROPOSICION:

Sea G un gda etiquetado; $u_1, \ldots, u_n$ nodos diferentes de G, $1 \leq i \leq n$. Sean $v_1, \ldots, v_n$ n nodos de G no necesariamente diferentes tal que para todo i, j, $1 \leq i, j \leq n$, $\neg(v_i \longrightarrow^{*} u_j)$.

Entonces: $G' = G[u_1 \longrightarrow v_1, \ldots, u_n \longrightarrow v_n]$ es un gda. etiquetado.

dem:

Las condiciones (1) (2) y (3) de la definición 5.1.1 y las de la definición 5.1.4 son inmediatas. Falta demostrar la aciclidad: Ningún camino en G' puede ser un ciclo. Observar que los arcos de un camino en G' pueden ser del gda G o bien de la forma $u' \longrightarrow v_i$ que reemplazan a los de G de la forma $u' \longrightarrow u_i$ para $1 \leq i \leq n$.

Sea un camino de G' cualquiera:

$$w_0 \longrightarrow w_1 \longrightarrow w_2 \longrightarrow \ldots \longrightarrow w_p = w_0$$

y que sea además un ciclo. Entonces al menos un arco del camino debe pertenecer sólo a G' (y no a G) porque si no ocurriese eso el camino sería de G contradiciendo el hecho que G es aciclico. Supongamos que el arco de G' (no de G) más a la izquierda en el camino es $u^i \longrightarrow v_r$ para un cierto r, $1 \leq r \leq n$. Es decir, el camino anterior se podría también representar por:

$$w_0 \longrightarrow^* u^i \longrightarrow v_r \longrightarrow^* w_0.$$

Si no hay ningún otro arco de G' (que no sea de G) tendríamos que se podría formar en G el camino:

$$v_r \longrightarrow w_0 \longrightarrow^* u^i \longrightarrow u_r.$$

contradiciendo la hipótesis que $\neg(v_i \longrightarrow^* u_j)$, $\forall i,j,\ 1 \leq i,j \leq n$.

Si hay otro arco, al menos, de G' (que no sea de G) tendríamos que estaría en el camino después de v_r y podemos suponer sin pérdida de generalidad que no hay otro arco de G' (no de G) intermedio:

$$w_0 \longrightarrow^* u^i \longrightarrow v_r \longrightarrow^* u^j \longrightarrow v_s \longrightarrow^* w_0$$

Entonces, se podría formar en G el camino:

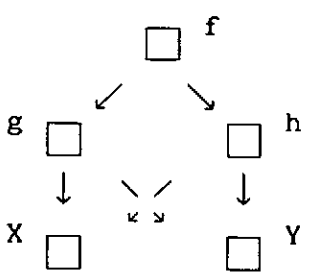
$$v_r \longrightarrow^* u^j \longrightarrow u_s$$

contradiciendo nuevamente la hipótesis $\neg(v_i \longrightarrow^* u_j)$, $\forall i,j,\ 1 \leq i,j \leq n$. ■

Observar que, en particular, $G[u \longrightarrow v] = \text{reemplaza}(G,u,v)$.

Por otra parte, el ejemplo siguiente muestra porqué hay que distinguir entre sustitución simultánea como se realiza por el procedimiento $G[u_1 \longrightarrow v_1, \ldots, u_n \longrightarrow v_n]$ y composición de sustituciones.

Sea G el gda etiquetado:



Entonces:

$$\text{expr}(G,u) = f(g(X,Y),h(Y,X)) = e$$

$$\text{expr}(G,v) = X;\ \text{expr}(G,w) = Y$$

e $\{X \rightarrow Y,\ Y \rightarrow X\}$ = f(g(Y,X),h(X,Y)) = expr(G[v→w,w→v]),u)

e $\{X \rightarrow Y\}\ \{Y \rightarrow X\}$ = f(g(X,X),h(X,X)) = expr(G[v→w][w→v]),u)

5.3. UNIFICACION

En esencia, el problema general de unificación sintáctica puede expresarse como sigue: Dadas dos expresiones que contienen algunas variables hallar, si es que existe, la sustitución más general que hace a las dos expresiones iguales. La sustitución resultante es el unificador más general y es único salvo renombramiento de variables. Se han propuesto muchos algoritmos de unificación desde el inicial de Robinson [Robinson 65] con objeto de superar la ineficiencia detectada en muchos casos. Nuestro estudio está próximo al algoritmo descrito utilizando gda's en [Corbin & Bidoit 83].

El problema de **unificación lineal** que se nos plantea, en el lenguaje que hemos descrito, tiene algunas condiciones más debido a la disciplina de constructores impuesta. Esto añade algunas características a las sustituciones que se obtienen.

5.3.1.-DEFINICION: d-sustitución, sust. lineal y sust. apartada

Decimos que una sustitución σ es una **d-sustitución** si y sólo si $X\sigma \in Term_{\Sigma}$ para todo $X \in dom(\sigma)$. (ver apartado 1.2.2)

Una sustitución σ es **lineal** si y sólo si:

1) $X\sigma$ es lineal para todo $X \in dom(\sigma)$

2) $var(X\sigma) \cap var(Y\sigma) = \emptyset$ si $X,Y \in dom(\sigma)$ y $X \neq Y$.

Una sustitución σ está **apartada** de un conjunto de variables $V \subseteq VAR$ si y sólo si $ran(\sigma) \cap V = \emptyset$. (ver 1.2.2) ■

Necesitamos describir dentro de un grafo dirigido acíclico un **algoritmo que nos permita unificar árboles y expresiones-gda**. Esto se corresponde con la unificación de lados izquierdos de reglas y expresiones objetivo o que surjan en el transcurso de la computación. Para ello, consideraremos el gda etiquetado como un **objeto global** que incluye el árbol asociado al lado izquierdo de la regla y la expresión-gda.

Podemos establecer el siguiente algoritmo:

ALGORITMO DE UNIFICACION

Consideramos globalmente un gda etiquetado G.

-input:

Un **par de nodos** (u,v) de G con algunas condiciones:

1) $u \neq v$

2) G/u y G/v no tienen nodos comunes.

3) expr(G,u) = l, lado izquierdo de una regla.

4) expr(G,v) = e, expresión.

Sean:

var(l) = $\{X_1,\ldots,X_p\}$, donde tenemos $G[u_1] = X_1, \ldots, G[u_p] = X_p$

var(e) = $\{Y_1,\ldots,Y_q\}$, donde $G[v_1] = Y_1, \ldots, G[v_q] = Y_q$

var(l) $\cap$ var(e) = $\emptyset$.

-output:

Un **quintuple** (G', infor, I', O', P') donde:

1) G' es un gda etiquetado;

2) **infor** es una información del resultado de **fallo** en la unificación;

3) I', O' son listas de nodos; y,

4) P' es un conjunto de **nodos pendientes** de unificación.

Lo que da origen a tres posibles resultados:

a) EXITO.

Si infor = NO, $I' = [u'_1,\ldots,u'_p]$, $O' = [v'_1,\ldots,v'_q]$, $P' = \{\}$ entonces devuelve como salida EXITO, y como resultado:

$$(G',\ \text{EXITO},\ [u'_1,\ldots,u'_p],\ [v'_1,\ldots,v'_q])$$

G' es el nuevo gda etiquetado;

$u'_1, \ldots, u'_p, v'_1, \ldots, v'_q$ son nodos de G'. Los que han sustituido a $u_1, \ldots, u_p, v_1, \ldots, v_q$ que son aquellos que inicialmente realizaban las variables $X_1, \ldots, X_p, Y_1, \ldots, Y_q$.

Debe cumplirse, en este caso, lo siguiente:

expr(G',v'_j) es un subtérmino s_j de l para todo $1 \leq j \leq q$.

expr(G',u'_i) es una expresión e_i para todo $1 \leq i \leq p$.

Las sustituciones:

$$\lambda = \{X_1 \longrightarrow e_1,\ldots,X_p \longrightarrow e_p\}$$

omitiendo $X_i \longrightarrow e_i$ si $X_i = e_i$

$$\sigma = \{Y_1 \longrightarrow s_1,\ldots,Y_q \longrightarrow s_q\}$$

omitiendo $Y_j \longrightarrow s_j$ si $s_j = Y_j$

verifican:

$$l\ \lambda = e\ \sigma.$$

b) FALLO.

Si infor = SI devuelve como salida FALLO y como resultado:

$$(G',\ \text{FALLO}).$$

c) UNIFICACION SUSPENDIDA.

Si $P' \neq \emptyset$ entonces devuelve como salida SUSPENDIDA y el resultado es:

$$(\ G',\ \text{SUSP},\ \{w_1,\ \ldots,\ w_k\}\ )$$

donde cada w_i es un nodo de G' tal que G'/w_i está etiquetado en la

raiz por un símbolo de operación. En este caso expr(G',w_1) para todo $1 \le i \le k$ son expresiones demandadas que requieren más evaluación antes de que la unificación pueda realizarse.

-algoritmo:

Una llamada:

unifica(G, u, v, [u_1, ..., u_p], [v_1, ..., v_q])

es posible con las condiciones impuestas a G, u y v; siendo [u_1, ..., u_p], y [v_1, ..., v_q] las listas de nodos que realizan las variables de expr(G,u) y expr(G,v) respectivamente.

Definimos:

unifica(G, u, v, I, O) = unif(G, u, v, I, O, {})

donde unif(G,u,v,I,O,P) se define recursivamente usando las reglas siguientes:

(U1) **G[u] = G[v] = k** $\in$ **CS$_n$** $\cup$ **FS$_n$**;

$\wedge$ **u** $\xrightarrow[i]{G}$ **u$_i$**, **v** $\xrightarrow[i]{G}$ **v$_i$**, **$1 \le i \le n$** ;

$\wedge$ **Us = [u$_1$,...,u$_n$], Vs = [v$_1$,...,v$_n$]** $\Longrightarrow$

unif(G,u,v,I,O,P) = unif-lis(G,Us,Vs,I,O,P)

donde

unif-lis(G,[],[],I,O,P) = (G,NO,I,O,P)

unif-lis(G,[u :Us],[v :Vs],I,O,P) =

let unif(G,u,v,I,O,P) = (G',infor,I',O',P')

in if infor = SI

then (G',SI,I',O',P')

else unif-lis(G',Us,Vs,I',O',P')

(U2) **expr(G,u) no variable**

$\wedge$ **expr(G,v) variable** $\Longrightarrow$

unif(G,u,v,I,O,P) = (G',NO,I',O',P)

donde G' = reemplaza(G,v,u)

I' = sustituye(I,v,u)

O' = sustituye(O,v,u).

(U3) **expr(G,u) variable** $\Longrightarrow$

unif(G,u,v,I,O,P) = (G',NO,I',O',P)

donde G' = reemplaza(G,u,v)

I' = sustituye(I,u,v)

O' = sustituye(O,u,v).

(U4) **G/u = c** $\in$ **CS** $\wedge$ **G/v = f** $\in$ **FS** $\Longrightarrow$

unif(G,u,v,I,O,P) = (G,NO,I,O,P $\cup$ {v}),

(U5) **G/u = c** $\in$ **CS** $\wedge$ **G/v = d** $\in$ **CS** $\wedge$ **c** $\neq$ **d** $\Longrightarrow$

unif(G,u,v,I,O,P) = (G,SI,I,O,P).

Donde la definición de

sustituye(L,u,v) (% sustituye en la lista L el elemento u por el elemento v) es recursivamente:

sustituye([],u,v) = []

sustituye([u : Ws],u,v) = [v : Ws]

sustituye([x : Ws],u,v) = [x : sustituye(Ws,u,v)], si x $\neq$ u. ■

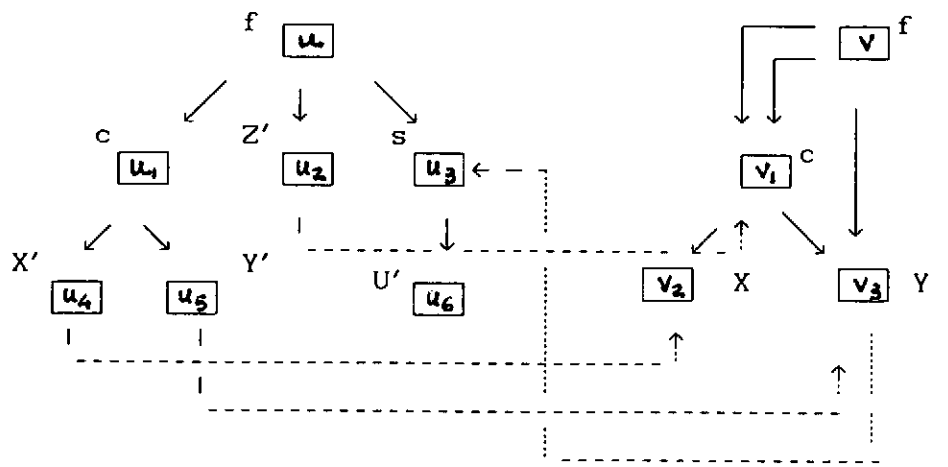
A continuación, se muestran algunos ejemplos para la mejor comprensión del funcionamiento del algoritmo.

EJEMPLO 1: Sea el gda etiquetado G:

donde:

$expr(G,u) = 1 = f(c(X',Y'),Z',s(U'))$

$expr(G,v) = e = f(c(X,Y),c(X,Y),Y)$



LLamada inicial: $unifica(G,u,v,[u_4,u_5,u_2,u_6],[v_2,v_3])$

$unifica(G,u,v,[u_4,u_5,u_2,u_6],[v_2,v_3]) \Longrightarrow$

$unif(G,u,v,[u_4,u_5,u_2,u_6],[v_2,v_3],\{\}) \Longrightarrow$

$unif\text{-}lis(G,[u_1,u_2,u_3],[v_1,v_1,v_3],[u_4,u_5,u_2,u_6],[v_2,v_3],\{\}) \Longrightarrow^*$

$unif\text{-}lis(G_2,[u_2,u_3],[v_1,v_3],[v_2,v_3,u_2,u_6],[v_2,v_3],\{\}) \Longrightarrow^*$

$unif\text{-}lis(G_3,[u_3],[v_3],[v_2,v_3,v_1,u_6],[v_2,v_3],\{\}) \Longrightarrow^*$

$unif\text{-}lis(G_4,[],[],[v_2,u_3,v_1,u_6],[v_2,u_3],\{\}) \Longrightarrow^*$

$(G_4,NO,[v_2,u_3,v_1,u_6],[v_2,u_3],\{\})$

ya que

$unif(G,u_1,v_1,[u_4,u_5,u_2,u_6],[v_2,v_3],\{\}) \Longrightarrow$

$unif\text{-}lis(G,[u_4,u_5],[v_2,v_3],[u_4,u_5,u_2,u_6],[v_2,v_3],\{\}) \Longrightarrow^*$

$unif\text{-}lis(G_1,[u_5],[v_3],[v_2,u_5,u_2,u_6],[v_2,v_3],\{\}) \Longrightarrow^*$

$unif\text{-}lis(G_2,[],[],[v_2,v_3,u_2,u_6],[v_2,v_3],\{\}) \Longrightarrow^*$

$(G_2,NO,[v_2,v_3,u_2,u_6],[v_2,v_3],\{\})$

donde

$$unif(G, u_4, v_2, [u_4, u_5, u_2, u_6], [v_2, v_3], \{\}) \Longrightarrow$$

$$(G_1, NO, [v_2, u_5, u_2, u_6], [v_2, v_3], \{\})$$

siendo G_1= reemplaza(G, u_4, v_2).

$$unif(G_1, u_5, v_3, [v_2, v_3, u_2, u_6], [v_2, v_3], \{\}) \Longrightarrow$$

$$(G_2, NO, [v_2, v_3, u_2, u_6], [v_2, v_3], \{\})$$

siendo G_2= reemplaza (G_1, u_5, v_3).

$$unif(G_2, u_2, v_1, [v_2, v_3, v_1, u_6], [v_2, v_3], \{\}) \Longrightarrow$$

$$(G_3, NO, [v_2, v_3, v_1, u_6], [v_2, v_3], \{\})$$

siendo G_3= reemplaza (G_2, u_2, v_1).

$$unif(G_3, u_3, v_3, [v_2, v_3, v_1, u_6], [v_2, v_3], \{\}) \Longrightarrow$$

$$(G_4, NO, [v_2, u_3, v_1, u_6], [v_2, u_3], \{\})$$

siendo G_4 =reemplaza (G_3, v_3, u_3).

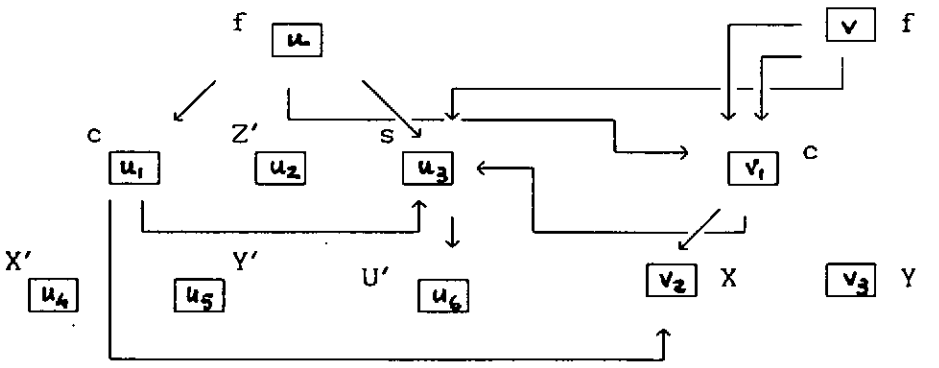
Resultado: $(G_4,$ EXITO, $[v_2, u_3, v_1, u_6]$, $[v_2, u_3])$

$\lambda = \{X' \rightarrow X,\ Y' \rightarrow s(U'),\ Z' \rightarrow c(X, s(U'))\}$

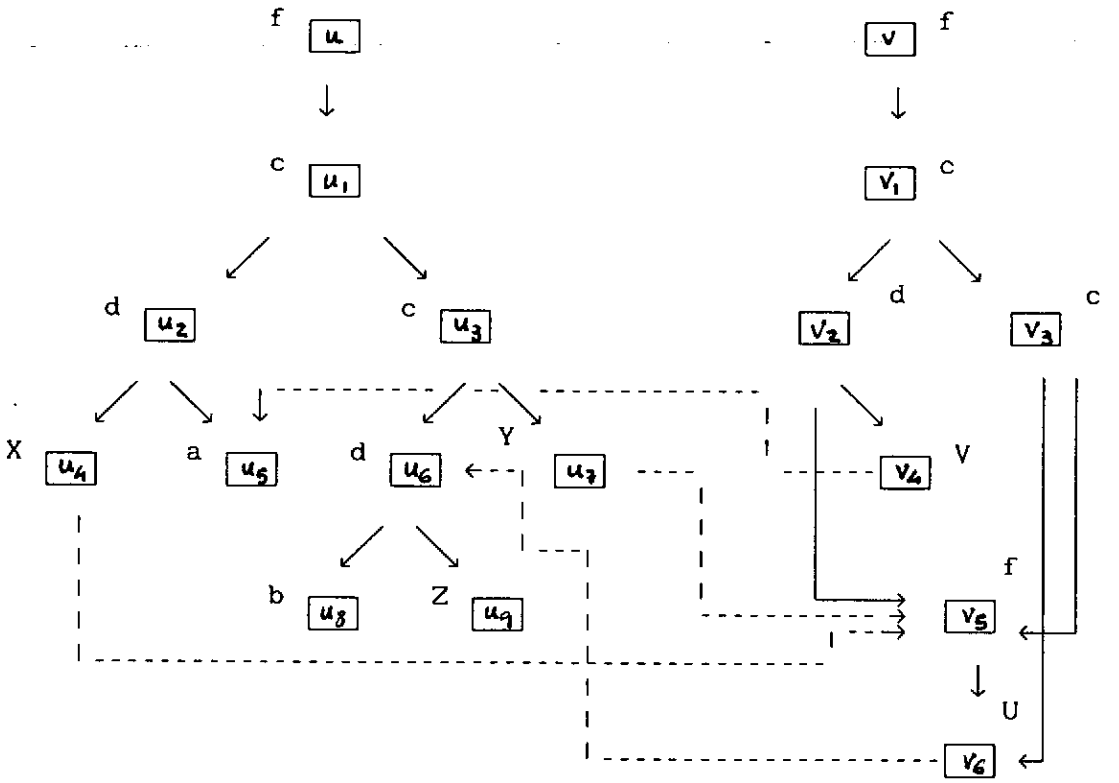
$\sigma = \{Y \rightarrow s(U')\}$.

$l\ \lambda = f(c(X, s(U')), c(X, s(U')), s(U')) = e\ \sigma$

El grafo final G_4 tendría la forma siguiente:



EJEMPLO 2: Sea G el gda etiquetado:



donde:

expr(G,u) = l = f(c(d(X,a),c(d(b,Z),Y)))

expr(G,v) = e = f(c(d(f(U),V),c(U,f(U))))

Llamada inicial: unifica(G,u,v,[u_4,u_9,u_7],[v_4,v_6])

unifica(G,u,v,[u_4,u_9,u_7],[v_4,v_6]) $\Longrightarrow$

unif(G,u,v,[u_4,u_9,u_7],[v_4,v_6],{}) $\overset{*}{\Longrightarrow}$

unif-lis(G,[u_1],[v_1],[u_4,u_9,u_7],[v_4,v_6],{}) $\overset{*}{\Longrightarrow}$

unif-lis(G_4,[],[],[v_5,u_9,v_5],[u_5,u_6],{}) $\overset{*}{\Longrightarrow}$

(G_4,NO,[v_5,u_9,v_5],[u_5,u_6],{})

ya que

unif(G,u_1,v_1,[u_4,u_9,u_7],[v_4,v_6],{}) $\overset{*}{\Longrightarrow}$

unif-lis(G,[u_2,u_3],[v_2,v_3],[u_4,u_9,u_7],[v_4,v_6],{}) $\overset{*}{\Longrightarrow}$

$unif\text{-}lis(G_2, [u_3], [v_3], [v_5, u_9, u_7], [u_5, v_6], \{\}) \Longrightarrow^*$

$unif\text{-}lis(G_4, [], [], [v_5, u_9, v_5], [u_5, u_6], \{\}) \Longrightarrow^*$

$(G_4, NO, [v_5, u_9, v_5], [u_5, u_6], \{\})$

y

$unif(G, u_2, v_2, [u_4, u_9, u_7], [v_4, v_6], \{\}) \Longrightarrow^*$

$unif\text{-}lis(G, [u_4, u_5], [v_5, v_4], [u_4, u_9, u_7], [v_4, v_6], \{\}) \Longrightarrow^*$

$unif\text{-}lis(G_1, [u_5], [v_4], [v_5, u_9, u_7], [v_4, v_6], \{\}) \Longrightarrow^*$

$unif\text{-}lis(G_2, [], [], [v_5, u_9, u_7], [u_5, v_6], \{\}) \Longrightarrow^*$

$(G_2, NO, [v_5, u_9, u_7], [u_5, v_6], \{\})$

y

$unif(G_2, u_3, v_3, [v_5, u_9, u_7], [u_5, v_6], \{\}) \Longrightarrow^*$

$unif\text{-}lis(G_2, [u_6, u_7], [v_6, v_5], [v_5, u_9, u_7], [u_5, v_6], \{\}) \Longrightarrow^*$

$unif\text{-}lis(G_3, [u_7], [v_5], [v_5, u_9, u_7], [u_5, u_6], \{\}) \Longrightarrow^*$

$unif\text{-}lis(G_4, [], [], [v_5, u_9, v_5], [u_5, u_6], \{\}) \Longrightarrow^*$

$(G_4, NO, [v_5, u_9, v_5], [u_5, u_6], \{\})$

donde

$unif(G, u_4, v_5, [u_4, u_9, u_7], [v_4, v_6], \{\}) \Longrightarrow$

$(G_1, NO, [v_5, u_9, u_7], [v_4, v_6], \{\})$

siendo G_1 = reemplaza(G, u_4, v_5).

$unif(G_1, u_5, v_4, [v_5, u_9, u_7], [v_4, v_6], \{\}) \Longrightarrow$

$(G_2, NO, [v_5, u_9, u_7], [u_5, v_6], \{\})$

siendo G_2 = reemplaza(G_1, v_4, u_5).

$unif(G_2, u_6, v_6, [v_5, u_9, u_7], [u_5, v_6], \{\}) \Longrightarrow$

$(G_3, NO, [v_5, u_9, u_7], [u_5, u_6], \{\})$

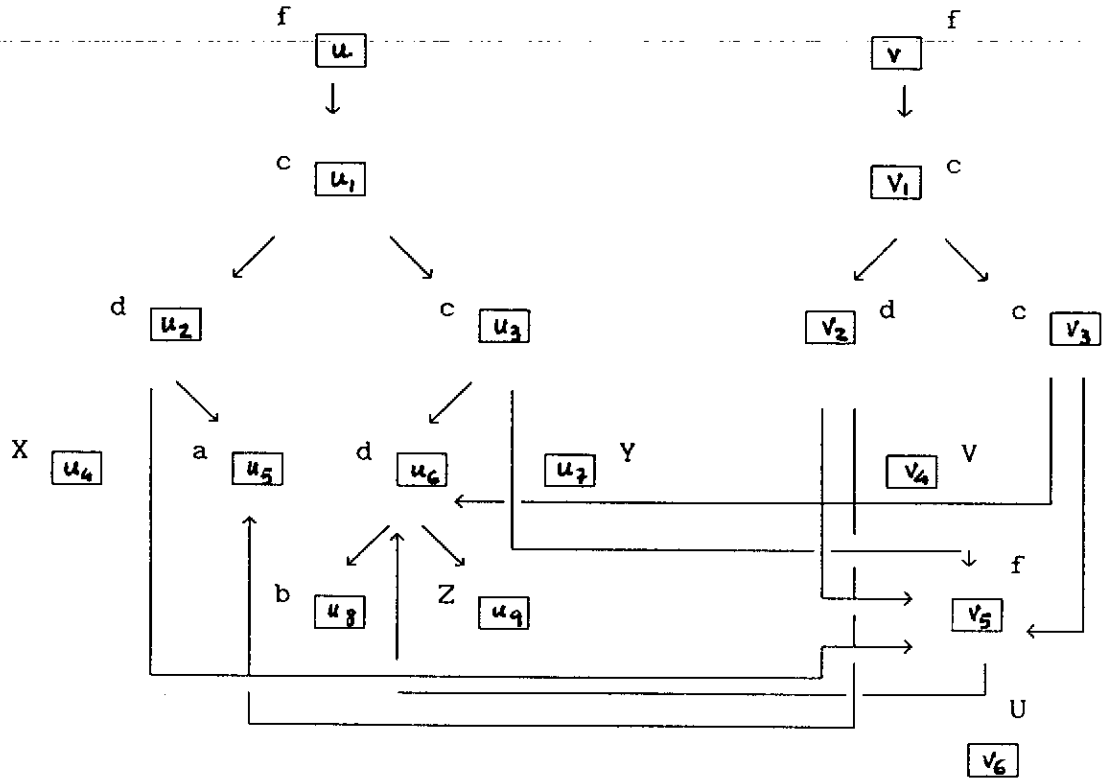
siendo G_3 = reemplaza(G_2, v_6, u_6).

$unif(G_3, u_7, v_5, [v_5, u_9, u_7], [u_5, u_6], \{\}) \Longrightarrow$

$(G_4, NO, [v_5, u_9, v_5], [u_5, u_6], \{\})$

siendo G_4 = reemplaza(G_3, u_7, v_5).

El grafo final G_4 tendría la forma siguiente:



donde:

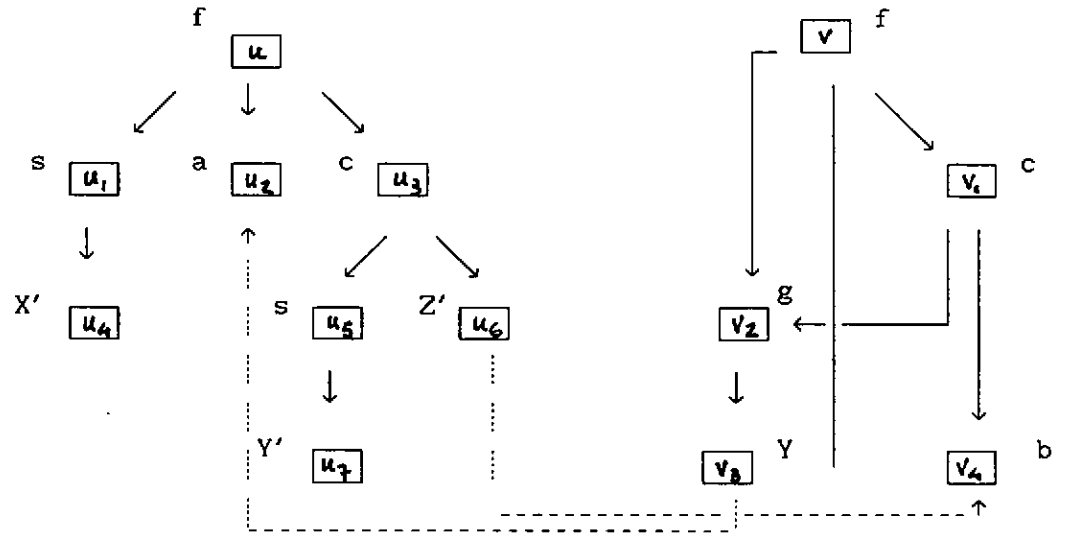
Resultado: (G_4, EXITO, [v_5, u_9, v_5], [u_5, u_6])

$\lambda = \{X \rightarrow f(d(b,Z),\ Y \rightarrow f(d(b,Z))\}$

$\sigma = \{V \rightarrow a,\ U \rightarrow d(b,Z)\}$.

$1\ \lambda = f(c(d(f(d(b,Z)),a),c(d(b,Z),f(d(b,Z))))) = e\ \sigma$

EJEMPLO 3: Sea el gda etiquetado G:



donde:

$expr(G,u) = l = f(s(X'),a,c(s(Y'),Z'))$

$expr(G,v) = e = f(g(Y),Y,c(g(Y),b))$

Llamada inicial: $unifica(G,u,v,[u_4,u_7,u_6],[v_3])$

$unifica(G,u,v,[u_4,u_7,u_6],[v_3]) \Longrightarrow$

$unif(G,u,v,[u_4,u_7,u_6],[v_3],\{\}) \Longrightarrow^*$

$unif\text{-}lis(G,[u_1,u_2,u_3],[v_2,v_4,v_1],[u_4,u_7,u_6],[v_3],\{\}) \Longrightarrow^*$

$unif\text{-}lis(G,[u_2,u_3],[v_4,v_1],[u_4,u_7,u_6],[v_3],\{v_2\}) \Longrightarrow^*$

$unif\text{-}lis(G_1,[u_3],[v_1],[u_4,u_7,u_6],[u_2],\{v_2\}) \Longrightarrow^*$

$unif\text{-}lis(G_2,[],[],[u_4,u_7,v_3],[u_2],\{v_2\}) \Longrightarrow^*$

$(G_2,NO,[u_4,u_7,v_3],[u_2],\{v_2\})$

ya que

$unif(G,u_1,v_2,[u_4,u_7,u_6],[v_3],\{\}) \longrightarrow$

$(G,NO,[u_4,u_7,u_6],[v_3],\{v_2\})$

y en u_1 hay un constructor y en v_2 hay un símbolo de función.

$unif(G,u_2,v_3,[u_4,u_7,u_6],[v_3],\{v_2\}) \Longrightarrow$

$(G_1, NO, [u_4, u_7, u_6], [u_2], \{v_2\})$

siendo G_1= reemplaza(G, v_3, u_2).

y hemos obtenido

unif$(G_1, u_3, v_1, [u_4, u_7, u_6], [u_2], \{v_2\}) \Longrightarrow$

unif-lis$(G_1, [u_5, u_6], [v_2, v_4], [u_4, u_7, u_6], [u_2], \{v_2\}) \overset{*}{\Longrightarrow}$

unif-lis$(G_1, [u_6], [v_4], [u_4, u_7, u_6], [u_2], \{v_2\}) \overset{*}{\Longrightarrow}$

unif-lis$(G_2, [], [], [u_4, u_7, v_3], [u_2], \{v_2\}) \overset{*}{\Longrightarrow}$

$(G_2, NO, [u_4, u_7, v_3], [u_2], \{v_2\})$

donde

unif$(G_1, u_5, v_2, [u_4, u_7, u_6], [u_2], \{v_2\}) \Longrightarrow$

$(G_1, NO, [u_4, u_7, u_6], [u_2], \{v_2\})$

ya que en u_5 hay un constructor y en v_2 hay un símbolo de función.

y

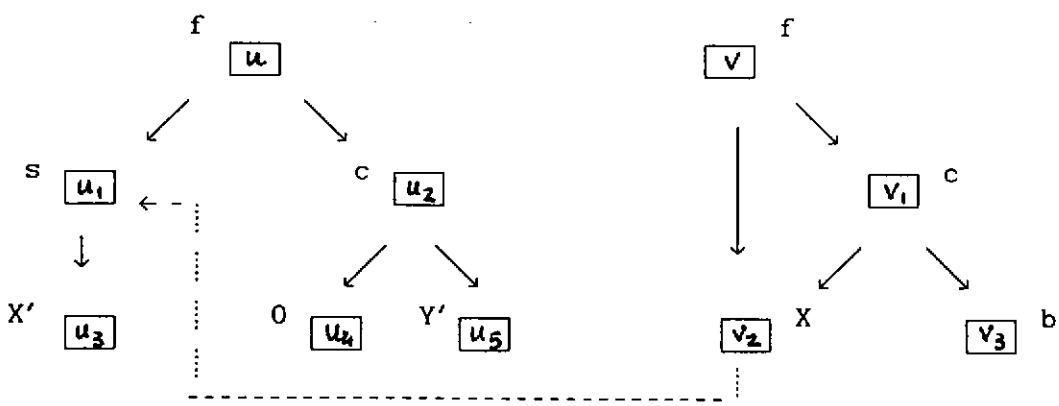
unif$(G_1, u_6, v_3, [u_4, u_7, u_6], [u_2], \{v_2\}) \longrightarrow$

$(G_2, NO, [u_4, u_7, v_3], [u_2], \{v_2\})$

siendo G_2= reemplaza(G_1, u_6, v_3).

Resultado: $(G_2,$ SUSP, $\{v_2\})$.

EJEMPLO 4: Sea el gda etiquetado G:



donde

$expr(G,u) = 1 = f(s(X'),c(0,Y'))$

$expr(G,v) = e = f(X,c(X,b))$

Llamada inicial: $unifica(G,u,v,[u_3,u_5],[v_2])$

$unifica(G,u,v,[u_3,u_5],[v_2]) \Longrightarrow$

$unif(G,u,v,[u_3,u_5],[v_2],\{\}) \Longrightarrow^{*}$

$unif\text{-}lis(G,[u_1,u_2],[v_2,v_1],[u_3,u_5],[v_2],\{\}) \Longrightarrow^{*}$

$unif\text{-}lis(G_1,[u_2],[v_1],[u_3,u_5],[u_1],\{\}) \Longrightarrow^{*}$

$(G_1,SI,,[u_3,u_5],[u_1],\{\})$

donde

$unif(G,u_1,v_2,[u_3,u_5],[v_2],\{\}) \Longrightarrow$

$(G_1,NO,[u_3,u_5],[u_1],\{\})$

siendo $G_1 = reemplaza(G,v_2,u_1)$.

y

$unif(G_1,u_2,v_1,[u_3,u_5],[u_1],\{\}) \Longrightarrow$

$unif\text{-}lis(G_1,[u_4,u_5],[u_1,v_3],[u_3,u_5],[u_1],\{\}) \Longrightarrow^{*}$

$(G_1,SI,,[u_3,u_5],[u_1],\{\})$

donde

$unif(G_1,u_4,u_1,[u_3,u_5],[u_1],\{\}) \Longrightarrow$

$(G_1,SI,,[u_3,u_5],[u_1],\{\})$

porque $G_1[u_4] = 0$; $G_1[u_1] = s$; 0 y s constructores distintos.

Resultado: $(G_1,$ FALLO).

OBSERVACION:

No presenta dificultad añadir el **occur-check** si se requiriera. En nuestro caso no es necesario por la linealidad de los lados izquierdos de reglas.

5.3.2. TEOREMA: (Unificación)

El algoritmo de unificación con una llamada inicial:

$$\text{unifica}(\ G,\ u,\ v,\ [u_1, \ldots, u_p],\ [v_1, \ldots, v_q]\ )$$

siempre termina. Además, devuelve EXITO si y sólo si expr(G,u), el lado izquierdo de una regla, y expr(G,v), la expresión correspondiente a una expresión-gda, son unificables. En este caso, se tiene que $\lambda \cup \sigma$ es un unificador más general (u.m.g.) de expr(G,u) y expr(G,v); y que σ es una d-sustitución lineal. Y, si G' es el grafo final, G'/u y G'/v son expresiones-gda semánticamente equivalentes.

dem:

Tanto la terminación como la obtención de un unificador más general en el caso que las dos expresiones sean unificables es consecuencia de que el algoritmo se comporta esencialmente como la unificación clásica de Robinson. Una modificación del algoritmo clásico próxima al nuestro puede verse en [Corbin & Bidoit 83]. Que σ es una d-sustitución lineal es inmediato por la linealidad de los lados izquierdos de reglas.

El ejemplo 1 anterior muestra que los grafos de las expresiones-gda que se obtienen al final del algoritmo pueden no ser idénticos pero sí son semánticamente equivalentes debido al determinismo de las constructoras. ■

5.4. REDUCCION POR ESTRECHAMIENTO EN GRAFOS

El problema general que afrontamos a continuación es el de **reducir** una expresión e utilizando las reglas de un programa Π. Ya hemos visto que es necesario para la corrección de la reducción

representar la expresión e como un gda. Esta representación puede hacerse como un gda etiquetado con raiz canónico G_e (ver 5.2) en el que sólo las variables, en su caso, aparecen compartidas. Pero, a lo largo del proceso de reducción pueden aparecer expresiones-gda más complejas (es decir, gda's con otros símbolos, constructores o/y operadores, diferentes de los de variable compartidos). Este hecho es necesario y fundamental tenerlo en cuenta en la construcción del proceso de reducción. Por este motivo estudiamos en el apartado anterior la unificación en el marco de grafos dirigidos aciclicos etiquetados considerando dos nodos u y v de un gda G de tal modo que el subgrafo G/u fuese un árbol correspondiente al lado izquierdo de una regla y el subgrafo G/v correspondiese a una expresión-gda.

Comenzamos estudiando el **paso de estrechamiento**. Partimos de la situación general siguiente: Sea una regla $l \longrightarrow r$ de un programa Π que deseamos aplicar a la expresión-gda G en la posición u. Podemos suponer las variables de G distintas de las de l y r; y que pueden existir **variables libres** en la regla (variables de r que no están en l).

Procedemos como sigue: Construimos un grafo $\mathfrak{G}$ en el que se representan los lados izquierdo l y derecho r de la regla y la expresión-gda G que intentamos estrechar en la posición u. Como $\mathfrak{G}$ debe ser un gda etiquetado las variables de l y r comunes deben estar en los mismos nodos (**variables compartidas**). El resto de los nodos del grafo en que se representan l, r, y G podemos suponerlos distintos sin pérdida de generalidad.

Para dar el paso de estrechamiento:

1) Se **unifican** el árbol correspondiente a l y la expresión-gda G/u. Si

se tiene EXITO puede proseguirse como se indica a continuación.

2) Se establece un **puntero** de u al nodo raiz de r. Observar que un puntero no es más que una ayuda gráfica equivalente a aplicar el procedimiento reemplaza anteriormente definido.

3) El resultado es el **grafo accesible** desde la antigua raiz de G.

Todo ello es formulable en un grafo con ayuda de punteros y depende de:

- ▷ El grafo total
- ▷ Los nodos raices de l, r y G
- ▷ La posición reducible u de G.

5.4.1.-DEFINICION:

Sea G una expresión-gda.

Sea $l \longrightarrow r$ una variante de regla de un programa Π, separada de G. O sea, $var(l \longrightarrow r) \cap var(G) = \varnothing$.

Sea $u \in O(G)$ una posición en la que vamos a aplicar la regla.

Sea σ una d-sustitución lineal.

Decimos que la expresión-gda G **estrecha** en la posición u **en un paso** a la expresión-gda G' y escribimos

$$G \xrightarrow[\sigma]{l \longrightarrow r, u} G'$$

si y sólo si G' se obtiene a partir de G según el siguiente procedimiento:

1) Construir $\mathfrak{G}_0$:

el gda etiquetado formado por l, r y G donde las variables comunes a l y r están compartidas y donde los nodos de G son distintos de los de l y r. Es decir, si ε_l, ε_r, ε_G son los nodos raiz de l, r y G respectivamente; $\mathfrak{G}_0$ es tal que $\mathfrak{G}_0/\varepsilon_l$ es el árbol correspondiente a l, $\mathfrak{G}_0/\varepsilon_r$ es el gda etiquetado con raiz que se asocia a r, y $\mathfrak{G}_0/\varepsilon_G$ es la

expresión-gda G.

2) Construir $\mathfrak{G}_1$:

el grafo que se obtiene al aplicar con EXITO el algoritmo de unificación al gda $\mathfrak{G}_0$ tomando como inputs los nodos u y ε_1. Llamamos σ a la d-sustitución sobre variables de G que se obtiene en el proceso de unificación:

$$expr(\mathfrak{G}_0, u)\ \sigma = expr(\mathfrak{G}_0, \varepsilon_1)\ \lambda = l\ \lambda.$$

donde λ es la sustitución que se obtiene ligando las variables de l con las expresiones que realizan las expresiones-gda que son destino de los punteros cuyo origen son las variables de l.

3) Obtener G':

Si $u \equiv \varepsilon_G$ entonces se toma como $G' = \mathfrak{G}_1/\varepsilon_r$. Si $u \not\equiv \varepsilon_G$ entonces se forma $\mathfrak{G}_2$ por reemplazamiento de u por ε_r en $\mathfrak{G}_1$, es decir, $\mathfrak{G}_2$= reemplaza($\mathfrak{G}_1$, u, ε_r) y se toma como $G' = \mathfrak{G}_2/\varepsilon_r$.

Si tenemos en cuenta que $\mathfrak{G}_1$= reemplaza($\mathfrak{G}_1$, u, u) podemos englobar ambos casos.

Si $\sigma = \{\}$ decimos que la expresión-gda G **reescribe en un paso** a la expresión-gda G':

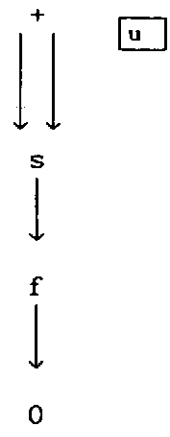
$$\therefore\ G \xrightarrow{l \to r, u} G' \ \blacksquare$$

Prescindiremos de escribir la regla (y/o la posición) a la que se refiere el paso de estrechamiento cuando esté claro por el contexto.

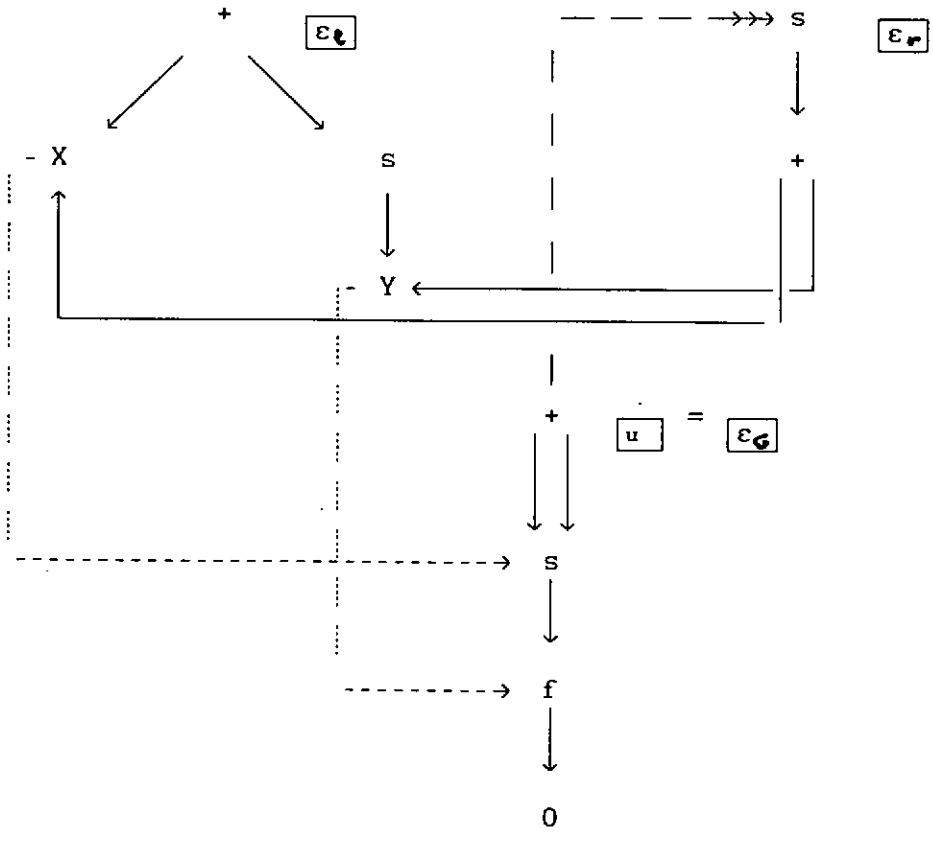
Los dos ejemplos siguientes permiten una mejor comprensión del mecanismo de cómputo en nuestro lenguaje.

EJEMPLO 1: Sea la regla $+(X,s(Y)) \longrightarrow s(+(X,Y))$, y sea la expresion-gda

G =



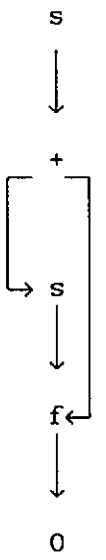
Construimos el grafo $\mathfrak{G}$:



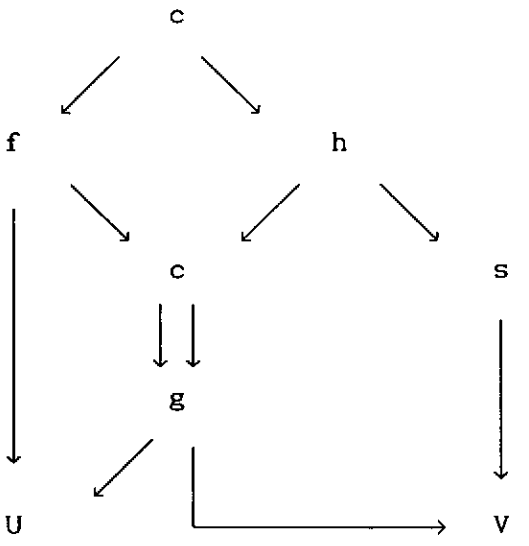
$\lambda = \{X \longrightarrow s(f(0)),\ Y \longrightarrow f(0)\}$ $\qquad \sigma = \{\}$

$$G \underset{\{\}}{\Longrightarrow} G'$$

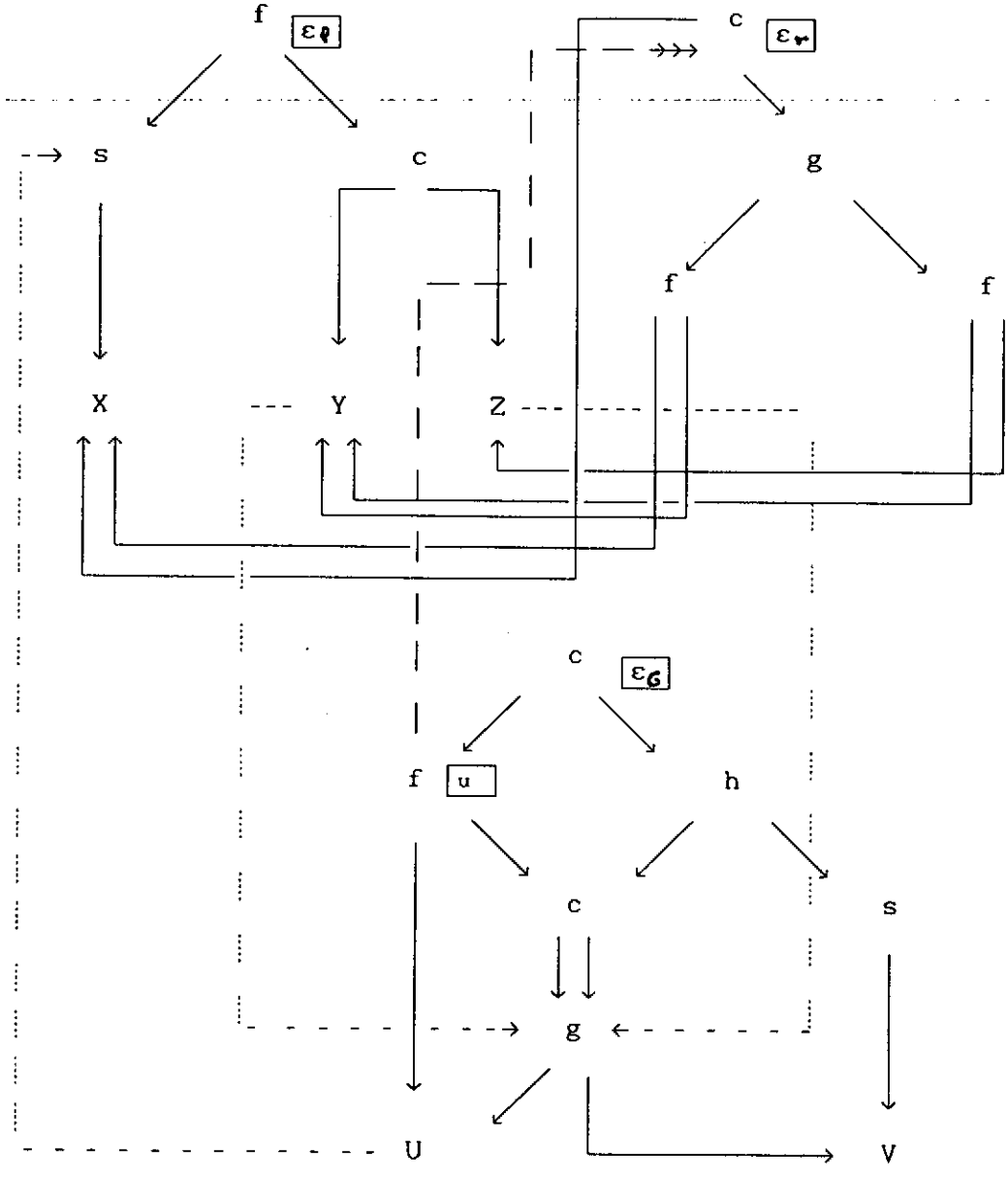
La expresión-gda G' resultante es:



EJEMPLO 2: Sea la regla f(s(X),c(Y,Z)) ⟶ c(X,g(f(X,Y),f(Y,Z))) y la expresión-gda G=



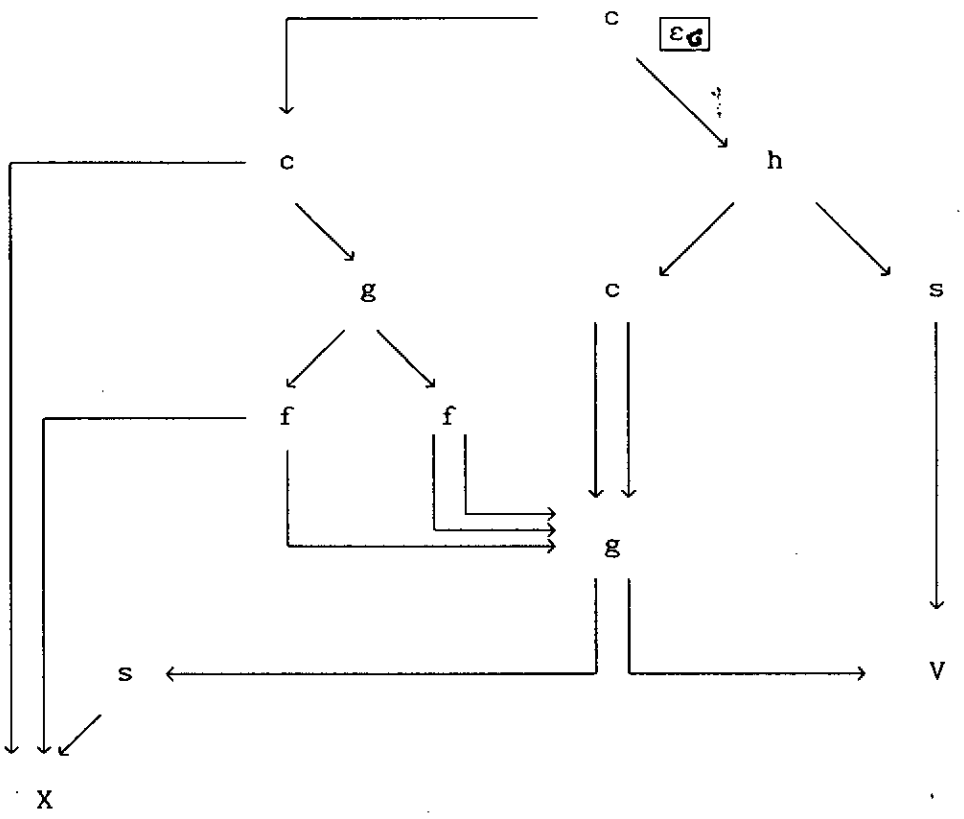
Construimos el grafo ⑤:



$\lambda = \{Y \longrightarrow g(s(X),V),\ Z \longrightarrow g(s(X),V)\}$ $\qquad$ $\sigma = \{U \longrightarrow s(X)\}$

$$G \underset{\sigma}{\Longrightarrow} G'$$

La expresión-gda G' resultante es:



NOTA: Si al aplicar la regla $l \longrightarrow r$ a la expresión-gda G en la posición u e intentar unificar el árbol correspondiente a l y la expresión-gda G/u el algoritmo de unificación da como resultado FALLO decimos que la regla $l \longrightarrow r$ **falla** o no es unificable con G/u; si, por otra parte, el algoritmo da como resultado UNIFICACION SUSPENDIDA con $\{w_1,\ldots,w_k\}$ como conjunto de nodos pendientes de unificación decimos que la regla $l \longrightarrow r$ está **suspendida** de aplicación pendiente de la evaluación de los nodos $\{w_1,\ldots,w_k\}$.

5.4.2.-DEFINICION:

Sea e una expresión. Sea G_e la expresión-gda canónica y Π un programa. Decimos que e se **reduce por estrechamiento** a la expresión-gda G' via

la sustitución σ_{out} cuando se puede establecer una sucesión de pasos de estrechamiento

$$G_e = G_0 \xrightarrow[\sigma_1]{} G_1 \xrightarrow[\sigma_2]{} G_2 \longrightarrow \ldots\ldots \xrightarrow[\sigma_n]{} G_n = G'$$

donde $\sigma_{out} =_{def} \sigma_1\sigma_2\ldots\sigma_n \restriction var(e)$.

Cuando $\sigma_{out} = \{\}$ sobre var(e) decimos que G_e se reduce a G' por **reescritura**. Observar que alguna σ_i puede ser distinta de {}. ■

Observar que σ_{out} puede conseguirse que sea una d-sustitución lineal apartada de cualquier conjunto finito de variables V ⊇ var(e) que se elija de antemano.

Podemos hacer un planteamiento más general de nuestro proceso de reducción por estrechamiento si consideramos el esqueleto ("shell") de una expresión-gda G.

5.4.3.-DEFINICION:

Sea G un gda etiquetado. Sea u ∈ O(G).

El **esqueleto** de una expresión-gda G/u es un elemento finito del dominio de Herbrand con variables $H = H_{\Sigma(VAR)}$ que se define por recursión:

$ar(u) = 0 \wedge G[u] = X \in VAR \Longrightarrow |G/u| = X$

$ar(u) = 0 \wedge G[u] = c \in CS^0_\Sigma \Longrightarrow |G/u| = c$

$ar(u) = n > 0 \wedge G[u] = c \in CS^n_\Sigma \wedge u \xrightarrow[i]{G} u_i \ (1 \le i \le n) \Longrightarrow$

$$|G/u| = c(|G/u_1|, \ldots, |G/u_n|)$$

$|G/u| = \perp_H$ en otro caso. ■

Entonces: Sea Π un programa. Sea e una expresión. Sea G_e la expresión-gda asociada a e. Sea

$$G_e = G_0 \xrightarrow[\sigma_1]{} G_1 \xrightarrow[\sigma_2]{} G_2 \longrightarrow \ldots\ldots \xrightarrow[\sigma_n]{} G_n = G'$$

un cómputo. Definimos:

$$\sigma =_{def} \sigma_1\sigma_2\ldots\sigma_n$$

$\sigma_{out} =_{def} \sigma \downarrow var(e)$

$s =_{def} | G' |$

Decimos que $(s : \sigma_{out})$ es la **solución calculada** para e por el cómputo.

Y que s es el **resultado** y σ_{out} la **respuesta.**

EJEMPLO: de cómputo y solución calculada

Sea el programa:

constructors:

zero / 0

succ / 1

nil / 0

cons / 2

operators:

from_pos / 1

from / 1

if / 2

less / 2

rules:

from_pos(X) ⟶ if(less(Y,X),from(X)).

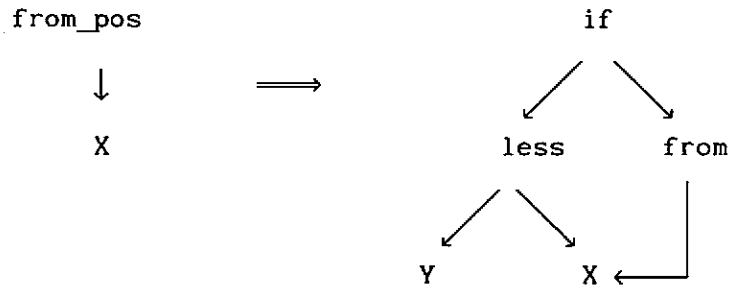
if(true,Y) ⟶ Y.

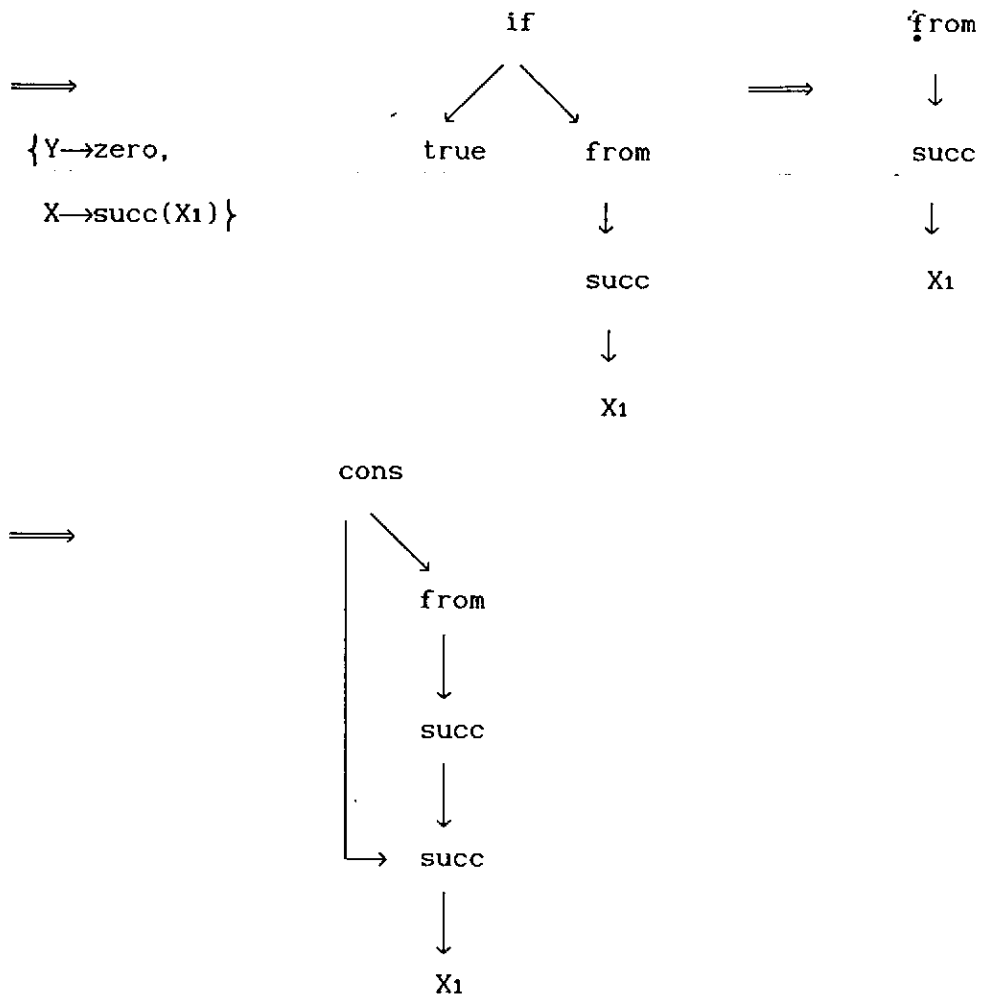
less(zero,succ(X)) ⟶ true.

less(succ(X),succ(Y)) ⟶ less(X,Y).

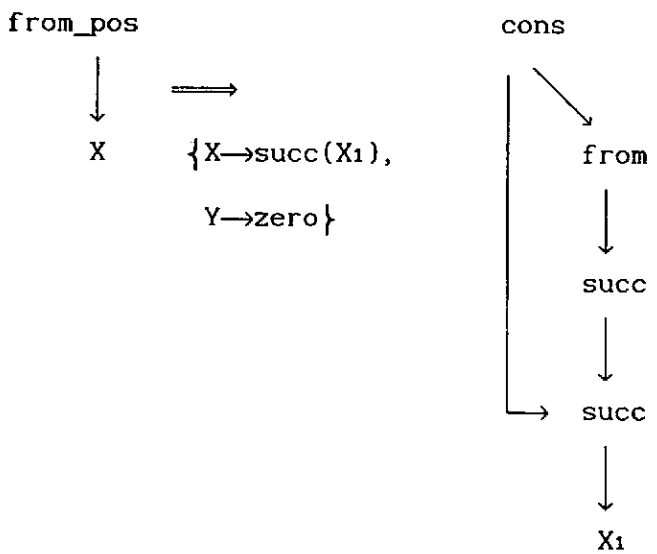
from(X) ⟶ cons(X,from(succ(X))).

Un cómputo para la expresión from_pos(X) sería el siguiente:





En resumen hemos obtenido el siguiente cómputo:



La solución calculada es:

$$\underbrace{cons(succ(X_1), \perp)}_{\text{(resultado)}} : \underbrace{\{X \longrightarrow succ(X_1)\}}_{\text{(respuesta)}}$$

6. RESULTADOS DE CORRECCION Y COMPLETITUD

En este capítulo se establecen los resultados más importantes de este trabajo: la corrección y completitud de nuestro lenguaje lógico aplicativo siguiendo las pautas marcadas por el trabajo de Clark para los programas lógicos [Clark 79]. (Ver también [Apt 90] y [Lloyd 87]) Podemos decir, de manera informal, que toda solución computada por estrechamiento es válida en cualquier modelo del programa (CORRECCION) y que para cualquier solución, válida en cualquier modelo del programa, puede calcularse al menos esa información usando estrechamiento (COMPLETITUD).

La introducción de expresiones-gda necesaria para definir una semántica operacional correcta plantea ciertas dificultades técnicas que complican las demostraciones. Por eso, comenzamos el capítulo con un apartado en el que definimos el importante concepto de evaluación de una expresión-gda así como el significado de aplicar una d-sustitución lineal a una expresión-gda. De paso, adquiere sentido la importancia que damos a este tipo de sustituciones. En el apartado segundo demostramos la corrección del paso de estrechamiento que hemos definido para expresiones-gda. Previamente establecemos un importante lema de sustitución. En el tercero vemos la corrección del estrechamiento general para expresiones-gda que admite el estrechamiento general para expresiones como caso particular. En el apartado cuarto establecemos el resultado final de nuestro trabajo: la

completitud.

6.1. NOTAS SOBRE EXPRESIONES-GDA

Necesitamos, además del concepto de evaluación de una expresión ya definido en el capítulo 4, el de evaluación de una expresión-gda para obtener resultados de corrección y completitud. La necesidad viene de nuestra semántica operacional basada en estrechamiento de grafos que hemos desarrollado en el capítulo anterior. La idea es que para evaluar una expresión-gda se precisa evaluar cada nodo del grafo etiquetado para que las partes compartidas de la expresión-gda tengan el mismo valor. La idea subyacente es que este concepto debe generalizar el de evaluación de una expresión.

6.1.1.-DEFINICION: Evaluación de una Expresión-Gda

Sea G una expresión-gda, I una Σ-interpretación, y $\rho : VAR \longrightarrow D_I$ una Σ-valoración. Una aplicación

$$\hat{\rho} : O(G) \longrightarrow Fin_{D_I}$$

es una **Σ-evaluación de G en I bajo ρ si** para $u \in dom(\hat{\rho})$:

- $ar(u) = 0 \wedge G[u] = X \in VAR \Longrightarrow \hat{\rho}(u) \ll \rho(X)$.
- $ar(u) = 0 \wedge G[u] = k \in CS^0_\Sigma \cup FS^0_\Sigma \Longrightarrow k_I \ni \hat{\rho}(u)$.
- $ar(u) = n > 0 \wedge u \xrightarrow[i]{G} u_i \ (1 \le i \le n) \wedge G[u] = k \in CS^n_\Sigma \cup FS^n_\Sigma$
  $\Longrightarrow k_I(\hat{\rho}(u_1), \ldots, \hat{\rho}(u_n)) \ni \hat{\rho}(u)$.

Definimos, entonces:

$\|G\|_I(\rho) = \{x \in Fin_{D_I} \mid$ existe $\hat{\rho}$ evaluación de G en I bajo ρ tal que $x \ll \hat{\rho}(\varepsilon)$ donde ε es el nodo raiz de $G\}$. ■

De la definición se deduce inmediatamente:

(1) $\|G\|_I(\rho) \in P(D_I)$: $\|G\|_I(\rho)$ es un elemento del dominio potencia $P(D_I)$ o un grafo del conjunto $[\{\iota\}\xrightarrow[n]{}D_I]$ en virtud del isomorfismo establecido en 3.3.4.

(2) Si $\hat{\rho}$ es una evaluación de G en I bajo ρ y $u \in O(G)$, $\hat{\rho}$ limitada a las O(G/u) es una evaluación del subgda G/u bajo ρ en I. Podemos entonces afirmar que:

$$\|G/u\|_I(\rho) \ni \hat{\rho}(u)$$

y, en particular, para G[u] = X ∈ VAR y $x \ll \rho(X)$ finito:

$$\|X\|_I(\rho) \ni x$$

que coincide con la intuición requerida.

(3) Se puede definir de la misma manera la evaluación de un gda cualquiera $\mathfrak{G}$ en I bajo la valoración ρ.

Recordemos que una expresión e puede representarse como una expresión-gda canónica G_e en la que únicamente se comparten las variables. Es fundamental para nuestro estudio que una expresión e y la expresión-gda G_e correspondiente evalúen a los mismos valores utilizando ambas definiciones.

6.1.2.-PROPOSICION:

Sea e una expresión y sea G_e la expresión-gda canónica correspondiente a e. Entonces para toda Σ-interpretación I y para toda Σ-valoración ρ : VAR $\longrightarrow D_I$ se cumple que:

$$\|e\|_I(\rho) \ni x \iff \|G_e\|_I(\rho) \ni x \text{ para todo } x \in Fin_{D_I}.$$

dem:

Sea ρ una valoración. Usamos la recursividad de la definición de e:

▷ Para los casos en que e ::= X ∈ VAR y e ::= k ∈ $CS^0_\Sigma \cup FS^0_\Sigma$, la demostración es trivial.

▷ Si e ::= $k(e_1,\ldots,e_n)$, k ∈ $CS^n_\Sigma \cup FS^n_\Sigma$, n>0 entonces G_e es un gda con

nodo raiz ε, $ar(\varepsilon) = n$, $G_e[\varepsilon] = k$, y existen nodos u_i tal que $\varepsilon \xrightarrow[i]{G} u_i$ $1 \le i \le n$ de modo que G_e/u_i es la expresión-gda correspondiente a e_i. Observar que G_e/u_i y G_e/u_j con $1 \le i \ne j \le n$ tienen todos sus nodos diferentes salvo los correspondientes a las variables comunes.

Si $\|G_e\|_I(\rho) \ni x$ entonces por 6.1.1 existe $\hat{\rho}$ evaluación de G_e en I bajo ρ tal que $x \ll \hat{\rho}(\varepsilon)$ y además $k_I(\hat{\rho}(u_1),\ldots,\hat{\rho}(u_n)) \ni \hat{\rho}(\varepsilon)$, de donde $k_I(\hat{\rho}(u_1),\ldots,\hat{\rho}(u_n)) \ni x$ por ser un conjunto cerrado por abajo de elementos finitos. Ahora bien, $\hat{\rho}$ limitada a $O(G_e/u_i)$ es una evaluación de G_e/u_i en I bajo ρ tal que:

$$\|G_e/u_i\|_I(\rho) \ni \hat{\rho}(u_i) \text{ para todo } i,\ 1 \le i \le n.$$

Por hipótesis de inducción:

$$\|G_e/u_i\|_I(\rho) \ni \hat{\rho}(u_i) \iff \|e_i\|_I(\rho) \ni \hat{\rho}(u_i) \text{ para todo } 1 \le i \le n.$$

Tenemos entonces que existe $\hat{\rho}(u_i) \in Fin_{D_I}$, $1 \le i \le n$, $\|e_i\|_I(\rho) \ni \hat{\rho}(u_i)$ y tal que $k_I(\hat{\rho}(u_1),\ldots,\hat{\rho}(u_n)) \ni x$.

Por la definición 4.1.5 obtenemos que:

$$\|k(e_1,\ldots,e_n)\|_I(\rho) \ni x \iff \|e\|_I(\rho) \ni x.$$

Reciprocamente, $\|k(e_1,\ldots,e_n)\|_I(\rho) \ni x \iff$ existe $x_i \in Fin_{D_I}$, $1 \le i \le n$, $\|e_i\|_I(\rho) \ni x_i$ y tal que $k_I(x_1,\ldots,x_n) \ni x$ (por 4.1.5). Por hipótesis de inducción $\|e_i\|_I(\rho) \ni x_i \iff \|G_e/u_i\|_I(\rho) \ni x_i$ para todo $1 \le i \le n$. Entonces para todo i, $1 \le i \le n$, $\|G_e/u_i\|_I(\rho) \ni x_i$ implica que existe una $\hat{\rho}_i$ evaluación de G_e/u_i en I bajo ρ con $x_i \ll \hat{\rho}_i(u_i)$.

Observar ahora que si w es un nodo correspondiente a una variable X compartida por varios G_e/u_i puede ocurrir que los $\hat{\rho}_i(w)$ sean distintos. Sin embargo, como todos ellos son un conjunto finito de aproximaciones finitas de $\rho(X)$ y el conjunto de las aproximaciones finitas de $\rho(X)$ es dirigido podemos encontrar una aproximación finita (lema 3.1.4) que sea mayor o igual que todos los $\hat{\rho}_i(w)$.

Definimos $\hat{\rho} : O(G_e) \longrightarrow Fin_{D_I}$ como sigue:

▷ $\hat{\rho}(u) = \hat{\rho}_i(u)$ si $u \in O(G_e/u_i)$ $1 \leq i \leq n \wedge u$ no es compartido

▷ $\hat{\rho}(u) = \bigsqcup \{\hat{\rho}_j(u) \in Fin_{D_I} \mid (G_e/u_j)[u] = X \in VAR \wedge j \in \{1,\ldots,n\}\}$

si u es un nodo de variable compartido

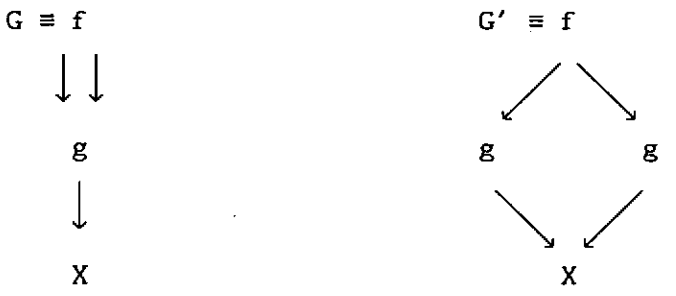
▷ $\hat{\rho}(\varepsilon) = x$ si ε es el nodo raiz de G_e.

La definición de $\hat{\rho}$ es correcta por la suposición que únicamente se comparten variables en G_e. Además se comprueba fácilmente que $\hat{\rho}$ es una evaluación de G_e en I bajo ρ con resultado $x \ll \hat{\rho}(u)$. Luego

$$\|G_e\|_I(\rho) \ni x. \blacksquare$$

Por el contexto distinguiremos perfectamente cuando evaluamos una expresión y cuando evaluamos una expresión-gda; o sea, un grafo. El siguiente ejemplo muestra, sin embargo, la necesidad de la distinción:

Dadas:



es sencillo encontrar una interpretación en la que G admite menos valores que G'.

Para demostrar que la semántica operacional que hemos definido es correcta necesitamos también dar un significado en nuestra representación de expresiones al hecho de aplicar una d-sustitución lineal a una expresión-gda.

6.1.3.-DEFINICION:

Sea G una expresión-gda. $V \supseteq var(G)$ un conjunto de variables.

Sea $v_i \in O(G)$ tal que $G[v_i] = X_i \in V$ $(1 \leq i \leq k)$. (las X_i todas distintas)

Sea $\sigma = \{X_1 \longrightarrow t_1, \ldots, X_k \longrightarrow t_k\}$ una d-sustitución lineal apartada de V.

Sea T_i el árbol correspondiente al término lineal t_i y supongamos que la raiz del árbol T_i es ε_i $(1 \leq i \leq k)$.

Evidentemente, podemos suponer que $O(T_i) \cap O(T_j) = \emptyset$ $(1 \leq i \neq j \leq k)$ y que $O(T_i) \cap O(G) = \emptyset$ $(1 \leq i \leq k)$.

Sea $\mathfrak{G}$ el gda etiquetado formado por G, $T_1, \ldots, T_k$, y sea

$$\mathfrak{G}' = \mathfrak{G}\,[v_1 \longrightarrow \varepsilon_1, \ldots, v_k \longrightarrow \varepsilon_k].$$

Entonces definimos:

$$G\sigma = \mathfrak{G}'/\varepsilon. \blacksquare$$

El siguiente lema tiene por objeto dar sentido a la notación "$G\sigma_1\sigma_2\ldots\sigma_n$". Es inmediato sin más que aplicar las definiciones:

6.1.4.-LEMA:

Sea G una expresión-gda.

Sea σ una d-sustitución lineal apartada de var(G) con $dom(\sigma) \subseteq var(G)$.

Sea ϑ una d-sustitución lineal apartada de $(var(G) \setminus dom(\sigma)) \cup ran(\sigma)$ con $dom(\vartheta) \subseteq (var(G) \setminus dom(\sigma)) \cup ran(\sigma)$.

Entonces: $(G\sigma)\vartheta = G(\sigma\vartheta)$. $\blacksquare$

6.2. CORRECCION DEL PASO DE ESTRECHAMIENTO

Para demostrar que la semántica operacional es lógicamente correcta se estudia, en este apartado, la corrección de un paso de estrechamiento. Posteriormente (en el siguiente) estudiaremos la del

estrechamiento general en varios pasos.

Para demostrar la corrección de un paso de estrechamiento y más adelante para demostrar la completitud es necesario el siguiente importante lema de sustitución.

6.2.1.-LEMA: (Lema de Sustitución de Gda's)

Sea G una expresión-gda con ε_G como nodo raíz.

Sean para $1 \leq i \leq k$, $G[u_i] = X_i \in VAR$ donde las X_i son todas distintas.

Sean las G_i, para $1 \leq i \leq k$, expresiones-gda con nodo raíz v_i.

Las expresiones-gda G_i pueden compartir nodos entre sí pero suponemos que no comparten nodos con G: $O(G_i) \cap O(G) = \varnothing$ para $1 \leq i \leq k$.

Sea $\mathfrak{G}$ el gda formado considerando conjuntamente todas las G_i $(1 \leq i \leq k)$ y la citada expresión-gda G.

Sea $\mathfrak{H}$ el grafo formado considerando solamente las G_i $(1 \leq i \leq k)$.

Sea $\mathfrak{G}' = \mathfrak{G}\,[u_1 \longrightarrow v_1, \ldots, u_k \longrightarrow v_k]$.

Entonces, para toda Σ-interpretación I, para toda Σ-valoración ρ sobre I y para todo $x \in Fin_{D_I}$ las dos afirmaciones siguientes son equivalentes:

(1) $\|\mathfrak{G}'/\varepsilon_G\|_I(\rho) \ni x$

(2) $\exists\, \hat{\rho}$ evaluación de $\mathfrak{H}$ en I bajo ρ tal que

$\|\mathfrak{G}/\varepsilon_G\|_I(\rho') \ni x$

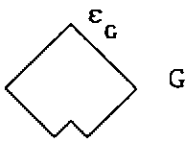
donde ρ' es la valoración definida como sigue:

$$\rho'(X_i) = \hat{\rho}(v_i) \text{ para } 1 \leq i \leq k$$

$$\rho'(X) = \rho(X) \text{ para } X \notin \{X_1, \ldots, X_k\}.$$

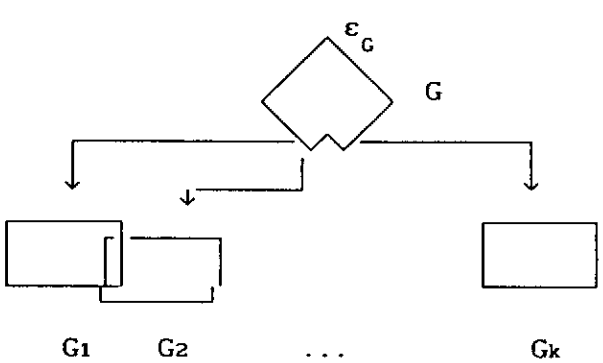
El siguiente esquema puede ayudar a entender el enunciado:

$\mathfrak{G}$



$\mathfrak{H}$

$\mathfrak{G}'$



dem:

"(1) $\Longrightarrow$ (2)"

$\|\mathfrak{G}'/\varepsilon_G\|_I(\rho) \ni x \Longleftrightarrow$ existe $\hat{\varphi}$ evaluación de $\mathfrak{G}'/\varepsilon_G$ en I bajo ρ con resultado $x \ll \hat{\varphi}(\varepsilon_G)$.

Restringiendo $\hat{\varphi}$ a las $O(\mathfrak{G}'/v_i)$ para $1 \leq i \leq k$ obtenemos evaluaciones de $\mathfrak{G}'/v_i$ en I bajo ρ con resultado $\hat{\varphi}(v_i)$.

Como $O(\mathfrak{H}) \subset O(\mathfrak{G}')$ podemos definir $\hat{\rho}$ como la restricción de $\hat{\varphi}$ a $O(\mathfrak{H})$:

$$\hat{\rho} =_{def} \hat{\varphi}|_{O(\mathfrak{H})}.$$

Entonces, es inmediato que $\hat{\rho}$ es una evaluación de $\mathfrak{H}$ en I bajo ρ.

Sea ρ' la valoración definida en el enunciado:

$$\rho'(X_i) = \hat{\rho}(v_i) \text{ para } 1 \leq i \leq k$$

$$\rho'(X) = \rho(X) \text{ para } X \notin \{X_1, \ldots, X_k\}.$$

Definimos una aplicación $\hat{\phi} : O(\mathfrak{G}/\varepsilon_G) \longrightarrow Fin_{D_I}$ como sigue:

$$\triangleright\ \hat{\phi}(u_i) = \rho'(X_i) = \hat{\rho}(v_i) \text{ para } 1 \leq i \leq k$$

$$\triangleright\ \hat{\phi}(u) = \hat{\varphi}(u) \text{ en otro caso.}$$

Es rutinario probar que $\hat{\phi}$ es una evaluación de $\mathfrak{G}/\varepsilon_G$ en I bajo ρ' con resultado $x \ll \hat{\phi}(\varepsilon_G) = \hat{\varphi}(\varepsilon_G)$.

"(2) $\Longrightarrow$ (1)"

Supongamos que existe $\hat{\rho}$ evaluación de $\mathfrak{H}$ bajo ρ tal que $\|\mathfrak{G}/\varepsilon_G\|_I(\rho') \ni x$ siendo ρ' la valoración del enunciado.

Entonces podemos encontrar para cada i, $1 \leq i \leq k$, una $\hat{\rho}_i$ evaluación de $\mathfrak{H}/v_i$ en I bajo ρ. Como $\mathfrak{H}/v_i = \mathfrak{G}'/v_i$ podemos considerar cada $\hat{\rho}_i$ como una evaluación de $\mathfrak{G}'/v_i$ en I bajo ρ.

Además las $\hat{\rho}_i$ son tales que si $v \in O(\mathfrak{G}'/v_i) \cap O(\mathfrak{G}'/v_j)$ se cumple que $\hat{\rho}_i(v) = \hat{\rho}_j(v)$ para todo $1 \leq i \neq j \leq k$.

Por la otra parte de la hipótesis tenemos que existe una $\hat{\rho}'$ evaluación de $\mathfrak{G}/\varepsilon_G$ en I bajo ρ' con resultado x.

Definimos una aplicación $\hat{\varphi} : O(\mathfrak{G}'/\varepsilon_G) \longrightarrow Fin_{D_I}$ como sigue:

$$\triangleright\ \hat{\varphi}(v) = \hat{\rho}'(v) \text{ si } v \in O(\mathfrak{G}/\varepsilon_G)$$

$$\triangleright\ \hat{\varphi}(v) = \hat{\rho}_i(v) \text{ si } v \in O(\mathfrak{G}'/v_i) = O(\mathfrak{H}/v_i) \text{ para todo } 1 \leq i \leq k.$$

Es inmediato que $\hat{\varphi}$ es una evaluación de $\mathfrak{G}'/\varepsilon_G$ en I bajo ρ con resultado x. ■

A continuación establecemos la corrección del paso de estrechamiento:

6.2.2.-TEOREMA (Corrección del Paso de Estrechamiento)

Supongamos que la Σ-expresion-gda G estrecha en la posición reducible u en un paso a G' aplicando la regla $l \longrightarrow r$:

$$G \xrightarrow[\sigma]{l \longrightarrow r, u} G'.$$

Entonces, para toda Σ-interpretación I tal que $I \models l \longrightarrow r$ y para toda Σ-valoración $\rho: VAR \longrightarrow D_I$ se verifica:

$$\|G'\|_I(\rho) \leq \|G\sigma\|_I(\rho).$$

dem:

Sea $\mathfrak{G}_0$ el gda formado por l, r y G donde las variables comunes a l y r están compartidas. Podemos suponer todos los demás nodos de $\mathfrak{G}_0$ distintos.

Sean ε_l, ε_r, y ε_G los nodos raices respectivos de los gdas correspondientes a l, r y G.

Sea $\mathfrak{G}_1$ el gda que se obtiene al aplicar el algoritmo de unificación en $\mathfrak{G}_0$ con nodos "inputs" u y ε_l.

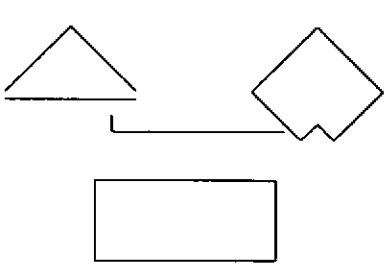
Sea σ la d-sustitución lineal obtenida en el algoritmo de unificación y sea λ la sustitución tal que:

$$expr(\mathfrak{G}_0, \varepsilon_l)\ \lambda = l\ \lambda = expr(\mathfrak{G}_0, u)\ \sigma$$

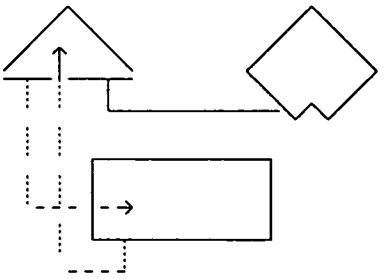
Para fijar la notación supongamos que λ liga las variables $X_1, \ldots, X_p$ de $\mathfrak{G}_0/\varepsilon_l$ (algunas compartidas con el gda $\mathfrak{G}_0/\varepsilon_r$) con los gdas $\mathfrak{G}_1/u_1, \ldots, \mathfrak{G}_1/u_p$ respectivamente.

Sea $\mathfrak{G}_2 = reemplaza(\mathfrak{G}_1, u, \varepsilon_r)$ donde u puede ser ε_G.

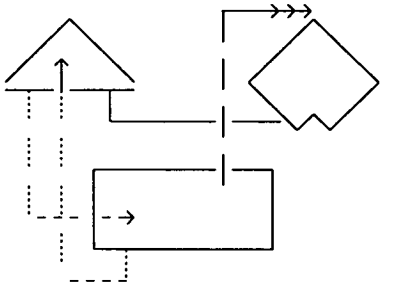
$\mathfrak{G}_0$



$\mathfrak{G}_1$



$\mathfrak{G}_2$



Sea $x \in Fin_{D_I}$. Hemos de probar que $x \in \|G'\|_I(\rho) \Longrightarrow x \in \|G\sigma\|_I(\rho)$.

Por definición $x \in \|G'\|_I(\rho) \Longleftrightarrow x \in \|\mathfrak{G}_2/\varepsilon_G\|_I(\rho)$. De aquí, existe una evaluación $\hat{\rho}$ del gda $\mathfrak{G}_2$ en I bajo ρ con resultado $x \ll \hat{\rho}(\varepsilon_G)$. Entonces, limitando $\hat{\rho}$ a las posiciones de $\mathfrak{G}_2/\varepsilon_r$ tenemos que existe un $y \in Fin_{D_I}$ tal que $y = \hat{\rho}(\varepsilon_r) \in \|\mathfrak{G}_2/\varepsilon_r\|_I(\rho)$.

Por el lema de sustitución de gda's tenemos que $y \in \|\mathfrak{G}_0/\varepsilon_r\|_I(\mu)$ siendo la valoración μ : VAR $\longrightarrow D_I$ tal que:

$$\mu(X_i) = \hat{\rho}(u_i) \text{ para } 1 \leq i \leq p$$

$$\mu(X) = \rho(X) \text{ en otro caso}$$

donde las $X_i \in dom(\lambda)$ para $1 \leq i \leq p$.

Por el lema 6.1.2 tenemos que $y \in \|r\|_I(\mu)$.

Como $I \models l \longrightarrow r$ (ver la definición 4.2.1) tenemos que $y \in \|l\|_I(\mu)$.

Aplicando nuevamente 6.1.2 obtenemos que $y \in \|\mathfrak{G}_0/\varepsilon_l\|_I(\mu)$.

Ahora, aplicamos otra vez más el lema de sustitución de gda's y el

hecho que $\mathfrak{G}_2/\varepsilon_1 = \mathfrak{G}_1/\varepsilon_1$ para obtener que $y \in \|\mathfrak{G}_1/\varepsilon_1\|_I(\rho)$.

Como los grafos $\mathfrak{G}_1/\varepsilon_1$ y $\mathfrak{G}_1/u$ son precisamente los que se han unificado por el Teorema de Unificación (5.3.2) se verifica que $y \in \|\mathfrak{G}_1/u\|_I(\rho)$.

La misma evaluación inicial $\hat{\rho}$ nos permite ahora obtener que $x \in \|\mathfrak{G}_1/\varepsilon_G\|_I(\rho)$. Pero esto es precisamente lo que buscabamos: que x es un elemento de $\|G\sigma\|_I(\rho)$. ■

6.3 CORRECCION

Aunque los cómputos que realizamos comienzan con expresiones es necesario considerar en las demostraciones por inducción que siguen cómputos intermedios que comienzan con expresiones-gda. Por este motivo, comenzamos este apartado precisando los dos conceptos de solución que hemos visto para expresiones-gda. Obviamente generalizan los conceptos para expresiones normales.

6.3.1.-DEFINICION: Solución Calculada para una Expresión-Gda

Sea Π un programa.

Sea G una expresión-gda.

Si

$$G = G_0 \xrightarrow[\sigma_1]{} G_1 \xrightarrow[\sigma_2]{} \cdots\cdots \xrightarrow[\sigma_n]{} G_n = G'$$

es un cómputo. Y si

$$\sigma =_{def} \sigma_1\sigma_2\ldots\sigma_n$$

$$\sigma_{out} =_{def} \sigma \restriction var(G)$$

$$s =_{def} |G'|.$$

Entonces $(s:\sigma_{out})$ es una **solución calculada** para G y Π. s es el **resultado** y σ_{out} la **respuesta**. ■

6.3.2.-DEFINICION: Término Parcial, Solución, Solución Correcta

Sea Π un programa. Sea G una expresión-gda.

Un **término parcial** es un término que permite el uso de $\perp$ como una constante (aridad 0).

La definición de sustitución parcial es obvia: sustitución de términos parciales en lugar de variables.

Una **solución** para G y Π es un par $(s:\vartheta)$ tal que ϑ es una d-sustitución lineal apartada de var(G) y s es un término parcial finito.

Las variables críticas de la solución son:

$$VS =_{def} var(G\vartheta) \cup var(s).$$

$(s:\vartheta)$ es una **solución correcta para G en el modelo mínimo de** Π sobre VS:

$$\Longleftrightarrow_{def} I_\pi(VS) \models G\vartheta \ni s.$$

$$\Longleftrightarrow \|G\vartheta\|_{I_\pi(VS)} \ni s$$

$$\Longleftrightarrow \|G\vartheta\|_{I_\pi(VS)} \supseteq \|s\|_{I_\pi(VS)}$$

$(s:\vartheta)$ es una **solución** (lógicamente) **correcta con respecto a** Π

$$\Longleftrightarrow_{def} \Pi \models G\vartheta \supseteq s$$

$\Longleftrightarrow$ para todo I $\models \Pi$ y para toda $\rho : VAR \longrightarrow D_I$

$$\|G\vartheta\|_I(\rho) \supseteq \|s\|_I(\rho). \blacksquare$$

De manera análoga se define para una expresión e y un programa Π solución, solución correcta en el modelo mínimo y solución correcta.

Para probar la corrección del estrechamiento general en varios pasos necesitamos dos lemas más:

6.3.3.-LEMA:

Sea G una expresión-gda. Entonces para toda interpretación I de un programa Π con signatura Σ y para toda valoración ρ se verifica:

$$\|G\|_I(\rho) \supseteq \| \ |G| \ \|_I(\rho)$$

dem:

Sobre la definición recursiva del esqueleto de G:

Sea u el nodo raiz de G

▷ $ar(u) = 0 \wedge G[u] = X \in VAR$:

$$\|G\|_I(\rho) = \|X\|_I(\rho) = \| \, |G| \, \|_I(\rho).$$

▷ $ar(u) = 0 \wedge G[u] = c \in CS^0_\Sigma$:

$$\|G\|_I(\rho) = \|c\|_I = \| \, |G| \, \|_I.$$

▷ $ar(u) = n \wedge G[u] = c \in CS^n_\Sigma$; $u \xrightarrow[i]{} u_i$ $1 \leq i \leq n$:

$$\|G\|_I(\rho) = c(\|G/u_1\|_I(\rho), \ldots, \|G/u_n\|_I(\rho))$$

donde $c(\|G/u_1\|_I(\rho), \ldots, \|G/u_n\|_I(\rho))$ representa el conjunto, elemento del dominio potencia $P(D_I)$, formado por elementos de la forma $c(x_1,\ldots,x_n)$ con $x_i \in \|G/u_i\|_I(\rho)$, para $1 \leq i \leq n$ donde cada $\|G/u_i\|_I(\rho)$ es a su vez un elemento del dominio potencia $P(D_I)$.

Como por hipótesis de inducción $\|G/u_i\|_I(\rho) \geq \| \, |G/u_i| \, \|_I(\rho)$ para $1 \leq i \leq n$ se deduce de inmediato que:

$$c(\| \, |G/u_1| \, \|_I(\rho), \ldots, \| \, |G/u_n| \, \|_I(\rho)) \leq c(\|G/u_1\|_I(\rho), \ldots, \|G/u_n\|_I(\rho)) = \|G\|_I(\rho).$$

▷ $ar(u) = n \wedge G[u] = f \in FS^n_\Sigma$:

$$\|G\|_I(\rho) \geq \perp = \| \, |G| \, \|_I(\rho). \quad \blacksquare$$

El siguiente lema es consecuencia del lema de sustitución de gda's y del concepto de evaluación determinista de un término lineal (ver 4.1.7).

6.3.4.-LEMA: (Lema de Sustitución de términos)

Sea G una expresión-gda. Sea $\sigma = \{ X_1 \longrightarrow t_1, \ldots, X_k \longrightarrow t_k \}$ una d-sustitución lineal apartada de V ≥ var(G). Entonces para toda $\Sigma(V)$-interpretación I y para toda $\Sigma(V)$-valoración ρ se verifica:

$$\|G\sigma\|_I(\rho) = \|G\|_I(\rho').$$

donde ρ' es la $\Sigma(V)$-valoración:

$$\rho'(X) = \|\sigma(X)\|_I^{det}(\rho)$$

dem:

Es suficiente probar que $\|G\sigma\|_I(\rho) \ni x \iff \|G\|_I(\rho') \ni x$ para todo x elemento finito de D_I.

Para probarlo basta tomar en el lema de sustitución de gda's como G_i las expresiones-gda correspondientes a los términos lineales t_i ($1 \le i \le k$) que cumplen las condiciones del lema. En este caso los G_i no comparten nodos entre sí. Si $\mathfrak{G}$, $\mathfrak{G}'$ y $\mathfrak{H}$ se definen como en 6.2.1 entonces $G\sigma \equiv \mathfrak{G}'/\varepsilon_G$ por 6.1.3 y $G \equiv \mathfrak{G}/\varepsilon_G$ siendo ε_G el nodo raiz de G. Además, ρ' es la valoración $\rho'(X_i) = \|\sigma(X_i)\|_I^{det}(\rho)$ ya que por la linelidad de los t_i se puede tomar como evaluación $\hat{\rho}$ de $\mathfrak{H}$ una evaluación determinista de los t_i. Y si $\sigma(X) = X$ entonces $\rho'(X) = \|X\|_I^{det}(\rho) = \rho(X)$. ■

El siguiente teorema prueba la corrección de la semántica operacional que hemos definido para grafos dirigidos acíclicos.

6.3.5.-TEOREMA: Corrección

Cualquier solución calculada para una expresión-gda G y un programa Π es correcta en todos los modelos del programa.

dem:

Sea $(s:\sigma_{out})$ una solución calculada para G y Π.

Sea el cómputo:

$$G = G_0 \xrightarrow[\sigma_1]{} G_1 \xrightarrow[\sigma_2]{} G_2 \longrightarrow \ldots\ldots \xrightarrow[\sigma_n]{} G_n = G'$$

donde $\sigma = \sigma_1\sigma_2\ldots\sigma_n$; $\sigma_{out} = \sigma \restriction var(G)$ es una d-sustitución lineal apartada de var(G) con $dom(\sigma_{out}) \subseteq var(G)$; y $s = |G'|$ es un término parcial finito. Sea $VS = var(G\sigma_{out}) \cup var(s)$.

Sean I un modelo cualquiera de Π y $\rho : VAR \longrightarrow D_I$ una valoración

cualquiera. En primer lugar, vemos por el lema 6.3.3 que

$$\|G'\|_I(\rho) \geq \| |G'| \|_I(\rho).$$

Entonces, el teorema estará demostrado si:

$$\|G\sigma_{out}\|_I(\rho) \geq \|G'\|_I(\rho)$$

Por inducción en el número de pasos de la secuencia de reducción por estrechamiento.

Para n = 0:

Tenemos el cómputo en 0 pasos de estrechamiento siguiente:

$$G \xRightarrow[\{\}]{} G.$$

y lo que hay que demostrar es trivial.

Para n > 0:

Podemos escribir abreviadamente:

$$G \xRightarrow[\sigma_1]{} G_1 \xRightarrow[\sigma_{resto}]{} G'$$

y podemos razonar:

$\|G\sigma_{out}\|_I(\rho)$

$= \|G(\sigma_1\sigma_{resto})\|_I(\rho)$

$= \|(G\sigma_1)\sigma_{resto}\|_I(\rho)$ (por el lema 6.1.4)

$= \|G\sigma_1\|_I(\rho')$ (por el lema de sustitución de términos)

$\geq \|G_1\|_I(\rho')$ (por la corrección del paso de estrechamiento)

$= \|G_1\sigma_{resto}\|_I(\rho)$ (por el lema de sustitución de términos)

$\geq \|G'\|_I(\rho)$ (por la hipótesis de inducción). ■

Observar que si partimos de una expresión e consideraríamos la expresión-gda canónica G_e y el teorema de corrección se aplica de la misma manera teniendo en cuenta que $\|e\sigma_{out}\|_I(\rho) = \|G_e\sigma_{out}\|_I(\rho)$ por la proposición 6.1.2.

El objetivo de este apartado es demostrar el resultado inverso de la corrección: la completitud. La intuición que subyace al teorema de completitud es que dada una solución correcta $(s:\vartheta)$ (en el modelo mínimo sobre VS) para una expresión-gda G, puede construirse un cómputo que calcula una respuesta más general σ. "Más general" significa que $(s:\vartheta)$ puede obtenerse a partir de σ particularizando las variables de σ mediante una sustitución ϑ'. Los valores de ϑ' deben ser términos con variables de VS, posiblemente parciales, y necesariamente lineales. También podemos hacer, por razones técnicas, la suposición razonable de que si una variable de G no es del dominio de la respuesta correcta ϑ entonces no es ni del dominio de la respuesta calculada σ ni del de la sustitución ϑ'.

Veamos un ejemplo:

constructors:	operators:
0 / 0 ; s / 1	f / 2
[] / 0 ; [. ¦ .] / 2	g / 2

rules:

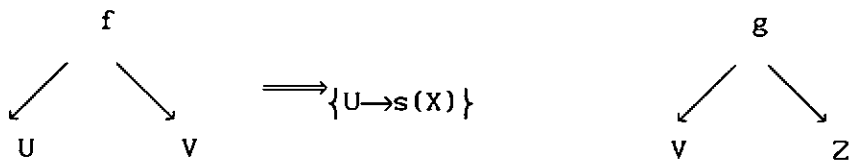
f(s(X),Y) ⟶ g(Y,Z).

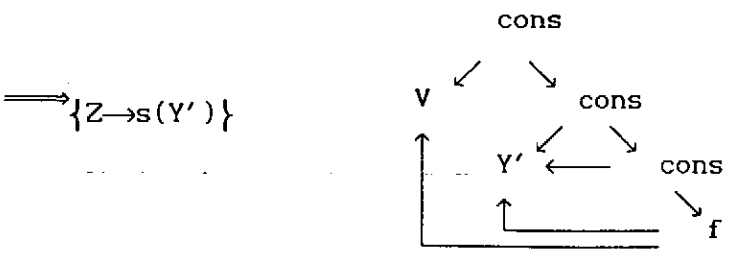
g(X,s(Y)) ⟶ [X,Y,Y ¦ f(X,Y)].

Una solución correcta para la expresión f(U,V) y este programa es:

$(s:\vartheta)$ = ([s(W),s(⊥),s(⊥) ¦ ⊥] : {U ⟶ s(0), V ⟶ s(W)})

Podemos hallar un cómputo:





La solución calculada es:

$$(|\ G'\ |\ :\ \sigma_{out}) = (\ [V,Y',Y'\ :\ \bot]\ :\ \{U \rightarrow s(X)\}\ )$$

Entonces la solución correcta $(s:\vartheta)$ es cubierta por la solución calculada $(|\ G'\ |\ :\ \sigma_{out})$ en el sentido siguiente:

Existe ϑ', d-sustitución parcial apartada de var(G'):

$$\vartheta' = \{X \rightarrow 0,\ V \rightarrow s(W),\ Y' \rightarrow s(\bot)\}$$

tal que:

1) $X \in var(G)\backslash dom(\vartheta) \Longrightarrow X \notin dom(\sigma_{out}) \cup dom(\vartheta')$

2) $\sigma_{out}\ \vartheta' \restriction dom(\vartheta) = \vartheta$

3) $|G'|\vartheta' \gg s$.

Observar que si las reglas del programa fuesen:

$$f(X,Y) \longrightarrow g(Y,Z).$$

$$g(X,s(Y)) \longrightarrow [X,Y,Y\ :\ f(X,Y)].$$

Entonces, una solución correcta para la expresión f(U,V) y este programa sería:

$$(s:\vartheta) = (\ [s(W),s(\bot),s(\bot)\ :\ \bot]\ :\ \{V \longrightarrow s(W)\}\ )$$

La solución calculada por un cómputo similar sería:

$$(|\ G'\ |\ :\ \sigma_{out}) = (\ [V,Y',Y'\ :\ \bot]\ :\ \{\}\ )$$

Entonces ϑ' sería:

$$\vartheta' = \{V \rightarrow s(W),\ Y' \rightarrow s(\bot)\}.$$

Se ve claramente como se cumplen las 3 condiciones. Observar, en particular que:

$U \in var(G)\backslash dom(\vartheta) \Longrightarrow U \notin dom(\sigma_{out}) \cup dom(\vartheta')$.

Comenzamos estableciendo la definición:

6.4.1.-DEFINICION: Cómputo que Cubre una Solución

Sea G una expresión-gda y Π un programa.

Sea $(s:\vartheta)$ una solución para G con conjunto de variables críticas

$VS = var(G\vartheta) \cup var(s)$.

Sea C un cómputo para G y Π

$$G = G_0 \xrightarrow[\sigma_1]{} G_1 \xrightarrow[\sigma_2]{} G_2 \longrightarrow \ldots\ldots \xrightarrow[\sigma_n]{} G_n = G'$$

Decimos que el cómputo C cubre la solución $(s:\vartheta) \Longleftrightarrow_{def}$

Existe ϑ', d-sustitución parcial apartada de $var(G')$ tal que:

$X \in var(G)\backslash dom(\vartheta) \Longrightarrow X \notin dom(\sigma) \cup dom(\vartheta')$

$\sigma\ \vartheta' \restriction dom(\vartheta) = \vartheta$

$|G'|\vartheta' \gg s$ (en $H_{\Sigma(VS)}$). ■

Para demostrar el teorema de completitud vamos a necesitar anotar las expresiones-gda; o sea, etiquetar cada nodo además de con su símbolo con un término parcial y un número natural. Para explicar la idea revisamos el ejemplo del comienzo del capítulo 5.

Sea el programa:

add(zero,X) ⟶ X

add(succ(X),Y) ⟶ succ(add(X,Y))

double(X) ⟶ add(X,X)

or(X,Y) ⟶ X

or(X,Y) ⟶ Y

Si representamos mediante subíndices las sucesivas potencias del operador T_π obtenemos los valores posibles:

$add_1(zero,\ t) \ni t$;

$add_1(succ(t'),\ t) \ni succ(\perp)$;

$double_1(t) \ni \bot;...$

$add_2(zero,\ t) \ni t;$

$add_2(succ(zero),\ t) \ni succ(t);$

$add_2(succ^{n+2}(t'),\ t) \ni succ(succ(\bot));...$

$double_2(zero) \ni zero;$

$double_2(succ(t)) \ni succ(\bot);...$

$double_3(zero) \ni zero;$

$double_3(succ(zero)) \ni succ(succ(zero));$

$double_3(succ(succ(t))) \ni succ(succ(\bot));...$

Entonces en el modelo mínimo $I_\pi = \coprod_{i<\omega} I_i$ se verifica:

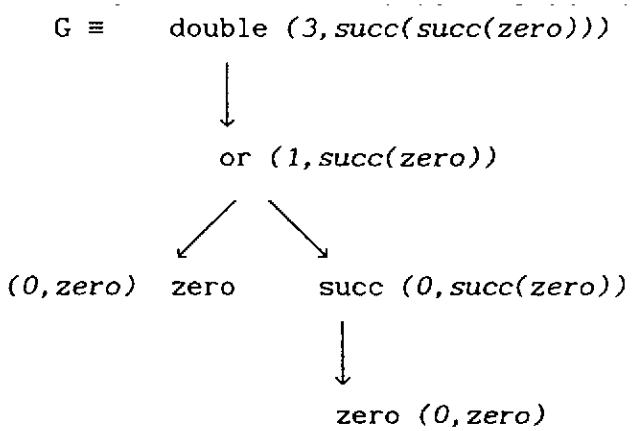
$$double_{I_\pi}(or_{I_\pi}(zero, succ(zero))) \ni succ(succ(zero))$$

Incluso podemos precisar las *menores* potencias:

$$double_3(or_1(zero, succ(zero))) \ni succ(succ(zero))$$

ya que $or_1(zero, succ(zero)) \ni succ(zero)$

Si G es el gda correspondiente a *double(or(zero,succ(zero)))* el gda **anotado** para **testificar** que G ∋ succ(succ(zero)) se obtendría marcando los nodos de función como sigue: en el nodo correspondiente a *double* el par (3,succ(succ(zero))) y en el nodo correspondiente a *or* el par (1,succ(zero)). Por otro parte, los demás nodos se marcarían como en el siguiente esquema:



Intentemos precisar lo anterior:

6.4.2.-DEFINICION: Expresiones-gda anotadas

Sea Π un programa.

Sea G una expresión-gda.

Sea ϑ una d-sustitución parcial.

Sea s un término parcial finito.

Sea VS = var(Gϑ) $\cup$ var(s).

Sea $I_\pi(VS)$ el $\Sigma(VS)$-modelo mínimo del programa Π e $I_k = T_\pi^k(\perp)$ (aproximaciones del modelo mínimo) las sucesivas iteraciones del operador T_π partiendo de la interpretación totalmente indefinida.

Supongamos que:

$$\|G\vartheta\|_{I_\pi(VS)} \ni s.$$

Entonces, al ser s finito, existe una evaluación $\hat{\vartheta}$ de G en $I_\pi(VS)$ bajo ϑ tal que $\hat{\vartheta}(\varepsilon) = s$ para ε nodo raiz de G. Para cada nodo u de G la restricción de $\hat{\vartheta}$ a los nodos del subgrafo G/u de G con raiz u es una evaluación de G/u en $I_\pi(VS)$ bajo ϑ con resultado $s_u = \hat{\vartheta}(u)$. En particular, si u es un nodo, etiquetado por la variable X, de G, entonces s_u es $\vartheta(X)$ que es finito al ser ϑ una d-sustitución parcial.

Además podemos asociar a cada $u \in O(G)$ un número natural k_u como sigue:

$$\triangleright \text{ Si } G[u] \in CS^n_\Sigma,\ n \geq 0 \Longrightarrow k_u = 0$$

Esto incluye el caso en que G[u] sea una variable, ya que estamos tratando las variables como constantes.

$$\triangleright \text{ Si } G[u] \in FS^n_\Sigma,\ n \geq 0 \wedge u \xrightarrow[i]{} u_i\ 1 \leq i \leq n \Longrightarrow$$

$$k_u =_{def} \min \{k \in \mathbb{N} \mid f_k(s_{u_1}, \ldots, s_{u_n}) \ni s_u\}$$

siendo $f_k = f_{I_k}$.

Observar que si $s_u = \perp$ tendríamos $k_u = 0$.

Diremos que el gda G con las dos informaciones (s_u, k_u) asociadas a cada nodo u es una **expresión-gda anotada para testificar que** $G\vartheta \ni s$.

Observar que las anotaciones del nodo raiz son $s_\varepsilon \equiv s$ y cierto $k_\varepsilon \in \mathbb{N}$.

Por abuso de notación, seguiremos llamando G a esta expresión-gda anotada.

Sea G un gda anotado para testificar que $G\vartheta \ni s$.

Sea G' otro gda anotado para testificar que $G'\vartheta' \ni s$.

Decimos que:

$G' \leq G \iff_{def}$

(1) Existe una posición u de G anotada con el par (k_u, s_u) tal que para todo $v \in O(G/u)$, $v \neq u$, $k_v = 0$.

(2) Existe R gda anotado para testificar que $R\mu \ni s_u$, para una d-sustitución parcial μ, con $k_w < k_u$ para todo $w \in O(R)$.

(3) G' se ha obtenido a partir de G reemplazando G/u por R (es decir: se quitan de G los nodos y arcos de G/u, se añaden los nodos y arcos de R y se hacen llegar a la raiz de R los arcos de G que antes llegaban a u).

(4) $\vartheta' = \mu \cup \vartheta$.

La relación anterior "$\leq$" no admite cadenas descendentes infinitas; o sea, no es posible una cadena:

$$G \equiv G_0 \geq G_1 \geq G_2 \geq \ldots \geq G_i \geq G_{i+1} \geq \ldots$$

La terminación de la cadena se deduce considerando, para cada G_i, el multiconjunto de números naturales k_u anotados en los nodos u. Al pasar de G_i a G_{i+1} se obtiene un nuevo multiconjunto donde uno de los números k_u ha sido sustituido por un multiconjunto finito de números menores o iguales que $k_u - 1$. Es decir: al pasar de G_i a G_{i+1}, el multiconjunto asociado a G_i decrece en el "orden de multiconjuntos",

que está bien formado (ver [Dershorwitz & Manna 79]).

La idea subyacente es que el multiconjunto de las anotaciones de los nodos de una expresión-gda nos da una medida de su "complejidad" respecto a una solución correcta $(s:\vartheta)$; es decir, lo que le "falta" a la expresión-gda para que su esqueleto contenga a s. Una complejidad vacía indica que el esqueleto de la expresión-gda contiene ya a s. Vamos a demostrar la completitud de nuestro lenguaje probando que al dar un paso de estrechamiento la complejidad de la nueva expresión-gda es "menor" que la de la antigua con el orden de multiconjuntos anterior. El hecho de que para la relación $\leq$ no existan cadenas descendentes infinitas indica que la sucesión de reducciones termina.

El lema siguiente se utilizará en la demostración del lema de reducción, que es la esencia de la prueba de la completitud de nuestro lenguaje. Necesitamos una definición:

6.4.3.-DEFINICION:

Sea μ una d-sustitución parcial. Entonces definimos:

$$|\mu| : VAR \longrightarrow H_{\Sigma(VAR)}$$

por $|\mu|(X) = |\mu(X)|$. ■

6.4.4.-LEMA:

Sean $t_1,\ldots,t_n$ términos lineales y $e_1, \ldots, e_n$ expresiones. Sean μ y ϑ d-sustituciones parciales tal que

$$(t_1,\ldots,t_n)\ \mu = |(e_1,\ldots,e_n)|\ \vartheta \qquad (1)$$

La tupla $(t_1,\ldots,t_n)$ debe ser lineal y suponemos además que $(t_1,\ldots,t_n)$ y $(e_1,\ldots,e_n)$ no comparten variables.

Entonces:

1) Existe una d-sustitución parcial $\bar{\mu}$ tal que:

$$(t_1,\ldots,t_n)\bar{\mu} = (e_1,\ldots,e_n)\vartheta$$

$$|\ \bar{\mu}\ | = \mu$$

$\bar{\mu} \cup \vartheta$ es un unificador de $(t_1,\ldots,t_n)$ y $(e_1,\ldots,e_n)$.

2) Además, el unificador más general idempotente $(\lambda \cup \sigma)$ de $(t_1, \ldots, t_n)$ y $(e_1, \ldots, e_n)$ con $dom(\lambda) \subseteq var(t_1, \ldots, t_n)$ y $dom(\sigma) \subseteq var(e_1, \ldots, e_n)$ cumple que:

$$\bar{\mu}(Z) \neq \mu(Z) \Longrightarrow Z \in var(t_1,\ldots,t_n)\backslash ran(\sigma)$$

dem:

La demostración es por inducción en el número p de apariciones de constructores en $(t_1,\ldots,t_n)$.

p = 0:

En este caso la igualdad (1) es de la forma:

$$(X_1,\ldots,X_n)\mu = |(e_1, \ldots, e_n)|\vartheta = (|e_1|,\ldots,|e_n|)\vartheta$$

donde las variables X_i $(1 \leq i \leq n)$ son distintas dos a dos. Entonces:

1) Definimos la sustitución $\bar{\mu}$:

$$\bar{\mu}(X_i) = e_i\vartheta \text{ para } 1 \leq i \leq n$$

$$\bar{\mu}(X) = \mu(X) \text{ en otro caso.}$$

que cumple las condiciones requeridas:

$$(X_1,\ldots,X_n)\bar{\mu} = (e_1,\ldots,e_n)\vartheta$$

$$|\ \bar{\mu}\ | = \mu.$$

2) Como en este caso $\sigma = \{\}$ es evidente que:

$$\bar{\mu}(Z) \neq \mu(Z) \Longrightarrow Z \in var(X_1,\ldots,X_n)\backslash ran(\sigma).$$

p > 0:

Podemos suponer, reordenando los t_i y los e_i si es preciso, que el primer término t_1 no es una variable. Tenemos la siguiente situación:

$$(c(s_1,\ldots,s_m),t_2,\ldots,t_n)\mu = (|e_1|,|e_2|,\ldots,|e_n|)\vartheta$$

A partir de ella distinguimos dos casos:

-caso 1:

$e_1 = c(u_1,\ldots,u_m)$ donde las u_j son expresiones para $1 \leq j \leq m$.

Entonces se deduce que:

$$(s_1,\ldots,s_m,t_2,\ldots,t_n)\mu = (|u_1|,\ldots,|u_m|,|e_2|,\ldots,|e_n|)\vartheta$$

y por hipótesis de inducción obtenemos la conclusión del lema.

caso 2:

Como

$$c(s_1,\ldots,s_m)\mu = |e_1|\vartheta \wedge e_1 \neq c(v_1,v_2,\ldots,v_m)$$

concluimos forzosamente que la única posibilidad que resta es que

$e_1 = Y \in VAR$ tal que $c(s_1,\ldots,s_m)\mu = |Y|\vartheta = Y\vartheta$.

Por otro lado, $(t_2,\ldots,t_n)\mu = (|e_2|,\ldots,|e_n|)\vartheta$, y por hipótesis de inducción:

1) Obtenemos $\bar{\mu}$ tal que:

$$(t_2,\ldots,t_n)\bar{\mu} = (e_2,\ldots,e_n)\vartheta$$
$$|\bar{\mu}| = \mu$$

2) Además, el unificador más general idempotente $(\lambda_r \cup \sigma_r)$ de $(t_2,\ldots,t_n)$ y $(e_1,\ldots,e_n)$ cumple que:

$$\bar{\mu}(Z) \neq \mu(Z) \Longrightarrow Z \in var(t_2,\ldots,t_n)\backslash ran(\sigma_r).$$

Entonces se deduce que:

1) Existe una d-sustitución parcial, la propia $\bar{\mu}$, tal que:

$$(t_1,\ldots,t_n)\bar{\mu} = (e_1,\ldots,e_n)\vartheta$$

ya que

$$t_1\bar{\mu} = c(s_1,\ldots,s_m)\bar{\mu} = c(s_1,\ldots,s_m)\mu = Y\vartheta = e_1\vartheta.$$

(porque $var(t_1) \cap var(t_2,\ldots,t_n) = \emptyset$ y $\bar{\mu}(Z) \neq \mu(Z)$ sólo si $Z \in var(t_2,\ldots,t_n)\backslash ran(\sigma_r)$)

y que cumple la otra condición:

$$| \bar{\mu} | = \mu$$

2) El u.m.g. idempotente en este caso es de la forma:

$$\lambda \cup \sigma = \lambda_r \cup \{Y \longrightarrow c(s_1, \ldots, s_m)\}\ \sigma_r.$$

como por hipótesis de inducción se cumple que

$$\bar{\mu}(Z) \neq \mu(Z) \Longrightarrow Z \in var(t_2, \ldots, t_n) \backslash ran(\sigma_r)$$

y $var(t_2, \ldots, t_n) \backslash ran(\sigma_r) \subseteq var(t_1, \ldots, t_n) \backslash ran(\sigma)$ obtenemos:

$$\bar{\mu}(Z) \neq \mu(Z) \Longrightarrow Z \in var(t_1, \ldots, t_n) \backslash ran(\sigma).$$

De donde para ambos casos se verifica lo que buscabamos. ■

Para demostrar la completitud probamos en primer lugar un lema de reducción que dice que al dar un paso de estrechamiento se obtiene una nueva expresión-gda que es "menos compleja" respecto al mismo resultado que la anterior.

6.4.5.-LEMA: (Lema de Reducción)

Si G es una expresión-gda anotada para testificar que $G\vartheta \ni s$ pero no verifica que $| G\vartheta | \gg s$, puede efectuarse un paso de estrechamiento

$$G \Longrightarrow_{\sigma} G'$$

de modo que exista una d-sustitución parcial, ϑ', apartada de var(G'), y se verifiquen las condiciones siguientes:

1) $X \in var(G) \backslash dom(\vartheta) \Longrightarrow X \notin dom(\sigma) \cup dom(\vartheta')$

2) $\sigma\vartheta' \restriction dom(\vartheta) = \vartheta$

3) G' puede anotarse para testificar $G'\vartheta' \ni s$ con una anotación tal que $G' \leq G$.

dem:

Sea $VS = var(G\vartheta) \cup var(s)$.

Sea $\|G\vartheta\|_{I_\pi(VS)} \ni s$. Y sea $\hat{\vartheta}$ una evaluación de G en $I_\pi(VS)$ bajo ϑ.

Sean para cada posición u de G sus anotaciones $s_u = \hat{\vartheta}(u)$ y $k_u \in \mathbb{N}$.

En G deben existir nodos u tal que $k_u > 0$, pues en caso contrario se

tendría $|G\vartheta| \gg s$. Podemos pues elegir un nodo u de rango mínimo entre todos los que verifiquen $k_u > 0$ de modo que los nodos v descendientes de u en G verifiquen $k_v = 0$. En estas condiciones, G[u] será necesariamente un símbolo de función f de cierta aridad n y si $u_1, \ldots, u_n$ son los hijos de u en G sabemos que $f_{k_u}(s_{u_1}, \ldots, s_{u_n}) \ni s_u$.

Además, podemos suponer que para $1 \leq i \leq n$, $s_{u_i} \ll |G/u_i|$ ya que todos los nodos v de G/u_i están anotados con $k_v = 0$.

Llamamos $s_i =_{def} |G/u_i|$ para $1 \leq i \leq n$.

Como $s_{u_i} \ll s_i$ $(1 \leq i \leq n)$, $f_{k_u}(s_{u_1}, \ldots, s_{u_n}) \ni s_u$ y f_{k_u} es una función monótona se tiene también que $f_{k_u}(s_1, \ldots, s_n) \ni s_u$.

Observar que el caso particular " n = 0 " está comprendido en el razonamiento que hacemos.

Del hecho que $f_{k_u}(s_1, \ldots, s_n) \ni s_u$ y por definición del operador T_π podemos elegir una variante de una regla, supongamos que sea $f(t_1, \ldots, t_n) \longrightarrow r$, de Π y una cierta $\mu : VAR \longrightarrow H_{\Sigma(VS)}$ que por monotonía y continuidad de la evaluación (4.2.2) se puede tomar como una d-sustitución lineal tal que

$$f(t_1, \ldots, t_n)\mu = f(s_1, \ldots, s_n)$$

y con $\|r\mu\|_{k_u - 1} \ni s_u$.

Observar que $dom(\mu) \cap dom(\vartheta) = \varnothing$

Entonces:

$$|G/u_i|\vartheta = s_i \wedge t_i\mu = s_i \quad (1 \leq i \leq n)$$

Sea $e_i = expr(G, u_i)$.

Por el lema 6.4.4 se deduce que $(t_1, \ldots, t_n)$ y $(e_1, \ldots, e_n)$ son unificables con unificador $\bar{\mu} \cup \vartheta$ y u.m.g. idempotente $\lambda \cup \sigma$ con $dom(\lambda)$

$\subseteq var(t_1,\ldots,t_n)$ y $dom(\sigma) \subseteq var(e_1,\ldots,e_n)$ que cumple la igualdad:

$$(\bar{\mu} \cdot \cup \vartheta) = (\lambda \cdot \cup \sigma)(\bar{\mu} \cdot \cup \vartheta)$$

y tal que $\bar{\mu}$ cumple además:

$$| \bar{\mu} | = \mu$$

$\bar{\mu}(Z) \neq \mu(Z)$ sólo si $Z \in var(t_1,\ldots,t_n)\backslash ran(\sigma)$.

Trasladamos lo anterior al estrechamiento en gda's que hemos definido como nuestra semántica operacional.

Sea $\mathfrak{G}_0$ el grafo formado por los gdas correspondientes a $l = f(t_1, \ldots, t_n)$, r y G.

Sea $\mathfrak{G}_1$ el grafo que queda después de aplicar el algoritmo de unificación con nodos de entrada ε_l (nodo raiz de l) y u; y posteriormente reemplazar el nodo u por el nodo ε_r (nodo raiz de r).

En el proceso se obtiene (ver 5.4.1):

▷ σ d-sustitución lineal que liga variables de G con subtérminos de $f(t_1,\ldots,t_n)$;

▷ la sustitución $\lambda = \{ X_1 \longrightarrow expr(\mathfrak{G}_1, v_1), \ldots, X_q \longrightarrow expr(\mathfrak{G}_1, v_q) \}$ donde para $1 \leq i \leq q$ las X_i son variables de $f(t_1,\ldots,t_n)$ cuyos nodos correspondientes están unidos por punteros a los nodos v_i de $\mathfrak{G}/\varepsilon_G$.

▷ $G' = \mathfrak{G}_1/\varepsilon_G$.

(NOTA: Si u es el nodo raiz de G entonces $G' = \mathfrak{G}_1/\varepsilon_r$)

Puede, por tanto, escribirse el paso de estrechamiento:

$$G \equiv \mathfrak{G}_0/\varepsilon_G \Longrightarrow_\sigma \mathfrak{G}_1/\varepsilon_G \equiv G'.$$

Tenemos que demostrar que existe una ϑ' tal que:

1) $X \in var(G)\backslash dom(\vartheta) \Longrightarrow X \notin dom(\sigma) \cup dom(\vartheta')$

2) $\sigma\vartheta' \restriction dom(\vartheta) = \vartheta$.

3) G' puede anotarse para testificar $G'\vartheta' \ni s$ con una anotación tal que $G' \leq G$.

Definimos:

$$\vartheta' =_{def} \mu \dot{\cup} \vartheta.$$

y vamos a probar 1) a 3):

1)

Probar que:

$$X \in var(G)\backslash dom(\vartheta) \Longrightarrow X \notin dom(\sigma) \cup dom(\vartheta')$$

es lo mismo que probar:

$$X \in var(G)\backslash dom(\vartheta) \Longrightarrow X \notin dom(\sigma) \cup dom(\mu)$$

ya que $\vartheta' = \mu \dot{\cup} \vartheta$.

Si $X \notin dom(\vartheta)$ entonces $X \notin dom(\sigma)$ pues σ es más general que ϑ en el sentido de la igualdad $(\bar{\mu} \dot{\cup} \vartheta) = (\lambda \dot{\cup} \sigma)(\bar{\mu} \dot{\cup} \vartheta)$.

Y por otra parte, si $X \in var(G)\backslash dom(\vartheta)$ se tiene que $X \notin dom(\mu)$ pues $var(G) \cap var(l \rightarrow r) = \varnothing$ y $dom(\mu) \subseteq var(l \rightarrow r)$.

2)

Tenemos:

$\sigma\vartheta' = \sigma(\mu \dot{\cup} \vartheta)$ (por definición de ϑ')

$\underset{dom(\vartheta)}{=} (\lambda \dot{\cup} \sigma)(\mu \dot{\cup} \vartheta)$ (ya que $dom(\lambda) \cap dom(\vartheta) = \varnothing$)

$\underset{dom(\vartheta)}{=} (\lambda \dot{\cup} \sigma)(\bar{\mu} \dot{\cup} \vartheta)$ (ya que si $X \in dom(\vartheta)$ entonces X no es del $dom(\lambda)$ ni del $dom(\mu)$ ni del $dom(\bar{\mu})$. Además hay dos posibilidades:

1) X no es del $dom(\sigma)$, y se obtiene:

$(\lambda \dot{\cup} \sigma)(\bar{\mu} \dot{\cup} \vartheta)(X) = \vartheta(X)$

$= (\lambda \dot{\cup} \sigma)(\mu \dot{\cup} \vartheta)(X)$.

2) Si $X \in dom(\vartheta) \cap dom(\sigma)$ basta ver que $\mu(Z) = \bar{\mu}(Z)$ para $Z \in \sigma(X)$ pero esto está garantizado por 6.4.4)

$= \bar{\mu} \dot{\cup} \vartheta$ (por ser u.m.g. idempotente).

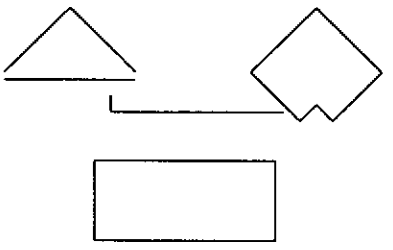
$\overset{=}{dom(\vartheta)}\ \vartheta$.

3)

Consideremos $(\mathfrak{G}_1/\varepsilon_r)\vartheta'$. Como ϑ' es una d-sustitución parcial podemos considerarla como una valoración para las variables del grafo $\mathfrak{G}_1$ y aplicar el lema de sustitución en gda's.

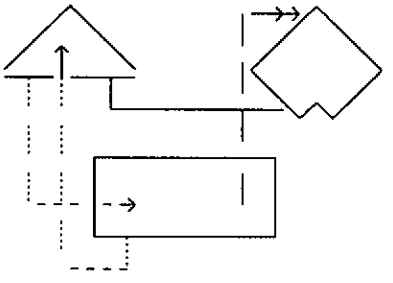
En un gráfico:

$\mathfrak{G}_0$



$\uparrow \varphi$

$\mathfrak{G}_1$



$\uparrow \vartheta' = \mu \cup \vartheta$

Sea $\mathfrak{H}$ el gda formado considerando conjuntamente las expresiones-gda $\mathfrak{G}_1/v_1$, ..., $\mathfrak{G}_1/v_p$ y donde v_i $(1 \leq i \leq p)$ son los nodos donde acaban los punteros que salen de las variables X_i compartidas por $f(t_1,\ldots,t_n)$ y r en la unificación.

Por el lema de sustitución en gda's se tiene que son equivalentes las afirmaciones siguientes para x finito:

(1) $\|\mathfrak{G}_1/\varepsilon_r\|_{k_u-1}(\vartheta') \ni x$

(2) Existe una $\hat{\phi}$ evaluación de $\mathfrak{H}$ bajo ϑ' tal que

$$\|\mathfrak{G}_0/\varepsilon_r\|_{k_u-1}(\varphi) \ni x$$

donde φ es la valoración tal que:

$$\varphi(X_i) = \hat{\phi}(v_i) \text{ para } 1 \leq i \leq k$$

$$\varphi(X) = \vartheta'(X) \text{ en otro caso.}$$

Veamos que podemos aplicar el resultado anterior tomando $\varphi = \mu \cup \vartheta$ y $x = s_u$.

▷ Para $1 \leq i \leq m$ tomamos como $\hat{\phi}(v_i)$ los siguientes:

$$\hat{\phi}(v_i) = |\mathfrak{G}_1/v_i|^{\vartheta'}$$

que es una elección correcta ya que

$$|\mathfrak{G}_1/v_i|^{\vartheta'} \in \|\mathfrak{G}_1/v_i\|_{k_u-1}(\vartheta').$$

Entonces tenemos:

$$\begin{aligned}
\varphi(X_i) &= \hat{\phi}(v_i) \\
&= |\mathfrak{G}_1/v_i|^{(\mu \dot{\cup} \vartheta)} && \text{(por definición de } \vartheta') \\
&= |\mathfrak{G}_1/v_i|^{(\bar{\mu} \dot{\cup} \vartheta)} && \text{(ya que si } Z \in \mathrm{var}(\mathfrak{G}_1/v_i) \text{ entonces} \\
& && Z \notin \mathrm{var}(f(t_1,\ldots,t_n)\backslash \mathrm{ran}(\sigma)) \text{ y por} \\
& && 6.4.4 \Longrightarrow \bar{\mu}(Z) = \mu(Z)) \\
&= |X_i\lambda|^{(\mu \dot{\cup} \vartheta)} && \text{(ya que el esqueleto de una} \\
& && \text{expresión coincide con el de su} \\
& && \text{correspondiente expresión-gda)} \\
&= |X_i\lambda(\mu \dot{\cup} \vartheta)| \\
&= |X_i(\lambda \dot{\cup} \sigma)(\mu \dot{\cup} \vartheta)| && \text{(ya que } \sigma \text{ no actúa sobre} \\
& && \text{las variables de } X_i) \\
&= |X_i(\mu \dot{\cup} \vartheta)| && \text{(por la propiedad del u.m.g.)} \\
&= X_i|\mu \dot{\cup} \vartheta| && \text{(por la definición 6.4.3)} \\
&= X_i(\mu \dot{\cup} \vartheta) && \text{(por el lema 6.4.4)}
\end{aligned}$$

▷ en otro caso:

$$\varphi(X) = \vartheta'(X) = (\mu \dot{\cup} \vartheta)(X)$$

En conclusión:

$$\varphi = \mu \dot{\cup} \vartheta.$$

Además, podemos razonar así:

$\|r\mu\|_{k_u-1} \ni s_u \iff \|(\mathfrak{G}_0/\varepsilon_r)\mu\|_{k_u-1} \ni s_u$ (por la proposición 6.1.2)

$\iff \|\mathfrak{G}_0/\varepsilon_r\|_{k_u-1}(\mu) \ni s_u$ (μ es d-sust. parcial)

$\iff \|\mathfrak{G}_0/\varepsilon_r\|_{k_u-1}(\varphi) \ni s_u$ ($\varphi = \mu$ para $X \in var(r)$)

$\iff \|\mathfrak{G}_1/\varepsilon_r\|_{k_u-1}(\vartheta') \ni s_u$ (por el lema de sustitución)

El apartado 3) del lema de reducción se prueba, entonces, en base a los siguientes hechos:

(3a) Por hipótesis, G es una expresión-gda anotada para testificar que $G\vartheta \ni s$ con $\hat{\vartheta}$ evaluación que hemos considerado para dicha anotación. En particular, vimos que existe una posición u de G anotada por el término parcial finito s_u, y el número natural $k_u > 0$.

(3b) Como $\|r\|_{k_u-1}(\mu) \ni s_u$ podemos considerar r como una expresión-gda anotada para testificar $r\mu \ni s_u$ donde hemos usado una $\hat{\mu}$ como evaluación de los nodos v de r y donde se verifica:

$$k_v \leq k_u - 1 \text{ para todo } v.$$

(3c) Teniendo en cuenta que $\|\mathfrak{G}_1/\varepsilon_r\|_{k_u-1}(\vartheta') \ni s_u$ demostramos que

$$\|\mathfrak{G}_1/\varepsilon_G\|_{I_\pi}(\vartheta') = \|G'\|_{I_\pi}(\vartheta') \ni s.$$

Para ello, definimos en G' una evaluación $\hat{\vartheta}'$ como sigue:

$$\hat{\vartheta}'(v) = \hat{\vartheta}(v) \text{ para todo } v \in O(G) \cap O(G')$$

$$\hat{\vartheta}'(v) = \hat{\mu}(v) \text{ para todo } v \in O(r)$$

siendo $\hat{\vartheta}$ y $\hat{\mu}$ las evaluaciones de G en I_π bajo ϑ con resultado s y de r en I_π bajo μ con resultado s_u citadas en (3a) y (3b).

Observar que todos los arcos de G con un hijo en u que evaluaba a s_u

se transforman en arcos de G' con un hijo en ε_r que también evalúa a s_u. Por otra parte, no hay problemas de evaluación planteados por nodos compartidos v de G descendientes de u debido a que están todos anotados con $k_v = 0$ y por tanto están etiquetados por constructoras o por símbolos de función evalúados por $\perp$.

(3d) G' con la evaluación $\hat{\vartheta}'$ se puede considerar un gda anotado para testificar que $G'\vartheta' \ni s$.

En efecto: Si $VS' =_{def} var(G'\vartheta') \cup var(s)$ entonces es evidente que $\|G'\|_{I_\pi(VS')}(\vartheta') \ni s$ siendo $I_\pi(VS')$ el $\Sigma(VS')$-modelo mínimo del programa Π. Además, los nodos de G' se anotan como los de G si pertenecen a $O(G) \cap O(G')$ y como los de r si pertenecen a O(r).

(3e) Que $G' \leq G$ es ahora inmediato de todos los subapartados anteriores y de la definición 6.4.2. ■

El teorema siguiente es el resultado culminante de nuestro trabajo y es el inverso del teorema de corrección (6.3.5). La demostración es una aplicación reiterada del lema de reducción anterior.

6.4.6.-TEOREMA: (Completitud)

Sea G una expresión-gda y Π un programa.

Cualquier solución $(s:\vartheta)$ para una expresión-gda G que sea correcta en el modelo mínimo de Π para $VS = var(G\vartheta) \cup var(s)$ está cubierta por una solución calculada.

dem:

Sea $(s:\vartheta)$ una solución para G que es correcta en el modelo mínimo de Π para $VS = var(G\vartheta) \cup var(s)$. O sea, $I_\pi(VS) \models G\vartheta \ni s$. O, de otro modo:

$$\|G\vartheta\|_{I_\pi(VS)} \ni s.$$

Utilizando las pautas señaladas en 6.4.2 podemos considerar G como una

expresión-gda anotada para testificar que $G\vartheta \ni s$.

Para demostrar el teorema, vamos a ver que dada una solución correcta $(s:\vartheta)$ en el modelo mínimo $I_\pi(VS)$ para una expresión-gda G se puede encontrar un cómputo:

$$G \equiv G^0 \Longrightarrow_{\sigma_1} G^1 \Longrightarrow_{\sigma_2} G^2 \Longrightarrow \ldots \Longrightarrow_{\sigma_n} G^n \equiv G'$$

que cubra la solución correcta $(s:\vartheta)$. Para ello, mientras no se tenga $|G^i\vartheta^i| \gg s$, realizamos un paso de estrechamiento $G^i \Longrightarrow G^{i+1}$ del modo que hemos visto en el lema de reducción. Hemos de razonar como se comporta la composición de estos pasos.

Aplicando el lema de reducción en el primer paso de estrechamiento tenemos que se verifica que:

(1^0) $X \in var(G^0)\backslash dom(\vartheta) \Longrightarrow X \notin dom(\sigma_1) \cup dom(\vartheta^1)$

(2^0) $\sigma_1\vartheta^1 \restriction dom(\vartheta) = \vartheta$.

(3^0) $I_\pi(VS^1) \models G^1\vartheta^1 \ni s$ donde $VS^1 = var(G^1\vartheta^1) \cup var(s)$

(4^0) G^1 para $(s:\vartheta^1) \leq G$ para $(s:\vartheta)$.

Después del segundo paso se verifica:

(1^1) $X \in var(G^1)\backslash dom(\vartheta^1) \Longrightarrow X \notin dom(\sigma_2) \cup dom(\vartheta^2)$

(2^1) $\sigma_2\vartheta^2 \restriction dom(\vartheta^1) = \vartheta^1$.

(3^1) $I_\pi(VS^2) \models G^2\vartheta^2 \ni s$ donde $VS^2 = var(G^2\vartheta^2) \cup var(s)$

(4^1) G^2 para $(s:\vartheta^2) \leq G^1$ para $(s:\vartheta^1)$.

Ahora probamos que:

▷ $X \in var(G^0)\backslash dom(\vartheta) \Longrightarrow X \notin dom(\sigma_1\sigma_2) \cup dom(\vartheta^2)$

En efecto: sea $X \in var(G^0)\backslash dom(\vartheta)$. Se tiene por (1^0) que $X \notin dom(\sigma_1)$ y $X \notin dom(\vartheta^1)$. Entonces hay dos posibilidades:

a) $X \in var(G^1)$. Luego $X \in var(G^1)\backslash dom(\vartheta^1) \Longrightarrow$ por (1^1) $X \in dom(\sigma_2) \cup dom(\vartheta^2) \Longrightarrow X \notin dom(\sigma_1\sigma_2) \cup dom(\vartheta^2)$ ya que $X \notin dom(\sigma_1)$.

b) $X \notin var(G^1)$. Entonces es inmediato que $X \notin dom(\sigma_2) \cup dom(\vartheta^2)$ y por

tanto que $X \notin dom(\sigma_1\sigma_2) \cup dom(\vartheta^2)$.

En ambos casos: $X \notin dom(\sigma_1\sigma_2) \cup dom(\vartheta^2)$.

▷ $\sigma_1\sigma_2\vartheta^2 \restriction dom(\vartheta) = \vartheta$.

Esto es inmediato de lo que hemos probado en el apartado anterior y de los hechos obtenidos en (2^1) y (2^0).

▷ $I_\pi(VS^2) \models G^2\vartheta^2 \ni s$ donde $VS^2 = var(G^2\vartheta^2) \cup var(s)$

▷ G^2 para $(s:\vartheta^2) \leq G^1$ para $(s:\vartheta^1) \leq G$ para $(s:\vartheta)$.

En consecuencia, aplicando reiteradamente el resultado anterior se obtiene:

(1) $X \in var(G)\backslash dom(\vartheta) \Longrightarrow X \notin dom(\sigma_1 \ldots \sigma_n) \cup dom(\vartheta^n)$

(2) $\sigma_1 \ldots \sigma_n\vartheta^n \restriction dom(\vartheta) = \vartheta$.

(3) $I_\pi(VS^n) \models G^n\vartheta^n \ni s$ donde $VS^n = var(G^n\vartheta^n) \cup var(s)$

(4) G^n para $(s:\vartheta^n) \leq \ldots \leq G$ para $(s:\vartheta)$.

En algún momento llegará a cumplirse que $|G^n\vartheta^n| \gg s$, ya que en otro caso obtendríamos una cadena infinita descendente

$$G \geq G^1 \geq \ldots \geq G^n \geq \ldots$$

lo cual es imposible como vimos en la definición 6.4.2. Una vez que se cumpla $|G^n\vartheta^n| \gg s$ habremos completado un cómputo que cubre la solución correcta $(s:\vartheta)$ dada inicialmente. ■

Como corolario del teorema podemos ver que el modelo mínimo de Herbrand tiene la propiedad de caracterizar los objetivos resolubles.

6.4.7.-COROLARIO: Canonicidad del Modelo Mínimo de Herbrand

Sea Π un programa. Sea G una expresión-gda, ϑ una d-sustitución lineal apartada de var(G), y sea s un término parcial finito. Sea $VS = var(G\vartheta) \cup var(s)$. Entonces son equivalentes las dos afirmaciones siguientes:

(1) $(s:\vartheta)$ es una solución correcta para G en el modelo mínimo de Π sobre VS.

(2) $(s:\vartheta)$ es una solución correcta con respecto a Π.

dem:

(2) $\Longrightarrow$ (1) Inmediato

(1) $\Longrightarrow$ (2) Si se cumple (1) la demostración del teorema de completitud que sólo usaba esta hipótesis nos asegura que alguna solución calculada cubre a $(s:\vartheta)$. Entonces, por el teorema de corrección la solución calculada, y con ella también $(s:\vartheta)$, es una solución correcta en todos los modelos. ■

7. UTILIZACION DEL LENGUAJE: UN COMPILADOR

Construido nuestro lenguaje, vamos a estudiar las posibilidades de utilizarlo como lenguaje de programación. No es nuestro objetivo en este capítulo diseñar, por ejemplo, una implementación del mismo; se deja para el futuro esta cuestión y vamos a insistir en aplicaciones inmediatas que muestren las buenas cualidades del lenguaje. Y como ejemplo vamos a construir un compilador para un subconjunto de PASCAL.

En este capítulo se estudia, en el primer apartado, algunos convenios para una eficaz utilización del lenguaje definiendo algunas funciones de uso común. Estas podrían considerarse como primitivas del mismo y quedarían implícitas en cualquier programa. El programa en el que las vamos a utilizar de inmediato va a ser precisamente el citado compilador que desarrollamos en el apartado final.

7.1. ALGUNAS POSIBLES FUNCIONES PRIMITIVAS

En una primera aproximación al tema, recordando los ejemplos del capítulo 2, parecen bastante evidentes cuales deberían ser estas funciones primitivas, definidas implícitamente por medio de reglas de nuestro lenguaje. Podemos clasificarlas como sigue:

Constantes booleanas:

Suponíamos en nuestro lenguaje $CS_{\Sigma} \neq \emptyset$.

Las dos constantes siguientes pueden considerarse incluídas en el mismo:

$$true \in CS^0_\Sigma$$
$$false \in CS^0_\Sigma.$$

Además suponemos que hay un número suficiente de constantes para nuestros programas.

Por ejemplo, para el compilador del apartado siguiente supondremos que tenemos suficientes **identificadores** que pueden ser constantes o términos de la forma #id(i), donde i es un término que representa un número natural.

Reglas para las conectivas lógicas:

Debido a la existencia de funciones indeterministas y la no existencia de problemas de confluencia se pueden considerar las reglas para la conjunción (and) y para la disyunción (or) paralelas que escritas en forma infija serían:

(true and Y) ⟶ Y.	(X and true) ⟶ X.
(false and Y) ⟶ false.	(X and false) ⟶ false.

(true or Y) ⟶ true.	(X or true) ⟶ true.
(false or Y) ⟶ Y.	(X or false) ⟶ X.

Estas reglas no cumplen la propiedad de no-ambigüedad de los sistemas de reescritura lineales de Huet y Levy [Huet & Levy 79] (aunque si

las condiciones de las reglas BABEL [Moreno & Rodriguez 89]).

Se suelen considerar, por ser más fácil su implementación reglas para la conjunción (and) y para la disyunción (or) secuenciales:

(true and Y) ⟶ Y. (true or Y) ⟶ true.

(false and Y) ⟶ false. (false or Y) ⟶ Y.

También podemos considerar las reglas para la negación:

not (true) ⟶ false.

not (false) ⟶ true.

Reglas condicionales:

Las siguientes reglas las escribimos, para mejorar su legibilidad, en forma infija-mixta:

(if true then X) ⟶ X.

(if true then X else Y) ⟶ X.

(if false then X else Y) ⟶ Y.

Otra regla a considerar en este apartado es:

(let true in X) ⟶ X.

El comportamiento de esta regla es formalmente idéntico a la regla para if_then pero puede ser más significativa en determinados contextos, como cuando el primer argumento es una conjunción de ecuaciones. En general, usaremos una u otra según convenga para

entender el sentido declarativo de las reglas.

Constructores numéricos y de listas:

Podemos suponer la existencia de $0 \in CS^0_\Sigma$ y $succ \in CS^1_\Sigma$ para construir los números naturales.

Igualmente podemos suponer la existencia de la lista vacía $[] \in CS^0_\Sigma$ y del constructor de listas *cons* o $[\ .\ :\ .\ ] \in CS^2_\Sigma$.

Reglas para la desigualdad:

Para números naturales se puede definir una regla para la desigualdad:

```
( succ(X) < 0 ) ⟶ false.
( 0 < succ(Y) ) ⟶ true.
( succ(X) < succ(Y) ) ⟶ X < Y.
```

Podemos suponer también que existe un orden lexicográfico que escribiremos "«". Este orden será necesario definirlo sobre los identificadores de variables del compilador como veremos más adelante.

Reglas para la igualdad:

El tratamiento de la igualdad merece algunos comentarios.

Lo más sencillo sería dar una definición similar a las funciones predefinidas anteriores. Por ejemplo, la siguiente:

```
(c = c) ⟶ true.                          para todo c ∈ CS⁰Σ.
( c(X1,...,Xn) = c(Y1,...,Yn) ) ⟶
     (X1 = Y1) and ... and (Xn = Yn).    para todo c ∈ CSⁿΣ, n ≥ 1
```

Sin embargo, el siguiente programa para concatenar dos listas expresado en nuestro lenguaje nos señala algunos problemas:

append([],Xs) $\longrightarrow$ Xs.

append([X : Xs],Ys) $\longrightarrow$ [X : append(Xs,Ys)].

Supongamos que tenemos el objetivo siguiente planteado por la igualdad:

append(Xs,Ys) = [Z_1,Z_2].

Sabemos que en PROLOG se obtienen las soluciones siguientes:

Xs = []; Ys = [Z_1,Z_2].

Xs = [Z_1]; Ys = [Z_2].

Xs = [Z_1,Z_2]; Ys = [].

mientras que con el estrechamiento definido en nuestro lenguaje pueden surgir computaciones como la siguiente:

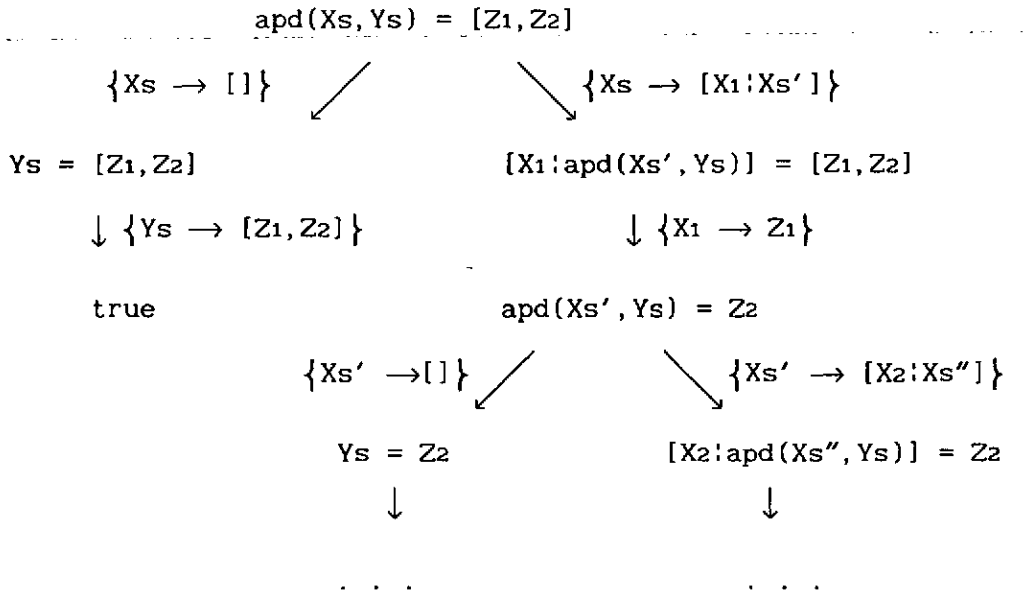
append(Xs,Ys) = [Z_1,Z_2]

$$\Longrightarrow \quad Ys = [Z_1, Z_2]$$
$$\{Xs \rightarrow []\}$$
$$\Longrightarrow \quad (Y_1 = Z_1) \text{ and } (Ys' = [Z_2])$$
$$\{Ys \rightarrow [Y_1 : Ys']\}$$
$$\Longrightarrow \quad Ys' = [Z_2]$$
$$\{Y_1 \rightarrow 0,\ Z_1 \rightarrow 0\}$$
$$\Longrightarrow \quad (Y_2 = Z_2) \text{ and } (Ys'' = [])$$
$$\{Ys' \rightarrow [Y_2 : Ys'']\}$$
$$\Longrightarrow \quad (Y_2' = Z_2') \text{ and } (Ys'' = [])$$
$$\{Y_2 \rightarrow s(Y_2'),\ Z_2 \rightarrow s(Z_2')\}$$
$$\Longrightarrow \quad Ys'' = []$$
$$\{Y_2' \rightarrow 0,\ Z_2' \rightarrow 0\}$$
$$\Longrightarrow \quad true$$
$$\{Ys'' \rightarrow []\}$$

que vincula las variables a valores particulares de la solución general.

Una solución para el problema es tratar en particular la igualdad como **unificación semántica**. Es decir, intentar unificar después de evaluar; o de otro modo, reducir ambos miembros de la igualdad a términos y entonces unificar.

Siguiendo el ejemplo propuesto tendríamos:



que permite obtener las mismas soluciones que en PROLOG.

Adoptar esta solución, la **unificación semántica**, nos llevaría a revisar la semántica de nuestro lenguaje a la que habría que "añadir" la semántica de la igualdad.

Para el ejemplo del compilador que vamos a desarrollar la ineficiencia que se observa con la definición de igualdad dada al principio es irrelevante. Adoptaremos, pues, ésta dejando para futuros trabajos el desarrollo de la otra definición de igualdad y las modificaciones semánticas que ocasiona.

Reglas para las operaciones aritméticas:

Supondremos también definidas las operaciones aritméticas elementales sobre números enteros: suma, resta, ... que representaremos por los signos habituales en forma infija: +, -, ...

7.2. UN COMPILADOR

Presentamos a continuación un compilador escrito en nuestro lenguaje para un subconjunto de PASCAL. Suponemos a partir de ahora definidas las reglas (primitivas) que hemos descrito y que disponemos de suficientes identificadores como indicabamos en el apartado anterior. También, haremos el convenio en este apartado de escribir los constructores no numéricos precedidos de un signo "#".

Primeramente, describimos el planteamiento general y los objetivos buscados. En sucesivos apartados analizamos las partes de que consta el programa.

7.2.1 PUNTO DE PARTIDA

El *lenguaje fuente* para el compilador es una versión simplificada de PASCAL. La sintáxis BNF del mismo es la siguiente:

```
<programa> ::= 'program' <identificador> ';'<sentencia>
<sentencia> ::=
     | 'read' <variable>
     | 'write' <expresion>
     | <variable> ':=' <expresion>
     | 'if' <condición> 'then' <sentencia> 'else' <sentencia>
     | 'while' <condición> 'do' <sentencia>
     | 'begin' <sentencias> 'end'
<sentencias> ::=
     | <sentencia>
     | <sentencia> ';' <sentencias>
```

No definimos la sintáxis de expresiones y condiciones que aparece más adelante.

Para mostrar el funcionamiento del compilador consideraremos tres ejemplos escritos en el lenguaje fuente anterior. El primero, muy sencillo, muestra el manejo de expresiones aritméticas. El segundo es una muestra simple del funcionamiento del condicional *if_then_else* que no se usa en el tercero, un programa algo más complejo.

EJEMPLO *first*:

```
program first ;
begin
    write (2 + 3 ) * 5
end
```

EJEMPLO *second*:

```
program second ;
begin
    if a < b then min := a else min := b
end
```

EJEMPLO *third*:

```
program third ;
begin
    sum := 0 ;
    count := 0 ;
    while count < 11 do
        begin
```

```
            sum := sum + count ;
            count := count + 1
        end ;
    write ( sum )
end.
```

El *lenguaje objeto* puede considerarse como un lenguaje máquina simple para una computadora que posee un acumulador y registros de memoria. Las instrucciones que consideramos son las siguientes:

aritméticas:

add <num>	suma al acumulador el contenido de la dirección de memoria <num>
sub <num>	idem
mul <num>	idem
div <num>	idem
loadc <cte>	carga <cte> en el acumulador.
load <num>	carga el contenido de la dirección <num> en el acumulador.
store <num>	almacena el contenido del acumulador en la dirección de memoria <num>.

de control:

jump <num>	salto incondicional a la instr. <num>
jumpeq <num>	salto condicional a la instr. <num>
jumpne <num>	idem
jumplt <num>	idem

jumpgt <num> idem

jumple <num> idem

jumpge <num> idem

de entrada/salida:

readop lectura

writeop escritura

halt parada

El compilador que vamos a mostrar traduce correctamente cualquier programa en el lenguaje fuente pero no maneja errores. Añadir esa posibilidad alargaría excesivamente el número de reglas del compilador.

La tarea de compilación consta de cuatro etapas. La primera transforma un programa fuente en una lista de unidades léxicas o *lexemas*. En la segunda etapa la lista de lexemas se analiza para dar una estructura fuente. En la tercera, la estructura fuente se transforma en código reubicable. En la cuarta, el código reubicable es ensamblado en código objeto absoluto.

Nuestro trabajo se limita a las tres últimas etapas que denominaremos respectivamente: analizador, generador de código y ensamblador. La primera etapa es sencilla y no la consideramos.

El punto de partida es la lista de lexemas *Tokens*. La función básica del programa es *compile(Tokens)* que toma como entrada la lista *Tokens*, construye un árbol de análisis *Statement*, código reubicable en el árbol *Code* y da como salida el código objeto absoluto *ObjectCode*:

```
compile(Tokens) ⟶
    let  #prog(Name,Statement) = parse(Tokens) and
         Code = encode(Statement,Table) and
         ObjectCode = assemble(Code,Table)
    in ObjectCode.
```

Se necesita como auxiliares para generar código un diccionario *Table* que asocia a las variables direcciones de memoria y tener en cuenta unas etiquetas (*labels*) de direcciones de programa.

Esta regla es el esquema básico del compilador. En ella se utilizan las propiedades más potentes y expresivas de nuestro lenguaje: las funciones indeterministas y las variables lógicas. Cada una de las ecuaciones de la parte derecha de la regla es una etapa del compilador.

En la primera etapa, el analizador, devuelve una estructura en árbol con el constructor *#prog* en la raiz y la estructura en árbol del programa en *Statement*. La variable auxiliar *Name* recoge un identificador que es el nombre del programa. La función que hay que definir en esta etapa es *parse(Tokens)*.

La segunda etapa, el generador de código, usa la función *encode(Statement,Table)* que codifica la estructura del programa *Statement* con auxilio de una tabla de símbolos *Table* que se realiza como un diccionario que es un árbol ordenado como en PROLOG [Sterling & Shapiro 86]. Este diccionario asocia variables y variables temporales (auxiliares) a direcciones de memoria de manera incompleta.

El código reubicable *Code* es transformado en código objeto *ObjectCode* mediante la función *assemble(Code,Table)* con ayuda de la tabla de direcciones *Table* que es en esta etapa precisamente

completada. Como final obtenemos una lista de instrucciones máquina y el tamaño del bloque (número de direcciones de memoria a parte de las que ocupa el programa objeto necesarias para su ejecución).

Previo a los comentarios de cada fase procedemos a escribir la lista de lexemas de los programas anteriores:

EJEMPLO *first*:

['program', 'first', ';', 'begin', 'write', '(', '2', '+', '3', ')', '', '5', 'end']*

EJEMPLO *second*:

['program', 'second', ';', 'begin', 'if', 'a', '<', 'b', 'then', 'min', ':=', 'a', 'else', 'min', ':=', 'b', 'end']

EJEMPLO *third*:

['program', 'third', ';', 'begin', 'sum', ':=', '0' , ';', 'count', ':=', '0' , ';', 'while', 'cont', '<' , '11', 'do', 'begin', 'sum', ':=', 'sum', '+', 'count', ';', 'count', ':=', 'count', '+', '1', 'end', ';', 'write', 'sum', 'end']

7.2.2 EL ANALIZADOR

En nuestro lenguaje el analizador puede ser escrito en forma sencilla utilizando reglas semejantes a las de una **gramática de cláusulas definidas** (DCG). Esto es una clara ventaja sobre otros lenguajes de programación, como PROLOG, donde la "traducción" de las reglas gramáticales a reglas del programa presenta ciertas complicaciones que dificultan la comprensión [Clocksin & Mellish 81]

[Clark & McCabe 84] [Sterling & Shapiro 86].

La primera regla del analizador es:

parse(Tokens) ⟶ *program(Tokens,[]).*

y dice que para analizar una lista de *Tokens* hay que analizar la lista diferencia *Tokens\[]* [Clark & Tarnlund 77].

La función *program* puede definirse:

program(Tokens,Rest) $\longrightarrow^{*}$ Program

y lo que hace es analizar una lista *diferencia* de lexemas *Tokens\Rest* dando como resultado un *Program*. Inicialmente, en la regla, el resto es [] porque buscamos precisamente analizar todos los *Tokens*.

La regla se escribe:

```
program( [ 'program', I, ';' | T], RT ) ⟶
    let ?identifier(I)
    in #prog ( #name(I), statement ( T, RT ) ).
```

y afirma que un programa se construye con el constructor de aridad 2 *#prog* cuyo primer argumento es *#name(I)* para *I* identificador del programa y el segundo argumento es la estructura del programa. Como ya dijimos supondremos, para simplificar el programa, que hay en el lenguaje fuente dos tipos de constantes en cantidad suficiente para nuestras necesidades: identificadores y términos.

Las siguientes reglas analizan las distintas estructuras que son posibles en un programa. La primera de ellas:

```
statement(['begin' | T],RT) ⟶
    #comp(statement(T,RT1),rest_statements(RT1,RT)).
```

es una declaración de que un programa fuente puede comenzar con *begin* y se compone de la primera sentencia y el resto de sentencias. Su análisis produce un árbol con el constructor de aridad 2 *#comp* en la

raiz del que cuelga en la rama izquierda la primera sentencia y en la rama derecha el resto. Como ambas o alguna de las sentencias pueden ser, a su vez, compuestas la estructura es recursiva.

Las sentencias en el lenguaje fuente están separadas por ";". Entonces el resto comienza con el lexema ';' y puede ir seguido de otro resto de sentencias o por el lexema *end* que señala cuando acaba una sucesión de sentencias. También *end* indica cuando acaba un programa en lenguaje fuente con lo que excluimos la posibilidad de que haya sentencias vacías:

```
rest_statements([';' | T],RT) ⟶
        #comp(statement(T,RT1),rest_statements(RT1,RT)).
rest_statements('end' | T],RT) ⟶
        let RT = T in #void.
```

El constructor de aridad 0 (constante) *#void* sirve para marcar en el programa analizado el fin de una sentencia compuesta. Posteriormente habrá que eliminarlo.

La segunda sentencia es la asignación:

```
statement([I,':=' | T],RT) ⟶
        let ?identifier(I)
        in #assign(#var(I),expression(T,RT)).
```

Su estructura es muy simple: una variable *I* seguida del signo ':=' y de una *expresion(T,RT)*. La estructura analizada consta de un constructor de aridad 2, *#assign*, que tiene como primer argumento un constructor de variables y como segundo argumento una *expresion(T,RT)*. El predicado *?identifier(I)* que no definimos en el programa se supone que devuelve *true* cuando una constante es un identificador posible.

El tratamiento de las expresiones tiene en cuenta las

prioridades de los operadores y se establece para una DCG de la forma siguiente [Aho & Ullmann 79]:

```
expression ::= expression + term
             | expression - term
             | term.
term ::= term * factor
       | term / factor
       | factor.
factor ::= constant
         | variable
         | (expression).
```

No consideramos la operación unitaria: negación de una expresión.

Las buenas cualidades de nuestro lenguaje se ponen de manifiesto en la traducción de esta gramática:

```
expression(T,RT) ⟶
     let  Exp = expression(T,['+' ¦ RT1]) and
          Term = term(RT1,RT)
     in   #exp('+',Exp,Term).
expression(T,RT) ⟶
     let  Exp = expression(T,['-' ¦ RT1]) and
          Term = term(RT1,RT)
     in   #exp('-',Exp,Term).
expression(T,RT) ⟶
     term(T,RT).
```

```
term(T,RT) ⟶
     let  Term = term(T,['*' ¦ RT1]) and
          Factor = factor(RT1,RT)
     in   #exp('*',Term,Factor).
term(T,RT) ⟶
     let  Term = term(T,['/' ¦ RT1]) and
          Factor = factor(RT1,RT)
     in   #exp('/',Term,Factor).
term(T,RT) ⟶
     factor(T,RT).
factor([X ¦ T],RT) ⟶
     let  ?number(X) and RT = T
     in   #cte(X).
factor([X ¦ T],RT) ⟶
     let  ?identifier(X) and RT = T
     in   #var(X).
factor(['(' ¦ T],RT) ⟶
     expression(T,[')' ¦ RT]).
```

Sin embargo, como se ve fácilmente estas reglas harían que el análisis de las expresiones en el compilador fuese muy indeterminista. Para evitarlo, hemos hecho en nuestro caso, un tratamiento más eficiente utilizando variables que van "acumulando" las expresiones o términos que se van formando a lo largo del cómputo. (Ver el programa al final del capítulo para apreciar las diferencias)

Otras dos estructuras posibles son las sentencias para *if_then_else* y *while_do* que tienen una fácil comprensión:

```
statement(['if' | T],RT) ⟶
    let  Test = test(T,['then' | RT1]) and
         ThenSt = statement(RT1,['else' | RT2]) and
         ElseSt = statement(RT2,RT)
    in   #if(Test,ThenSt,ElseSt).
statement(['while' | T],RT) ⟶
    let  Test = test(T,['do' | RT1]) and
         Body = statement(RT1,RT)
    in   #while(Test,Body).
```

La estructura analizada de *if_then_else* está formada por un constructor de aridad 3 siendo los argumentos: el *Test*, la declaración *ThenSt* y la declaración *ElseSt*. la estructura de *while_do* está formada por un constructor *#while* con dos argumentos: el *Test* y el *Body*. La novedad es que plantean una nueva función: *test*. Un "test" se realiza entre dos expresiones que son comparadas mediante una operación:

```
test(T,RT) ⟶
    let  Exp1 = expression(T,[Op | RT1]) and
         comp_op(Op) and
         Exp2 = expression(RT1,RT)
    in   #compare(Op,Exp1,Exp2).
```

Las posibles operaciones para realizar el mencionado "test" son las clásicas y son explícitamente escritas. (Ver el programa completo al final del capítulo)

Finalmente, tenemos sentencias de lectura y escritura de interpretación inmediata:

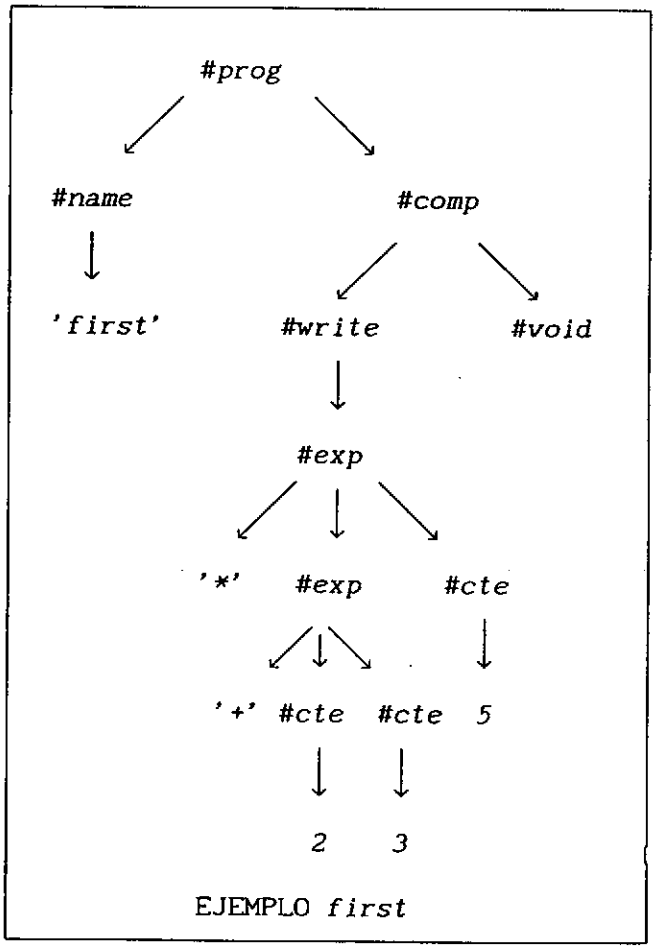
```
statement(['read',I | T],RT) ⟶
    let  ?identifier(I) and RT = T
```

```
        in   #read(#var(I)).

statement(['write' | T],RT) ⟶

        #write(expression(T,RT)).
```

Todo el conjunto anterior constituye el analizador y como muestra de su funcionamiento después de la primera etapa obtendríamos para los ejemplos en lenguaje fuente señalados al comienzo lo siguiente:

EJEMPLO *first*:

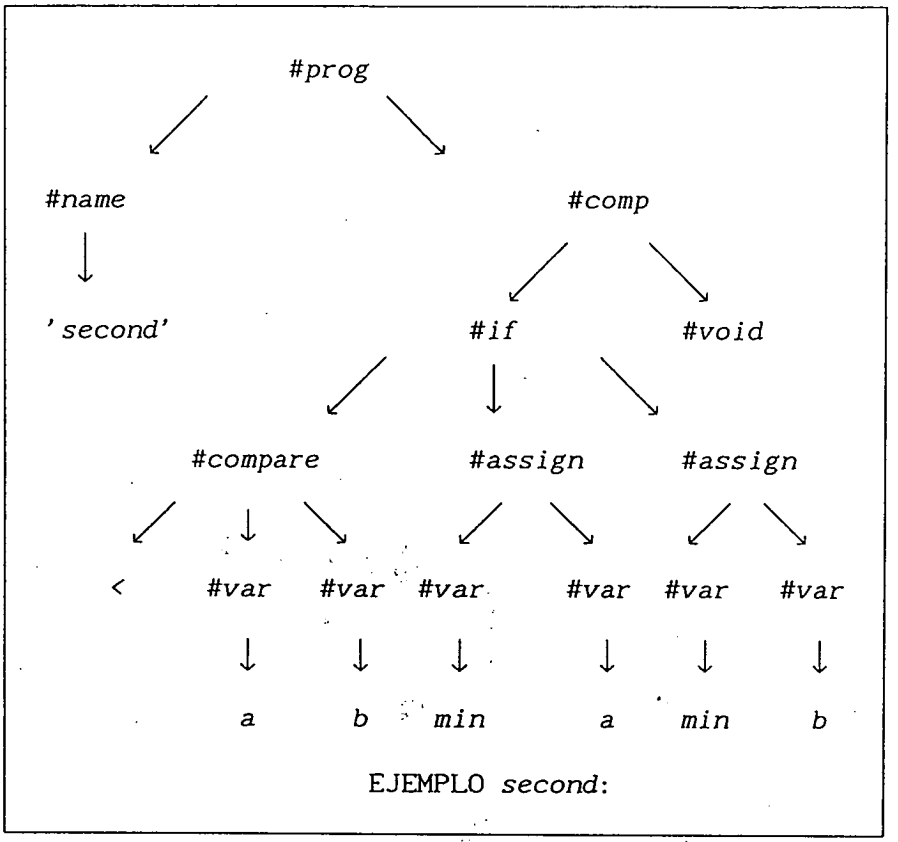
```
#prog ( #name ( 'first' ), #comp ( #write ( #exp ( '*', #exp ( '+',
#cte(2), #cte(3) ), #cte(5) ) ), #void ) ).
```



EJEMPLO *first*

EJEMPLO *second*:

#prog (#name ('second'), #comp (#if (#compare ('<', #var(a), #var(b)), #assign(#var(min), #var(a)), #assign (#var(min), #var(b))), #void)).

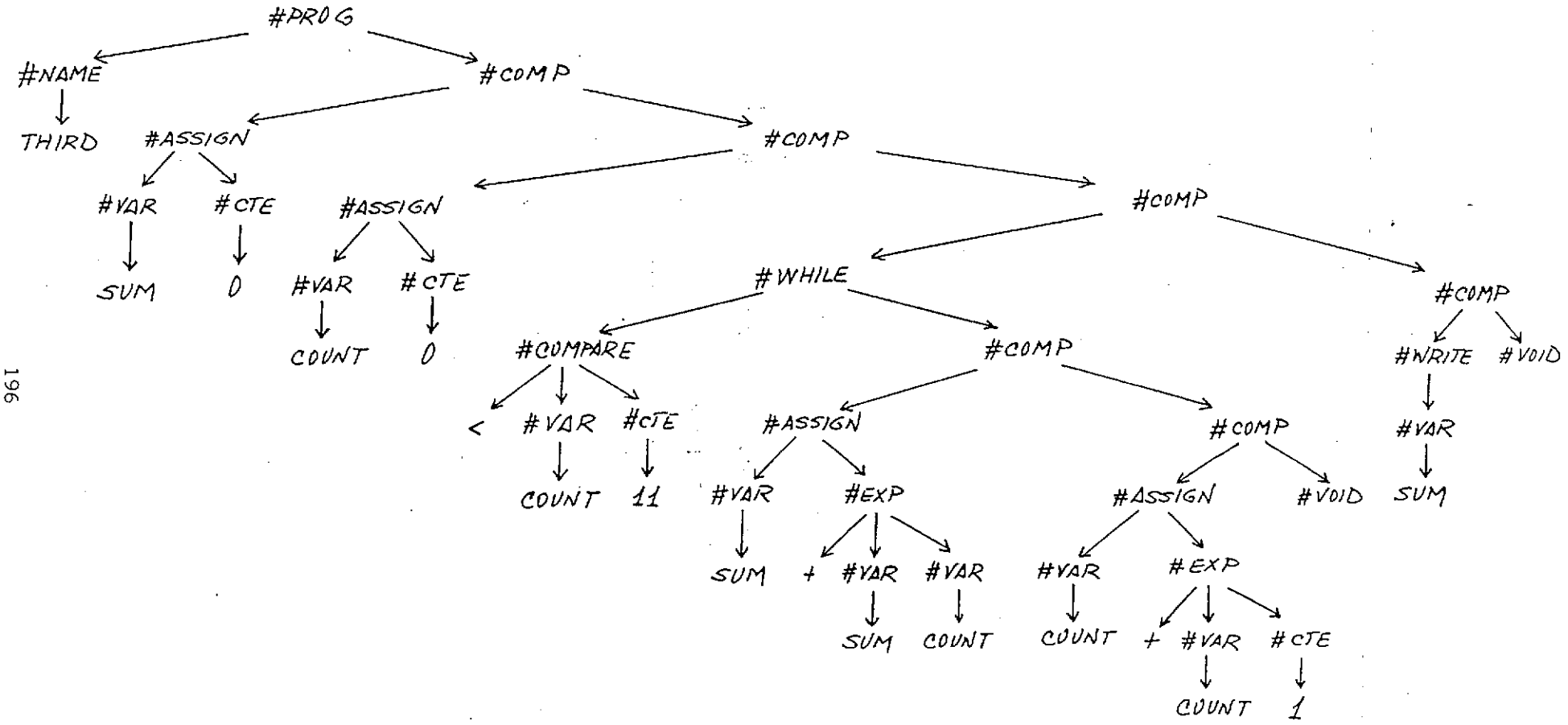


EJEMPLO *second*:

EJEMPLO *third*:

#prog (#name ('third'), #comp (#assign (#var(sum), #cte(0)), #comp (#assign (#var(count), #cte(0)), #comp (#while (#compare ('<', #var(count), #cte(11)), #comp (#assign (#var(sum), #exp ('+', #var(sum), #var(count))), #comp (#assign (#var(count), #exp ('+', #var(count), #cte(1))), #void))), #comp (#write (#var(sum)), #void))))).

Se ha desarrollado el árbol correspondiente en la página siguiente:



EJEMPLO : third

7.2.3 EL GENERADOR DE CODIGO REUBICABLE

Statement es la variable donde en la etapa anterior se ha recogido la estructura del programa que vamos a codificar.

encode(Statement, Table) es la función básica que va a generar código a partir de *Statement* usando *Table* como tabla auxiliar que guarda las direcciones de memoria que se adjudican a las variables y las direcciones de instrucciones que se adjudican a las etiquetas (*labels*). *Table* es una variable lógica, que representa un árbol ordenado, inicialmente vacío que se va formando mediante la función *lookup* y cuyo contenido se completa en la última etapa por medio del predicado *allocate*. (Ver el ejemplo 2 del capítulo 2)

La función *lookup* es conocida y ha sido usada para definir estructuras de datos incompletas en forma de listas de tuplas o en forma de árboles binarios [Sterling & Shapiro 86]. En nuestro caso se define como sigue:

```
lookup(#mkdic(Key,Value,Left,Right),K) ⟶
        if K = Key then Value.
lookup(#mkdic(Key,Value,Left,Right),K) ⟶
        if before(K,Key)
        then lookup(Left,K).
        else lookup(Right,K).
```

y como se ve, elige ramas a izquierda o derecha en el árbol según indica la función *before* según unas claves (*Key*) y deja unos valores (*Value*) pendientes que son otras variables lógicas que se llenarán de contenido en la última etapa (ensamblador).

Antes de definir *before* conviene decir que para el diseño del compilador hemos tenido en cuenta dos tipos de variables: ordinarias (

precedidas del constructor *#ordinary*) y temporales (precedidas del constructor *#temporary*) según que correspondan a variables del lenguaje fuente o variables auxiliares (temporales) introducidas por el compilador. Entre ellas hay un orden establecido: las variables ordinarias van antes que las temporales; las ordinarias tienen entre sí un orden lexicográfico definido previamente y las temporales el orden numérico clásico. En resumen:

before(#ordinary(X),#temporary(Y)) $\longrightarrow$ *true.*

before(#temporary(X),#ordinary(Y)) $\longrightarrow$ *false.*

before(#temporary(X),#temporary(Y)) $\longrightarrow$ $X < Y$.

before(#ordinary(X),#ordinary(Y)) $\longrightarrow$ $X \ll Y$.

Después de indicar como se forma la tabla *Table* para generar código vamos a ver como se genera éste para las distintas declaraciones que pueden ocurrir en el programa ya analizado.

El constructor *#comp* de la etapa anterior es en esta etapa sustituido, por razones de claridad expositiva, por el constructor *#seq*. Así la primera regla para codificar la composición de declaraciones es de lectura inmediata:

encode(#comp(ST1,ST2),Table) $\longrightarrow$

#seq(encode(ST1,Table),encode(ST2,Table)).

Es decir, la estructura correspondiente a una composición de estructuras con *#comp* es una composición (que por no confundir la etapa en que nos encontramos podíamos decir sucesión) de sus estructuras constituyentes con *#seq*.

La primera regla para codificar que consideramos es la de asignación:

```
encode(#assign(Var,Exp),Table) ⟶
    let Addr = lookup(Table,name_of(Var))
    in #seq(encode_exp(Exp,Table),#instr(store Addr)).
```

que se puede leer: Para generar código para una asignación de una expresión a una variable con ayuda de una tabla de direcciones *Table* se pone en sucesión el código generado por la expresión con auxilio de *Table* y una instrucción *store* cuyo argumento es la dirección de memoria correspondiente a la variable. Se necesita introducir la función *name_of* para asegurarnos que la variable usada es una variable ordinaria:

```
name_of(#var(I)) ⟶ #ordinary(I).
```

Además *lookup* ha abierto una entrada en el árbol ordenado de direcciones *Table* cuya clave es la variable ordinaria y el valor es, en el momento de utilizar la regla, todavía una variable no vinculada.

Para generar código para las expresiones necesitamos introducir una numeración que nos permita establecer un orden para las variables temporales que, recordemos, tenían la ordenación numérica mientras que las variables ordinarias tenían una ordenación lexicográfica. Este es el objetivo de la regla siguiente:

```
encode_exp(Exp,Table) ⟶
    encode_from(0,Exp,Table).
```

Ahora hay tres casos que considerar para *encode_from:*

```
encode_from(N,#var(I),Table) ⟶
    let Addr = lookup(Table,#ordinary(I))
    in #instr(load Addr).
encode_from(N,#cte(X),Table) ⟶
    #instr(loadc X).
```

```
encode_from(N,#exp(Op,E1,E2),Table) ⟶
    let TempAddr = lookup(Table,#temporary(N)) and
        Code2 = encode_from(N,E2,Table) and
        Code1 = encode_from(#succ(N),E1,Table) and
        OpCode = op1(Op,TempAddr)
    in sequence([Code2,
                 #instr(store TempAddr),
                 Code1,
                 OpCode]).
```

donde hemos definido *sequence* como sigue:

```
sequence ([X]) ⟶ X.
sequence([X,Y | Zs]) ⟶
    #seq(X,sequence([Y|Zs])).
```

Claramente, el caso importante para *encode_from* es el tercero. Para generar código para *#exp(Op,E1,E2)* se compone en secuencia sucesivamente: el código generado para la segunda expresión *Code2* que utiliza, si es necesario, una dirección temporal *N* para guardar valores intermedios; una instrucción *store* cuyo argumento es *TempAddr*, una dirección temporal introducida para realizar la operación demandada entre las dos expresiones *E1* y *E2*; el código generado para la primera expresión *Code1* que utiliza, si la necesita, la dirección temporal *N + 1*; y el código generado para la operación.

Las dos primeras reglas para *encode_from* son las dos maneras posibles de terminar la generación de código para una expresión: 1) generar código para una variable, en cuyo caso es necesario introducir una nueva clave y valor en el árbol *Table* y añadir una instrucción *load* con la dirección dada por la variable no vinculada del árbol; y 2)

generar código para una constante lo que se traduce simplemente en añadir una instrucción *loadc* con la constante como argumento.

Es posible hacer una optimización para la codificación de las expresiones considerando que una o ambas de ellas son variables y/o constantes; pero no nos detenemos en ella para no alargar el programa.

La codificación de *#read* y *#write* no presenta ninguna novedad:

```
encode(#read(Var),Table) ⟶
        let Addr = lookup(Table,name_of(Var))
        in #seq(#instr(readop),#instr(store Addr)).
encode(#write(Exp),Table) ⟶
        #seq(encode_exp(Exp,Table),#instr(writeop)).
```

por lo que pasamos a considerar la codificación del condicional *#if* :

```
encode(#if(Test,ThenSt,ElseSt),Table) ⟶
        let  TestCode = encode_test(Test,Table,ElseAddr) and
             ThenCode = encode(ThenSt,Table) and
             ElseCode = encode(ElseSt,Table)
        in   sequence([TestCode,
                       ThenCode,
                       #instr(jump EndAddr),
                       #label(ElseAddr),
                       ElseCode,
                       #label(EndAddr)]).
```

La idea es sencilla aunque el proceso es largo: se genera código para *TestCode*, *ThenCode* y *ElseCode* a los que se añade en los lugares adecuados una instrucción de salto incondicional al final de la

declaración del *#if* si se ha ejecutado *ThenCode* marcada por la etiqueta *#label(EndAddr)*. En *TestCode* debe de haber una instrucción de salto condicional para lo que es necesaria la etiqueta *#label(ElseAddr)*. *ElseAddr* y *EndAddr* son variables lógicas, que representan dos direcciones de instrucción nuevas y aún indefinidas, que elegimos de manera indeterminista.

La codificación del test presenta algunas cosas de interés:

```
encode_test(Test,Table,Ad) ⟶
    let  Code = encode_exp(test_exp(Test),Table) and
         JumpCode = test_jump(Test,Ad)
    in   #seq(Code,JumpCode).
test_exp(#compare(Op,E1,E2)) ⟶
    #exp('-',E1,E2).
test_jump(#compare('=',E1,E2),Ad) ⟶
    instr(jumpne Ad).
    ............
```

Lo primero hay que tranformar el test implícito en *#compare(Op,E1,E2)* como sigue: Si se compara mediante *Op*, la expresión *E1* con la *E2* vamos a transformarla de modo que se compare también mediante *Op*, la expresión *(E1 - E2)* con *0*. Esto es lo que realiza precisamente la función *test_exp*. Entonces, el problema de codificar un test se reduce a generar código para una expresión que nos permita evaluar la expresión *(E1 - E2)* y añadir en secuencia un test de salto condicional según que *(E1 - E2)* no sea igual a *0*, no sea distinto de *0*, etc... Observar que además obtenemos la dirección de la etiqueta *ElseAddr* (La definición completa para *test_jump* puede verse al final del capítulo).

La codificación de *#while* no presenta ninguna novedad sobre

el caso *#if* tratado anteriormente:

encode(#while(Test,Body),Table) ⟶

```
let  TestCode = encode_test(Test,Table,EndAddr) and
     BodyCode = encode(Body,Table)
in   sequence([#label(LoopAddr),
               TestCode,
               BodyCode,
               #instr(jump LoopAddr),
               #label(EndAddr)]).
```

Finalmente *#void* se codifica intoduciendo una instrucción "fantasma" *#no_op* que debe desaparecer en la tercera etapa:

encode(#void,Table) ⟶ *#no_op.*

Después de esta segunda etapa los tres ejemplos citados quedarían:

EJEMPLO *first*:

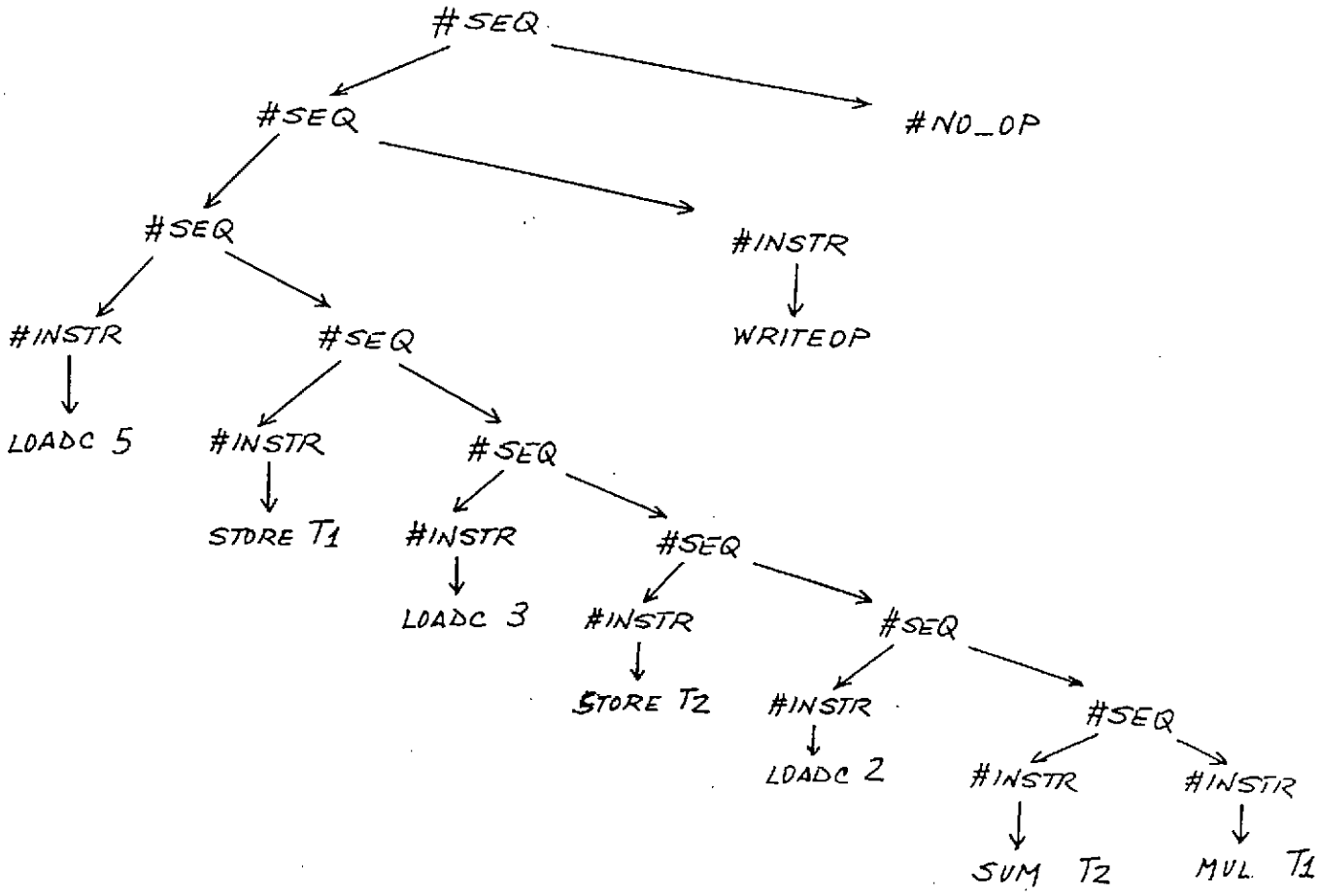
```
#seq ( #seq ( #seq ( #instr (loadc 5), #seq ( #instr (store T1), #seq
( #instr (loadc 3), #seq ( #instr (store T2), #seq ( #instr (loadc 2),
#seq ( #instr (sum T2), #instr (mul T1) ) ) ) ) ) ), #instr ( writeop
) ), #no_op )
```

EJEMPLO *second*:

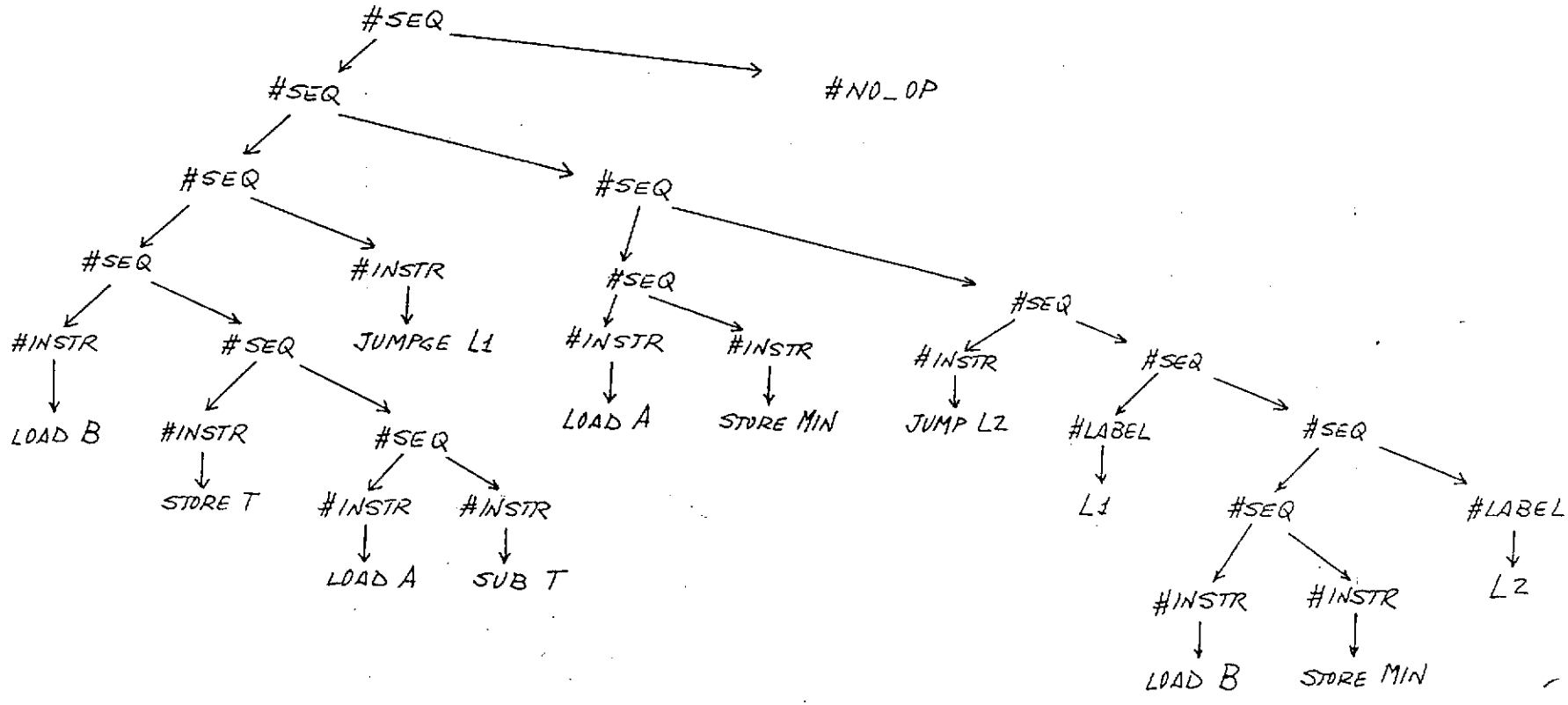
```
#seq ( #seq ( #seq ( #seq ( #instr (load B), #seq ( #instr (store T),
#seq ( #instr (load A), #instr (sub T) ) ) ), #instr (jumpge L1) ),
#seq ( #seq ( #instr (load A), #instr (store MIN) ), #seq ( #instr
(jump L2), #seq ( #label (L1), #seq ( #seq ( #instr (load B), #instr
(store MIN) ), #label (L2) ) ) ) ) ), #no_op )
```

EJEMPLO *third*:

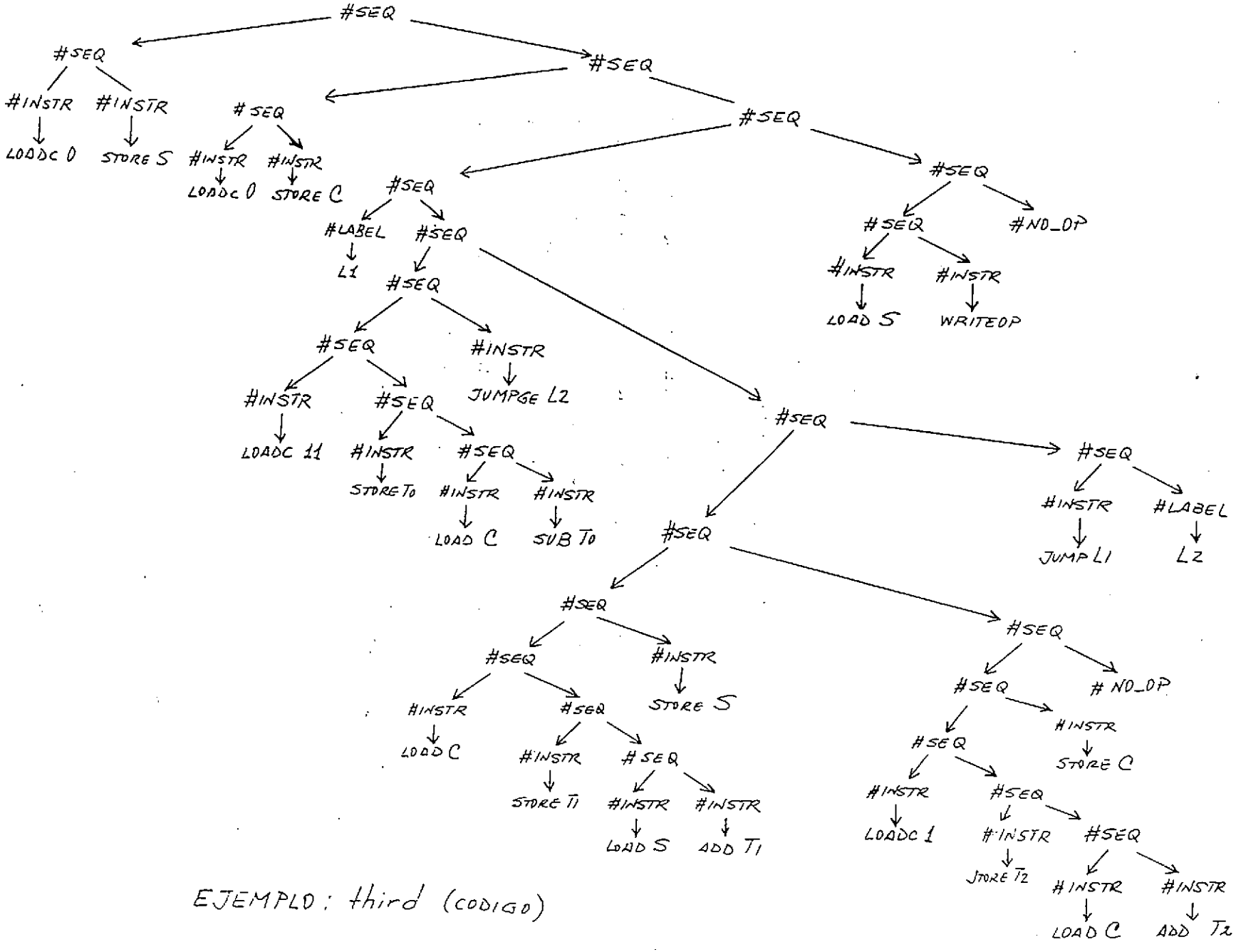
```
#seq ( #seq ( #instr (loadc 0), #instr (store S) ), #seq ( #seq (
#instr (loadc 0), #instr (store C) ), #seq ( #seq ( #label (L1), #seq
( #seq ( #seq ( #instr (loadc 11), #seq ( #instr (store T0), #seq (
#instr (load C), #instr (sub T0) ) ) ), #instr (jumpge L2) ), #seq (
#seq ( #seq ( #seq ( #instr (load C), #seq ( #instr (store T1), #seq (
#instr (load S), #instr (add T1) ) ) ), #instr (store S) ), #seq (
#seq ( #seq ( #instr (loadc 1), #seq ( #instr ( store T2), #seq (
#instr (load C), #instr (add T2) ) ) ), #instr (store C) ), #no_op )
), #seq ( #instr (jump L1), #label (L2) ) ) ) ), #seq ( #seq ( #instr
(load S), #instr (writeop) ), #no_op ) ) ) )
```



EJEMPLO: first (CODIGO)



EJEMPLO : second (CODIGO)



EJEMPLO: third (CODIGO)

7.2.4 ENSAMBLADOR

En esta etapa partimos del código *Code* obtenido en la etapa anterior y de una tabla de direcciones *Table* aún incompleta: tenemos ya instanciada la clave (*Key*) pero no el valor (*Value*) de cada nodo del árbol. Al final de ella queremos obtener **código objeto absoluto** en la lista *InstrList*, y el **tamaño del bloque** (mínimo número de direcciones de memoria que es necesario reservar) que viene dado en la regla siguiente por la variable *BlockSize*

assemble(Code,Table) ⟶

let *#mkpair(InstrList,CodeSize)* =

flatten_and_count(#seq(Code,#instr(halt)),succ(0)) *and*

allocate(Table,CodeSize,EndBlock) *and*

BlockSize = EndBlock - CodeSize

in *#objcode(InstrList,BlockSize).*

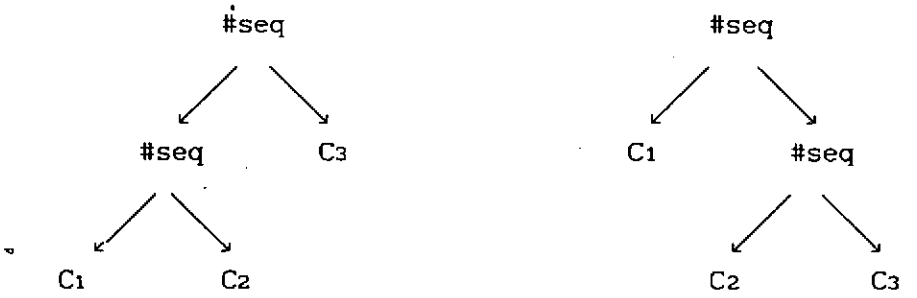
La idea es construir un par ordenado donde se va formando por una parte *InstrList*, la lista de instrucciones a la que se añade para acabar la instrucción *halt* y por otra parte en *CodeSize* se van acumulando a partir de 1 el número de instrucciones de programa. La función que hace esto es *flatten_and_count*. A continuación, se llena de contenido, por medio del operador *allocate*, el árbol ordenado *Table* tomando como primera dirección libre *CodeSize* y como última *EndBlock*. Entonces, al terminar de llenar *Table* se puede determinar precisamente la variable que buscabamos *BlockSize = EndBlock - CodeSize*.

La función importante en esta parte es *flatten_and_count*. Su comportamiento es bastante sencillo de describir. La primera regla:

flatten_and_count(#seq(#seq(C1,C2),C3),N) ⟶

flatten_and_count(#seq(C1,#seq(C2,C3)),N).

simplemente transforma el árbol de la izquierda en el de la derecha:



El problema se reduce entonces a estudiar árboles como el de la derecha. La segunda, tercera y cuarta regla:

```
flatten_and_count(#seq(#instr(I),C),N) ⟶
        let  flatten_and_count(C,#succ(N)) = #mkpair(Is,S)
        in   #mkpair([I | Is],S).
flatten_and_count(#seq(#label(M),C),N) ⟶
        let  M = N
        in   flatten_and_count(C,N).
flatten_and_count(#seq(#no_op,C),N) ⟶
        flatten_and_count(C,N).
```

corresponden a las tres posibilidades de encontrar como hoja más a la izquierda una instrucción, una etiqueta o el identificador *#no_op*. En el primer caso, se coloca la instrucción en la cabeza de la lista de instrucciones que vamos formando y se aumenta en una unidad el valor de *CodeSize*. En el segundo caso, se limita a identificar el valor de la etiqueta con el valor de *CodeSize* consiguiendo de este modo señalar las direcciones de las instrucciones de salto del programa objeto. En el tercer caso, simplemente se ignora la instrucción y se continua generando código objeto absoluto.

La quinta, sexta y última regla corresponden a las tres posibles hojas en las ramas derechas del árbol correspondiente al código de

instrucciones:

flatten_and_count(#instr(I),N) ⟶

mkpair([I],#succ(N)).

flatten_and_count(#label(M),N) ⟶

let M = N

in mkpair([],N).

flatten_and_count(#no_op,N) ⟶

#mkpair([],N).

Y lo que hacemos es evidente.

El último operador (que es un predicado) de esta etapa que consideramos es *allocate*. Sus reglas son:

allocate(#empty,M,N) ⟶ *M = N.*

allocate(#mkdic(Key,Value,Left,Right),M,N) ⟶

allocate(Left,M,Value) and

allocate(Right,#succ(Value),N).

Este operador es el que llena de contenido las variables lógicas donde se reservaban las direcciones de memoria del árbol ordenado *Table* que hemos formado en la etapa anterior. Así, *allocate*, se aplica en:

allocate(Table,CodeSize,EndBlock)

tomando como primera dirección libre *CodeSize* y va llenando el árbol ordenado, recorriendolo una sola vez, comenzando por las variables ordinarias en orden lexicográfico y continuando con las temporales en orden numérico. Cuando termina el recorrido conoce el valor de *EndBlock* y puede determinar el tamaño del bloque *BlockSize = EndBlock - CodeSize*.

Después de esta etapa *el código objeto* para nuestros tres ejemplos sería:

EJEMPLO *first*:

#objcode ([(loadc 5), (store 10), (loadc 3), (store 11), (loadc 2), (sum 11), (mul 10), (writeop), (halt)], 2)

EJEMPLO *second*:

#objcode ([(load 13), (store 15), (load 12), (sub 15), (jumpge 9), (load 12), (store 14)), (jump 11), (load 13), (store 14)), (halt)], 4)

EJEMPLO *third*:

#objcode([(loadc 0), (store 25), (loadc 0), (store 24), (loadc 11), (store 26), (load 24), (subl 26), (jumpge 21), (load 24), (store 27), (load 25), (add 27), (store 25) (loadc 1), (store 28), (load 24), (add 28), (store 24), (jump 5), (load 25), (writeop)), (halt)], 5)

Finalmente, en la página siguiente, presentamos agrupados todos los operadores correspondientes al programa:

COMPILADOR

```
compile(Tokens) ⟶
     let  #prog(Name,Statement) = parse(Tokens) and
          Code = encode(Statement,Table) and
          ObjectCode = assemble(Code,Table)
     in   ObjectCode.
```

ANALIZADOR

```
parse(Tokens) ⟶
     program(Tokens,[]).

program(['program',I,';' ¦ T],RT) ⟶
     let  ?identifier(I)
     in   #prog(#name(I),statement(T,RT)).

statement(['begin' ¦ T],RT) ⟶
     #comp(statement(T,RT1),rest_statements(RT1,RT)).
statement([I,':=' ¦ T],RT) ⟶
     let  ?identifier(I)
     in   #assign(#var(I),expression(T,RT)).
statement(['if' ¦ T],RT) ⟶
     let  Test = test(T,['then' ¦ RT1]) and
          ThenSt = statement(RT1,['else' ¦ RT2]) and
          ElseSt = statement(RT2,RT)
     in   #if(Test,ThenSt,ElseSt).
```

```
statement(['while' ! T],RT) ⟶
     let  Test = test(T,['do' ! RT1]) and
          Body = statement(RT1,Rt)
     in   #while(Test,Body).
statement(['read',I ! T],RT) ⟶
     let  ?identifier(I) and RT = T
     in   #read(#var(I)).
statement(['write' ! T],RT) ⟶
     #write(expression(T,RT)).


rest_statements([';' ! T],RT) ⟶
     #comp(statement(T,RT1),rest_statements(RT1,RT)).
rest_statements('end' ! T],RT) ⟶
     let  RT = T
     in   #void.


expression(T,RT) ⟶
     let  Term = term(T,RT1)
     in   complete_exp(Term,RT1,RT).


complete_exp(Exp,['+' ! T],RT) ⟶
     let  Term = term(T,RT1)
     in   complete_exp(#exp('+',Exp,Term),RT1,RT).
complete_exp(Exp,['-' ! T],RT) ⟶
     let  Term = term(T,RT1)
     in   complete_exp(#exp('-',Exp,Term),RT1,RT).
```

```
complete_exp(Exp,T,RT) ⟶
     let  ?no_term(T) and RT = T
     in  Exp.

term(T,RT) ⟶
     let  Factor = factor(T,RT1)
     in   complete_term(Factor,RT1,RT).

complete_term(Term,['*' ¦ T],RT) ⟶
     let  Factor = factor(T,RT1)
     in   complete_term(#exp('*',Term,Factor),RT1,RT).
complete_term(Term,['/' ¦ T],RT) ⟶
     let  Factor = factor(T,RT1)
     in   complete_term(#exp('/',Term,Factor),RT1,RT).
complete_term(Term,T,RT) ⟶
     let  ?no_factor(T) and RT = T
     in   Term.

factor([X ¦ R],T) ⟶
     let  ?number(X) and R = T
     in   #cte(X).
factor([X ¦ R],T) ⟶
     let  ?identifier(X) and R = T
     in   #var(X).
factor(['(' ¦ T],RT) ⟶
     expression(T,[')' ¦ RT]).
```

```
?no_term([]) ⟶ true.

?no_term([X ¦ RT] ⟶

     if X = '+ ' or X = '-'

     then false

     else true.


?no_factor([]) ⟶ true.

?no_factor([X ¦ RT] ⟶

     if X = '* ' or X = '/'

     then false

     else true.


test(T,RT) ⟶

     let  Exp1 = expression(T,[Op ¦ RT1]) and

          comp_op(Op) and

          Exp2 = expression(RT1,RT)

     in   #compare(Op,Exp1,Exp2).


comp_op('=') ⟶ true.

comp_op('≠') ⟶ true.

comp_op('>') ⟶ true.

comp_op('<') ⟶ true.

comp_op('≥') ⟶ true.

comp_op('≤') ⟶ true.
```

```
encode(#comp(ST1,ST2),Table) ⟶
     #seq(encode(ST1,Table),encode(ST2,Table)).
encode(#assign(Var,Exp),Table) ⟶
     let  Addr = lookup(Table,name_of(Var))
     in   #seq(encode_exp(Exp,Table),#instr(store Addr)).
encode(#read(Var),Table) ⟶
     let  Addr = lookup(Table,name_of(Var))
     in   #seq(#instr(readop),#instr(store Addr)).
encode(#write(Exp),Table) ⟶
     #seq(encode_exp(Exp,Table),#instr(writeop)).
encode(#if(Test,ThenSt,ElseSt),Table) ⟶
     let  TestCode = encode_test(Test,Table,ElseAddr) and
          ThenCode = encode(ThenSt,Table) and
          ElseCode = encode(ElseSt,Table)
     in   sequence([TestCode,
                    ThenCode,
                    #instr(jump EndAddr),
                    #label(ElseAddr),
                    ElseCode,
                    #label(EndAddr)]).
encode(#while(Test,Body),Table) ⟶
     let  TestCode = encode_test(Test,Table,EndAddr) and
          BodyCode = encode(Body,Table) and
     in   sequence([#label(LoopAddr),
                    TestCode,
```

```
                    BodyCode,
                    #instr(jump LoopAddr),
                    #label(EndAddr)]).
encode(#void,Table) ⟶
    #no_op.

encode_exp(Exp,Table) ⟶
    encode_from(0,Exp,Table).

encode_from(N,#var(I),Table) ⟶
    let  Addr = lookup(Table,#ordinary(I))
    in   #instr(load Addr).
encode_from(N,#cte(X),Table) ⟶
    #instr(loadc X).
encode_from(N,#exp(Op,E1,E2),Table) ⟶
    let  TempAddr = lookup(Table,#temporary(N)) and
         Code2 = encode_from(N,E2,Table) and
         Code1 = encode_from(#succ(N),E1,Table) and
         OpCode = opl(Op,TempAddr)
    in   sequence([Code2,
                    #instr(store TempAddr),
                    Code1,
                    OpCode]).

opl('+',A) ⟶ #instr(add A).
opl('-',A) ⟶ #instr(sub A).
opl('*',A) ⟶ #instr(mul A).
```

```
opl('/',A) ⟶ #instr(div A).

encode_test(Test,Table,Ad) ⟶
     let  Code = encode_exp(test_exp(Test),Table) and
          JumpCode = test_jump(Test,Ad)
     in   #seq(Code,JumpCode).

test_exp(#compare(Op,E1,E2)) ⟶
     #exp('-',E1,E2).

test_jump(#compare('=',E1,E2),Ad) ⟶
     instr(jumpne Ad).
test_jump(#compare('≠',E1,E2),Ad) ⟶
     instr(jumpeq Ad).
test_jump(#compare('>',E1,E2),Ad) ⟶
     instr(jumple Ad).
test_jump(#compare('<',E1,E2),Ad) ⟶
     instr(jumpge Ad).
test_jump(#compare('≥',E1,E2),Ad) ⟶
     instr(jumplt Ad).
test_jump(#compare('≤',E1,E2),Ad) ⟶
     instr(jumpgt Ad).

sequence([X]) ⟶ X.
sequence([X,Y : Zs]) ⟶
     #seq(X,sequence([Y : Zs])).
```

```
name_of(#var(I)) ⟶ #ordinary(I).


lookup(#mkdic(Key,Value,Left,Right),K) ⟶
     if K = Key
     then Value.
lookup(#mkdic(Key,Value,Left,Right),K) ⟶
     if before(K,Key)
     then lookup(Left,K).
     else lookup(Right,K).


before(#ordinary(X),#temporary(Y)) ⟶ true.
before(#temporary(X),#ordinary(Y)) ⟶ false.
before(#temporary(X),#temporary(Y))⟶ X < Y.
before(#ordinary(X),#ordinary(Y)) ⟶ X « Y.
```

ENSAMBLADOR

```
assemble(Code,Table) ⟶
     let  #mkpair(InstrList,CodeSize) =
          flatten_and_count(#seq(Code,#instr(halt)),succ(0)) and
          allocate(Table,CodeSize,EndBlock) and
          BlockSize = EndBlock - CodeSize
     in   #objcode(InstrList,BlockSize).


flatten_and_count(#seq(#seq(C1,C2),C3),N) ⟶
     flatten_and_count(#seq(C1,#seq(C2,C3)),N).
```

```
flatten_and_count(#seq(#instr(I),C),N) ⟶
     let  flatten_and_count(C,#succ(N)) = #mkpair(Is,S)
     in   #mkpair([I : Is],S).
flatten_and_count(#seq(#label(M),C),N) ⟶
     let  M = N
     in   flatten_and_count(C,N).
flatten_and_count(#seq(#no_op,C),N) ⟶
     flatten_and_count(C,N).
flatten_and_count(#instr(I),N) ⟶
     mkpair([I],#succ(N)).
flatten_and_count(#label(M),N) ⟶
     let  M = N
     in   mkpair([],N).
flatten_and_count(#no_op,N) ⟶
     #mkpair([],N).

allocate(#empty,M,N) ⟶ M = N.
allocate(#mkdic(Key,Value,Left,Right),M,N) ⟶
     allocate(Left,M,Value) and
     allocate(Right,#succ(Value),N).
```

CONCLUSIONES

Hemos estudiado en este trabajo un lenguaje que integra la programación lógica y funcional en el siguiente sentido:

1) Permite la coexistencia de funciones y predicados (tratados como funciones booleanas).

2) Permite la existencia de funciones indeterministas que aumentan la potencia y expresividad del lenguaje.

3) Permite la existencia de constructores, cualidad apreciada en lenguajes funcionales y en PROLOG.

4) Permite la existencia de variables libres. Con lo que podemos simular el comportamiento de las variables lógicas y las cualidades más poderosas de la programación lógica.

5) Usa estrechamiento como mecanismo de cómputo que permite simular al mismo tiempo reescritura y SLD-resolución.

Etc ...

Los programas en este lenguaje son llamados sistemas de reescritura regulares. La semántica declarativa ha sido estudiada en el marco de los dominios de Scott. Al no haber funciones en sentido matemático hemos desarrollado el estudio de los grafos correspondientes a las funciones indeterministas. A través de este estudio se ha conseguido construir un dominio de funciones indeterministas que contiene como subdominio al de las funciones continuas (deterministas). Estos resultados se han comparado con los

dominios potencia, concretamente el de Hoare.

La semántica operacional usada ha sido estrechamiento en grafos dirigidos acíclicos. La necesidad de utilizarla viene de la incorrección que, en algunos casos, puede originar el estrechamiento clásico no "innermost". Hemos probado que esta semántica operacional es correcta y completa respecto a la semántica declarativa anteriormente definida.

El lenguaje, aún no implementado y reducido a un núcleo básico, permite sin embargo aplicaciones importantes. Como muestra hemos desarrollado un compilador para un lenguaje ideal que es un subconjunto de PASCAL.

Posibles lineas en que se puede ampliar nuestro trabajo son las siguientes:

1) El lenguaje permite añadirle todas las buenas cualidades de los lenguajes funcionales: tipos, orden superior, computación perezosa, ...

2) El estudio de la igualdad como unificación semántica de la que se habla en el capítulo 7 es otra posible linea de trabajo. Por este camino sería necesaria una ampliación (y una nueva definición) de la semánticas operacional y declarativa y la necesidad de una nueva demostración de completitud.

3) Una posible implementación del lenguaje.

Otra linea de actuación es buscar un nuevo enfoque para salvar los problemas que plantea la incorrección del estrechamiento y estudiar los sistemas de reescritura regulares con una formulación de la semántica operacional más fácilmente manejable. Algunos esfuerzos

se están realizando precisamente por este camino, reemplazando los grafos dirigidos acíclicos por construcciones semánticas mejor estructuradas.

REFERENCIAS

[Abramson 86] E. ABRAMSON. A Prological Definition of HASL: A Purely Functional Language with Unification Based Conditional Binding Expressions. In: [DeGroot & Lindstrom 86]. pp. 73-129.

[Aho & Ullman 79] A. V. AHO and J. D. ULLMAN. Principles of Compiler Design. Addison-Wesley. 1979.

[Andreka & Nemeti 78] H. ANDREKA and I. NEMETI. The Generalizad Completeness of Horn Predicate Logic as a Programming Language. Acta Cybernetica 4 1978 pp. 3-10.

[Apt 90] K. R. APT. Logic Programming. In Jan van Leuwen (ed.): Handbook of Theoretical Computer Science. Vol. B. Elsevier North Holland. 1990. pp. 495-574.

[Apt & Van Emden 82] K. R. APT and M. H. VAN EMDEN. Contributions to the Theory of Logic Programming. J.A.C.M. Vol 29 No.3 july 1982. pp. 841-862.

[Backus 78] J. BACKUS. Can Programming Be Liberated from the von Neumann Style ?. A Functional Style and its Algebra of Programs. Commun. A.C.M. Vol. 21. 1978. pp. 613-641.

[Barbuti & al. 86] R. BARBUTI, M. BELLIA and G. LEVI. LEAF: A Language which Integrates Logic, Equations and Functions. In: [DeGroot & Lindstrom 86] pp. 201-238.

[Barendregt 84] H. P. BARENDREGT. The Lambda Calculus: Its Syntax and Semantics. Revised edition. North Holland. 1984.

[Barendreght & al. 87] H. P. BARENDREGT, M. C. J. D. VAN EEKELEN, J. R. W. GLAUERT, J. R. KENNAWAY, M. J. PLASMEIJER and M. R. SLEEP. Term Graph Rewriting. Technical Rapport. Univ. of Nijmegen. Holanda. 1987.

[Bellia & Levi 86] M. BELLIA & G. LEVI. The Relation Between Logic and Functional Languages: A Survey. Journal of Logic Programming no. 3 pp. 217-236. 1986.

[Bermudez 85] J. BERMUDEZ. Una Exposición de los Fundamentos Matemáticos de la Semántica Denotacional. Tesina. Facultad de Ciencias Matemáticas U.C.M. Madrid. 1985.

[Blair 82] H. A. BLAIR. The Recursion-Theoretic complexity of predicate logic as a programming language. Information and Control Vol. 54 No. 1-2. 1982. pp 25-47.

[Bowen & al. 85] D. L. BOWEN, L. BYRD, D. FERGUSON and W. KORNFELD. Quintus Prolog Reference Manual. Quintus Computer Systems. 1985.

[Broy & Wirsing 81] M. BROY and M. WIRSING. On the Algebraic Specification of Nondeterministic Programming Languages. In: E Astesiano & C. Böhm (eds.) 6th. Coll. on Trees in Algebra and Programming. LNCS 112. Springer Verlag. 1981. pp. 162-179.

[Burstall & al. 80] R. M. BURSTALL, D. B. MACQUEEN and D. T. SANNELLA. HOPE: An Experimental Applicative Language. In: The 1980 LISP Conference. Stanford Univ. The USP Co. pp. 136-143.

[Cardelli 83] L. CARDELLI. The Functional Abstract Machine. Bell Labs. Technical Report TR-107. 1983.

[Church 41]. A. CHURCH. The Calculi of Lambda Conversion. Princeton University Press. 1941.

[Clark 77] K. L. CLARK. Negation as failure. In: Logic and Data bases. H. Gallaire, J. Minker (eds.). Plenum Press 1977.

[Clark 79] K. L. CLARK. Predicate Logic as a computacional formalism. Research Report DOC 79/59. Department of Computing. Imperial College of London. 1979.

[Clark & Gregory 86] K. L. CLARK and S. GREGORY. PARLOG: A Parallel

Logic Programming Language. ACM Trans. on Programming Languages and Systems 8, 1 (Jan 1986) pp. 1-49.

[Clark & McCabe 84] K. L. CLARK and F. G. McCABE. Micro-PROLOG: Programming in Logic. Prentice-Hall. 1984

[Clark & Tärnlund 77] K. L. CLARK and S. A. TARNLUND. A First Order Theory Of Data and Programs. In: Proceedings IFIP'77. North-Holland 1977. pp. 939-944.

[Clocksin & Mellish 87] W. F. CLOCKSIN and C. S. MELLISH. Programming in PROLOG. Springer-Verlag. 2a. ed. 1987

[Cohen 86] S. COHEN. The APPLOG Language. In: [DeGroot & Lindstrom 86] pp. 239-276.

[Colmerauer & al. 73] A. COLMERAUER, H. KANOUI, P. ROUSSEL and R. PASERO: Un Systeme de Communication Homme-Machine en Francais. Groupe de Recherche en Intelligence Artificielle. Université de Aix in Marseille. 1973.

[Corbin & Bidoit 83] J. CORBIN and M. BIDOIT. A Rehabilitation of Robinson's Unification Algorithm. R. E. A. Mason (ed.) Elsevier North Holland. 1983.

[Curry & Feys 58] H. B. CURRY and R. FEYS. Combinatory Logic. Vol. 1. North Holland. 1958.

[Darlington & al 86] J. DARLINGTON, A. J. FIELD and H. PULL. The Unification of Functional and Logic Languages. In: [DeGroot & Lindstrom 86]. pp. 37-70.

[Darlington & Guo 89] J. DARLINGTON and Y. GUO. Narrowing and Unification in Functional Programming. An Evaluation Mechanism for Absolute Set Abstraction. Technical Report. Working Draft. Nov. 1988.

[DeGroot & Lindstrom 86] D. DEGROOT and G. LINDSTROM (eds.) Logic Programming: Functions, Relations and Equations. Prentice-Hall. 1986.

[Dershowitz & Jouannaud 90] N. DERSHOWITZ and J. P. JOUANNAUD. Rewrite Systems. In Jan van Leuwen (ed.): Formal models and semantics, Handbook of Theoretical Computer Science, Vol. B. Elsevier North Holland. 1990. Chapter 6 pp. 243-320.

[Dershowitz & Manna 79] N. DERSHOWITZ and Z. MANNA. Proving termination with multiset orderings. Comm. of the A.C.M. 22 (8). 1979. pp. 465-476.

[Dershowitz & Okada 90] N. DERSHOWITZ and M. OKADA. A Rationale for Conditional Equational Programming. Theoretical Computer Science 75. 1990. pp. 111-138.

[Dershowitz & Plaisted 85] N. DERSHOWITZ and D. A. PLAISTED. Logic Programming cum Applicative Programming. In: Proceedings of the IEEE International Symposium on Logic Programming. July 1985. IEEE Computer Society Press. pp. 54-66.

[Fay 79] M. FAY. First-Order Unification in a Equational Theory. In: Proceedings of the 4th. Workshop on Automated Deduction. Austin (Texas) 1979. pp. 161-167.

[Farmer & Watro 91] W. M. FARMER and R. J. WATRO. Redex Capturing in Term Graph Rewriting. In: R. V. Book (ed.) Proceedings of RTA'91. LNCS 488. Springer Verlag. 1991. pp. 1-12.

[Field & Harrison 88] A. J. FIELD and P. G. HARRISON. Functional Programming. Addison-Wesley. 1988.

[Fribourg 85] L. FRIBOURG. SLOG: A Logic Programming Language Interpreter based on Clausal Superposition and Rewriting. Symposium on Logic Programming. IEEE Computer Society Press 1985. pp. 172-184.

[Goguen & al. 77] J. A. GOGUEN, J. W. THATCHER, E. G. WAGNER and J. B. WRIGHT. Initial Algebra Semantics and Continuous Algebras. J.A.C.M. Vol. 24 No. 1 January 1977. pp. 68-95.

[Goguen & Meseguer 86] J. A. GOGUEN and J. MESEGUER. EQLOG: Equality, Types, Modules and Generics for Logic Programming. In: [DeGroot & Lindstrom 86]. pp. 295-363.

[Goguen & Tardo 79] J. A. GOGUEN and J. J. TARDO. An Introduction to OBJ: A Language for Writing and Testing Formal Algebraic Program Specifications. Specifications of Reliable Software. IEEE 1979. pp. 170-189.

[Goguen & Thatcher & Wagner 78] J. A. GOGUEN, J. W. THATCHER, E. G. WAGNER. An Initial Algebra Approach to the Specification, Correctness, and Implementation of Abstract Data Types. In: Current Trends in Programming Methodology. Vol. 4 Ed. Yeh R. Prentice-Hall 1978. pp. 80-149.

[Gordon & al. 79] M. J. GORDON, R. MILNER and C. P. WADSWORTH. Edinburgh LCF. Springer-Verlag LNCS 78. 1979.

[Heckmann 90] R. HECKMANN. Set Domains. In: N. Jones (ed.). Proceedings E.S.O.P.'90. LNCS. Springer Verlag. 1990.

[Hill 74] R. HILL. Lush-Resolution and its completeness. DCL Memo 78. Department of Artificial Intelligence. University of Edimburg. 1974.

[Hoffmann & O'Donnell 82] C. M. HOFFMANN and M. J. O'DONNELL. Programming with Equations. A.C.M. Vol 4 No 1. 1982.

[Holm 90] K. H. HOLM. Graph Matching in Operational Semantics and Typing. In: A. Arnold (ed.). Proceedings CAAP'90. LNCS 431. Springer Verlag. 1990. pp. 191-205.

[Hudak 89] P. HUDAK. Conception, Evolution, and Application of Functional Programming Languages. A.C.M. Computing Surveys Vol. 21 No. 3. September 1989 pp. 359-411.

[Huet 77] G. HUET. Confluent Reductions: Abstract Properties and Applications to Term Rewriting Systems 18th IEEE Symposium on

Foundations of Computer Science (1977). pp 30-45.

[Huet & Levy 79] G. HUET and J. J. LEVY. Computations in Nonambiguous Linear Rewriting Systems. Technical Report 359. I.N.R.I.A. 1979.

[Huet & Oppen 80] G. HUET and D. C. OPPEN. Equations and Rewrite Rules: A Survey. In: R. V. Book (ed.) :"Formal Language Theory: Perspectives and Open Problems". Academic Press. 1980. pp 349-405.

[Hullot 80] J. M. HULLOT. Canonical Forms and Unification. Proc. of the 5th Conference on Automated Deduction. Lecture Notes in Computer Science 87. Springer-Verlag. 1980. pp. 318-334.

[Hussmann 88] H. HUSSMANN. Nondeterministic Algebraic Specifications and Nonconfluent Term Rewriting. In: J. Grabowski, P. Lescanne and W. Wechler (eds.). Proceedings of the Algebraic and Logic Programming International Workshop. Gaussig GDR november 1988. Springer-Verlag. pp. 31-40.

[Iverson 62] K. IVERSON. A Programming Language. Wiley. 1962.

[Kaplan 88] S. KAPLAN. Rewriting with a Nondeterministic Choice Operator. Theoretical Computer Sciences No. 56. 1988. pp. 37-57.

[Klop 90] J. W. KLOP. Term Rewriting Systems. Technical Report CS-R 9073. CWI. Amsterdam. December 1990.

[Kornfeld 86] W. A. KORNFELD. Equality for Prolog. In: [DeGroot & Lindstrom 86]. pp. 279-294.

[Knuth & Bendix 70] D. KNUTH and P. BENDIX. Simple Word Problems in Universal Algebras. In J. Leach (ed): Computacional Problems in Abstract Algebra. Pergamon Press 1970. pp. 263-297.

[Kowalski 74] R. A. KOWALSKI. Predicate Logic as a Programming Language. In: Proc. IFIP'74. Stockholm. North Holland. 1974. pp 569-574.

[Kowalski 79] R. A. KOWALSKI. Logic for Problem Solving. North

Holland. 1979.

[Kuchen & al. 90] H. KUCHEN, R. LOOGEN, J. J. MORENO and M. RODRIGUEZ ARTALEJO. Graph-Based Implementation of a Functional Logic Language. Technical Report Univ. Comp. Madrid. March 1990.

[Landin 66] The Next 700 Programming Languages. Commun. A.C.M. Vol. 9. 1966. pp. 157-166.

[Levi & al 87] G. LEVI, P. G. BOSCO, E. GIOVANNETTI, C. MOISO and C. PALAMIDESI. A Complete Semantic Characterization of K-LEAF. In Proceedings 4th. Symposium on Logic Programming. San Francisco 1987. pp. 1-27.

[Lloyd 87] J. W. LLOYD. Foundations of Logic Programming. 2a. extended edition. Springer-Verlag. 1987.

[McCarthy 60] J. McCARTHY. Recursive Functions of Symbolic Expressions and Their Computation by Machine. A.C.M. Vol. 3. 1960. pp. 184-195.

[Milner 87] R. MILNER. A Proposal for Standard ML. In Proceedings A.C.M. Conference on LISP and Functional Programming 1987.

[Moreno 89] J. J. MORENO. Diseño, Semántica e Implementación de BABEL: Un Lenguaje que Integra la Programación Funcional y Lógica. Tésis. Facultad de Informática U.P.M. Madrid. 1989.

[Moreno & Rodriguez 89] J. J. MORENO and M. RODRIGUEZ ARTALEJO. Logic Programming with Functions and Predicates: The Language BABEL. Informe Técnico. Departamento de Informática y Automática U.C.M. (To appear in Journal of Logic Programming). 1989.

[Mulmuney 86] K. MULMUNEY. Full Abstraction and Semantic Equivalence. ACM Doctoral Dissertation Award 1986. MIT Press 1987.

[O'Donnell 85] M. J. O'DONNELL. Equational Logic as a Programming Language. M.I.T. Press. 1985.

[Peyton Jones 87] S. L. PEYTON JONES. The Implementation of Functional

Programming Languages. Prentice-Hall. 1987

[Plotkin 76] G. PLOTKIN. A Powerdomain Construction. S.I.A.M. Vol. 5. 1976. pp. 452-487.

[Reddy 85] U. S. REDDY. Narrowing as the Operational Semantics of Functional Languages. In: Proceedings of the IEEE International Symposium on Logic Programming. IEEE Computer Society Press. July 1985. pp. 138-151.

[Reddy 86] U. S. REDDY. On the Relationship Between Logic and Functional Languages. In: [DeGroot & Lindstrom 86] pp. 3-36.

[Robinson 65] J. A. ROBINSON. A machine-oriented Logic based on the Resolution Principle. J. A. C. M. vol. 12 n. 1 1965. pp 23-41.

[Robinson & Sibert 82] J. A. ROBINSON and E. E. SIBERT. LOGLISP: An Alternative to PROLOG. In: Hays, Mitchie & Yao (eds.): Machine Intelligence 10. Wiley and Sons. 1982. pp. 399-498.

[Scott 81] D. S. SCOTT. Lectures on a Mathematical Theory of Computation. Technical Monograph PRG-19. Oxford University Computing Laboratory. 1981.

[Scott 82] D. S. SCOTT. Domains for Denotational Semantics. In: Proceedings ICALP'82. LNCS 140. Springer Verlag. 1982.

[Scott & Strachey 71] D. S. SCOTT and C. STRACHEY. Toward a Mathematical Semantics for Computer Languages. Technical Monograph PRG-6. Oxford Computing Laboratory. 1971.

[Shapiro 83] E. SHAPIRO. A Subset of Concurrent PROLOG and its Interpreter. Technical Report. Tokyo. 1983.

[Smyth 78] M. SMYTH. Powerdomains. Journal of Computer & System Science Vol. 16 No. 1. 1978.

[Sterling & Shapiro 86] L. STERLING and E. SHAPIRO. The Art of Prolog. MIT Press. 1986.

[Stoy 77] J. E. STOY. Denotational Semantics: The Scott-Strachey Approach to Programming Language Theory. MIT Press. 1977.

[Subrahmanyam 81] P. A. SUBRAHMANYAM. Nondeterminism in Abstract Data Types. In: S. Even & O. Kariv (eds.) ICALP'81. LNCS 115. Springer Verlag. 1981. pp. 148-164.

[Subrahmanyam & You 86] P. A. SUBRAHMANYAM and J. H. YOU. FUNLOG: A Computational Model Integrating Logic Programming and Functional Programming. In: [DeGroot & Linsdtrom 86]. pp. 157-198.

[Tarski 55] A. TARSKI. A lattice-theoretical fixpoint theorem and its applications. Pacific Journal Math. vol. 5 1955. pp 285-309.

[Turner 76] D. A. TURNER. SASL Language Manual. Technical Report. Univ. St Andrews. 1976.

[Turner 79] D. A. TURNER. A New Implementation Technique for Applicative Languages. Softw. Pract. Exper.9. 1979. pp. 31-49.

[Turner 81] D. A. TURNER. The Semantic Elegance of Applicative Languages. In: Proceedings of the 1981 Conference on Functional Programming Languages and Computer Architecture A.C.M. 1981. pp. 85-92.

[Turner 85] D. A. TURNER. MIRANDA: A Non-Strict Functional Languages with Polymorphic Types. In: Functional Programming Languages and Computer Architecture. Springer-Verlag LNCS 201. 1985. pp. 1-16.

[Van Emden & Kowalski 76] M. H. VAN EMDEN and R. A. KOWALSKI. The semantics of Predicate Logic as a programming language. J.A.C.M. vol. 23 n.4 1976. pp 733-742.

[Warren 77] D. H. D. WARREN. Applied Logic: Its Use and Implementation as a Programming Tool. Ph. D. Thesis. University of Edimburgh, Scotland 1977.

[Winskel 83a] G. A. WINSKEL. A Note on Powerdomains and Modality. In:

Proceedings of the Conference F.C.T. Sweden. August 1983. LNCS. Springer Verlag.

[Winskel 83b] G. A. WINSKEL. Nondeterministic Recursive Program Schemes and Powerdomains. Computer Laboratory University of Cambridge. 1983.