Transformadores de Predicados y Semántica de Programas

A la memoria de mi padre,
Manuel Ruiz Hoyos,
sencillo matemático y gran Profesor.

A mi nieto recién nacido,
Manuel Ruiz Sánchez,
con el deseo de que complete la
cuarta generación de matemáticos.

Blas C. Ruiz Jiménez

Blas Carlos Ruiz Jiménez

Profesor Titular de Universidad
Departamento de Lenguajes y Ciencias de la Computación

E.T.S.I. Informática. Universidad de Málaga

TRANSFORMADORES DE PREDICADOS Y SEMÁNTICA DE PROGRAMAS

(CORRECCIÓN DE LA 2ª ED. – SETIEMBRE DE 2003)

Málaga, Octubre de 2010

IMPRIME: *IMAGRAF–Impresores.*
C/Nabuco, Nave 14-D. 29006–Málaga. Tel.: 2328597

I.S.B.N.: **84-607-5971-7**
Depósito Legal: MA-1203-2003

Composición realizada por el autor en LaTeX2ϵ.

Prólogo

Quien puede hace, quien no puede enseña
G. Bernard Shaw, *Man and superman*, 1903[1].

El objetivo del presente libro es el de servir como *libro de texto* de la asignatura LENGUAJES DE PROGRAMACIÓN, correspondiente al tercer curso de los estudios de Ingeniería Informática de la Universidad de Málaga. Este texto es el resultado de mi experiencia docente durante más de trece años.

A grandes rasgos, la asignatura está dedicada a la *Semántica de los Lenguajes de Programación*, haciendo especial énfasis en dos modelos semánticos para un lenguaje imperativo simple: el modelo de *transformadores de predicados* de E.W. Dijkstra, y la lógica de C.A.R. Hoare. Con objeto de dar una visión más amplia, la asignatura se complementa con aspectos relativos a los estilos semánticos operacionales y denotacionales. Tales modelos son aplicados sistemáticamente a la escritura de programas, o *programación*, que es como se conoce esta disciplina usualmente en la literatura científica.

La programación, en un ámbito académico, y por consiguiente en este texto, incluye necesariamente el término *programación correcta*, que es aún temido en la enseñanza quizás porque exige ciertas habilidades matemáticas.

La Programación como una Actividad Científica La historia del calculador está enormemente ligada a la historia de las descripciones algorítmicas; Blas Pascal, aprovechando el mecanismo contador a base de ruedas dentadas (odómetro, descrito por Herón de Alejandría varios siglos antes) y las descripciones algorítmicas facilitadas para las operaciones aritméticas elementales, construye su máquina de sumar alrededor del año 1642. La asociación de estas dos ideas, mecanismo dentado y algoritmo, parece que constituye el nacimiento del calculador. Artilugios parecidos, como la máquina multiplicadora de Leibniz aparecida treinta años más tarde, van surgiendo con esta asociación de ideas hasta hace muy pocos años.

Sin menospreciar las ideas de ilustres matemáticos como Pascal o Leibniz, la noción de computador se debe al también matemático inglés Charles Babbage, quien, alrededor del año 1860, diseña su *Máquina Analítica*; esta admitía una programación (externa) basada en el mecanismo de tarjetas perforadas del telar de Jacquard (1801); desde una librería *física* se montaban paquetes de tarjetas correspondientes a *rutinas* para funciones matemáticas; estas rutinas controlaban el cálculo (movimiento de ruedas dentadas) a realizar con los datos suministrados por otro juego de tarjetas perforadas. Desde que Ada Augusta de Lovelace (hija del poeta Byron) *programó* el famoso ingenio de Charles Babbage, se vislumbró la capacidad conceptual de abstracción sobre una máquina. El propio Babbage escribiría ([Babbage, 1864]):

[1] Obras completas, traducción en Ed. Suramericana, Buenos Aires, 1950, página 320.

> *. . . ahora todos los desarrollos y operaciones del análisis matemático pueden ser ejecutados mediante máquinas . . . La Máquina Analítica influirá decisivamente en el desarrollo de la ciencia. Cuando pretendamos resolver un problema con su ayuda, necesariamente hemos de preguntarnos ¿qué procedimiento de cálculo emplearemos para que la máquina obtenga el resultado en el menor tiempo posible?*

Desde ese año (1864) se acepta que la programación obedece a unas necesidades concretas: la comunicación de algoritmos para su realización automática en un computador. Los algoritmos se describen a través de *programas*. Curiosamente, el problema de la *corrección de los programas* surge ya en estos años. Parafraseando a Ada Augusta de Lovelace: las máquinas, muy testarudas, se empeñan en hacer lo que se les dice, no lo que el programador quería decirles.

¿Es la Programación una ciencia?

La pregunta anterior es crucial. Si admitimos una respuesta afirmativa tendremos dos conclusiones importantes; por un lado, es tradicional incluir dentro de la cultura general de cada individuo los aspectos fundamentales de cada ciencia; por otro, el carácter científico (aspectos teóricos, modelado, etc) debe reflejarse en su enseñanza.

Durante el nacimiento de la programación su enseñanza y práctica eran simples: dibujo de un diagrama de flujo, codificación, test y depuración. La componente *artística* era muy elevada y los Cursos de Programación se limitaban prácticamente a describir algún lenguaje. Investigaciones posteriores modificaron tal punto de vista aunque hoy en día, incluso en ámbitos universitarios, todavía quedan rasgos de tal componente artística. Según [Arsac, 1985], la evolución de las técnicas de programación queda reflejada en tres extraordinarios textos: *The art of computer programming* [Knuth, 1968], *A discipline of programming* [Dijkstra, 1976] y *The science of programming* [Gries, 1981]. En 13 años pasó de ser un arte a ser una ciencia, lo que obligó a los enseñantes a cambiar sus métodos. Tal evolución debe entenderse desde el punto de la enseñanza superior (los tres libros citados no son en ninguna forma elementales).

Numerosos autores justifican una respuesta a la pregunta anterior describiendo *la naturaleza de la programación como una actividad científica* [Hoare, 1971], y aún más lejos, otros sostienen que *los conceptos de la programación son tan universales como las matemáticas* [Hebenstreit, 1985]. Un ejemplo de este punto de vista lo proporciona brillantemente Edsger W. Dijkstra, que en múltiples artículos y ensayos muestra una profunda analogía entre la actividad del matemático y la actividad del programador; así, comienza su artículo *Why correctness must be a mathematical concern* ([Dijkstra, 1981]) con el siguiente problema:

> una urna contiene bolas blancas y negras; se extraen dos bolas, añadiéndose posteriormente una negra si estas eran del mismo color o una blanca si eran de colores diferentes; este paso se repite cuantas veces sea posible; puesto que en cada paso decrece en una unidad el total de bolas (se extraen dos y se añade una) en un número finito de pasos quedará una sola bola; la pregunta es ¿qué puede afirmarse sobre el color de la última bola en función del contenido inicial de la urna?

La simulación del juego en casos particulares conduce a conjeturar: *la paridad del número de bolas blancas es decisiva*; en efecto, es fácil ver que en cada paso no se altera la paridad del número de bolas blancas; por ello, si el número inicial de bolas blancas es impar, la última será blanca. Dijkstra observa que tal argumento permite solucionar el problema en todos los casos, para todos los estados iniciales y para todos los juegos posibles. Por ello la solución dada tiene las siguientes características propias de las matemáticas: (a) la respuesta es general (para todos los juegos posibles), (b) es precisa y (c) está justificada por un razonamiento convincente y exacto. En definitiva, el programador, como el matemático, debe trabajar con generalidad, precisión y convicción. La programación es pues una actividad de indudable naturaleza matemática; [Dijkstra, 1981]:4 añade

> *. . . por un desafortunado accidente histórico, la programación se ha convertido en una actividad industrial en los Estados Unidos en un momento de profunda influencia en la educación . . .*

Sobre el lenguaje de programación Desde un punto de vista educacional, quiero recoger algunas observaciones interesantes sobre el papel que puede jugar la elección del lenguaje de programación en la escritura de un texto de programación. En tales observaciones veremos el descontento con los lenguajes existentes por parte de autores de indudable importancia.

Donald Knuth, en el prólogo del primer volumen de *The Art of Computer Programming* ([Knuth, 1968]:x) dice: *. . . un programador está profundamente influenciado por el lenguaje en el que escribe sus programas*. A continuación justifica la elección de un lenguaje de bajo nivel como MIX frente a un lenguaje (que el llama algebraico) como ALGOL, diciendo que en muchos problemas de interés tratados en su texto es más importante el *arte del programador* (por razones de eficiencia) que la comodidad en la escritura de programas. Otros autores también reflejan tal componente artística. Por ejemplo, para [Wirth, 1973], *la programación debe entenderse como el arte o técnica de construir y formular algoritmos en forma sistemática. Se trata de una disciplina constructiva, sintetizadora, y con entidad propia*.

En *Fundamentals of Computer Algorithms* [Horowitz y Sahni, 1978], en la página 4 de la introducción aparece:

> *. . . la elección de un lenguaje para la descripción de algoritmos es difícil . . . consideramos inicialmente algunos lenguajes existentes, como* ALGOL, FORTRAN *o* PASCAL. *En primer lugar, deseamos escribir nuestros algoritmos sin enfatizar las idiosincrasias de un lenguaje determinado. En segundo lugar, cada lenguaje tiene sus seguidores y detractores . . .*

Así, proponen un nuevo lenguaje que llaman SPARKS, con profundo sabor pascaliano y que incorpora algunos conceptos, como el polimorfismo; sin embargo, tal lenguaje incorpora idiosincrasias propias, como la construcción *loop ... if ... exit* o variables globales declaradas en procedimientos. Por ello, las razones expuestas por los autores no están muy justificadas; más aún si tenemos en cuenta que enfatiza demasiado la existencia de un preprocesador para un compilador de FORTRAN ¿Quizás la verdadera razón de los autores sea el querer contentar a todos los programadores de lenguajes como PASCAL o FORTRAN?

Edsger [Dijkstra, 1976] en el prefacio de su texto *A Discipline of Programming* (xiii–xiv), dice:

> *. . . responder a la pregunta ¿qué lenguaje de programación vamos a utilizar? no es una mera cuestión de presentación. Cualquier herramienta crea hábitos y un lenguaje de programación, como herramienta, influye en el hábito de pensar . . .*

Llega a la conclusión de que ninguno de los lenguajes existentes responde a su propósito, y que no es necesario diseñar para la exposición de su texto un lenguaje completo, sino solamente un mini-lenguaje en el que las construcciones fundamentales estén profundamente justificadas por la metodología a desarrollar. Afortunadamente, esta *nueva forma de programar* ha sido exportada brillantemente por muchos autores utilizando lenguajes existentes, por lo que la metodología propuesta por Dijkstra ha sido más general de lo que se pensaba, enfatizando que *la programación es una rama formal de la matemática* [Dijkstra y Feijen, 1984]:v., en la cual la lógica matemática es una herramienta indispensable.

Los textos antes citados contienen los fundamentos de la programación de computadores y son considerados esenciales desde el punto de vista educacional. En los tres se justifica la introducción de un nuevo lenguaje para enfatizar que la línea de exposición del texto hace necesaria la introducción de un lenguaje cómodo para la descripción de las ideas que se desarrollan. Aún así, el punto de vista de Dijkstra es el que más ha influido en el desarrollo de gran parte del presente texto, así como en el enfoque que he dado a la asignatura.

Sobre el contenido del presente texto El texto está dividido en cinco partes. La primera de ellas está dedicada a los conceptos matemáticos y herramientas más útiles en programación. Además de un capítulo introductorio, que puede leerse como complementario a este prólogo, el capítulo primero está dedicado al cálculo de predicados sobre espacios de estados. En él desarrollamos la lógica de predicados que será utilizada en el resto del libro, así como una herramienta esencial en la formación de un Informático: el principio de inducción, dando distintas equivalencias del concepto de conjunto bien construido.

El estilo utilizado en las demostraciones es el propuesto por Edsger W. Dijkstra, aparece suficientemente justificado en [Dijkstra y Scholten, 1990], y ya fue utilizado por otros matemáticos del siglo pasado, como Cauchy. La razón fundamental de su uso es que permite *comunicar* de forma clara y paso a paso una demostración matemática. El uso de una única notación para este propósito es esencial: el profesor y todos los alumnos, durante las clases, en sus ejercicios personales, en los exámenes, etc, deberán acostumbrarse a exponer los pasos de un razonamiento riguroso de forma inequívoca, lo que facilita la tarea de revisión de los ejercicios por los tutores. Tal estilo fue ampliamente discutido y aceptado durante la celebración del *University of Texas Year Programming* y es utilizado sistemáticamente por distintos autores, entre ellos [Bird y Wadler, 1988], y los colaboradores de [Dijkstra, 1990, Huet, 1990]. El mismo estilo es también utilizado en otras asignaturas de nuestra escuela [Ruiz Jiménez et al., 2000] en las que se hace necesaria una descripción formal de las demostraciones.

También en el Capítulo 2 de esta primera parte se exponen las herramientas más útiles de la teoría de dominios, como son los teoremas del punto fijo

y las técnicas de construcción de dominios. Tanto desde el punto de vista de la semántica denotacional, como desde el punto de vista de los transformadores de predicados, los resultados de este capítulo son imprescindibles para modelar (semánticamente) la recursión y los procedimientos.

En la segunda parte se desarrolla la semántica de Dijkstra relacionándola con la lógica de Hoare. Para ello comenzamos describiendo un lenguaje muy simple indeterminista y con bucles, dando la semántica de los bucles en forma inductiva, para después completarla vía los teorema del punto fijo. Con numerosos ejemplos se enfatiza la necesidad de proceder a una construcción metódica de los programas, sobre todo en aquellos que intervienen bucles, describiendo distintas estrategias de diseño basadas en invariantes y contadores generalizados de los bucles. Durante la exposición se muestra como se usan las herramientas expuestas en la primera parte del texto y de forma unificada. Por ejemplo, el principio de inducción se utilizará para probar distintas propiedades del lenguaje (salubridad, continuidad, etc) o para encontrar la semántica de programas en presencia de bucles, o incluso probar propiedades de la lógica de Hoare, a través de inducción sobre las derivaciones en la lógica. Esta parte termina con un capítulo final dedicado a las construcciones recursivas y los procedimientos con parámetros.

La tercera parte del texto está dedicada a otros estilos semánticos: operacionales y denotacionales, dedicando un capítulo a cada uno. Una vez descrito el estilo operacional con distintos ejemplos introductorios, se procede a modelar dos lenguajes imperativos, uno determinista, y otro indeterminista, que es el utilizado en la segunda parte del texto. De esta forma, será fácil establecer la conexión con la lógica de Hoare y los transformadores de predicados.

En el siguiente capítulo exponemos una introducción a los modelos denotacionales, haciendo especial hincapié en la necesidad de hacer uso de los resultados de la teoría de dominios con objeto de modelar la no terminación o el indeterminismo, donde se justifica las distintas construcciones correspondientes a los dominios potencia. Además de un lenguaje funcional simple, se expone la semántica denotacional de los mismos lenguajes utilizados en las partes precedentes. De esta forma el alumno deberá comparar y valorar los tres estilos semánticos esenciales.

Muchos ejercicios y ejemplos tratados en el texto *proceden* de ejercicios propuestos en clases o en exámenes realizados entre los años 1989 y 2003. En estos se reflejan los tópicos más interesantes de la asignatura, ya que estos son los que mayormente aparecen en los exámenes.

Terminamos el texto con las soluciones a los ejercicios propuestos, dando en cada caso soluciones alternativas o incluso relacionándolos con otros ejercicios o resultados ya demostrados en otros capítulos.

En un principio pretendí que este texto fuera una revisión corregida de la edición que con idéntico título publiqué en el año 1999; durante el proceso de revisión he introducido muchos cambios, tanto de contenido y de notación, como de estructura, por lo que el resultado final es un texto muy diferente.

Málaga, Setiembre de 2003

BLAS C. RUIZ

Índice general

El estilo Semántico de Dijkstra 41

Índice de figuras

Capítulo 0

Introducción

0.0. Modelos Semánticos

Un programa es correcto si cumple su especificación. Desde el punto de vista del programador la *especificación* de un programa es una formulación lógica y/o algebraica de lo que debe realizar. La *verificación* consiste en demostrar matemáticamente que un programa es correcto. Es imposible demostrar la corrección de un programa si no se puede describir formalmente cada construcción del lenguaje. La corrección estimula un enfoque riguroso de la programación y motiva una definición formal de la sintaxis y de la semántica de los lenguajes. La corrección cada vez tiene más influencia en el diseño de lenguajes.

La definición de un lenguaje (sintaxis y semántica) tiene dos objetivos; uno es determinar aspectos para una correcta implementación: diseñar un compilador que se ajuste a la estructura semántica. El otro objetivo es de cara al programador: *sintetizar* programas correctos así como *verificar* programas ya escritos.

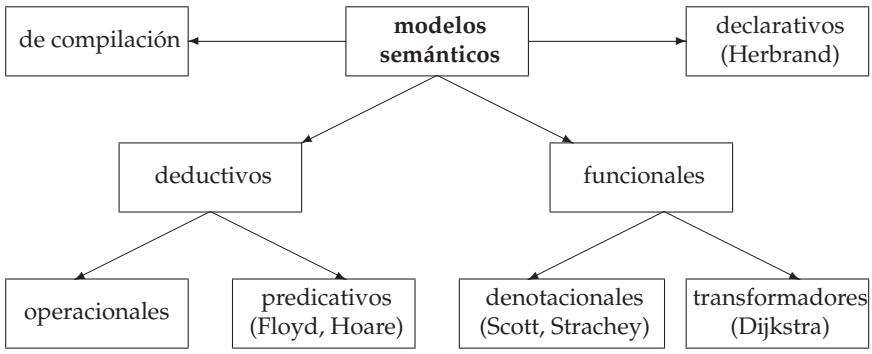


Figura 0: Modelos Semánticos

Siguiendo a [Wegner, 1984] y [Schmidt, 1988]:1–4, un *modelo semántico* es una terna $\mathcal{M} = (\mathcal{L}, \mathcal{D}, \phi)$, donde

- ✓ $\mathcal{L}$ es un *dominio sintáctico*, o conjunto de programas.
- ✓ $\mathcal{D}$ es un *dominio semántico* o de especificaciones.
- ✓ ϕ es una aplicación: $S \in \mathcal{L} \mapsto \phi(S) \in \mathcal{D}$,

de forma que $\phi(S)$ es el significado o comportamiento de S.

Los modelos semánticos se clasifican en términos de la naturaleza del dominio semántico[1]:

— **Modelos de compilación**. El dominio semántico es un conjunto de programas objetos. La semántica de un programa es un código ejecutable por cierto procesador.

— **Modelos deductivos**. Están basados en sistemas de inferencia o teorías, donde los elementos de la teoría capturan los posibles cómputos. Ejemplos de estos son los

  ✓ Operacionales: especifican los cómputos a través de sistemas de transición que vienen determinados por un sistema de inferencia.

  ✓ Axiomáticos predicativos: capturan a través del cálculo de predicados las especificaciones, y a través de una teoría o cálculo, la relación entre especificaciones de entrada y salida (pre y poscondiciones). Ejemplos de ellos son los modelos de Floyd y de Hoare.

— **Modelos funcionales**. En estos el "significado"de un programa está determinado por una función abstracta que permite obtener lo que *computa* un programa. La definición está estructurada: partiendo de la función correspondiente a cada construcción del lenguaje y con ciertas reglas, se define la función final. Las estructuras de control del lenguaje se definen a partir de la composición de funciones elementales; por ejemplo, el bucle da lugar a una función recursiva. Ejemplos de estos modelos son el

  ✓ *denotacional* de Scott/Strachey, combinación del rigor matemático del modelo de Dana Scott con la notación *elegante* de Christopher Strachey.

  ✓ *de Dijkstra*, basado en transformadores de predicados.

— **Modelos Declarativos**. Modelan lenguajes lógicos. Con ciertas restricciones, las consecuencias lógicas de un programa se identifican con un modelo (mínimo modelo de Herbrand), que a su vez puede también identificarse como el punto fijo de cierta funcional. En la literatura anglosajona se describen con *Model-theoretic semantics* [Kowalski, 1979].

Todos los modelos son interesantes y están interrelacionados. Desde el punto de vista de la programación, los modelos axiomáticos son más apropiados ya que capturan las propiedades deseadas cómodamente en forma predicativa, por lo que son interesantes de cara a la síntesis de programas y su verificación. Una vez formuladas tales propiedades, una semántica denotacional proporciona el significado del programa; una demostración formal probará que tal semántica captura las propiedades del sistema axiomático (se trata de dar la equivalencia de tales semánticas). Finalmente, la semántica operacional permite construir un intérprete. La demostración de la equivalencia semántica de la denotacional con la operacional permite probar la corrección del intérprete.

Damos a continuación algunas ideas más precisas de los modelos que estudiaremos.

[1] En [Berg y o., 1982]: 15 puede verse un diagrama completo de los distintos modelos con los principales trabajos de investigación y aportaciones.

0.1. Modelos operacionales

En estos se describe cómo se ejecutan los programas sobre una computadora virtual, abstracción del modelo de implementación. La función semántica viene dada por una relación entre estados iniciales y finales, que modela las transformaciones sobre el espacio de estados $\mathcal{E}$. La semántica de un programa sería la traza que produciría el intérprete (o el conjunto de posibles trazas, si el programa es indeterminista).

Hay varios estilos para definir la relación de transición. Normalmente se hace obedeciendo a la estructura del programa, por lo que podemos hablar de *semántica operacional estructurada* [Hennessy, 1990]. Por ejemplo, la *semántica operacional natural* [Nielson y Nielson, 1992] para un lenguaje imperativo es una relación

$$\twoheadrightarrow_{\mathcal{N}} : \mathcal{E} \times \mathcal{P}rog \mapsto \mathcal{E}$$

donde $\mathcal{P}rog$ es el conjunto de todas las sentencias, y $\mathcal{E}$ el conjunto de estados posibles (que a su vez están descritos por los valores de las variables). La expresión $(\rho, S) \twoheadrightarrow_{\mathcal{N}} \rho'$ significa que la ejecución de la sentencia S, partiendo del estado ρ, puede terminar en el estado ρ'. La relación $\twoheadrightarrow_{\mathcal{N}}$ puede ser descrita con un sistema de inferencia o cálculo, y normalmente se describe en forma estructurada. Por ejemplo, para la composición de sentencias tenemos la regla

$$\frac{(\rho, S) \twoheadrightarrow_{\mathcal{N}} \rho' \qquad (\rho', T) \twoheadrightarrow_{\mathcal{N}} \rho''}{(\rho, S; T) \twoheadrightarrow_{\mathcal{N}} \rho''}$$

en la cual se indica que es posible realizar la transición $(\rho, S; T) \twoheadrightarrow_{\mathcal{N}} \rho''$ si es posible realizar las transiciones particulares de la parte superior de la regla.

Las principales *desventajas* de los modelos operacionales son

- ✓ La semántica es el conjunto de trazas del programa.
- ✓ La descripción algorítmica del intérprete puede ser tan compleja o más que el propio lenguaje en sí.
- ✓ Es difícil demostrar la equivalencia de los programas, y por tanto, es difícil desarrollar una metodología simple de cara a la programación.

0.2. Modelos denotacionales

Como en todos los modelos funcionales, en estos se da la función que transforma *directamente* un programa en su significado, utilizando una función valor denotada usualmente con $[\![\_]\!]$:

$$S \to [\![S]\!] \in \mathcal{D} \qquad \text{— denotación o significado}$$

Al igual que en la semántica operacional, el valor $[\![S]\!]$ se determina en forma estructurada, por lo que la función $[\![\_]\!]$ se puede obtener a través de otras más sencillas, en las que pueden intervenir varios dominios de denotaciones; en estos modelos la teoría de dominios es esencial, y tiene la ventaja adicional de poder interpretar las propiedades de los programas viendo cómo se reflejan en propiedades de los dominios involucrados. Tales modelos son más abstractos

(no especifican transiciones) y permiten estudiar las partes individuales de la semántica sin examinar la definición entera.

A modo de ejemplo, consideremos como dominio semántico $\mathcal{D}$ el espacio de funciones $\mathcal{E} \to \mathcal{E}$, de forma que $[\![S]\!]$ es una función que a cada entorno o estado inicial le hace corresponder otro estado final. En ese caso, la denotación de la composición $begin\ S; T\ end$ se puede obtener en la forma siguiente:

$$[\![begin\ S; T\ end]\!].\rho \doteq [\![T]\!].([\![S]\!].\rho)$$

Este modelo tan simple no puede contemplar ni errores, ni la *no terminación*, ni el indeterminismo. Para obtener indeterminismo necesitamos admitir un posible conjunto de estados finales, por lo que el valor $[\![S]\!].\rho$ sería un subconjunto de $\mathcal{E}$, y por tanto $[\![S]\!] \in \mathcal{E} \to \mathbb{P}(\mathcal{E})$. Para contemplar los errores de evaluación de expresiones hay que introducir un valor especial, normalmente denotado con $\perp$, que representa un estado (abstracto) al que conduce las evaluaciones erróneas, de forma que el dominio a utilizar sería ahora la suma disjunta de dos dominios, $\mathcal{D} = \mathcal{E} + \{\perp\}$. Pero aquí no acaban los inconvenientes ya que para capturar la no terminación debemos incorporar un nuevo elemento. De esta forma, la semántica de la composición secuencial ya no es tan simple, y los dominios utilizados deben ser estudiados convenientemente con objeto de formalizar otras construcciones, como la recursión o los procedimientos.

0.3. Modelos axiomáticos predicativos

Estos parten de la posibilidad de capturar subconjuntos de estados y configuraciones del sistema vía el cálculo de predicados. Un predicado caracteriza un conjunto de estados. Se caracterizan las construcciones del lenguaje a través de una *teoría* asociando *teoremas especiales* de la teoría a construcción del lenguaje. Por ejemplo, en el modelo de Hoare, estos teoremas son los tripletes $\{Y\}S\{X\}$. Tales modelos son más abstractos que los operacionales, ya que la semántica de S es el conjunto de todos los posibles tripletes que pueden ser inferidos en la teoría.

Los trabajos de Robert Floyd Los primeros trabajos sobre la definición de un lenguaje de programación en forma *axiomática* aparecen en los años 60. Ejemplo de ello es el trabajo de Robert [Floyd, 1967] publicado con el título *Assigning meaning to programs*, que será posteriormente formalizado por Tony [Hoare, 1969] y Zohar [Manna, 1974].

El método de Floyd consiste en asignar asertos o predicados a ciertos puntos del diagrama de flujo correspondiente a un programa. Tales asertos describen relaciones entre las variables en tal punto del programa. En este modelo el problema de la verificación se establece en la forma siguiente: dadas una poscondición (predicado de salida, o *output predicate*, como denomina Floyd), una precondición (predicado de entrada o *input predicate*) y un programa, se trata de demostrar que las ejecuciones del programa con entradas satisfaciendo la precondición conducen a su terminación verificando la poscondición. Para ello define el significado o semántica de las primitivas del lenguaje en términos de pre y poscondiciones. A través de reglas de inferencia define la composición de sentencias. Para un bucle o ciclo del diagrama de flujo, Floyd fija un punto arbitrario del bucle llamado punto de corte (*cut point*) asociándole un predicado

I e intenta probar que si la ejecución comienza en tal punto con el predicado I cierto debería ser cierto al volver a dicho punto. Esta idea inspirará más tarde a Dijkstra para introducir su concepto de invariante.

Floyd afirma que el método de verificación basado en inducción por asertos (*inductive-assertions method*) parte de la necesidad de una adecuada definición del lenguaje de programación. Puede verse con detalle tal método en [Manna, 1974]:174–189, y también descripciones breves en [Knuth, 1968]:15 y en [Wirth, 1983]: 21–26.

Los trabajos de Hoare Basándose en los trabajos de Floyd, Carlos Antonio Ricardo (C.A.R.) Hoare escribe en 1969 su célebre artículo *An Axiomatic Basis for Computer Programming* [Hoare, 1969]. En este trabajo desarrolla una notación lineal del formalismo de Floyd, introduciendo el triplete $\{X\}S\{Y\}$[2] para expresar la siguiente idea: *si la ejecución del código S comienza en un estado satisfaciendo la precondición X, y si la ejecución termina, el estado final verificará la poscondición Y*. A continuación da una serie de axiomas y reglas de inferencia, que son descritas brevemente tomando como base el lenguaje ALGOL. De esta forma se pueden caracterizar programas completos. Entre las reglas aparecen las siguientes:

$$\frac{\{X\}S\{Y\} \qquad \{Y\}T\{Z\}}{\{X\}begin\ S;T\ end\{Z\}} \qquad\qquad \frac{\{P \wedge b\}S\{Q\} \qquad \{P \wedge \neg b\}T\{Q\}}{\{P\}if\ b\ then\ S\ else\ T\ fi\{Q\}}$$

y para los bucles considera la regla

$$\frac{\{I \wedge b\}S\{I\}}{\{I\}while\ b\ do\ S\{I\}}$$

Una regla especial, llamada refinamiento, permite obtener tripletes a partir de teoremas del cálculo de predicados y de otros tripletes:

$$\frac{[P \Rightarrow P'] \qquad \{P'\}S\{Q'\} \qquad [Q' \Rightarrow Q]}{\{P\}S\{Q\}}$$

La lógica de Hoare mezcla tripletes con teoremas del cálculo de predicados sobre un espacio de estados. C.A.R. Hoare escribiría en su artículo de 1969:

> ... Cuando la corrección del programa, el compilador y el hardware del computador han sido establecidos con certeza matemática, es posible asignar gran confianza a sus resultados, y predecir sus propiedades con una seguridad limitada solamente por la fiabilidad de la electrónica ...

El estilo axiomático de Hoare será aplicado posteriormente a PASCAL. Uno de los primeros trabajos es [Wirth y Hoare, 1973], *An axiomatic Definition of the Programming Language Pascal*. En éste trabajo, aparecen reglas para las estructuras de control y otras para las estructuras de datos, operadores, etc. Los trabajos de Hoare crean escuelas en varios campos. Por un lado, el campo de la enseñanza de la programación: desde el año 72 se enseñaba en cursos de programación en las universidades. Numerosos textos han estado influenciado por

[2] En su trabajo original los tripletes los describe en la forma $X\{S\}Y$, aunque en trabajos posteriores usa la notación del presente texto.

estas ideas, por ejemplo, [Wirth, 1973, Wirth, 1976, Alagic y Arbib, 1978]. Otro campo de investigación donde influyen los trabajos de Hoare es en la especificación de tipos abstractos de datos [Liskov y Zilles, 1974]. El tercer campo es en el diseño de posteriores lenguajes. Como hemos dicho, tales ideas se aplicaron a PASCAL, pero se han tenido presentes en el diseño de nuevos lenguajes, como EUCLID [Popek y Horning, 1977] o EIFFEL [Meyer, 1988]. Finalmente citaremos también la influencia del método de Floyd–Hoare en sistemas de verificación automática, basados a su vez en axiomas y reglas de inferencia.

El método de Hoare tiene una ventaja clara: modela directamente la no terminación, y con ligeros cambios también es posible modelar el indeterminismo. Por el contrario, tiene dos inconvenientes importantes; el primero es que no puede ser utilizado para calcular Y en función de S y de X, con objeto de que se cumpla $\{Y\}S\{X\}$. El segundo es que el cálculo captura *corrección parcial* ([Manna, 1974]:170), [Gries, 1981]:109).

El problema de la *corrección total* fue resuelto por Robert Floyd y formalizado por Zohar [Manna, 1974] utilizando el método de los conjuntos bien construidos (*well-founded-set method*) (un conjunto parcialmente ordenado se dice bien construido si toda sucesión decreciente es finita). Para probar la terminación de un bucle se asocia al predicado I del punto de corte un valor dentro de un conjunto bien construido, de tal forma que en cada paso los valores forman una sucesión decreciente, por lo que el número de pasos será finito y el bucle debe terminar. Este método será mejorado posteriormente por Dijkstra introduciendo el concepto de *contador* de un bucle.

Manna y otros autores ([Gries, 1981]) utilizan la notación $\{P\}S\{Q\}$ para capturar corrección total: *si la ejecución del código S comienza en un estado satisfaciendo la precondición P, entonces la ejecución de S termina en un estado final verificando Q*. Como curiosidad diremos que la notación anterior es utilizada por Manna indistintamente para corrección total y parcial, mientras que Wirth la utiliza en su *Systematic Programming* para la corrección parcial guiado por el criterio de Pascal de incluir entre llaves los comentarios que conforman la documentación de un programa (véase [Wirth, 1973]:37). En este trabajo de Wirth ya se utiliza con precisión la idea actual de invariante: dado el bucle $while\ b\ do\ S$, un invariante es un predicado I verificando $\{I \wedge b\}S\{I\}$. [Wirth, 1973]:24 diría al respecto:

> *...la lección que todo programador debe recordar es que la indicación implícita del invariante esencial de cada repetición representa el elemento de más valor de la documentación de un programa...*

Los métodos axiomáticos son deductivos y tienen las limitaciones de los sistemas deductivos. Una limitación importante es que una demostración puede contener errores. Otra limitación es que se ha demostrado [Berg y o., 1982] que existen construcciones cuya semántica no puede ser expresada completamente por reglas de inferencia en un sistema deductivo. Según el teorema de incompleción de Gödel (en todo sistema deductivo existen teoremas que no pueden ser demostrados en el sistema) se deduce que un modelo semántico deductivo contiene programas cuya corrección no puede demostrarse.

Los trabajos de Dijkstra En los trabajos citados anteriormente se hace especial énfasis en la verificación y no en la síntesis de programas correctos. Por

ejemplo, en muchos casos el invariante del bucle es conocido por el programador antes de la verificación, pero esto no es necesariamente cierto y a veces el verdadero problema es buscar el invariante; por otro lado, cierta información sobre el invariante permitirá la síntesis de programas con bucles.

En el texto *A Discipline of Programming*, Edsger [Dijkstra, 1976] desarrolla un método axiomático con un enfoque constructivo haciendo hincapié en el desarrollo de programas correctos más que en la verificación. Estudia la solución mínima de la ecuación en Y: $\{Y\}S\{X\}$, que es denotada con $wp(S, X)$ (*weakest precondition*), de forma que

$$\{Y\}S\{X\} \equiv [Y \ \Rightarrow \ wp(S, X)]$$

y la implicación debe leerse *para todos los estados*. Dijkstra parte de la idea de que podemos conocer la semántica de un programa o mecanismo S si conocemos su transformador de predicados, es decir, una regla que permita derivar $wp(S, \_)$, y ello se consigue en forma composicional. La estrategia seguida por Dijkstra en su modelo es: (1) definir $wp(S, \_)$ para sentencias simples, y (2) dar reglas para calcular $wp(S, \_)$ a partir de los transformadores de las componentes de S.

Así, la síntesis de programas consiste en: dada una poscondición X, buscar un programa S de forma que el valor $wp(S, X)$ puede ser satisfecho para una amplia clase de situaciones. O sea, dada una descripción estática de X (un predicado) pasar a una descripción dinámica S.

El modelo de Dijkstra es utilizado a veces como sinónimo de *síntesis de programas correctos en forma disciplinada*, y opera *hacia atrás* ya que permite calcular la precondición de un programa para una especificación dada. Como dice David Gries: *ahora podemos mostrar que la programación puede ser practicada como una ciencia* [Gries, 1981]:301.

Dijkstra hace especial hincapié en su texto en la búsqueda de invariantes y desarrolla estrategias muy generales para ello. La demostración de la terminación de bucles la hace en forma paralela a la síntesis de programas correctos, introduciendo los contadores enteros basados en los conjuntos bien construidos usados por Robert Floyd.

El lenguaje propuesto por Dijkstra es simple, elegante y contiene dos aportaciones importantes: (1) el uso de secuencias guardadas y (2) el indeterminismo implícito en el lenguaje. La asimetría de la notación de ALGOL $if \ x \geq y \ then \ m := x \ else \ m := y \ fi$ trata la sentencia $m := y$ como un *defecto*, y como dice [Dijkstra, 1976]:214, *los defectos provocan desconfianza*. Así, Dijkstra utiliza la siguiente notación simétrica y elegante:

$$\begin{array}{lll} if & x \geq y & \rightarrow \quad m := x \\ \Box & y \geq x & \rightarrow \quad m := y \\ fi & & \end{array}$$

Los conceptos introducidos por Dijkstra tendrán importancia en el diseño de posteriores notaciones para la concurrencia, como CSP (*Communicating Sequential Processes* [Hoare, 1978, Hoare, 1985], ADA [ANSI-83, 1983] y GHC (*Guarded Horn Clauses*) [Ueda, 1985].

El texto de Dijkstra es difícil de leer y ha servido de base para la escritura de numerosos textos de metodología de la programación; entre ellos citemos [Gries, 1981] y [Dijkstra y Feijen, 1984], los cuales *suavizan* el trabajo anterior.

Dijkstra no fue el primero en considerar los transformadores de predicados para definir la semántica de las construcciones. Floyd, en su modelo original utilizaba un transformador de predicados $svc(S, Q)$, siendo tal predicado la poscondición más fuerte (*strongest verifiable consequent*) que verifica $\{Q\}S\{R\}$, de forma que se verifica:

$$\{P\}S\{R\} \equiv [svc(S, P) \ \Rightarrow \ R]$$

Así, por ejemplo tendríamos

$$svc(\text{máx} := b, \text{máx} = a \wedge \ \text{máx} < b) \equiv \exists x :: \text{máx} = b \wedge x = a \wedge x < b$$

Al contrario que el modelo de Dijkstra, el modelo anterior opera *hacia adelante*, ya que el transformador de predicados svc permite calcular la poscondición. Por cuestiones metodológicas, el modelo de Dijkstra es preferible al de Floyd por dos razones: es más sencillo y desde el punto de vista de lo que se quiere obtener (la poscondición) es más natural pensar en una función $wp(S, \_)$, que aplicada a X permita obtener la situación de partida más favorable.

En [Dijkstra, 1976] cada programa S está caracterizado por el transformador de predicados $wp(S, \_)$. En un trabajo posterior, [Dijkstra y Scholten, 1990], catorce años más tarde, cada programa S queda caracterizado por los valores $wp.S.Cierto$ y $wlp.S.X$, de forma que se tiene la relación

$$[\ wp.S.X \ \equiv \ wp.S.Cierto \wedge wlp.S.X\ ]$$

donde la notación *punto*[3] representa la *aplicación*. Es decir, $wp.S.X$ debe leerse como la aplicación de la función $wp.S$ al argumento X, y a su vez, $wp.S$ denota la aplicación de la función wp al argumento S. Así, $wp.S$ es una *aplicación parcial*. La funcional wlp *weakest liberal precondition* es el transformador de predicados que captura la corrección parcial.

En la ecuación anterior vemos que en vez de dar $wp.S.X$ se da su descomposición. Aunque este punto de vista puede tener ciertas ventajas, nos parece oportuno seguir en este texto el punto de vista original y, por tanto, la semántica de un programa S estará definida por el transformador $wp.S.X$. Los tripletes de Hoare podrían obtenerse vía la ecuación

$$\{Y\}S\{X\} \equiv [Y \ \Rightarrow \ wp(S, X)]$$

Las propiedades del lenguaje pueden estudiarse directamente a través de las propiedades de los transformadores. Por ejemplo la conjuntividad de los transformadores permite deducir la propiedad de monotonía, y ésta, permite comprobar que la lógica de Hoare es en cierta forma equivalente a la descripción vía transformadores de predicados.

[3] Introducida ya por Cauchy en su Curso de Análisis [Cauchy, 1821].

Capítulo 1

Cálculo con Estructuras Booleanas

1.0. Predicados sobre un espacio de estados

Denotemos con $\mathcal{E}$ el conjunto de los estados posibles en los que puede encontrarse una máquina, cuyo funcionamiento podemos describir con un programa $\mathcal{S}$. Normalmente, el conjunto de estados queda caracterizado por el conjunto de variables del programa. A modo de ejemplo, supongamos que las variables de cierto programa vienen declaradas en la forma

$$u, v :\in \mathcal{N}; w :\in \mathcal{B}$$

Como es usual, esto significa que u y v son variables de tipo $\mathcal{N}$ (el conjunto de los naturales) y w de tipo $\mathcal{B}$, siendo éste el conjunto $\{Cierto, Falso\}$ (escalares lógicos) con los operadores lógicos usuales: $\equiv$ (equivalencia), $\wedge$ (conjunción), $\vee$ (disyunción), $\Rightarrow$ (implicación), $\neg$ (negación). En ese caso el conjunto de estados puede describirse como el producto cartesiano de los conjuntos de valores asociados a las variables: $\mathcal{N} \times \mathcal{N} \times \mathcal{B}$ y cada variable es vista como una proyección:

$$w : \mathcal{E} \to \mathcal{B}$$

Hablaremos de estructuras como sinónimo de expresiones; existen estructuras enteras, booleanas, matriciales, funcionales, etc. La estructura booleana

$$u > 3 \wedge v \leq 0$$

asocia a cada estado un valor lógico; los operadores lógicos sobre estructuras booleanas se indicarán con los mismo símbolos que los correspondientes a $\mathcal{B}$: $(\equiv, \wedge, \vee, \Rightarrow, \neg)$. Estos pueden definirse a partir de los operadores lógicos sobre los escalares booleanos. Por ejemplo, para un estado $\sigma \in \mathcal{E}$ definimos

$$(u > 3 \wedge v \leq 0).\sigma \doteq (u > 3).\sigma \wedge (v \leq 0).\sigma$$

que se lee: el valor de la función $u > 3 \wedge v \leq 0$ en el estado σ es la conjunción de valores de los predicados $u > 3$ y $v \leq 0$ en el estado σ. De esta forma se puede definir el valor de una estructura booleana conocida su sintaxis. Nótese que el significado del operador infijo $\wedge$ en la expresión $u > 3 \wedge v \leq 0$ es diferente al significado del mismo operador en la expresión $Cierto \wedge Falso$: sobrecargamos los operadores ya que no es posible confundirlos. En definitiva, el conjunto $\mathcal{P}$ de predicados o estructuras booleanas es un álgebra.

Un predicado X puede ser cierto para todos los estados del sistema; ello significa que X (como función) es constante, por lo que tendremos

$$(\forall \sigma : \sigma \in \mathcal{E} : X.\sigma) \equiv Cierto$$

[Dijkstra y Scholten, 1990] introducen una notación más compacta

$$[X] \equiv Cierto$$

con el uso del operador *everywhere* descrito con corchetes $[X]$ que se leerá 'X es cierto para todos los estados' (o también, X *ptle*); el operador $[\circ]$ es un cuantificador universal

$$[\circ] : \mathcal{P} \to \mathcal{B}$$

EJEMPLO 1.0 Para el espacio de estados correspondiente a la declaración

$$u :\in \{-1, 0, 1\}$$

se verifica $[u \leq 1] \equiv Cierto$, que escribiremos simplemente $[u \leq 1]$. De la misma forma $[u > 1] \equiv Falso$. EJEMPLO

En general, dada una estructura arbitraria X, $[X]$ es o $Cierto$ o $Falso$, aunque X pueda tomar valores no constantes; esto se indica diciendo que X es una **estructura** booleana, mientras que $[X]$ es un **escalar** booleano. Puesto que $\mathcal{B}$ contiene dos escalares, existirán dos estructuras booleanas constantes:

$$Cierto, Falso : \mathcal{E} \to \mathcal{B}$$
$$[Cierto] \equiv Cierto$$
$$[Falso] \equiv Falso$$

De la misma forma que hemos sobrecargado los operadores lógicos, sobrecargamos $Cierto$ y $Falso$; esto permite considerar el operador $[\circ]$ sobre el espacio trivial $\mathcal{B}$ de forma que este operador es idempotente

$$[\,[[X]] \equiv [X]\,]$$

Ya que la equivalencia anterior es sobre el espacio trivial suprimimos los corchetes redundantes

$$[[X]] \equiv [X]$$

Los escalares booleanos son las únicas soluciones de la ecuación anterior.

Se podría definir el cálculo de predicados sobre un espacio de estados a partir de los operadores lógicos sobre estructuras booleanas para dar a continuación las propiedades de tales operadores como teoremas; por ejemplo, serían teoremas la ley de Augusto de Morgan

$$[\neg(X \vee Y) \equiv \neg X \wedge \neg Y]$$

o la regla de oro

$$[(X \wedge Y \equiv X) \equiv (Y \equiv X \vee Y)]$$

[Dijkstra y Scholten, 1990] sugieren un método alternativo muy original: partimos de axiomas para los operadores lógicos (compatibles con la lógica de $\mathcal{B}$) deduciremos como teoremas las identidades más importantes. De esta forma, nuestro modelo sobre un espacio de estados concreto (dado por las variables de un programa) quedará validado si demostramos que los operadores elementales verifican los axiomas correspondientes. En la Sección 1.6 volveremos a tratar el punto de vista tradicional.

La regla de Leibniz. Esquemas de demostración.

Pretendemos establecer la conservación de la igualdad con la aplicación. Si x e y denotan estructuras sobre un espacio de estados, la regla de Leibniz se escribe

$$x = y \;\Rightarrow\; \forall f :: f.x = f.y \qquad (leib)$$

o también, teniendo en cuenta la función identidad

$$x = y \equiv (\forall f :: f.x = f.y) \qquad (leib)$$

También se utilizará el principio de extensionalidad: dos funciones son iguales si y solo si toman los mismos valores; es decir:

$$f = g \equiv (\forall x :: f.x = g.x) \qquad (ext)$$

expresión dual de la regla de Leibniz.

Si X y X' son predicados, la igualdad de éstos se establece como $[X \equiv X']$. Si f es un transformador de predicados (una función de $\mathcal{P}$ en $\mathcal{P}$), aplicando la regla de Leibniz obtenemos

$$[X \equiv Y] \;\Rightarrow\; [f.X \equiv f.Y]$$

mientras que para la función $[f] : \mathcal{P} \to \mathcal{B}$ se tendría

$$[X \equiv Y] \;\Rightarrow\; [f.X] \equiv [f.Y]$$

Las implicaciones anteriores pueden describirse vía *esquemas de demostración*[1] en la forma siguiente:

$$\begin{array}{l} [f.X] \\ = \quad \because [X \equiv Y], leib \\ [f.Y] \end{array} \qquad\qquad \begin{array}{l} f.X \\ = \quad \because [X \equiv Y], leib \\ f.Y \end{array}$$

Obsérvese que con los esquemas anteriores se obtienen distintas conclusiones: con el primero obtenemos $[f.X] \equiv [f.Y]$, mientras que con el segundo obtenemos $[f.X \equiv f.Y]$; pero ambos usan la regla de Leibniz.

Una consecuencia importante de la regla de Leibniz.– Los esquemas de demostración del presente texto se basan en la propiedad de que los transformadores de predicados que verifican la regla de Leibniz son *sustitutivos*: si sustituimos parte de un predicado por un término equivalente (*ptle*) se obtiene otro predicado equivalente *ptle*. En particular, para la función identidad, se verifica

$$[X \equiv Y] \;\Rightarrow\; [X] \equiv [Y]$$

NOTA 1.1 *La implicación contraria no es cierta; por ejemplo, para el espacio de estados dado por la declaración $x :\in \mathcal{N}$ tenemos*

$$[x > 2] \equiv [x > 3]$$

ya que ambos coinciden con Falso, mientras que $[x > 2 \equiv x > 3]$ es Falso ya que para los estados dado $x = 3$ los predicados $x > 2$ y $x > 3$ tienen valores distintos.

[1] Seguiremos las recomendaciones de [Huet, 1990, Dijkstra, 1990] para describir esquemas.

La regla de Leibniz permite obtener

$$[X \equiv Y] \;\Rightarrow\; ([Y \equiv Z] \equiv [X \equiv Z])$$

de donde se deduce,

$$[X \equiv Y] \wedge [Y \equiv Z] \;\Rightarrow\; [X \equiv Z]$$

que es una versión débil de la *transitividad*, pero permite deducir la corrección del siguiente esquema de demostración de la identidad $[X \equiv Z]$ a partir de las identidades $[X \equiv Y]$ e $[Y \equiv Z]$:

$$\begin{array}{ll} & X \\ = & \quad \because [X \equiv Y] \\ & Y \\ = & \quad \because [Y \equiv Z] \\ & Z \end{array}$$

1.1. Equivalencia, conjunción e implicación

POSTULADO 1.2 (Equivalencia) *La equivalencia es asociativa y simétrica:*

$$[(X \equiv (Y \equiv Z)) \equiv ((X \equiv Y) \equiv Z)] \qquad (aso \equiv)$$

$$[(X \equiv Y) \equiv (Y \equiv X)] \qquad (sim \equiv)$$

El operador equivalencia es esencial y el primero que se introduce. De los sus postulados obtenemos

$$\begin{array}{ll} & [X \equiv (Y \equiv Y \equiv X)] \\ = & \quad \because aso \equiv \\ & [(X \equiv Y) \equiv (Y \equiv X)] \\ = & \quad \because sim \equiv \\ & Cierto \end{array}$$

y de la misma forma

$$[(X \equiv Y \equiv Y) \equiv X]$$

de donde el predicado $(Y \equiv Y)$ es neutro para la equivalencia (es el único neutro salvo equivalencias) y lo llamaremos $Cierto$, sobrecargando la correspondiente constante de $\mathcal{B}$,

$$[X \equiv Cierto \equiv X] \qquad (neu \equiv)$$

A partir de la siguiente prueba

$$\begin{array}{ll} & [X \equiv X] \\ = & \quad \because neu \equiv \\ & [X \equiv (\text{Cierto } \equiv X)] \\ = & \quad \because neu \equiv \\ & Cierto \end{array}$$

obtenemos la reflexividad

$$[X \equiv X] \qquad (ref \equiv)$$

POSTULADO 1.3 (Conjunción) *Postulamos para el operador $\wedge$ (conjunción), con más precedencia que $\equiv$, los axiomas de asociatividad, idempotencia y la pseudo-distributividad*

$$[X \wedge (Y \equiv Z) \equiv X \wedge Y \equiv Z \wedge X \equiv X] \qquad (dis \wedge \equiv)$$

Del axioma $dis \wedge \equiv$, tomando todas las letras iguales a X, se obtiene

$$[X \wedge Cierto \equiv X] \qquad (neu \wedge)$$

y tomando Y por Z se deduce la simetría

$$[X \wedge Y \equiv Y \wedge X] \qquad (sim \wedge)$$

Aplicando $dis \wedge \equiv$ dos veces, seguida de la asociatividad de $\equiv$, tenemos

$$[X \wedge (Y \equiv Z \equiv T) \equiv X \wedge Y \equiv Z \wedge X \equiv X \wedge T] \qquad (dis \wedge \equiv\equiv)$$

así como

$$\begin{array}{l} [(X \equiv Y) \wedge (Z \equiv T) \\ \equiv \\ X \wedge Z \equiv X \wedge T \equiv Y \wedge Z \equiv Y \wedge T \equiv X \equiv Y \equiv Z \equiv T] \end{array} \qquad (dis \wedge \equiv 2)$$

El hecho de que $[\,]$ es un cuantificador universal se captura con el siguiente

POSTULADO 1.4 *El operador $[\,]$ es conjuntivo: $[X \wedge Y] \equiv [X] \wedge [Y]$.*

DEFINICIÓN 1.5 (Implicación) *El operador $\Rightarrow$ queda definido por*[2]

$$[\, X \;\Rightarrow\; Y \;\equiv\; (X \wedge Y \equiv X)\,]$$

De la definición, junto a $(neu \wedge)$ obtenemos $[X \;\Rightarrow\; Cierto]$.

LEMA 1.6 *El operador $[\,]$ es monótono*

$$\forall X, Y :: [X \;\Rightarrow\; Y] \;\Rightarrow\; ([X] \;\Rightarrow\; [Y])$$

Demostración.–

$$\begin{array}{ll} & [X \;\Rightarrow\; Y] \\ = & \quad \text{⋱ Definición 1.5} \\ & [X \wedge Y \equiv X] \\ \Rightarrow & \quad \text{⋱ } leib^{3} \\ & [X \wedge Y] \equiv [X] \\ = & \quad \text{⋱ } [\,] \text{ es conjuntivo – Postulado 1.4} \\ & [X] \wedge [Y] \equiv [X] \\ = & \quad \text{⋱ Definición 1.5} \\ & [X] \;\Rightarrow\; [Y] \end{array}$$

LEMA

OBSERVACIÓN.– Definimos la $X \leq Y \;\doteq\; [X \;\Rightarrow\; Y]$. El lector puede comprobar que $\leq$ es una relación de orden sobre $\mathcal{P}$. El Lema 1.6 se interpreta diciendo el operador $[\,]$ conserva la relación $\leq$.

OBS

[2] El axioma que define el operador $\Rightarrow$ también se suele denominar *regla de oro*.

[3] Obsérvese que este paso es una implicación.

De la Definición 1.5 se deducen trivialmente las propiedades:

$$[X \Rightarrow X] \qquad [X \Rightarrow (Y \Rightarrow Z) \equiv X \wedge Y \Rightarrow Z]$$
$$[X \Rightarrow Cierto] \qquad [X \wedge (X \Rightarrow Y) \equiv X \wedge Y]$$
$$[X \wedge Y \Rightarrow X] \qquad [(X \Rightarrow Y) \equiv (Y \Rightarrow X) \equiv X \equiv Y]$$

y de ellas la transitividad de $\Rightarrow$; en efecto

$$\begin{array}{ll} & [(X \Rightarrow Y) \wedge (Y \Rightarrow Z) \Rightarrow (X \Rightarrow Z)] \\ = & \quad \because [X \Rightarrow (Y \Rightarrow Z) \equiv (X \wedge Y \Rightarrow Z)] \\ & [(X \Rightarrow Y) \wedge (Y \Rightarrow Z) \wedge X \Rightarrow Z] \\ = & \quad \because \text{asociativa, conmutativa} \\ & [X \wedge (X \Rightarrow Y) \wedge (Y \Rightarrow Z) \Rightarrow Z] \\ = & \quad \because [X \wedge (X \Rightarrow Y) \equiv X \wedge Y] \\ & [X \wedge Y \wedge (Y \Rightarrow Z) \Rightarrow Z] \\ = & \\ & [X \wedge Y \wedge Z \Rightarrow Z] \\ = & \quad \because [X \wedge Y \Rightarrow X] \\ & Cierto \end{array}$$

1.2. Sustitutividad y puntualidad

DEFINICIÓN 1.7 *Un transformador f se dice (fuertemente) sustitutivo si verifica*

$$[(X \equiv Y) \wedge f.X \equiv (X \equiv Y) \wedge f.Y] \qquad (sust)$$

Ejemplos de transformadores sustitutivos son las constantes y la identidad:

$$\begin{array}{ll} & [(X \equiv Y) \wedge X \equiv (X \equiv Y) \wedge Y] \\ = & \quad \because dis \wedge \equiv \\ & [(X \equiv Y) \wedge (X \equiv Y) \equiv (X \equiv Y)] \\ = & \quad \because idem \wedge \\ & Cierto \end{array}$$

Si tomamos dos funciones f y g sustitutivas arbitrarias, tenemos *ptle*:

$$\begin{array}{lll} & (X \equiv Y) \wedge (f.X \equiv g.X) & (1) \\ = & \quad \because dis \wedge \equiv & \\ & (X \equiv Y) \wedge f.X \equiv (X \equiv Y) \wedge g.X \equiv X \equiv Y & \\ = & \quad \because f \text{ y } g \text{ sustitutivas} & \\ & (X \equiv Y) \wedge f.Y \equiv (X \equiv Y) \wedge g.Y \equiv X \equiv Y & \\ = & \quad \because \text{idem} & \\ & (X \equiv Y) \wedge (f.Y \equiv g.Y) & (2) \end{array}$$

O sea: $[(1) \equiv (2)]$, y la función h dada por $h.X \doteq f.X \equiv g.X$ es sustitutiva. En particular, tomando f como la identidad, se obtiene que $\equiv$ es sustitutiva en sus argumentos. Además, *ptle*

$$\begin{array}{lll} & (X \equiv Y) \wedge (f.X \wedge g.X) & (3) \\ = & \quad \because asoc \wedge, idem \wedge, sim \wedge & \\ & (X \equiv Y) \wedge f.X \wedge (X \equiv Y) \wedge g.X & \\ = & \quad \because f \text{ y } g \text{ sustitutivas} & \end{array}$$

$$
\begin{array}{ll}
 & (X \equiv Y) \wedge f.Y \wedge (X \equiv Y) \wedge g.Y \\
= & \quad \because \text{idem} \\
 & (X \equiv Y) \wedge (f.Y \wedge g.Y) \qquad (4)
\end{array}
$$

de donde $[(3) \equiv (4)]$, y la función $h.X \doteq f.X \wedge g.X$ es sustitutiva. En definitiva, hemos demostrado

LEMA 1.8 *Se obtienen transformadores sustitutivos combinando variables y constantes con los operadores $\wedge$ y $\equiv$. En particular, la implicación es sustitutiva.*

La sustitutividad es una herramienta muy útil en la demostración de otras propiedades; por ejemplo, la transitividad de $\equiv$

$$
\begin{array}{ll}
 & [(X \equiv Y) \wedge (Y \equiv Z) \;\Rightarrow\; (X \equiv Z)] \\
= & \quad \because \text{definición } \Rightarrow \text{ (regla de oro)} \\
 & [(X \equiv Y) \wedge (Y \equiv Z) \wedge (X \equiv Z) \equiv (X \equiv Y) \wedge (Y \equiv Z)] \\
= & \quad \because f.X == (Y \equiv Z), \text{ y } (X \equiv Z) \text{ es sustitutiva} \\
 & [(X \equiv Y) \wedge (Y \equiv Z) \wedge (Y \equiv Z) \equiv (X \equiv Y) \wedge (Y \equiv Z)] \\
= & \quad \because idem \wedge \\
 & Cierto
\end{array}
$$

LEMA 1.9 $[(X \;\Rightarrow\; Y) \wedge (Y \;\Rightarrow\; X) \equiv X \equiv Y]$

Demostración.– En efecto, *ptle*

$$
\begin{array}{ll}
 & (X \;\Rightarrow\; Y) \wedge (Y \;\Rightarrow\; X) \\
= & \quad \because \text{definición } \Rightarrow \text{ (regla de oro)} \\
 & (X \wedge Y \equiv X) \wedge (X \wedge Y \equiv Y) \\
= & \quad \because \text{sustitutividad} \\
 & (X \wedge Y \equiv X) \wedge (X \equiv Y) \\
= & \quad \because \text{sustitutividad} \\
 & (X \wedge X \equiv X) \wedge (X \equiv Y) \\
= & \quad \because idem \wedge, neu \equiv \\
 & X \equiv Y
\end{array}
$$

LEMA

Diferencia entre sustitutividad y regla de Leibniz.– Por la pseudo-distributividad la sustitutividad equivale a

$$[(X \equiv Y) \wedge (f.X \equiv f.Y) \equiv X \equiv Y]$$

y también a

$$[(X \equiv Y) \;\Rightarrow\; (f.X \equiv f.Y)]$$

expresión más fuerte que la regla de Leibniz: $[X \equiv Y] \;\Rightarrow\; [f.X \equiv f.Y]$. Dijkstra llama *puntuales* a las funciones que verifican la implicación

$$[x = y \;\Rightarrow\; f.x = f.y] \qquad (punt)$$

Por tanto, las funciones sustitutivas son puntuales. Es más, la composición de funciones puntuales es puntual, así como las constantes y la identidad, por lo que los resultados anteriores son un caso particular.

1.3. La disyunción y la negación

DEFINICIÓN 1.10 *El operador* $\vee$ *se define como aquel que verifica la* regla de oro

$$[X \wedge Y \equiv X \equiv Y \equiv X \vee Y] \qquad (oro)$$

De tal definición se deduce que el operador $\vee$ es simétrico, idempotente, admite a $Cierto$ como elemento absorbente:

$$[X \vee Cierto \equiv Cierto] \qquad (abs\vee)$$

y distribuye a la equivalencia

$$[X \vee (Y \equiv Z) \equiv X \vee Y \equiv X \vee Z] \qquad (dis\vee \equiv)$$

Si calculamos $X \vee (Y \vee Z)$ utilizando la regla de oro dos veces tenemos

$$[X \vee (Y \vee Z) \equiv X \wedge (Y \wedge Z) \equiv X \wedge Y \equiv X \wedge Z \equiv Y \wedge Z \equiv X \equiv Y \equiv Z]$$

y permutando las letras

$$[Z \vee (X \vee Y) \equiv Z \wedge (X \wedge Y) \equiv X \wedge Y \equiv X \wedge Z \equiv Y \wedge Z \equiv X \equiv Y \equiv Z]$$

de donde, debido a la asociatividad y simetría de $\wedge$, tenemos la asociatividad de $\vee$. La propiedades distributivas de $\vee$ respecto de $\wedge$ se deducen fácilmente ya que, *ptle*

$$\begin{array}{ll} & (X \wedge Y) \vee (X \wedge Z) \\ = & \quad \because oro \\ & (X \wedge Y) \wedge (X \wedge Z) \equiv X \wedge Y \equiv X \wedge Z \\ = & \quad \because aso\wedge, idem\wedge, sim\wedge \\ & X \wedge (Y \wedge Z) \equiv X \wedge Y \equiv X \wedge Z \\ = & \quad \because dis\wedge \equiv\equiv \\ & X \wedge (Y \wedge Z \equiv Y \equiv Z) \\ = & \quad \because oro \\ & X \wedge (Y \vee Z) \end{array}$$

Es decir, se verifica

$$[\,(X \wedge Y) \vee (X \wedge Z) \equiv X \wedge (Y \vee Z)\,] \qquad (dis\vee\ \wedge)$$

Además, *ptle*

$$\begin{array}{ll} & X \vee (Y \wedge Z) \\ = & \quad \because oro \\ & X \wedge (Y \wedge Z) \equiv X \equiv Y \wedge Z \\ = & \quad \because dis\vee \equiv, aso\vee \\ & X \vee Y \vee Z \equiv X \vee Y \equiv X \vee Z \\ = & \quad \because idem\vee \\ & (X \vee Y) \vee (X \vee Z) \equiv X \vee Y \equiv X \vee Z \\ = & \quad \because oro \\ & (X \vee Y) \wedge (X \vee Z) \end{array}$$

y hemos obtenido

$$[\, X \vee (Y \wedge Z) \equiv (X \vee Y) \wedge (X \vee Z) \,] \qquad (dis \wedge \ \vee)$$

Otras propiedades inmediatas son las de *absorción*:

$$[X \vee (X \wedge Y) \equiv X] \qquad [X \wedge (X \vee Y) \equiv X] \qquad (absor)$$

El operador de negación $\neg$ se puede definir de varias formas. Dijkstra lo introduce con dos axiomas:

$$[\neg(X \equiv Y) \equiv \neg X \equiv Y] \qquad [\neg X \vee X]$$

Quizás sea más simple con un sólo axioma,

DEFINICIÓN 1.11 (Negación, Falso) *El operador $\neg$ queda definido por el axioma*

$$[X \wedge Y \equiv X \equiv \neg X \vee Y] \qquad (neg)$$

y la constante $Falso$ queda definida en la forma $[Falso \equiv \neg Cierto]$.

Una definición alternativa al axioma (neg) es el axioma equivalente

$$[X \ \Rightarrow \ Y \equiv \neg X \vee Y] \qquad (neg)$$

Para los escalares booleanos se verifica

$$[\neg b] \equiv \neg b \qquad [b] \equiv b$$

Del axioma (neg) deducimos la **ley del tercio excluido**

$$[\neg X \vee X \equiv Cierto]$$

y $Falso$ es neutro para la disyunción

$$[Falso \vee X \equiv X] \qquad (abs\vee)$$

Del esquema

$$\begin{array}{ll} & \neg\neg X \vee Y \\ = & \quad \because \text{definición} \\ & \neg X \wedge Y \equiv \neg X \\ = & \quad \because oro \\ & \neg X \vee Y \equiv Y \\ = & \quad \because \text{definición} \\ & X \wedge Y \equiv X \equiv Y \\ = & \quad \because oro \\ & X \vee Y \end{array}$$

tomando $Y == Falso$ y aplicando $(abs\vee)$ se deduce que $\neg$ es su propio inverso

$$[\neg\neg X \equiv X]$$

y en particular $[\neg Falso \equiv Cierto]$. Entonces, es fácil probar las propiedades:

$$\begin{array}{lll} [Falso \wedge X \equiv Falso] & [Falso \ \Rightarrow \ X] & [\neg X \equiv (X \equiv Falso)] \\ \neg[X] \vee [X] \equiv Cierto & \neg[X] \wedge [X] \equiv Falso & \end{array}$$

y de aquí que el operador $\neg X$ sea sustitutivo. Veamos, por ejemplo, la última:

$$
\begin{array}{ll}
 & \neg[X] \wedge [X] \\
= & \quad \because \text{definición} \\
 & [X] \equiv \neg[X] \vee \neg[X] \\
= & \quad \because idem\vee, \neg[X] \equiv [X] \equiv Falso \\
 & [X] \equiv \neg[X] \\
= & \\
 & Falso
\end{array}
$$

A partir de las identidades anteriores es fácil probar las leyes de Augusto de Morgan

$$[\neg X \vee \neg Y \equiv \neg(X \wedge Y)] \qquad [\neg X \wedge \neg Y \equiv \neg(X \vee Y)] \qquad (de\ Morgan)$$

y la regla de intercambio

$$[\,(X \wedge Y \;\Rightarrow\; Z) \;\equiv\; (X \;\Rightarrow\; \neg Y \vee Z)] \qquad (inter)$$

así como las propiedades

$$[(X \;\Rightarrow\; Y) \vee (Z \;\Rightarrow\; T) \equiv X \wedge Y \;\Rightarrow\; Z \vee T]$$
$$[(X \;\Rightarrow\; Z) \wedge (Y \;\Rightarrow\; Z) \equiv X \vee Y \;\Rightarrow\; Z]$$

Veamos por ejemplo la segunda ley $de\ Morgan$; tenemos, *ptle*

$$
\begin{array}{ll}
 & \neg X \wedge \neg Y \\
= & \\
 & (X \equiv Falso) \wedge (Y \equiv Falso) \\
= & \quad \because dis \wedge \equiv 2, [Falso \wedge \_ = Falso] \\
 & X \wedge Y \equiv Falso \equiv Falso \equiv Falso \equiv X \equiv Y \equiv Falso \equiv Falso \\
= & \quad \because neu \equiv \\
 & X \wedge Y \equiv X \equiv Y \equiv Falso \\
= & \quad \because \text{oro} \\
 & X \vee Y \equiv Falso \\
= & \\
 & \neg(X \vee Y)
\end{array}
$$

OBSERVACIÓN.– El operador $[\,]$ no es disyuntivo como muestra el ejemplo siguiente (sobre el espacio de los naturales):

$$[x > 2 \vee x \leq 2] \equiv Cierto$$

$$[x > 2] \vee [x \leq 2] \equiv Falso$$

pero sí se cumplen las siguientes propiedades. OBS

LEMA 1.12 *En general, el operador* $[\,]$ *no es disyuntivo, pero verifica*

(i) $[X] \vee [Y] \;\Rightarrow\; [X \vee Y]$
(ii) $[X \vee b] \equiv [X] \vee b$
(iii) $[[X] \;\Rightarrow\; X]$
(iv) $[f.Cierto] \wedge [f.Falso] \;\Rightarrow\; [f.X]$, *si* f *es sustitutivo*

Demostración.–

(i):

$$
\begin{array}{ll}
 & [X] \vee [Y] \;\Rightarrow\; [X \vee Y] \\
= & \\
 & ([X] \;\Rightarrow\; [X \vee Y]) \wedge ([Y] \;\Rightarrow\; [X \vee Y]) \\
\Leftarrow & \quad \because\, leib \\
 & ([X \;\Rightarrow\; X \vee Y]) \wedge ([Y \;\Rightarrow\; X \vee Y]) \\
= & \\
 & Cierto
\end{array}
$$

(iii):

$$
\begin{array}{ll}
 & [[X] \;\Rightarrow\; X] \\
= & \\
 & [\neg[X] \vee X] \\
= & \quad \because \neg[X] \text{ es booleano} \\
 & [\neg[X] \vee [X]] \\
= & \quad \because \text{tercio excluido} \\
 & Cierto
\end{array}
$$

(ii):

$$
\begin{array}{ll}
 & [X \vee b] \equiv [X] \vee b \\
= & \\
 & ([X \vee b] \;\Rightarrow\; [X] \vee b) \wedge ([X] \vee [b] \;\Rightarrow\; [X \vee b]) \\
= & \quad \because [[X] \vee [b] \;\Rightarrow\; [X \vee b]) \equiv Cierto], \text{ ley intercambio} \\
 & \neg b \wedge [X \vee b] \;\Rightarrow\; [X] \\
= & \quad \because b \text{ booleana} \\
 & [\neg b] \wedge [X \vee b] \;\Rightarrow\; [X] \\
= & \quad \because [\,] \text{ es conjuntivo} \\
 & [\neg b \wedge (X \vee b)] \;\Rightarrow\; [X] \\
\Leftarrow & \quad \because [\,] \text{ monótono} \\
 & [\neg b \wedge (X \vee b) \;\Rightarrow\; X] \\
= & \quad \because dis \wedge \vee, neu \\
 & [\neg b \wedge X \;\Rightarrow\; X] \\
= & \\
 & Cierto
\end{array}
$$

(iv): Sea f es transformador sustitutivo; entonces, *ptle*

$$
\begin{array}{ll}
 & f.X \\
= & \quad \because \text{tercio excluido, distributividad} \\
 & X \wedge f.X \vee \neg X \wedge f.X \\
= & \quad \because neu \\
 & (X \equiv Cierto) \wedge f.X \vee (X \equiv Falso) \wedge f.X \\
= & \quad \because f \text{ es sustitutivo} \\
 & (X \equiv Cierto) \wedge f.Cierto \vee (X \equiv Falso) \wedge f.Falso \\
= & \quad \because \text{razonando como antes} \\
 & X \wedge f.Cierto \vee \neg X \wedge f.Falso \\
= & \quad \because [f.Cierto], [f.Falso], \text{ idem} \\
 & X \wedge Cierto \vee \neg X \wedge Cierto \\
= & \quad \because \text{tercio excluido} \\
 & Cierto
\end{array}
$$

LEMA

NOTA 1.13 (Pruebas a partir de una tabla de verdad) *La propiedad Lema 1.12(iv) recuerda la regla de comprobación de identidades del cálculo de predicados a partir de una tabla de verdad. En particular, para la función (sustitutiva)*

$$f.X \;\doteq\; \neg X \vee \neg Y \equiv \neg(X \wedge Y)$$

tendremos $[f.Cierto \equiv Cierto]$ y $[f.Falso \equiv Cierto]$, de donde $[f.X]$, que sería una prueba de la ley de De Morgan.

Otro ejemplo: probemos $[M \;\Rightarrow\; \neg B \vee B \wedge M]$; sea $f.Z \;\doteq\; M \;\Rightarrow\; \neg Z \vee Z \wedge M$. Entonces, es fácil obtener $[f.Cierto \equiv Cierto]$, y $[f.Falso \equiv Cierto]$; ya que f es sustitutivo, aplicamos Lema 1.12(d), para obtener $[f.Z]$, y de aquí $[M \;\Rightarrow\; \neg Z \vee Z \wedge M]$. Obviamente encontramos una prueba directa por aplicación de la regla de intercambio.

Probemos, a modo de ejercicio, el siguiente

LEMA 1.14 *Se verifica la siguiente identidad, ptle:*

$$(b \;\Rightarrow\; P) \wedge (\neg b \;\Rightarrow\; Q) \equiv b \wedge P \vee \neg b \wedge Q$$

Demostración.– Aplicamos el Lema 1.12(d) con $f.X \doteq (b \;\Rightarrow\; P) \wedge (\neg b \;\Rightarrow\; Q) \equiv b \wedge P \vee \neg b \wedge Q$. LEMA

OBSERVACIÓN.– En general, el operador $[\,]$ no es sustitutivo; si lo fuera

$$[(X \equiv Y) \wedge [X] \equiv (X \equiv Y) \wedge [Y]]$$

tomando $Y == Cierto$, llegamos a $[X \wedge [X] \;\equiv\; X]$, que no se cumple, por ejemplo, para $X == x > 1$ con $x :\in \mathcal{N}$. OBS

1.4. Cuantificadores

El cuantificador universal Una estructura cuantificada universalmente la escribimos como $\forall x :: f.x$, donde los :: indican un predicado/rango igual a Cierto. La variable x es una variable *muda* y consideraremos la identificación:

$$[(\forall x :: f.x) \equiv (\forall y :: f.y)]$$

Para dos variables se considera $[(\forall x :: (\forall y :: g.x.y) \equiv (\forall y :: (\forall x :: g.x.y)]$, y podemos definir:

$$[(\forall x, y :: g.x.y) \equiv (\forall y :: (\forall x :: g.x.y)]$$

Para un rango no idénticamente $Cierto$ consideraremos la

DEFINICIÓN 1.15 $[(\forall x : r.x : f.x) \equiv (\forall x :: r.x \;\Rightarrow\; f.x)]$

Se observa que el rango genera una implicación y deducimos la siguiente regla de intercambio entre rangos

$$[(\forall x : r.x : s.x \vee f.x) \equiv (\forall x :: \neg s.x : \neg r.x \vee f.x)]$$

De ciertos postulados o axiomas para el cuantificador $\forall$ deduciremos las propiedades más útiles en el cálculo.

DEFINICIÓN 1.16 (Axiomas para el cuantificador $\forall$)

$$[X \vee (\forall x :: f.x) \equiv (\forall x :: X \vee f.x)] \qquad (dis\forall)$$
$$[(\forall x :: f.x) \wedge (\forall x :: g.x) \equiv (\forall x :: f.x \wedge g.x)] \qquad (con\forall)$$
$$[(\forall x : x = y : f.x) \equiv f.y] \qquad (punt)$$

El último axioma se llamará *regla puntual* (*one-point rule*) y en ésta es necesario que las variables x e y sean del mismo tipo. El siguiente teorema es consecuencia inmediata de la definición.

TEOREMA 1.17 *Se verifican las siguientes identidades, ptle*

(i) $X \vee (\forall x : r.x : f.x) \equiv (\forall x : r.x : X \vee f.x)$
(ii) $(\forall x : r.x : Cierto) \equiv Cierto$
(iii) $(\forall x : Falso : f.x) \equiv Cierto$
(iv) $(\forall x : [x = y] : Falso) \equiv Falso$
(v) $(\forall x : r.x : f.x) \wedge (\forall x : r.x : g.x) \equiv (\forall x : r.x : f.x \wedge g.x)$
(vi) $(\forall x : r.x : f.x) \wedge (\forall x : s.x : f.x) \equiv (\forall x : r.x \vee s.x : f.x)$
(vii) $(\forall x : r.x : (\forall y : s.y : g.x.y) \equiv (\forall y : s.y : (\forall x : r.x : g.x.y)$
$(viii)$ $(\forall x : r.x : f.x \equiv g.x) \Rightarrow (\forall x : r.x : f.x) \equiv (\forall x : r.x : g.x)$
(ix) $(\forall x : r.x : f.x \Rightarrow g.x) \Rightarrow ((\forall x : r.x : f.x) \Rightarrow (\forall x : r.x : g.x))$

La prueba del siguiente teorema ilustra el cálculo con cuantificadores.

TEOREMA 1.18 *Se verifican las siguientes propiedades:*

(i) $[(\forall x : x \in W : Falso) \equiv W = \emptyset]$
(ii) $[(\forall x : x \in W : X) \equiv X \vee W = \emptyset]$
(iii) $W \neq \emptyset \Rightarrow [(\forall x : x \in W : X \wedge f.x) \equiv X \wedge (\forall x : x \in W : f.x)]$

Demostración.– Demostraremos (i) en los casos $W \neq \emptyset$ y $W = \emptyset$; tenemos, *ptle*

✓ si $W = \emptyset$:

$\forall x : x \in W : Falso$
$=$ $\quad \because leib, W = \emptyset$
$\forall x : x \in \emptyset : Falso$
$=$ $\quad \because x \in \emptyset = Falso$
$\forall x : Falso : Falso$
$=$ $\quad \because$ Teorema 1.17(iii)
$Cierto$

✓ si $W \neq \emptyset$, con $y \in W$:

$\forall x : x \in W : Falso$
$=$ $\quad \because$ Teorema 1.17(vi), CP
$\forall x : x \in W : Falso$
$\wedge$
$\forall x : x \in W : \neg(x = y)$
$=$ $\quad \because$ intercambio rango
$\forall x : x \in W : Falso$
$\wedge$
$\forall x : x = y : x \notin W$
$=$ $\quad \because punt$ con $f.x == x \notin W$
$(\forall x : x \in W : Falso) \wedge y \notin W$
$=$ $\quad \because y \in W$
$Falso$

(ii):

$\forall x : x \in W : X$
$=$ $\quad \because$ CP
$\forall x : x \in W : X \vee Falso$
$=$ $\quad \because$ Teorema 1.17(i)
$X \vee \forall x : x \in W : Falso$
$=$ $\quad \because$ Teorema 1.18(i)
$X \vee W = \emptyset$

TEOREMA

(iii):

$\forall x : x \in W : X \wedge f.x$
$=$ $\quad \because$ Teorema 1.17(v)
$(\forall x : x \in W : X) \wedge (\forall x : x \in W : f.x)$
$=$ $\quad \because$ Teorema 1.18(i)
$(X \vee W = \emptyset) \wedge (\forall x : x \in W : f.x)$
$=$ $\quad \because$ si $W \neq \emptyset$
$X \wedge (\forall x : x \in W : f.x)$

El cuantificador existencial se introduce con las reglas

$$[(\exists x :: f.x) \equiv \neg(\forall x : \neg f.x)] \qquad [(\exists x : r.x : f.x) \equiv \exists x :: r.x \wedge f.x)]$$

Es fácil probar el siguiente teorema.

TEOREMA 1.19 *Se verifican las siguientes identidades, ptle*

(i) $X \wedge (\exists x : r.x : f.x) \equiv (\exists x : r.x : X \wedge f.x)$
(ii) $(\exists x : r.x : Falso) \equiv Falso$
(iii) $(\exists x : Falso : f.x) \equiv Falso$
(iv) $(\exists x : x = y : f.x) \equiv f.y$ — regla puntual
(v) $(\exists x : r.x : f.x) \vee (\exists x : r.x : g.x) \equiv (\exists x : r.x : f.x \vee g.x)$
(vi) $(\exists x : r.x : f.x) \wedge (\exists x : s.x : f.x) \equiv (\exists x : r.x \vee s.x : f.x)$

1.5. Conjuntos bien construidos

Aunque este apartado será utilizado a partir del capítulo segundo, lo incluimos aquí ya que utiliza propiedades del cálculo de predicados antes descrito; las ideas fundamentales están extraídas de [Dijkstra y Scholten, 1990]:174–189.

En lo sucesivo se considera un conjunto $(\mathcal{D}, \leq)$ parcialmente ordenado.

DEFINICIÓN 1.20 (Conjunto bien construido) *Un subconjunto $\mathcal{C} \subseteq \mathcal{D}$ se dice bien construido (well-founded) si verifica, $\forall \mathcal{S} : \mathcal{S} \subseteq \mathcal{D}$,*

$$(\exists x :: x \in \mathcal{S} \cap \mathcal{C}) \equiv \mathcal{C} \cap \mathcal{S} \text{ tiene elemento mínimo} \qquad (bc)$$

Recordemos que

$$x \text{ es mínimo de } A \;\equiv\; x \in A \wedge (\forall y : y < x : y \notin A)$$

Podemos manipular brevemente el predicado (bc) para eliminar la noción de mínimo; así, $\forall \mathcal{S} : \mathcal{S} \subseteq \mathcal{D}$:

$(\exists x :: x \in \mathcal{S} \cap \mathcal{C}) \equiv \mathcal{C} \cap \mathcal{S}$ tiene elemento mínimo
$=$ $\quad \because$ izquierda: intercambio en rango; derecha: definición de mínimo
$(\exists x : x \in \mathcal{S} : x \in \mathcal{C}) \equiv (\exists x : x \in \mathcal{C} \cap \mathcal{S} \wedge (\forall y : y < x : y \notin \mathcal{C} \cap \mathcal{S}))$
$=$ $\quad \because$ intercambio en rango
$\equiv$ $\exists x : x \in \mathcal{C} : x \in \mathcal{S}$
$\exists x : x \in \mathcal{C} : x \in \mathcal{S} \wedge (\forall y : y \in \mathcal{C} \wedge y < x : y \notin \mathcal{S})$

DEFINICIÓN 1.21 (Inducción sobre un conjunto) *Se dice que la propiedad de inducción es válida sobre $\mathcal{C}$ (para simplificar, $\mathcal{C}$ es inductivo) si se tiene, para cualquier predicado $f : \mathcal{D} \to \mathcal{B}$,*

$$\begin{array}{l} \forall x : x \in \mathcal{C} : f.x \\ \equiv \\ \forall x : x \in \mathcal{C} : (\forall y : y \in \mathcal{C} \wedge y < x : f.y) \;\Rightarrow\; f.x \end{array} \qquad (ci)$$

Las definiciones anteriores son equivalentes:

TEOREMA 1.22 *$\mathcal{C}$ es bien construido sii la inducción sobre $\mathcal{C}$ es válida*

Demostración.– Para probar $(ci) \equiv (bc)$ observamos que a cada función $f : \mathcal{D} \to \mathcal{B}$ le podemos hacer corresponder el subconjunto $\mathcal{S} = \{x \in \mathcal{D} \mid \neg f.x\}$, y recíprocamente; por tanto tenemos:

(bc)

$$
\begin{array}{ll}
= & \\
 & \forall f : f : \mathcal{D} \to \mathcal{B} : (\exists x : x \in \mathcal{C} : \neg f.x) \equiv \\
 & \qquad (\exists x : x \in \mathcal{C} : \neg f.x \wedge (\forall y : y \in \mathcal{C} \wedge y < x : f.y)) \\
= & \quad \because \text{cálculo de predicados} \\
 & \forall f : f : \mathcal{D} \to \mathcal{B} : (\forall x : x \in \mathcal{C} : f.x) \equiv \\
 & \qquad (\forall x : x \in \mathcal{C} : f.x \vee \neg(\forall y : y \in \mathcal{C} \wedge y < x : f.y)) \\
= & \quad \because \text{definición de } \Rightarrow \\
 & (ci)
\end{array}
$$

TEOREMA

TEOREMA 1.23 *$\mathcal{C}$ es bien construido sii toda cadena decreciente en $\mathcal{C}$ es finita.*

Demostración.– Demostremos que $\mathcal{C}$ no es bien construido sii existe una cadena decreciente en $\mathcal{C}$ infinita:

$$
\begin{array}{ll}
 & \mathcal{C} \text{ no es bien construido} \\
= & \quad \because \text{neg. cuantificador, intercambio en rango} \\
 & \exists \mathcal{S} : \mathcal{S} \subseteq \mathcal{D} : (\exists x : x \in \mathcal{C} \cap \mathcal{S} : Cierto) \not\equiv \\
 & \qquad (\exists x : x \in \mathcal{C} \cap \mathcal{S} : (\forall y : y < x : y \notin \mathcal{S} \cap \mathcal{C})) \\
= & \quad \because (A \;\Rightarrow\; B) \;\Rightarrow\; (A \not\equiv B) \equiv B \wedge \neg A \\
= & \quad \because (\exists x : r.x : f.x) \;\Rightarrow\; (\exists x : r.x : Cierto) \\
 & (\exists \mathcal{S} : \mathcal{C} \subseteq \mathcal{S} : (\exists x : x \in \mathcal{C} \cap \mathcal{S} : Cierto) \wedge \\
 & \qquad (\forall x : x \in \mathcal{C} \cap \mathcal{S} : (\exists y : y < x : y \in \mathcal{S} \cap \mathcal{C})) \qquad (nbc)
\end{array}
$$

Si $\mathcal{C}$ contiene una cadena decreciente infinita $\{x_n\}$, tomando $\mathcal{S} = \{x_n\} = \mathcal{C} \cap \mathcal{S}$ se obtiene el predicado (nbc). Recíprocamente; supongamos (nbc); entonces existe $x_0 \in \mathcal{C} \cap \mathcal{S}$. En general, dado $x_n \in \mathcal{C} \cap \mathcal{S}$, debe darse

$$(\exists y : y < x_n : y \in \mathcal{S} \cap \mathcal{C})$$

de donde existe un $x_{n+1} \in \mathcal{C} \cap \mathcal{S}$ tal que $x_{n+1} < x_n$ y de aquí se obtiene una cadena de $\mathcal{C}$ decreciente e infinita. TEOREMA

EJEMPLO 1.24 (Una curiosa aplicación del concepto de conjunto inductivo) Tratemos de probar que el siguiente juego termina:

> Dado un saco finito de números naturales (por ejemplo, un saco de bolas cada una numerada con un natural) un movimiento consiste en extraer una bola x y añadir un saco de bolas (con un número arbitrario de ellas) con números $< x$.

La solución al problema la da el siguiente razonamiento; supongamos que n es una cota de los índices de las bolas del saco: todas las bolas aparecen numeradas con un número $\leq n$). El estado del saco viene determinado por un juego de frecuencias $\mathcal{F} \doteq (f_n, \ldots, f_0) \in \mathbb{N}^{n+1}$, siendo f_i el número de bolas numeradas con i; consideremos ahora el orden lexicográfico sobre $\mathbb{N}^{n+1}$, resultando un conjunto bien construido; ahora bien, el estado $\mathcal{F}'$ después de un movimiento verifica $\mathcal{F}' < \mathcal{F}$ ya que si se extrae una bola de índice x, se tiene:

$$(\forall j : j > x : f_j = f'_j) \wedge f'_x = f_x - 1$$

por consiguiente toda secuencia de juegos de frecuencias es finita, por ser $\mathbb{N}^{n+1}$ bien construido. EJEMPLO

1.6. Programas y Algebras

Teoría sobre un Algebra

Las expresiones de un lenguaje de programación imperativo se construyen a partir de operadores y variables. Por ejemplo, las expresiones enteras se construyen con el uso de los operadores $+$, $*$, mientras que las lógicas a través de los operadores $=$, no (negación), etc. Tales expresiones se consideran sintácticas, aunque tienen asignados valores dependiendo de la representación de los datos utilizados. Así, en un programa pueden aparecer símbolos de constantes, como $cero$, $falso$, y expresiones en las que intervienen funciones, tales como $suma(sucesor\ cero, cero)$, o $no(x = cero)$, para representar las operaciones $1+0$ o $\neg(valor\_de\_x = 0)$ en el álgebra de los enteros y los booleanos. Así, un lenguaje de programación tiene asociada una signatura o conjunto de símbolos de funciones y predicados.

Una *signatura* Σ es un conjunto de símbolos de función y símbolos de predicados conjuntamente con su signatura (número de argumentos)

$$\Sigma = \{f : s^n \to s, p : s^n\}$$

Los objetos con signatura nula son las constantes.

Dado un conjunto $\mathcal{V}$ de variables, el conjunto $\mathcal{T}$ de Σ–términos se define en forma inductiva

$$\begin{array}{rcll} t & ::= & v & \text{— } v \in \mathcal{V} \\ & | & f(t, \ldots, t) & \text{— si } f \text{ es un símbolo de función} \end{array}$$

El conjunto $\mathcal{P}$ de predicados o Σ–fórmulas sobre la signatura se define también en forma inductiva

$$\begin{array}{rcl} \varphi & ::= & p(t, \ldots, t) \quad \text{— predicados simples} \\ & | & \neg\varphi \mid \varphi \wedge \varphi \mid \varphi \vee \varphi \mid \varphi \Rightarrow \varphi \mid \varphi \equiv \varphi \mid \forall x :: \varphi \mid \exists x :: \varphi \end{array}$$

Programas sobre una signatura. Σ-álgebra.

Para cada signatura Σ con conjunto de variables $\mathcal{V}$, conjunto de términos $\mathcal{T}$, y conjunto de predicados $\mathcal{P}$, se puede considerar un conjunto de sentencias, descritos por ejemplo con la sintaxis

$$S ::= x := t \mid [\![b \to S \,\square\, T]\!] \mid *[\![b \to S]\!] \mid S; T$$

donde $t \in \mathcal{T}$ y $b \in \mathcal{P}$. Y recíprocamente. Por ejemplo, para un lenguaje de programación que utilice solamente las constantes lógicas ($cierto, falso$), la constante $cero$, las variables x, y, b, y las operaciones $suma$, $\neg$, -", nos bastará con la signatura

$$\Sigma \doteq \{suma : s^2 \to s, cero : s, cierto : s^0, falso : s^0, \ldots\}$$

con la cual se pueden escribir términos y fórmulas: $suma(cero, x)$ $x = cero$.

Normalmente, la representación de los datos y operaciones se realiza a través de una interpretación o álgebra. Una Σ–interpretación o Σ–álgebra $\mathcal{A}$

consiste en una asignación de funciones y predicados sobre un dominio $\mathbf{A} = dom\ \mathcal{A}$. Tal interpretación asigna a cada símbolo de función f de aridad n una función

$$f_A : \mathbf{A}^n \to \mathbf{A}$$

o una constante de $\mathbf{A}$ si la aridad es cero, y a cada símbolo de predicado p de aridad n una relación

$$p_A \subseteq \mathbf{A}^n$$

o un valor booleano si la aridad es cero.

La memoria en los lenguajes imperativos. Entornos

Los lenguajes imperativos están caracterizados por utilizar una memoria que recuerda el estado de las variables de forma que

- ✓ La memoria es esencial para la evaluación de sentencias y expresiones.
- ✓ La memoria es un medio de comunicación entre sentencias: el comportamiento de una sentencia depende de la memoria, que a su vez depende de las sentencias ejecutadas con anterioridad.

Desde un punto de vista operacional, las estructuras de un lenguaje imperativo quedan caracterizadas por ciertas relaciones de transición en las que intervienen las sentencias y la memoria. Veamos en primer lugar como se modela la memoria. El conjunto de variables de un programa es visto como una función que asocia a cada símbolo de variable su valor; el conjunto de valores de todas las variables en cierto instante es un *entorno*, y tal entorno permite capturar el estado del sistema (la memoria) en cierto punto de un programa; por ejemplo, para el programa:

$$\begin{array}{l} var\ x, y : integer; z : real; b : boolean; \\ x := 1; y := x + 1; z := -1{,}3; b := true; \\ \{E_1\} \end{array}$$

En el punto $\{E_1\}$ el entorno ρ será

$$\rho.x = 1 \qquad \rho.y = 2 \qquad \rho.z = -1{,}3 \qquad \rho.b = true$$

o también

$$x \overset{\rho}{\mapsto} 1 \qquad y \overset{\rho}{\mapsto} 2 \qquad z \overset{\rho}{\mapsto} -1{,}3 \qquad b \overset{\rho}{\mapsto} true,$$

Dada una Σ–álgebra $\mathcal{A}$, un estado o entorno ρ es una función $\rho : \mathcal{V} \to \mathbf{A}$ que asocia a cada símbolo de variable un valor de un subconjunto $Dom.v \subseteq \mathbf{A}$; tal conjunto $Dom.v$ queda determinado por el tipo de la variable. Denotaremos con $\mathcal{E}$ el conjunto de todos los entornos.

Los entornos permiten asignar valores a los términos que tienen variables ya que cada término t es visto como una función $t : \mathcal{E} \to \mathbf{A}$ que se define en forma inductiva

$$\begin{array}{lcll} v.\rho & \doteq & \rho.v & \text{— para variables} \\ f(t_1, \ldots, t_n).\rho & \doteq & f_A(t_1.\rho, \ldots, t_n.\rho) & \text{— para términos} \end{array}$$

Los entornos también permiten asignar valores a las fórmulas en forma inductiva y utilizando el cálculo de $\mathcal{B}$; los casos base corresponden a símbolos

constantes, cuyos valores no dependen del entorno. Para los símbolos de predicados definimos

$$p(t_1, \ldots, t_n).\rho \doteq Cierto, \text{ si } (t_1.\rho, \ldots, t_n.\rho) \in p_A$$

y para el resto de fórmulas

$$\begin{array}{lcl}
(\neg\varphi).\rho & \doteq & \neg(\varphi.\rho) \\
(\varphi \wedge \psi).\rho & \doteq & \varphi.\rho \wedge \psi.\rho \\
(\varphi \vee \psi).\rho & \doteq & \varphi.\rho \vee \psi.\rho \\
(\varphi \Rightarrow \psi).\rho & \doteq & \varphi.\rho \Rightarrow \psi.\rho \\
(\varphi \equiv \psi).\rho & \doteq & \varphi.\rho \equiv \psi.\rho \\
(\forall x :: \varphi).\rho & \doteq & \forall a : a \in \mathbf{A} : \varphi.\rho\{x := a\} \\
(\exists x :: \varphi).\rho & \doteq & \exists a : a \in \mathbf{A} : \varphi.\rho\{x := a\}
\end{array}$$

donde $\rho\{x := a\}$ es un *entorno punteado*, y representa la misma colección de funciones, salvo el valor correspondiente a la variable x:

$$\rho\{x := a\}.y \doteq \begin{cases} \rho.y, & \text{si } y \not\equiv x \\ a_\rho, & \text{si } y \equiv x \end{cases}$$

Modelos y consecuencia lógica

Un Σ–álgebra $\mathcal{A}$ es un *modelo de una fórmula* φ si

$$\forall \rho :: \varphi.\rho$$

que escribiremos, como es habitual, utilizando el operador $[\,]$ (*ptle*) $[\varphi]$, o si queremos hacer notar el álgebra, en la forma $\mathcal{A} \models [\varphi]$. Si la fórmula φ no tiene variables libres entonces su valor coincide con el de $[\varphi]$. $\mathcal{A}$ es modelo de un conjunto de fórmulas M si lo es de cada una; esto lo escribiremos en la forma $\mathcal{A} \models M$. Es fácil ver que las reglas usuales del cálculo de predicados son válidas también sobre espacios de estados; como por ejemplo, la regla de oro

$$[\,(\varphi \Rightarrow \psi) \equiv (\varphi \wedge \psi \equiv \varphi)\,]$$

Se dice que una fórmula φ es *consecuencia lógica* de un conjunto de fórmulas M, que escribiremos en la forma $M \models \varphi$, si todo modelo de M es modelo de φ.

Sea $\mathcal{A}$ una Σ–interpretación; la teoría $T(\mathcal{A})$ es el conjunto de fórmulas cerradas (sin variables) válidas en $\mathcal{A}$; dos interpretaciones $\mathcal{A}$ y $\mathcal{B}$ son equivalentes si tienen la misma teoría, es decir $T(\mathcal{A}) = T(\mathcal{B})$.

Capítulo 2

Elementos de la Teoría de Dominios

2.0. Continuidad

A lo largo de esta sección, D será un conjunto dotado de un orden parcial $\leq$.

DEFINICIÓN 2.0 (Retículo. Retículo completo)

1. *$x \cup y$ denota el supremo de x e y, o sea,* $\min\{z \in D | x, y \leq z\}$.
2. *$x \cap y$ denota el mínimo de x e y, o sea,* $\max\{z \in D | z \leq x, y\}$.
3. *Si $X \subseteq D$, se definen en forma similar el supremo $\cup X$ y el ínfimo $\cap X$.*
4. *$\perp \doteq \cap D$, y $\top \doteq \cup D$. Entonces, $\cup\emptyset \doteq \perp$.*
5. *D se dice un* retículo, *si para cuales quiera $x, y \in D$, existen $x \cup y$, $x \cap y$.*
6. *D se dice un* retículo completo *si todo subconjunto $X \subseteq D$ tiene supremo $\cup X$ e ínfimo $\cap X$. Luego en un retículo completo existen $\top$ y $\perp$.*

NOTA 2.1 *Para cada retículo completo puede probarse que si $\forall X :: \exists(\cup X)$, entonces $\forall X :: \exists(\cap X)$. Luego una de las dos condiciones de la definición no es necesaria.*

DEFINICIÓN 2.2 (Conjunto dirigido y cadena)

1. *Un conjunto $X \subseteq D$ se dice* dirigido *si es no vacío y cualesquiera para de elementos $x, y \in X$ admite un $z \in X$ verificando $x, y \leq z$ (o equivalentemente, todo subconjunto finito de X tiene cota superior).*
2. *Un conjunto $X \subseteq D$ se dice una* cadena *si es no vacío y para cualesquiera $x, y \in X$, se verifica $x \leq y \vee y \leq x$.*

De la definición se deduce que toda cadena es un conjunto dirigido, y de todo conjunto dirigido se puede extraer una cadena.

LEMA 2.3 *Si X es dirigido, existe una cadena $\{x_n\} \subseteq X$ tal que $\cup X = \cup\{x_n\}$.*

DEFINICIÓN 2.4 (Dominio y $\perp$–Dominio)

1. *D se dice un* dominio *si el orden es completo; es decir, todo subconjunto dirigido X tiene un supremo $\cup X$ (o equivalentemente, toda cadena tiene supremo).*
2. *D se dice* $\perp$–dominio (dominio con punta*) si existe el mínimo $\perp$.*

3. *D es un* **dominio plano** *si $a \leq b \Longleftrightarrow a = \bot' \vee a = b$ para cierto elemento $\bot'$ (todo dominio plano es un dominio y $\bot' \equiv \bot$).*

NOTA 2.5 *En los textos anglófilos se utiliza cpo (*complete partial order*) en lugar de dominio y* pointed cpo *en lugar de $\bot$–dominio.*

OBSERVACIÓN.– Todo retículo completo es un $\bot$-dominio, y en particular un dominio ya que la condición de la definición de retículo es más restrictiva. Para las necesidades de la semántica denotacional basta con la condición de dominio. OBS

DEFINICIÓN 2.6 (Aplicación monótona) *Sean D y D' dos conjuntos parcialmente ordenados. Una función $T : D \to D'$ es* monótona *si verifica: $\forall x, y :: x \leq y \Rightarrow T(x) \leq T(y)$.*

NOTA 2.7 *Para ser más precisos deberíamos escribir $T(x) \leq' T(y)$ para hacer ver que la comparación es con el orden de D'.*

DEFINICIÓN 2.8 (Topología de Scott) *Sea D un dominio. Un subconjunto O se dice abierto en la topología de Scott si verifica las condiciones:*

(a) $x \in O \wedge x \leq y \Rightarrow y \in O$.

(b) $\cup X \in O$, *con X dirigido* $\Rightarrow X \cap O \neq \emptyset$

Tales abiertos inducen una topología trivialmente.

LEMA 2.9 *$V[x] = \{z \mid z \not\leq x\}$ es abierto*

Demostración.– Trivial. LEMA

En lo que sigue D y D' son dominios.

DEFINICIÓN 2.10 (Continuidad) *Una aplicación $f : D \to D'$ se dice*

1. continua *si es continua en la topología de Scott.*
2. semicontinua *si verifica la propiedad:*

$$\forall X : X \text{ dirigido } \subseteq D : f(\cup X) = \cup f(X)$$

3. continua por cadenas *si verifica la propiedad:*

$$\forall C : C \text{ cadena } \subseteq D : f(\cup C) = \cup f(C)$$

Veamos que las tres definiciones de continuidad son equivalentes.

TEOREMA 2.11 *Se verifican las siguientes propiedades*

(i)	*f continua*	$\Rightarrow$	*f monótona*
(ii)	*f semicontinua*	$\Rightarrow$	*f monótona*
(iii)	*f monótona y X dirigido*	$\Rightarrow$	*$f(X)$ dirigido*
(iv)	*f continua*	$\Longleftrightarrow$	*f semicontinua*
(v)	*f continua por cadenas*	$\Rightarrow$	*f continua*
(vi)	*f continua*	$\Longleftrightarrow$	*f continua por cadenas*

Demostración.– Sea $g(Y) = f^{-1}(Y)$. (iii) y (vi) son triviales. TEOREMA

(i) si $x \leq y$ y $f(x) \not\leq f(y)$ entonces $f(x) \in V[f(y)]$ abierto, de donde $x \in g(V[f(y)])$ que es abierto y por $x \leq y$ se tiene $y \in g(V[f(y)])$, luego $f(y) \in V[f(y)]$ que es absurdo.

(ii) si $x \leq y$, entonces $\{x, y\}$ es dirigido, luego

$$\cup f(\{x, y\}) = f(y) \;\Rightarrow\; f(x) \leq f(y)$$

$(iv) \Rightarrow$: sea X dirigido; por monotonía, $\forall x \in X :: f(x) \leq f(\cup X)$, luego $\cup f(X) \leq f(\cup X)$. Si fuera $f(\cup X) < \cup f(X) (\doteq y)$, entonces $f(\cup X) \in V[y]$ abierto, luego $\cup X \in g(V[y])$ abierto, y X dirigido, de donde $\cup X \bigcap g(V[y]) \neq \emptyset$, luego $\exists a \in X$ tal que $f(a) \in V[y] \;\Rightarrow\; f(a) \not\leq \cup f(X)$ que es absurdo.

$(iv) \Leftarrow$: sea $O' \subseteq D'$ abierto y sea $O \doteq g(O')$; veamos que O es abierto probando las dos condiciones de la Definición 2.8:

(a) Si $x \in O$, $x \leq y$ entonces, $f(x) \in O'$, y por (ii), $f(x) \leq f(y)$; luego por ser O' abierto, $f(y) \in O'$, de donde $y \in O$.

(b) Sea $\cup X \in O, X$ dirigido; entonces $f(\cup X) \in O', f(X)$ dirigido, luego por ser f semicontinua, $\cup f(X) \in O', f(X)$ dirigido; entonces por ser O' abierto se tiene que $f(X) \cap O' \neq \emptyset$, de donde $X \cap O \neq \emptyset$.

(v) Sea O' abierto en $D', O = g(O')$; veamos que O es abierto:

(a) si $x \in O, y \geq x$, entonces $f(x) \in O'$ abierto, y por ser $\{x, y\}$ una cadena, $f(y) \geq f(x)$; luego $f(y) \in O'$, de donde $y \in O$.

(b) Sea X dirigido, $\cup X \in O$. Existe una cadena $\{x_n\} \subseteq X$ verificando $\cup\{x_n\} = \cup X$; entonces $\cup\{x_n\} \in O$, de donde $\cup f\{x_n\} = f(\cup\{x_n\}) \in O'$; luego $\cup f\{x_n\} \in O'$ y $\{f(x_n)\}$ es una cadena, y de aquí (definición de abierto): $\{f(x_n)\} \cap O' \neq \emptyset$, de donde $\{x_n\} \cap O \neq \emptyset$. TEOREMA

2.1. Teoremas del Punto Fijo

DEFINICIÓN 2.12 *Sea $T : D \to D$;*

1. *x se dice un* punto fijo *de T si $T(x) = x$.*
2. *a se llama* menor punto fijo *de T, y se designa con $mpf(T)$, si es un punto fijo para cualquier otro punto fijo x, $a \leq x$.*
3. *de la misma forma se define el* mayor punto fijo *de T, $Mpf(T)$.*

Si existe $mpf(T)$, éste es único. Es trivial que $mpf(T) \leq Mpf(T)$

TEOREMA 2.13 (Tarski–Knaster, 1955) *Sea D un retículo completo y $T : D \to D$ monótona; entonces existen $Mpf(T)$ y $mpf(T)$; además:*

(i) $mpf(T) = \inf\{x \,|\, T(x) = x\} = \inf\{x \,|\, T(x) \leq x\}$

(ii) $Mpf(T) = \sup\{x \,|\, T(x) = x\} = \sup\{x \,|\, x \leq T(x)\}$

Demostración.– Sea $G = \{x \,|\, T(x) \leq x\}$; G es no vacío ya que $T(\top) \leq \top$, y existe $g = \inf(G)$; vamos a probar en primer lugar que g es un punto fijo.

$g = \inf(G)$
$\Rightarrow$ $\because$ definición de G
$\forall x : x \in G : g \leq x$
$\Rightarrow$ $\because$ T es monótona
$\forall x : x \in G : T(g) \leq T(x)$
$\Rightarrow$ $\because$ def. de G: $T(x) \leq x$; $\leq$ trans.
$\forall x : x \in G : T(g) \leq x$
$\Rightarrow$ $\because$ luego $T(g)$ es cota inferior de G:
$T(g) \leq g$

$T(g) \leq g$
$\Rightarrow$ $\because$ T es monótona
$T(T(g)) \leq T(g)$
$\Rightarrow$ $\because$ definición de G
$T(g) \in G$
$\Rightarrow$ $\because$ $g = \inf(G)$
$T(g) \geq g$

Luego $T(g) = g$. Por otro lado $\{x \,|\, T(x) = x\} \subseteq \{x \,|\, T(x) \leq x\}$, de donde

$$g' \doteq \inf\{x \,|\, T(x) = x\} \geq \inf\{x \,|\, T(x) \leq x\} = g$$

Luego $g' \geq g$. Pero, ya que $g \in \{x \,|\, T(x) = x\}$, entonces $g \geq g'$. (ii) se prueba de forma parecida. TEOREMA

TEOREMA 2.14 (Stephen Kleene [1909-1994]) *Sea D un $\perp$–dominio y $T : D \to D$ continua; entonces $mpf(T) = \sup\{t_n\}$, donde $t_0 \doteq \perp$, y $t_{n+1} \doteq T(t_n)$.*

Demostración.– Veremos sucesivamente

✓ El conjunto $\{t_n, n \geq 0\}$ es una cadena (se prueba por inducción).

✓ $\sup\{t_n\}$ es punto fijo

$\sup\{t_n, n \geq 0\}$
$=$ $\because$ $t_0 = \perp$
$\sup\{t_{n+1}, n \geq 0\}$
$=$ $\because$ definición de t_n
$\sup\{T(t_n), n \geq 0\}$
$=$ $\because$ T continua
$T(\sup\{t_n, n \geq 0\})$

✓ $\sup\{t_n\} = mpf(T)$

En efecto. Si $x = T(x)$,
$\perp \leq x$
$\Rightarrow$ $\because$ inducción sobre n
$\forall n : n \geq 0 : t_n \leq x$
$\Rightarrow$
$\sup\{t_n\} \leq x$

TEOREMA

2.2. Construcción de Dominios

LEMA 2.15 (Dominio Extendido (*lifted domain*), función extendida)

(i) *Sea D un dominio sin ínfimo (sin punta) y $\perp \notin D$. Definimos $D_\perp \doteq D \cup \{\perp\}$. Entonces $D_\perp$ es un $\perp$–dominio con la relación de orden extendida:*

$$\forall a, b \in D_\perp :: a \leq b \Longleftrightarrow a = \perp \text{ o bien } a, b \in D \wedge a \leq_D b$$

(ii) *Sean dos dominios D y D', y sus extensiones $D_\perp$ y $D'_\perp$; sea una aplicación $f : D \to D'_\perp$. Entonces, si $f : D \to D'_\perp$ es continua, también lo es la aplicación extendida $f_\perp : D_\perp \to D'_\perp$ definida como*

$$f_\perp(x) \;=\; (\lambda_\perp a \to f(a))x \;\doteq\; \langle x \neq \perp \to f(x) \,\square\, \perp\rangle$$

donde la expresión $\langle a \to A \,\square\, A'\rangle$ es una simplificación de la expresión si a entonces A en otro caso A'.

Demostración.– Sea C una cadena en $D_\perp$. Si $C = \{\perp\}$, es trivial que $\cup f_\perp(C) = f_\perp(\cup C)$.

— Si $\perp \notin C$,

$$\begin{array}{ll} & f_\perp(\cup C) \\ = & \quad \because \perp \notin C \\ & f(\cup C) \\ = & \quad \because \text{por ser } f \text{ continua} \\ & \cup f(C) \\ = & \quad \because f(C) = f_\perp(C) \\ & \cup f_\perp(C) \end{array}$$

— Si $\perp \in C \neq \{\perp\}$, entonces $C' = C - \{\perp\}$ es una cadena, y

$$\begin{array}{ll} & f_\perp(\cup C) \\ = & \quad \because \cup C = \cup C' \\ & f_\perp(\cup C') \\ = & \quad \because \text{por lo anterior} \\ & \cup f_\perp(C') \\ = & \quad \because \text{ya que } \perp \notin C' \\ & \cup f_\perp(C) \end{array}$$

LEMA

TEOREMA 2.16 (Dominio Producto) *Si $(D, \leq)$ y $(D', \leq)$ son dominios (o bien $\perp$–dominios), los es también $(D \times D', \leq)$, donde*

$$(x, y) \leq (x', y') \Longleftrightarrow x \leq x' \wedge y \leq y'$$

Demostración.– Si son $\perp$–dominios, $(\perp, \perp')$ es el mínimo de $D \times D'$. Sea entonces X dirigido $\subseteq D \times D'$, entonces también son dirigidos las proyecciones:

$$\begin{array}{l} X_0 = \{x \in D \mid \exists x' \in D' :: (x, x') \in X\} \\ X_1 = \{x' \in D' \mid \exists x \in D :: (x, x') \in X\} \end{array}$$

y además $\cup X = (\cup X_0, \cup X_1)$.

TEOREMA

El Dominio de las Funciones Continuas El conjunto de funciones continuas de D en D', que denotaremos con $[D \to D']$, admite como orden parcial:

$$f \leq g \Longleftrightarrow \forall x :: f(x) \leq g(x)$$

Es fácil ver que $[D \to D']$ es un dominio y que el supremo de una colección de funciones, $\cup\{f | f \in F\}$, es la función

$$(\cup\{f, f \in F\})(x) = \cup\{f(x), f \in F\}$$

Si D' es un $\perp$–dominio, la función $\lambda x.\perp'$ es el mínimo de $[D \to D']$.

TEOREMA 2.17 *Se verifican las siguientes propiedades*

(i) *$f : D \times D' \to D''$ es continua sii lo es cada aplicación parcial:*

$$\lambda x \to f(x, y) \qquad\qquad \lambda y \to f(x, y)$$

(ii) *Continuidad de la aplicación. La función* aplicación A *es continua:*

$$A : [D \to D'] \times D \to D' \qquad A(f, x) \doteq f(x)$$

(iii) *Continuidad de la abstracción. Sea* $f \in [D \times D' \to D'']$ *y sea*

$$F\, x \doteq \lambda y \to f(x, y) : D \to [D' \to D'']$$

Entonces, F *y* $\lambda f \to F$ *son continuas.*

Dominio Unión Disjunta Sean dos conjuntos A y B; la unión disjunta o unión etiquetada de A y B se obtiene uniendo los elementos de A y B previamente etiquetados; si utilizamos dos etiquetas, en_A y en_B, definimos:

$$A + B = \{(en\_A, a) \,|\, a \in A\} \cup \{(en\_B, b) \,|\, b \in B\}$$

Si A y B son dominios, el conjunto $A + B$ con la relación de orden:

$$\begin{array}{lll} (en\_A, a) \leq (en\_A, a') & \Longleftrightarrow & a \leq a' \\ (en\_B, b) \leq (en\_B, b') & \Longleftrightarrow & b \leq b' \end{array}$$

es un dominio. ($A + B$ no es un $\perp$–domino, aunque los sean A y B.)

Definimos dos funciones para 'sumergir' A y B en $A + B$:

$$\begin{array}{ll} s_A : A \to A + B, & s_A\, a \doteq (en\_A, a) \\ s_B : B \to A + B, & s_B\, b \doteq (en\_B, b) \end{array}$$

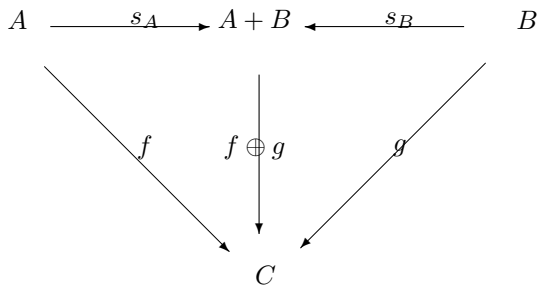
Sean ahora dos funciones $A \xrightarrow{f} C, B \xrightarrow{g} C$; entonces podemos considerar una nueva función $(f \oplus g) : A + B \to C$, definida en la forma:

$$\begin{array}{l} \forall a \in A :: (f \oplus g)(en\_A, a) \doteq f\, a \\ \forall b \in B :: (f \oplus g)(en\_B, b) \doteq g\, b \end{array}$$

Se cumple:

$$(f \oplus g) \circ s_A = f \qquad (f \oplus g) \circ s_B = g$$

Es decir, el siguiente diagrama es conmutativo:



TEOREMA 2.18 *Las inmersiones* s_A, s_B *son continuas; si* f *y* g *son continuas entonces lo es* $f \oplus g$.

COROLARIO 2.19 *Toda función construida usando la notación funcional vista hasta el momento es continua.*

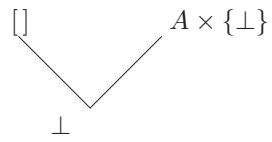


Figura 2.0: Diagrama de Hasse para el dominio L_1.

2.3. Especificación Recursiva de Dominios

Sea F una funcional construida utilizando las operaciones con dominios $\_+\_$, $\_\times\_$, $\_\_\perp$ y $[\_ \to \_]$, y sea la definición recursiva $D = F(D)$. Tales definiciones corresponden a la definición de estructuras recursivas, como

$$\begin{array}{lcl} Arbol & = & (Vacio \ + (Base \times Arbol \times Arbol))_\perp \\ Lista & = & (Vacia \ + (Base \times Lista))_\perp \end{array}$$

Estas definiciones recursivas son imprescindibles para modelar otras estructuras, como los procedimientos de ALGOL, donde puede pasarse un procedimiento como parámetro:

$$\begin{array}{lcl} Proc & = & Param \to Mem \to Mem_\perp \\ Param & = & Int + Real + Proc \end{array}$$

El problema que abordamos es tratar de encontrar un dominio solución de la ecuación $D = F(D)$.

Antes de ver la construcción general veamos un ejemplo de cómo se realiza la construcción.

Un Ejemplo. Las Listas Sea la ecuación

$$L = (Vacia + A \times L)_\perp (\ \doteq \ F(L)\ ) \tag{1}$$

donde el conjunto $Vacia$ es unitario y se confundirá con la lista sin elementos $[\,]$. La idea es generar una sucesión $\{L_i\}_{i\geq 0}$ de $\perp$–dominios que correspondan a aproximaciones de la solución de la ecuación (1), partiendo, por ejemplo, de $L_0 = \{\perp\}$, y tomando $L_i = F(L_{i+1})$, al igual que se hace para la construcción de un punto fijo para las funciones continuas.

Entonces $L_1 = ([\,] + A \times L_0)_\perp = ([\,] + A \times \{\perp\})_\perp$; sabemos que L_0 se puede sumergir en L_1 a través de una aplicación continua $\sigma \in [L_0 \to L_1]$, y de la misma forma L_1 se puede proyectar sobre L_0 a través de una aplicación continua $\pi \in [L_1 \to L_0]$. Si prescindimos, por comodidad de escritura, de tal inmersión, según la definición de orden en $\_ + \_$, tenemos, para L_1, el diagrama de Hasse de la Figura 2.0, y las listas de L_1 serán, prescindiendo de inmersiones:

$$L_1 = \{\perp, [\,]\} \cup \{a : \perp \mid a \in A\}$$

Utilizamos la notación $a : l$ en lugar de (a, l) para denotar el correspondiente elemento del producto cartesiano; los elementos de la forma $a : [\,]$ del producto

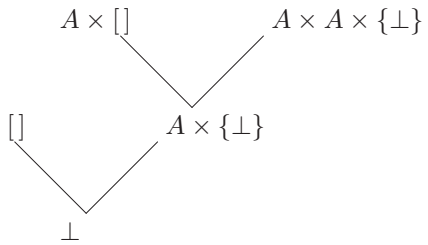


Figura 2.1: Diagrama de Hasse para el dominio L_2.

cartesiano $A \times [\,]$ se denotarán con $[a]$. Las listas de la forma $a : \bot$ son listas parciales, o con información parcial (tienen un elemento y algo más). Veamos como será L_2 (se trata de tomar el diagrama de la Figura 2.0, multiplicado por A, y añadir $[\,]$ y $\bot$. De esta forma se obtiene el diagrama de la Figura 2.1. Obsérvese que las listas de L_2 ya tienen más información

$$\begin{array}{rcl} L_2 & = & \{\bot, [\,]\} \cup \{a : \bot \mid a \in A\} \\ & & \cup \{[a] \mid a \in A\} \cup \{a_0 : a_1 : \bot \mid a_0, a_1 \in A\} \end{array}$$

El elemento $\bot$ juega un doble papel:

- modela la no terminación, y
- modela el conocimiento parcial: $[a_0, a_1|\bot]$ (el programa genera los dos primeros elementos y después diverge).

Si nos preguntamos por el límite $\lim L_n$, podríamos considerar, salvo inmersiones, algo como:

$$\lim_n L_n = \cup_n L_n$$

y tal conjunto contiene todas las listas finitas y las listas parciales. Pero tal límite no es un dominio ya que la cadena:

$$\bot \qquad a_0 : \bot \qquad a_0 : a_1 : \bot \qquad \ldots a_0 : a_1 : \ldots : a_n : \bot \qquad \ldots$$

no tiene límite en $\bigcup L_i$; el remedio es añadir todas las listas infinitas, de forma que se obtiene el diagrama de Hasse para el dominio L_∞ (Figura 2.2). Puesto que sumergir significa obtener una tupla, podemos considerar que cada lista se identifica (al sumergir) con una tupla de tuplas:

$$\begin{array}{lcl} a_0 : a_1 : [\,] \in L_3 & \equiv & (\bot, a_0 : \bot, a_0 : a_1 : \bot, a_0 : a_1 : [\,]) \\ a_0 : a_1 : a_2 : \bot \in L_3 & \equiv & (\bot, a_0 : \bot, a_0 : a_1 : \bot, a_0 : a_1 : a_2 : \bot) \end{array}$$

Lo que se hace es, partiendo del objeto ir suprimiendo información sucesivamente:

$$a_0 : a_1 : a_2 : \bot \longrightarrow a_0 : a_1 : \bot \longrightarrow a_0 : \bot \longrightarrow \bot$$

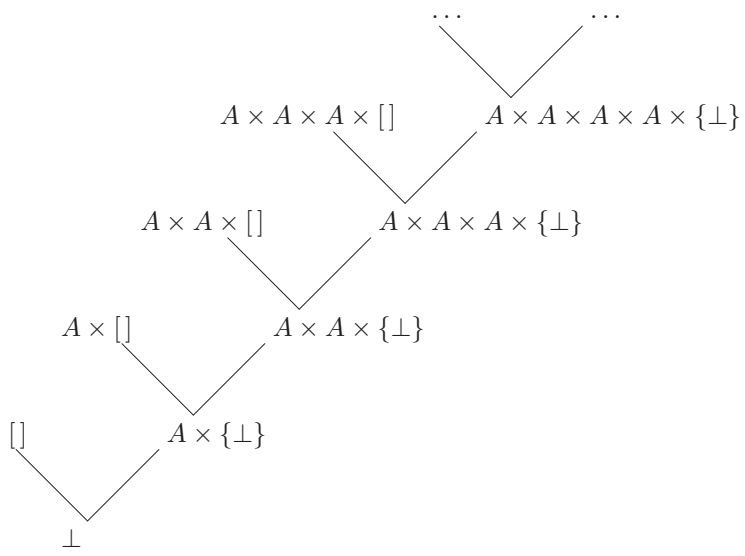


Figura 2.2: Diagrama de Hasse para el dominio L_∞.

Sea la cadena de proyecciones e inmersiones

$$L_0 \underset{\pi_0}{\overset{\sigma_0}{\rightleftarrows}} L_1 \underset{\pi_1}{\overset{\sigma_1}{\rightleftarrows}} L_2 \underset{\pi_2}{\overset{\sigma_2}{\rightleftarrows}} \ldots \quad L_i \underset{\pi_i}{\overset{\sigma_i}{\rightleftarrows}} L_{i+1} \quad \ldots$$

Obsérvese que la composición $\pi_i \circ \sigma_i$ (sumergir y proyectar) no pierde información, mientras que la composición $\sigma_i \circ \pi_i$ (proyectar y sumergir) pierde información; por consiguiente

$$\pi_i \circ \sigma_i = i_{L_i} \qquad \sigma_i \circ \pi_i \leq i_{L_{i+1}}$$

Obsérvese entonces que los elementos de L_n son las $(n+1)$–tuplas de la forma: $(x_0, x_1, \ldots, x_n)$, donde $x_n = x$ y $x_i = \pi_i(x_{i+1})$ para $i \geq n-1$.

Las listas infinitas admiten una representación infinita mediante listas finitas; así, una lista infinita de a-es se representa:

$$(\bot, a : \bot, a : a : \bot, a : a : a : \bot, \ldots)$$

de forma que podemos identificar el dominio L_∞ solución de la ecuación $D = F(D)$ como el conjunto:

$$L_\infty = \{\mathbf{x} = (x_0, x_1, \ldots, x_n, \ldots) \,|\, x_n \in L_n, x_n = \pi_n(x_{n+1})\}$$

con la relación de orden:

$$\mathbf{x} \leq \mathbf{y} \Longleftrightarrow \forall n :: \mathbf{x}\downarrow n \leq \mathbf{y}\downarrow n$$

Vamos a formalizar lo anterior

Límite Proyectivo Inverso y Dominio D_∞

Sean dos $\perp$–dominios D y D'; un *par de Scott* es una pareja de funciones continuas (σ, π), $\sigma \in [D \to D']$, $\pi \in [D' \to D]$, tales que

$$\pi \circ \sigma = i_D \qquad\qquad \sigma \circ \pi \leq i_{D'}$$

σ se llama inmersión y π proyección; en otros textos se habla de *retraction par*. Escribiremos, para simplificar $(\sigma, \pi) : [D \rightleftarrows D']$.

LEMA 2.20 *Se verifican las propiedades*

(i) *Si* $(\sigma, \pi) : [D \rightleftarrows D']$ *y* $(\sigma, \pi') : [D \rightleftarrows D']$ *son dos pares de Scott, entonces* $\pi = \pi'$.

(ii) *Si* $(\sigma, \pi) : [D \rightleftarrows D']$ *y* $(\sigma', \pi) : [D \rightleftarrows D']$ *son dos pares de Scott, entonces* $\sigma = \sigma'$.

(en definitiva, cada elemento del par caracteriza al otro).

Demostración.– Veamos (i):

$\sigma \circ \pi \leq i_{D'}$
$\Rightarrow$ $\because \pi'$ es monótona
$\pi' \circ \sigma \circ \pi \leq \pi' \circ i_{D'}$
$\Rightarrow$ $\because \pi' \circ \sigma = i_D$ y asociatividad
$i_D \circ \pi \leq \pi' \circ i_{D'}$
$\Rightarrow$ $\because$ propiedades de las identidades
$\pi \leq \pi'$

e igualmente encontramos $\pi \leq \pi'$; (ii) se prueba igual. LEMA

LEMA 2.21 *Si* $(\sigma, \pi) : [D \rightleftarrows D']$ *es un par de Scott, entonces* σ *y* π *son estrictas.*

Demostración.– Si $\perp$ y $\perp'$ son los ínfimos de D y D', entonces,

$\perp \leq \pi\perp'$
$\Rightarrow$ $\because \sigma$ es monótona
$\sigma\perp \leq \sigma\pi\perp'$
$\Rightarrow$ $\because \sigma \circ \pi \leq i_{D'}$
$\sigma\perp \leq \perp'$
$\Rightarrow$ $\because$ unicidad del ínfimo
$\sigma\perp = \perp'$ (luego σ es estricta)
$\Rightarrow$ $\because$ Leibniz
$\pi\sigma\perp = \pi\perp'$
$\Rightarrow$ $\because \pi \circ \sigma = i_D$
$\perp = \pi\perp'$ (luego π es estricta) LEMA

DEFINICIÓN 2.22 *Sean* $(\sigma, \pi) : [D \rightleftarrows D']$, $(\alpha, \omega) : [D' \rightleftarrows D'']$ *dos pares de Scott. Entonces se definen:*

— $(\sigma, \pi)^r : [D' \rightleftarrows D]$, $\qquad (\sigma, \pi)^r \doteq (\pi, \sigma)$
(no necesariamente es un par de Scott)

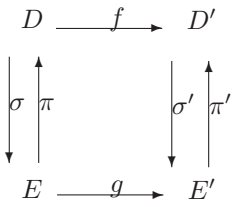


Figura 2.3: La operación $[p \to p']$.

— $i_{D \rightleftarrows D} \doteq (i_D, i_D)$
(es un par de Scott)

— $(\sigma, \pi) \circ (\alpha, \omega) \doteq (\alpha \circ \sigma, \pi \circ \omega) : [D \rightleftarrows D'']$

Ahora asociamos una operación con pares de Scott a cada operación con dominios:

DEFINICIÓN 2.23 *Sean $p = (\sigma, \pi) \in [D \rightleftarrows E]$, $p' = (\sigma', \pi') \in [D' \rightleftarrows E']$ dos pares de Scott; entonces se definen los nuevos pares inducidos por las operaciones sobre dominios $\_ \times \_$, $\_ + \_$, $[\_ \to \_]$, $(\_)_\perp$:*

1.	$p \times p'$	$\in$	$[D \times D' \rightleftarrows E \times E']$
	$p \times p'$	$\doteq$	$(\ \lambda(x,y) \to (\sigma x, \sigma' y),\ \lambda(x,y) \to (\pi x, \pi y)\ )$
2.	$p \oplus p'$	$\in$	$[D + D' \rightleftarrows E + E']$
	$p \oplus p'$	$\doteq$	$(\ s_E \circ \sigma \oplus s_{E'} \circ \sigma',\ s_D \circ \pi \oplus s_{D'} \circ \pi'\ )$
3.	$[p \to p']$	$\in$	$[D \to D'] \rightleftarrows [E \to E']$
	$[p \to p']$	$\doteq$	$(\ \lambda f \to \sigma' \circ f \circ \pi,\ \lambda g \to \pi' \circ g \circ \sigma\ )$ – *véase Figura 2.3*
4.	$(p)_\perp$	$\in$	$[D_\perp \rightleftarrows E_\perp]$
	$(p)_\perp$	$\doteq$	$(\ \lambda_\perp x \to \sigma x,\ \lambda_\perp x \to \pi x\ )$

Es fácil demostrar, que en efecto, tales operaciones definen pares de Scott.

DEFINICIÓN 2.24 (Funcional Constructor de Dominios) *Una funcional F sobre dominios construida con las operaciones $\_ \times \_$, $\_ + \_$, $[\_ \to \_]$ y $(\_)_\perp$ se dirá un* constructor de dominios.

Vamos a considerar la operación $F(p)$ sobre un par de Scott.

LEMA 2.25 *Si p es un par de Scott, y F es un constructor de dominios, entonces*

(i) *$F(p)$ es un par de Scott.*

(ii) $F(i_{D \rightleftarrows D}) = i_{F(D) \rightleftarrows F(D)}$.

(iii) $F(p) \circ F(q) = F(p \circ q)$, *si* $p \in [D \rightleftarrows D'], q \in [D' \rightleftarrows D'']$.

(iv) $F(p^r) = F(p)^r$.

DEFINICIÓN 2.26 (Límite inverso) *Un* sistema proyectivo (D_n, p_n) *está formado por una sucesión* $\{D_n\}_{n\geq 0}$ *de* $\perp$*–dominios y una sucesión* $\{p_n\} = \{(\sigma_n, \pi_n)\}$ *de pares de Scott* $(\sigma_n, \pi_n) : [D_n \rightleftarrows D_{n+1}]$. *El* límite inverso proyectivo *es el conjunto*

$$D_\infty = \{\mathbf{x} = (x_n) \,|\, x_i \in D_i, \pi_i(x_{i+1}) = x_i\}$$

TEOREMA 2.27 *El límite inverso* D_∞ *es un* $\perp$*–dominio con la relación*

$$\mathbf{x} \leq \mathbf{y} \Longleftrightarrow \forall i :: x_i \leq y_i$$

Demostración.– Ver [Schmidt, 1988]:261. TEOREMA

TEOREMA 2.28 *Sea* F *un constructor de dominios. Entonces la secuencia* (D_n, p_n) *es un sistema proyectivo, donde:*

$$\begin{array}{lcllcl} D_0 & \doteq & \{\perp\}, & D_{n+1} & \doteq & F(D_n), \\ \sigma_0 & \doteq & \lambda x \to \perp, & (\sigma_{i+1}, \pi_{i+1}) & \doteq & F(\sigma_i, \pi_i), \\ \pi_0 & \doteq & \lambda x \to \perp. & & & \end{array}$$

Demostración.– Basta aplicar el lema anterior, ya que $F(\sigma_i, \pi_i)$ es un par de Scott si lo es (σ_i, π_i). TEOREMA

TEOREMA 2.29 D_∞ *es isomorfo a* $F(D_\infty)$.

Demostración.– Ver [Schmidt, 1988]:260–264. TEOREMA

Para el caso particular $D_\infty \cong [D_\infty \to D_\infty]$ puede verse la construcción (no menos complicada) en [Barendregt, 1984].

2.4. Dominios Potencias

Los dominios potencia aparecen normalmente al modelar lenguajes donde interviene el indeterminismo, en forma pura o junto a la concurrencia. La naturaleza del dominio base utilizado es muy importante; comenzamos viendo los dominios potencia discretos: con dominio base plano.

Dominio Potencia Relacional Discreto Sea D un dominio plano; es decir un dominio con la relación $a \leq b \Longleftrightarrow a = \perp \vee a = b$. Tal dominio plano modela cómputos, tales que si son propios (distintos de $\perp$) no están relacionados. Sea $\mathbb{P}_d(D)$ (dominio potencia relacional discreto) el conjunto

$$\mathbb{P}_d(D) = \{A \subseteq D \,|\, \perp \notin A\}$$

con la relación de orden $\leq \equiv \subseteq$. Por ser D plano, tenemos

$$A \leq B \Longleftrightarrow (\forall a \in A :: \exists b \in B :: a \leq b)$$

Al eliminar $\perp$, obtenemos $\mathbb{P}_d(\mathbb{N}) = \mathbb{P}_d(\mathbb{N}_\perp)$ y por ello, la terminación no puede modelarse; se trata de una construcción natural en dominios sin ínfimo. Si incluimos el ínfimo se obtiene el dominio de *Egli–Milner*.

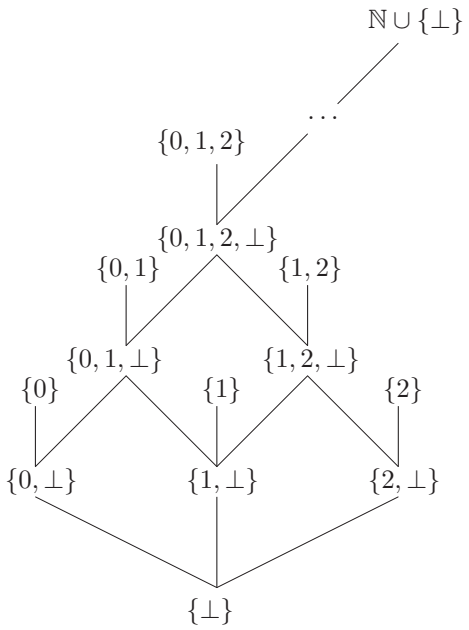


Figura 2.4: Diagrama de Hasse para el dominio $\mathbb{P}_{em}(\mathbb{N}_\bot)$.

Dominio Potencia de Egli–Milner Sea D un $\bot$–dominio. El dominio de Egli-Milner,

$$\mathbb{P}_{em}(D) = \{A \subseteq D \mid A \neq \emptyset \wedge A \text{ es finito } \vee \bot \in A\}$$

queda definido con la relación de orden $\leq$

$$A \leq B \iff (\forall a \in A :: \exists b \in B :: a \leq b) \wedge (\forall b \in B :: \exists a \in A :: a \leq b)$$

Consideramos conjuntos no vacíos puesto que normalmente cada conjunto modela respuestas a un cómputo. Por otro lado, los conjuntos infinitos deben contener el elemento $\bot$: si $\bot$ denota *no terminación*, un conjunto infinito captura un cómputo infinito. En la Figura 2.4 aparece el diagrama de Hasse de $\mathbb{P}_{em}(\mathbb{N}_\bot)$.

Un conjunto finito $\{m_1, \ldots, m_n\}$ no conteniendo a $\bot$ es el final de un cómputo; un conjunto que contiene $\bot$ (podría ser infinito), tal como $\{\bot, m_1, \ldots\}$ se interpreta como un cómputo parcial. Veamos como se interpreta la relación de orden; en primer lugar observamos que si $A \leq B$ entonces $A - \{\bot\} \subseteq B$; por otro lado, si $\bot \notin A$, debe tenerse $B \subseteq A$; por tanto,

$$\{1\} \not\leq \{1,2\} \qquad\qquad \{1,\bot\} \leq \{1,2,3\}$$

y en general

(a) $A \cup \{\bot\} \leq A \cup X$, para cualquier X
(b) $A - \{\bot\} \leq X \iff A = X$

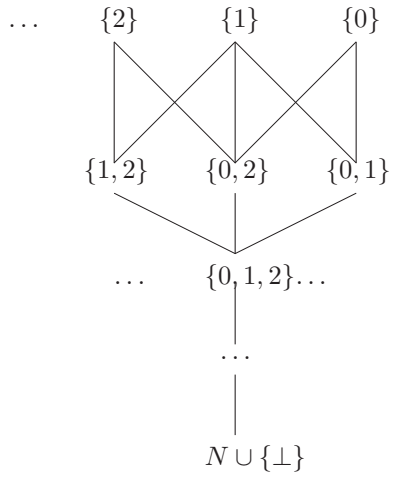


Figura 2.5: Diagrama de Hasse para el dominio discreto de Schmidt $\mathbb{P}_s(\mathbb{N}_\perp)$.

y de aquí se concluye que $\{\perp\}$ es el elemento ínfimo. Supongamos ahora dos programas $\mathcal{S}$ y $\mathcal{T}$ verificando, para cualquier ρ,

$$[\![\mathcal{S}]\!]\,\rho \leq [\![\mathcal{T}]\!]\,\rho$$

Entonces, si, desde ρ, S siempre termina, por (b), debe darse $[\![\mathcal{S}]\!]\,\rho = [\![\mathcal{T}]\!]\,\rho$, por lo que el programa $\mathcal{T}$ no conduce a más estados que $\mathcal{S}$; por el contrario, si $\mathcal{S}$ no termina desde ρ, entonces $[\![\mathcal{T}]\!]\,\rho$ debe contener a $[\![\mathcal{S}]\!]\,\rho$ más algunos otros estados que potencialmente podría alcanzar $\mathcal{S}$ si terminara. Todo esto puede verse como cierta relación de determinismo.

Dominio Potencia Discreto de Schmidt Para un domino D plano, sea $\mathbb{P}_s(D)$, el dominio discreto de Schmidt, dado por el conjunto

$$\mathbb{P}_s(D) = \{A \subseteq D \mid A \neq \emptyset \wedge A \text{ es finito } \wedge \perp \notin A\} \cup D$$

con la relación de orden

$$A \leq B \Longleftrightarrow \forall b \in B :: (\exists a \in A :: a \leq b)$$

Podemos ver que, en $\mathbb{P}_s(D)$, $A \leq B \Longleftrightarrow B \subseteq A$, y la interpretación que tiene es: un cómputo en el dominio de Schmidt es el conjunto de aquellos resultados no deseados. Puesto que el mínimo es el propio D, se interpreta que el mínimo es lo que no se desea, ya que el resultado del cómputo es arbitrario. En la Figura 2.5 aparece el diagrama de Hasse de $\mathbb{P}_s(\mathbb{N}_\perp)$.

Capítulo 3

Programas como Transformadores

3.0. La funcional wp (*weakest precondition*)

La semántica de un programa como función entre estados. Problemas. Una forma de capturar el significado de un programa S es considerar el estado final del cómputo como función del estado inicial. Es decir, la semántica de S sería una función $\mathcal{S} : \mathcal{E} \to \mathcal{E}$, donde $\mathcal{E}$ denota el espacio de estados. Ello trae consigo dos serios inconvenientes:

(i) podría ocurrir que S no termine partiendo de ciertos estados, y

(ii) para un programa indeterminista, la semántica de S no puede venir dada por una función $\mathcal{S} : \mathcal{E} \to \mathcal{E}$, ya que el estado final puede no ser único.

El primer problema exige que la función $\mathcal{S}$ sea una función parcial, indefinida para aquellos estados iniciales para los que el programa no termine. Este problema puede resolverse completando el conjunto de estados con un estado especial $\perp$ al que se le asocia la no terminación. Las desventajas de esta solución son: (a) se destruye la homogeneidad del espacio de estados, y (b) hay que trabajar con funciones *completadas*. Esta solución es la considerada en algunas semánticas operacionales y denotacionales tal como la que veremos en §11.2.

Admitimos que un sistema (programa, máquina o mecanismo) es *determinista* si el estado final depende únicamente del estado inicial; o sea, con idénticos estados iniciales se llega a idénticos estados finales, y el comportamiento del sistema es *reproducible*. Por el contrario, en un sistema indeterminista, el estado inicial puede conducir a distintos estados finales.

El segundo inconveniente (ii) puede resolverse si a cada estado se asocia una colección de estados, por lo que la semántica de S viene dada por una función $\mathcal{S} : \mathcal{E} \to \mathbb{P}(\mathcal{E})$ (subconjuntos o partes de $\mathcal{E}$); el inconveniente esencial de esta solución es la dificultad en obtener una semántica composicional: la semántica de las componentes determina la de la composición.

Modelando cómputos y estados vía predicados Parece interesante, al igual que hicieron Floyd y Hoare (recordemos §0.3, así como §1.0), asociar predicados con conjuntos de estados del sistema; un predicado X divide el espacio de estados en dos clases disjuntas, aquellos que verifican X y aquellos que verifican $\neg X$; por la ley del tercio excluido $[\neg X \vee X \equiv Cierto]$ obtenemos dos clases disjuntas. Puesto que un predicado X es visto como una función $X : \mathcal{E} \to \{C, F\}$, el predicado X se identifica con el conjunto de estados donde toma el valor C.

Un predicado X puede capturar un conjunto de estados finales o un conjunto de estados iniciales, y así podemos **clasificar cómputos**. La decisión de relacionar X con el estado final o con el inicial obedece a razones técnicas; es un hecho demostrado que se obtienen semánticas más sencillas si relacionamos cada predicado con un conjunto de estados finales; [Dijkstra y Scholten, 1990]:201–215 exponen algunos argumentos interesantes sobre esta ventaja. Así, se consideran dos transformadores de predicados, wp y wlp:

$$wp.S.X \quad \equiv \quad \text{estados iniciales para los cuales todo cómputo de } S \text{ termina en un estado verificando } X.$$

$$wlp.S.X \quad \equiv \quad \text{estados iniciales para los cuales todo cómputo de } S, \text{ o no termina, o termina en un estado verificando } X.$$

Así, $wp.S.X$ es el predicado que únicamente es cierto en aquellos estados iniciales para los cuales todo cómputo de S termina en un estado verificando X. O lo que es lo mismo, partiendo de cualquier estado del conjunto $wp.S.X$, el programa S termina en un estado verificando X.

NOTA 3.0 *Las funciones wp y wlp tienen dos argumentos, siendo el primero un programa y el segundo un predicado; la notación $wp.S$ indica la parcialización de wp con respecto al segundo argumento; es decir, $wp.S : X \mapsto wp.S.X$. Las siglas wp y wlp son las abreviaturas introducidas en [Dijkstra, 1976] para $weakest\ precondition$ y $weakest\ liberal\ precondition$. Asociando conjuntos de estados finales podemos considerar el transformador de predicados sq:*

$$sq.S.X \quad \equiv \quad \textit{aquellos estados finales procedentes de cómputos a partir de estados iniciales verificando } X.$$

Las siglas sq significan strongest postcondition, *o poscondición más fuerte, y fue introducida por [Floyd, 1967].*

NOTA 3.1 (Caracterización de Dijkstra/Scholten) *[Dijkstra, 1976] caracteriza un programa S por el transformador $wp.S$; en un trabajo posterior, [Dijkstra y Scholten, 1990], catorce años más tarde, cada programa S queda definido por los valores $wp.S.Cierto$ y $wlp.S.X$, verificándose:*

$$[\ wp.S.X \ \equiv \ wp.S.Cierto \wedge wlp.S.X\ ]$$

A pesar de que este último punto de vista puede tener ciertas ventajas, nos parece más simple seguir modelo original.

3.2 *¿Es posible deducir $wlp.S$ conocido el transformador $wp.S$?*

CONVENIO 3.3 (Una simplificación importante) Ya que $wp.S$ caracteriza a S, es interesante 'confundir' $wp.S$ con S con objeto de obtener una notación más compacta y expresiva; así, el predicado $wp.S.X$ será denotado simplemente con '$S.X$'. El predicado $wlp.S.X$ será escrito como $\widehat{S}.X$; por tanto, la caracterización semántica de Dijkstra/Scholten se escribiría con nuestra notación en la forma[1]:

$$[\ S.X \ \equiv \ S.C \wedge \widehat{S}.X\ ] \qquad \boxed{\text{CONVENIO}}$$

[1] Simplificaremos el predicado $Cierto$ con C, así como el predicado $Falso$ con F.

Tripletes y transformadores. Por el Convenio 3.3, el significado o semántica de un programa S está completamente definida si es posible conocer, para cada predicado X, los valores $S.X$ y $\widehat{S}.X$.

DEFINICIÓN 3.4 (Tripletes à la Dijkstra) *Introducimos el triplete* $\vdash_{\mathcal{D}} \{Y\}S\{X\}$ *en sentido de Dijkstra con la definición:*

$$\vdash_{\mathcal{D}} \{Y\}S\{X\} \doteq [Y \Rightarrow S.X]$$

Para simplificar escribiremos $\{Y\}S\{X\}$ en lugar de $\vdash_{\mathcal{D}} \{Y\}S\{X\}$.

La Definición 3.4 tiene tres consecuencias importantes:

1ª consecuencia: Un triplete es un escalar booleano: $Cierto$ o $Falso$. La implicación $Y \Rightarrow S.X$ podría no ser constante, pero el operador $[\,]$ la transforma en una constante.

2ª consecuencia: El cálculo de tripletes se reduce al cálculo de predicados.

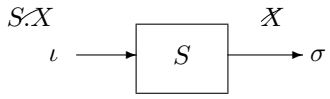
3ª consecuencia: Si consideramos **corrección total**, $wp.S.X$ es la solución más débil de la ecuación (en Y):

$$Y : \{Y\}S\{X\}$$

Así, de la Definición 3.4 deducimos la interpretación del triplete $\{Y\}S\{X\}$: *partiendo de un estado verificando* Y*, el programa* S *termina en un estado verificando* X. Luego,

> $S.X$ DENOTA LA PRECONDICIÓN MÁS DÉBIL QUE GARANTIZA LA TERMINACIÓN DE S SATISFACIENDO EL PREDICADO X.

En otras palabras: $S.X$ captura el mayor conjunto de estados iniciales que garantiza la terminación en un estado determinado por X. Gráficamente:



✓ Si consideramos **corrección parcial**, $wlp.S.X$ es la solución más débil de la ecuación en Y:

$$Y : \{Y\}\widehat{S}\{X\}$$

Es decir, $\{Y\}\widehat{S}\{X\}$ se interpreta: *partiendo de un estado verificando* Y*, si el programa* S *termina, lo hace en un estado verificando* X. En definitiva,

$$[Y \Rightarrow S.X] \equiv \{Y\}S\{X\} \qquad \text{— corrección total,}$$
$$[Y \Rightarrow \widehat{S}.X] \equiv \{Y\}\widehat{S}\{X\} \qquad \text{— corrección parcial.}$$

3.1. Capturando propiedades de Programas vía propiedades de Transformadores

Si para cada sentencia del lenguaje podemos derivar su transformador de predicados entonces el lenguaje está *bien definido*. Además, dos sentencias S y S' sintácticamente diferentes pero con idénticos transformadores:

$$\forall X : X \in \mathcal{P} : [S.X \equiv S'.X]$$

no serán distinguibles ya que según la Definición 3.4,

$$\forall X, Y : X, Y \in \mathcal{P} : \{Y\}S\{X\} \equiv \{Y\}S'\{X\}.$$

Entonces, utilizando la interpretación de los tripletes, es fácil deducir que no podemos distinguir experimentalmente la ejecución de S de la de S'.

3.5 *[255] Probad la última afirmación. Para ello, consideremos para cada estado σ el predicado P^σ, definido por*

$$P^\sigma.\alpha = \begin{cases} C & \text{, si } \sigma \equiv \alpha \\ F & \text{, en otro caso.} \end{cases}$$

P^σ se llama el indicador *del estado σ, y determina un conjunto con un único estado (el propio σ). Justificad/interpretad que si $S = S'$ entonces se verifica*

$$\forall \iota, \sigma : \iota, \sigma \in \mathcal{E} : \{P^\iota\}S\{P^\sigma\} \equiv \{P^\iota\}S'\{P^\sigma\}.$$

OBSERVACIÓN.– Interpretemos el predicado $S.C$; puesto que los estados verificando el predicado C ($Cierto$) son todos los estados del sistema, para que se cumpla $\{Q\}S\{C\}$ es suficiente que S *termine*:

$$S.C \quad \equiv \quad \text{estados iniciales para los cuales } S \text{ termina.}$$

Por tanto, si se verifica $[S.C] \equiv C$, el programa S termina siempre. OBS

EJEMPLO 3.6 Sea $\mathcal{M}$ ($\mathcal{M}oneda$) el mecanismo o programa que simula la tirada de una moneda asignando a la variable x el resultado $cara$ o el resultado $cruz$. Analicemos el valor $\mathcal{M}.(x = cara)$. Si el mecanismo *simula perfectamente* el lanzamiento de una moneda entonces se debe verificar $[F \equiv \mathcal{M}.(x = cara)]$. En efecto; si para cierto estado ι se tiene $(\mathcal{M}.(x = cara))_\iota$, entonces, partiendo de ι aseguramos que $\mathcal{M}$ termina con $x = cara$, que es imposible ya que ningún estado inicial asegura el resultado final del lanzamiento. En definitiva deberán cumplirse las identidades,

$$[F \equiv \mathcal{M}.(x = cara)] \qquad [F \equiv \mathcal{M}.(x = cruz)]. \tag{1}$$

Obsérvese que de las anteriores se deducen los tripletes:

$$\{F\}\mathcal{M}\{x = cara\} \qquad \{F\}\mathcal{M}\{x = cruz\}.$$

Por otro lado, será interesante asegurar que $\mathcal{M}$ siempre termina y que el resultado es o $cara$ o $cruz$:

$$[C \equiv \mathcal{M}.(x = cara \vee x = cruz)] \tag{2}.$$

Las ecuaciones (1) y (2) caracterizan a $\mathcal{M}$: siempre termina y lo hace asignando a la variable x el valor $cruz$ o el valor $cara$, y de forma indeterminista. EJEMPLO

Aunque después veremos la definición formal de sustitución (véase Definición 4.2 en página 54), con objeto de dar otro ejemplo interesante, adelantemos el transformador sustitución. Si Z es un predicado donde puede aparecer la variable x, el predicado $x := E.Z$ se obtiene reemplazando en Z todas las apariciones *libres* de la variable x por el valor E.

EJEMPLO 3.7 Sea $\mathcal{N}$ el transformador de predicados definido, $\forall Z$, y *ptle*,

$$\mathcal{N}.Z \doteq x := cara.Z \wedge x := cruz.Z$$

Con muy poquito cálculo podemos deducir las siguientes propiedades, *ptle*:

$$F \equiv \mathcal{N}.(x = cara) \qquad F \equiv \mathcal{N}.(x = cruz) \qquad C \equiv \mathcal{N}.(x = cara \vee x = cruz)$$

En efecto; veamos la primera

$$\begin{array}{ll} & \mathcal{N}.(x = cara) \\ = & \quad \text{definición} \\ & x := cara.(x = cara) \wedge x := cruz.(x = cara) \\ = & \quad \text{definición de sustitución} \\ & cara = cara \wedge cruz = cara \\ = & \\ & Falso \end{array}$$

Las propiedades anteriores conducen a los tripletes:

$$\{F\}\mathcal{N}\{x = cara\} \qquad \{F\}\mathcal{N}\{x = cruz\} \qquad \{C\}\mathcal{N}\{x = cara \vee x = cruz\}.$$

En definitiva el transformador $\mathcal{N}$ satisface las propiedades (1) y (2) del programa $\mathcal{M}oneda$. La diferencia esencial entre $\mathcal{N}$ y $\mathcal{M}$ es que el transformador $\mathcal{N}$ está completamente determinado, pero $\mathcal{M}$ no es conocido para todos los predicados. Por ejemplo, si z es una variable diferente de x, tendremos, *ptle*

$$(z = a) \equiv \mathcal{N}.(z = a)$$

que conduce al triplete $\{(z = a)\}\mathcal{N}\{z = a\}$, y se interpreta en forma trivial: el programa $\mathcal{N}$ no altera el valor de la variable z. Como se desprende fácilmente de la definición, $\mathcal{N}$ solo altera el valor de la variable x. Por el contrario, no sabemos nada sobre $\mathcal{M}.(z = a)$. EJEMPLO

En este punto, el lector debería saber definir un transformador de predicados que represente la tirada de un dado.

EJEMPLO 3.8 Otro transformador igual de interesante que los anteriores es el definido por la propiedad, $\forall Z$, y *ptle*,

$$\boxed{[Z] \;=\; desastre.Z}$$

denominado *havoc* en [Dijkstra y Scholten, 1990]:133. Veamos la interpretación del programa $desastre$ y de ésta la justificación de su nombre. Si consideramos el espacio de estados correspondiente a la declaración de variable $x :\in \{-1, 0, 1\}$, es fácil comprobar $\forall i : -1 \leq i \leq 1 : [x = i] \equiv F$, y de aquí, *ptle*

$$C \equiv desastre.C \qquad\qquad \forall i : -1 \leq i \leq 1 : F \equiv desastre.(x = i)$$

que se interpreta en la forma siguiente: la ejecución de $desastre$ termina siempre pero conduce a un estado arbitrario, ya que no es posible asegurar el valor final de ninguna variable. Es un mecanismo completamente indeterminista.

Consideremos ahora el espacio de estados correspondiente a la única declaración $x :\in \mathbb{N}$; en ese caso tenemos

$$\forall i : i \in \mathbb{N} : \{X\} desastre \{x = i\} \equiv Falso$$

para cualquier predicado X que no sea idénticamente Falso.

El tipo de indeterminismo que introduce $desastre$ es no acotado en espacios de estado con una variable natural x, ya que ningún estado inicial garantiza que el valor final de la variable x quede acotado después de la ejecución. EJEMPLO

3.9 *Cuál de los programas es más indeterminista, $Azar_x$ o $desastre$, donde, ptle,*

$$Azar_x.Z \doteq (\forall i : i \in \mathbb{N} : x := i.Z)$$

3.2. Propiedades de salubridad

Cualquier transformador de predicados no sirve para capturar cómputos desde el punto de vista operacional; esto significa que existen transformadores que no corresponden a ningún programa real. Así, los transformadores de predicados correspondientes a las sentencias de un lenguaje *real* deben verificar una serie de propiedades conocidas como propiedades de *salubridad*; daremos una justificación de estas propiedades utilizando la interpretación de wp. Utilizaremos también el término *sano* como sinónimo de *salubre*.

La ley del milagro excluido Sea S un programa; si $S.F$ no es idénticamente F ($Falso$) existe un estado ι verificando $S.F$; partiendo de este estado el programa S terminará en otro estado σ verificando F, lo cual es absurdo. Por tanto deberá tenerse

LEY DEL MILAGRO EXCLUIDO: $[S.F \equiv F]$

Dijkstra llama a la ley anterior *Law of the Excluded Miracle*.

3.10 *[255] ¿Bajo qué condiciones se cumple $\vdash_{\mathcal{D}} \{Falso\}S\{Q\}$? ¿Y $\vdash_{\mathcal{D}} \{P\}S\{Falso\}$?*

DEFINICIÓN 3.11 *En el conjunto $\mathcal{T}$ de transformadores de predicados definimos la relación $\leq$ como:*

$$S \leq T \doteq \forall X : X \in \mathcal{P} : [\, S.X \Rightarrow T.X \,]$$

Puesto que el ínfimo de $\mathcal{P}$ es el predicado idénticamente $Falso$, la 'Ley del Milagro Excluido' viene a decir que el transformador $S : \mathcal{P} \to \mathcal{P}$ correspondiente a un programa *real* debe ser estricto. Denotaremos con $\mathcal{T}^{\perp}$ el conjunto de transformadores estrictos. Entonces el conjunto $\mathcal{P}rog$ de programas sanos debe cumplir $\mathcal{P}rog \subseteq \mathcal{T}^{\perp}$.

3.12 *Probad e interpretad la siguiente propiedad*

$$S \leq T \quad \Rightarrow \quad \forall X, Y : X, Y \in \mathcal{P} : \{Y\}S\{X\} \Rightarrow \{Y\}T\{X\}.$$

Conjuntividad Sea $\mathcal{C} = \{R, R', \dots\}$ una colección arbitraria de predicados y S un programa; entonces, para cada estado inicial ι deducimos la siguiente intepretación[2]

$$
\begin{array}{ll}
 & (S.(R \wedge R' \wedge \dots))_\iota \\
= & \quad \because \text{interpretación del transformador } S\text{; véase página 43: } S.X \text{ DENOTA } \dots \\
 & \textit{partiendo de } \iota\textit{, } S \textit{ termina en un estado final verificando } R \wedge R' \wedge \dots \\
= & \\
 & \textit{partiendo de } \iota\textit{, } S \textit{ termina en un estado final verificando } R \\
 & \wedge \\
 & \textit{partiendo de } \iota\textit{, } S \textit{ termina en un estado final verificando } R' \\
 & \wedge \\
 & \dots \\
= & \\
 & (S.R)_\iota \wedge (S.R')_\iota \wedge \dots \\
= & \quad \because \text{definición} \\
 & (S.R \wedge S.R' \wedge \dots)_\iota
\end{array}
$$

En definitiva: $(S.(R \wedge R' \wedge \dots))_\iota \equiv (S.R \wedge S.R' \wedge \dots)_\iota$, para cualquier estado ι; o sea, la propiedad de conjuntividad (universal):

CONJUNTIVIDAD:
$[S.(R \wedge R' \wedge \dots) \equiv S.R \wedge S.R' \wedge \dots]$

Con una notación más rigurosa la propiedad anterior se escribe en la forma:

$$\forall \mathcal{C} : \mathcal{C} \subseteq \mathbb{P}(\mathcal{P}) : S.(\bigwedge_{X \in \mathcal{C}} X) = \bigwedge_{X \in \mathcal{C}} S.X$$

Es decir, los transformadores de predicados asociados a los programas deben ser conjuntivos ($\in \mathcal{T}^\wedge$); por tanto $\mathcal{P}rog \subseteq \mathcal{T}^\wedge \cap \mathcal{T}^\perp = \mathcal{T}^{\wedge\perp}$.

EJEMPLO 3.13 No todos los transformadores son conjuntivos; un ejemplo simple es el transformador $T.X \doteq \neg X$ para el cual se verifica:

$$T.C \wedge T.F = F \neq C = T.(C \wedge F)$$

No existe un programa *real* con tal transformador; por otro lado es también evidente que tal transformador no es estricto. Conclusión: no existe un programa S que resuelva el problema $\forall X : X \in \mathcal{P} : \{X\}S\{\neg X\}$ EJEMPLO

DEFINICIÓN 3.14 *Diremos que un mecanismo o programa S es sano si su transformador de predicados S es estricto y conjuntivo.*

NOTA 3.15 (Caracterización semántica de Dijkstra/Scholten) *Como dijimos en Nota 3.1, [Dijkstra y Scholten, 1990] caracterizan una sentencia vía los transformadores wp y wlp. Se exige que $wp.S$ sea estricto (ley del milagro excluido) y que $wlp.S$ sea conjuntivo. Luego $wlp.S$ no necesariamente debe ser estricto, pero $wp.S$ debe ser conjuntivo.*

[2] Como es habitual, la aplicación tiene precedencia sobre los operadores lógicos; y por ello

$$S.A \wedge B \equiv (S.A) \wedge B \not\equiv S.(A \wedge B).$$

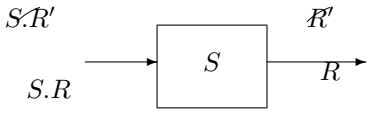
TEOREMA 3.16 (Monotonía de los programas) *Todo transformador conjuntivo es también monótono.*

Demostración.– Hay que probar: $[X \ \Rightarrow \ Y] \ \Rightarrow \ [S.X \ \Rightarrow \ S.Y]$. En efecto,

$$
\begin{array}{ll}
 & [X \ \Rightarrow \ Y] \\
= & \quad \because \text{regla de oro} \\
 & [X \wedge Y = X] \\
\Rightarrow & \quad \because \text{regla de Leibniz} \\
 & [S.(X \wedge Y) = S.X] \\
= & \quad \because S \text{ es conjuntivo} \\
 & [S.X \wedge S.Y = S.X] \\
= & \quad \because \text{regla de oro} \\
 & [S.X \ \Rightarrow \ S.Y]
\end{array}
$$

TEOREMA

Luego, todos los programas sanos satisfacen la propiedad de monotonía. Tal propiedad tiene una interpretación sencilla: para obtener una poscondición R más restrictiva que R', tendremos que partir de un conjunto de estados $S.R$ más pequeño que $S.R'$:



Concepto de programa útil Un programa o mecanismo S es útil según [Hehner, 1984]:148, si existe un predicado R tal que $[S.R]$; es decir, $\{C\}S\{R\}$. Por monotonía obtenemos $[S.R \ \Rightarrow \ S.C]$, y de aquí la equivalencia

$$[S.C] = \exists R : R \in \mathcal{P} : [S.R]$$

En consecuencia, un programa es útil si termina siempre. Hay programas útiles, como por ejemplo $x := x$, y programas que no lo son, como por ejemplo

$$x, y :\in Int; x := x \div y$$

ya que no se puede garantizar que termine para todos los estados iniciales. Realmente aquí útil significa *terminación propia*. El conjunto de programas útiles será denotado con $\mathcal{T}^{\top}$.

3.3. Determinismo y disyuntividad

Los transformadores asociados a los programas NO son necesariamente disyuntivos; un ejemplo de ello lo proporciona el programa $\mathcal{M}$ del Ejemplo 3.6, para el cual tenemos

$$\mathcal{M}.\{x = cara\} \vee \mathcal{M}.\{x = cruz\} \not\equiv \mathcal{M}.\{x = cara \vee x = cruz\}$$

Es fácil justificar que los programas deterministas verifican la siguiente propiedad de *disyuntividad*, válida para toda colección de predicados $\{R, R' \ \ldots\}$:

$$[S.(R \vee R' \vee \ \ldots) \equiv S.R \vee S.R' \vee \ \ldots]$$

Justificación.- Por aplicación del Lema 1.9 (página 15), basta justificar o probar las dos siguientes implicaciones, *ptle*:

$$S.R \vee S.R' \vee \ldots \;\Rightarrow\; S.(R \vee R' \vee \ldots) \tag{1}$$

$$S.(R \vee R' \vee \ldots) \;\Rightarrow\; S.R \vee S.R' \vee \ldots \tag{2}$$

La implicación (1) es consecuencia inmediata de la propiedad de monotonía. y bastaría justificar la implicación (2). Consideremos un estado inicial ι verificando $S.(R \vee R' \vee \ldots)$; entonces,

$$\text{por ser } S \text{ determinista, existe un único estado final } \sigma \text{ verificando } R \vee R' \vee \ldots \tag{3}$$

Supongamos que $(S.R)_\iota$ sea falso; entonces, partiendo del estado ι, el único estado final posible σ no puede verificar R (de lo contrario, si podemos asegurar que el estado final satisface R, es que el inicial aparece en $S.R$, que es absurdo). Pero si R_σ es falso, el estado final no puede aparecer en R. Por tanto, si todos los predicados $(S.R)_\iota$, $(S.R')_\iota$, $\ldots$, son falsos, el estado final σ no puede aparecer en $R \vee R' \vee \ldots$, que está en contradicción con (3). Luego alguno de los predicados $(S.R)_\iota$, $(S.R')_\iota$, $\ldots$, debe ser cierto. ■

El razonamiento anterior justifica la siguiente definición:

DEFINICIÓN 3.17 *Sea S un programa (sano); S se dice determinista si su transformador es disyuntivo. S de dice indeterminista si su transformador no es disyuntivo.*

3.18 *[255] Justificad la Definición 3.17 probando la propiedad: si S es sano y disyuntivo, entonces, dado un estado inicial ι para el cual S termina, el estado final es único.*

3.19 *[256]* **(Febrero, 97)** *Sea S sano y no disyuntivo. Dad otra justificación de la Definición 3.17 probando las siguientes propiedades:*

(A) Existe un estado ι y dos predicados A y B tales que $(S.A)_\iota \equiv Falso$, $(S.B)_\iota \equiv Falso$ y $(S.(A \vee B))_\iota \equiv Cierto$.

(B) Partiendo de ι, S puede terminar en, al menos, dos estados distintos.

Observe el lector que los dos ejercicios anteriores justifican completamente la Definición 3.17: S es disyuntivo si y solo si, el estado final, si existe, es único.

DEFINICIÓN 3.20 (Conjugado de un transformador) *Dado un transformador S, definimos el conjugado S^* en la forma, ptle, $S^*.X \doteq \neg S.\neg X$*

TEOREMA 3.21 *Se verifican las siguientes propiedades*

(i) $S \in \mathcal{T}^\wedge \Longleftrightarrow S^* \in \mathcal{T}^\vee$, $\quad S \in \mathcal{T}^\vee \Longleftrightarrow S^* \in \mathcal{T}^\wedge$

(i') $S \in \mathcal{T}^\top \Longleftrightarrow S^* \in \mathcal{T}^\bot$, $\quad S \in \mathcal{T}^\bot \Longleftrightarrow S^* \in \mathcal{T}^\top$

(ii) *Si* $S \in \mathcal{T}^{\wedge\bot}$ *entonces* $S \leq S^*$

(iii) *Si* $S \in \mathcal{T}^{\wedge\bot}$ *entonces*
$S = S^* \Longleftrightarrow S$ *es determinista y termina siempre*

Demostración.– Veamos solo algunas implicaciones.
(iii) Sea $S \in \mathcal{T}^{\wedge\bot}$; entonces, por $(i) - (i')$, $S^* \in \mathcal{T}^{\vee\top}$; de donde $S = S^* \;\Rightarrow\; S \in \mathcal{T}^{\vee\top}$. Para probar la otra implicación, teniendo en cuenta (ii), basta probar $S \in \mathcal{T}^{\vee\top} \;\Rightarrow\; S^* \leq S$; veamos:

$S^* \leq S$
$=$ $\quad \because$ definición de orden en $\mathcal{P}$
$\forall X : X \in \mathcal{P} : [S^*.X \ \Rightarrow \ S.X]$
$=$ $\quad \because$ definición de S^*
$\forall X : X \in \mathcal{P} : [\neg S.\neg X \ \Rightarrow \ S.X]$
$=$ $\quad \because$ cálculo
$\forall X : X \in \mathcal{P} : [S.\neg X \vee S.X]$
$=$ $\quad \because$ S es disyuntivo, tercio excluido
$\forall X : X \in \mathcal{P} : [S.C]$
$=$ $\quad \because$ S es útil
$Cierto$

$\underline{(i)}$. Sea $S \in \mathcal{T}^{\wedge}$; tenemos, *ptle*

$S^*.(A \vee B \vee \ldots)$
$=$ $\quad \because$ definición de S^*
$\neg S.\neg(A \vee B \vee \ldots)$
$=$ $\quad \because$ de Morgan
$\neg$S.$(\neg A \wedge \neg B \wedge \ldots)$
$=$ $\quad \because$ conjuntividad
$\neg(S.\neg A \wedge$ S.$\neg B \wedge \ldots)$
$=$ $\quad \because$ de Morgan
$\neg S.\neg A \vee \neg S.\neg B \vee \ldots$
$=$ $\quad \because$ definición de S^*
$S^*.A \vee S^*.B \vee \ldots$

$\underline{(ii)}$. Sea $S \in \mathcal{T}^{\wedge\perp}$

$S \leq S^*$
$=$ $\quad \because$ definición de orden en $\mathcal{P}$
$\forall X : X \in \mathcal{P} : [S.X \ \Rightarrow \ S^*.X]$
$=$ $\quad \because$ definición de S^*
$\forall X : X \in \mathcal{P} : [S.X \ \Rightarrow \ \neg S.\neg X]$
$=$ $\quad \because$ cálculo
$\forall X : X \in \mathcal{P} : [\neg S.X \vee \neg S.\neg X]$
$=$ $\quad \because$ definición de S^*
$\forall X : X \in \mathcal{P} : [S^*.\neg X \vee S^*.X]$
$=$ $\quad \because$ (i), S^* es disyuntivo
$\forall X : X \in \mathcal{P} : [S^*.C]$
$=$ $\quad \because$ por (i'), S es útil
$Cierto$

TEOREMA

NOTA 3.22 Determinismo según Dijkstra/Scholten *Por Teorema 3.20(ii), si S es autoconjugado entonces es determinista. [Dijkstra y Scholten, 1990]:131 dan una definición más restrictiva: S es determinista sii $S^* = \widehat{S}$.*

3.23 *[256] ¿Es equivalente la definición de la Nota 3.22 con la dada en la Definición 3.17?*

3.24 *[256] Siendo $[U.X \ \doteq \ b \vee S.X]$, probad que U es sana.*

3.25 *Sea S una sentencia que conserva dos predicados P y Q en presencia de otro predicado b; es decir, verificando $\{b \wedge P\}S\{P\}$ y $\{b \wedge Q\}S\{Q\}$. Estudiad la forma más general del operador $f : \mathcal{P} \to \mathcal{P} \to \mathcal{P}$ de forma que S conserve el predicado $f.P.Q$; es decir: $\{b \wedge f.P.Q\}S\{f.P.Q\}$.*

3.26 **(Diciembre, 99)** *El siguiente ejercicio trata de mostrar que conocido el transformador de predicados de cierta sentencia es posible deducir algunas propiedades de ésta. Así pues, siendo x una variable natural, y N una constante natural, sea la sentencia $Azar_{xN}$ con transformador de predicados, ptle*

$$Azar_{xN}.Z \ \doteq \ \forall i : 0 \leq i \leq N : x := i.Z$$

Probad que $Azar_{xN}$ es indeterminista y termina siempre.

3.27 *Sean dos transformadores f y g tales que f es más indeterminista que g; es decir, $f \leq g$. Probad:*

(A) $[g.X \equiv Falso] \ \Rightarrow \ [f.X \equiv Falso]$ ¿Qué interpretación tiene?

(A') $[f.X] \ \Rightarrow \ [g.X]$ ¿Qué interpretación tiene?

(B) El transformador $Azar_{x\infty}$ es más indeterminista que $Azar_{xN}$.

3.28 *Demostrad que el transformador $desastre$ del Ejemplo 3.8 es conjuntivo y estricto. Además es indeterminista para espacios de estados con más de un estado.*

Capítulo 4

Un lenguaje de Programación simple

La semántica de un programa S es su transformador de predicados, que según el Convenio 3.3, identificamos con su forma sintáctica. Por ello, sería interesante dar un método para caracterizar la semántica de S conocida su forma sintáctica. Una forma elegante de conseguirlo es a través de una definición semántica *composicional*, que sigue dos fases

$(\mathcal{F}_1)$ dar la semántica de las sentencias simples, y

$(\mathcal{F}_2)$ describir reglas para obtener la semántica de las sentencias compuestas a partir de la semántica de sus componentes.

En términos de trasformadores de predicados, esto significa que debemos

$(\mathcal{F}_1)$ dar los transformadores de predicados de las sentencias simples, y

$(\mathcal{F}_2)$ describir reglas para obtener el transformador de cada sentencia compuesta, conocidos los trasformadores de sus componentes.

La fase $\mathcal{F}_2$ se logra asociando un constructor de transformadores para cada constructor sintáctico del lenguaje. Debemos asegurar que los constructores del lenguaje conserven las propiedades de salubridad. Por ejemplo, si el constructor $\ll;\gg$ que genera la composición secuencial $\ll S; S' \gg$ tiene asociado la composición funcional, ésta debe conservar las propiedades de salubridad; o sea, si S y S' son sanas, $S \circ S'$ deberá ser sana.

4.0. Las sentencias más simples: $nada$ y $aborta$

Los transformadores o funciones $S : \mathcal{P} \to \mathcal{P}$ más simples son la identidad y las constantes. Consideremos el primer caso, o sea, la función identidad, que será denotada con $nada$, y queda definida así:

$$\boxed{\forall X :: [nada.X \doteq X]}$$

Si recordemos la Definición 3.4 de triplete en el sentido de Dijkstra:

$$\{Y\}S\{X\} \doteq [Y \Rightarrow S.X]$$

la sentencia $nada$ verificará, para todo predicado R, el triplete $\{R\}nada\{R\}$, por lo que para obtener la poscondición R después de ejecutar $nada$ es suficiente partir de la misma precondición R. Experimentalmente (o sea, observando

la ejecución de la sentencia) es imposible distinguir $nada$ de la sentencia que *no hace nada* (no modifica el estado inicial del sistema). En efecto: para cada estado inicial ι, el indicador P^ι (véase Ejercicio 3.5) verifica $\{P^\iota\}nada\{P^\iota\}$, y el estado final después de ejecutar $nada$ coincide con el estado inicial.

La sentencia $nada$ aparece en algunos lenguajes de programación con un identificador especial. En el lenguaje ADA se escribiría $null$; en un lenguaje ensamblador se denota con $skip$; en PASCAL o MODULA sería la sentencia 'vacía'.

Estudiemos ahora los transformadores constantes; existen dos posibilidades ya que dicha constante puede ser F o C, y tendremos dos transformadores, que llamaremos $aborta$ y $milagro$.

EJEMPLO 4.0 El transformador $milagro$ es el transformador constante

$$\forall X :: [milagro.X \doteq C]$$

y no verifica la *Ley del Milagro Excluido*, ya que según la definición anterior $[milagro.F = C]$. Por tanto tal sentencia no aparecerá en ningún lenguaje de programación *real*. Un razonamiento simple nos convence de que esta sentencia no es *implementable*. En efecto, para cualquiera poscondición R obtendríamos $\{C\}milagro\{R\}$. Es decir, *tendríamos resuelto el problema de la programación*; por ejemplo

$$\{C\}milagro\{x = \text{máx}(a, b, c)\}$$
$$\{C\}milagro\{\text{la tabla } a[1..n] \text{ está ordenada}\}$$

Tales milagros serán excluidos de nuestro lenguaje. EJEMPLO

Interpretemos el otro transformador constante que llamaremos $aborta$:

$$\boxed{\forall X :: [aborta.X \doteq F]}$$

Veamos que $aborta$ no puede terminar una vez iniciada. En efecto, sea ι un estado inicial. Entonces, por definición de $aborta$, $\{P^\iota\}aborta\{C\} \equiv F$. Si terminase la sentencia $aborta$ en cierto estado σ, tal estado verificaría el predicado C, lo que contradice el triplete anterior. Así, la sentencia $aborta$ permite capturar aquellos mecanismos que provocan errores o no terminación.

Es un ejercicio simple probar que los transformadores $nada$ y $aborta$ son estrictos y conjuntivos, y además deterministas; por consiguiente tienen asociados dos sentencias *reales*.

NOTA 4.1 *Si seguimos a [Dijkstra y Scholten, 1990] (véase Nota 3.1) y definimos la sentencia* $aborta$ *en la forma*

$$\forall X :: [wlp.aborta.X \doteq Cierto] \wedge [wp.aborta.Cierto \doteq Falso]$$

obtenemos la definición anterior:

$$\forall X :: [wp.aborta.X \equiv Falso]$$

de donde (obsérvese que wlp *no es estricto)* $aborta$ *es determinista en la dos definiciones y además* $aborta$ *no es autoconjugado, ya que,* $ptle$*:*

$$\begin{aligned} & aborta^*.X = aborta.X \\ = \quad & \langle\cdot\cdot\rangle \text{ definición de } S^*, \text{ semántica} \end{aligned}$$

$$
\begin{array}{ll}
 & \neg aborta.\neg X = F \\
= & \quad \because \textit{semántica, CP} \\
 & F
\end{array}
$$

Esto es debido a que, según el Teorema 3.21(i), la equivalencia entre determinismo y autoconjugación es válida solamente para programas útiles (que terminan siempre) y $aborta$ no termina. Obsérvese además que $[wlp.aborta.X = C]$ se interpreta con tripletes parciales:

$$\{C\}\widehat{aborta}\{X\}$$

que puede leerse en la forma: 'si $aborta$ termina, lo hace verificando X'.

Otros transformadores interesantes Además de los anteriores, existen otros transformadores simples, unos sanos y otros no; por ejemplo, cada predicado P tiene asociado un transformador $\widetilde{P}$ dado por

$$\boxed{[\widetilde{P}.X \;\doteq\; P \wedge X]}$$

Trivialmente $\widetilde{P}$ es estricto y conjuntivo, por lo que podríamos hacerle corresponder un mecanismo. La utilidad de tal transformador consiste en introducir comentarios en los programas. Además es útil en un *lenguaje de especificaciones*, como el propuesto en [Hehner, 1984] o en [Morris, 1990]. Obsérvese que se verifica $\widetilde{Cierto} = nada$ y $\widetilde{Falso} = aborta$.

Otra forma de obtener un transformador a partir de un predicado y una sentencia es en la forma siguiente

$$[\langle B \to S\rangle.X \;\doteq\; B \;\Rightarrow\; S.X]$$

Es fácil ver que si S es conjuntivo también lo es $\langle B \to S\rangle$, sin embargo, el transformador $\langle B \to S\rangle$ no es estricto, puesto que puede introducir milagros si es ejecutado en el entorno del predicado $\neg B$. En efecto, para *ptle*

$$
\begin{array}{ll}
 & \langle B \to S\rangle.F \\
= & \quad \because \text{definición} \\
 & B \;\Rightarrow\; S.F \\
= & \quad \because S \text{ es estricto} \\
 & B \;\Rightarrow\; F \\
= & \quad \because \text{cálculo} \\
 & \neg B
\end{array}
\qquad\qquad
\begin{array}{ll}
 & \langle B \to S\rangle.(X \wedge Y \wedge \ldots) \\
= & \quad \because \text{definición} \\
 & B \;\Rightarrow\; S.(X \wedge Y \wedge \ldots) \\
= & \quad \because S \text{ es conjuntivo} \\
 & B \;\Rightarrow\; S.X \wedge S.Y \wedge \ldots \\
= & \quad \because \text{cálculo} \\
 & (B \;\Rightarrow\; S.X) \wedge (B \;\Rightarrow\; S.Y) \wedge \ldots \\
= & \quad \because \text{definición} \\
 & \langle B \to S\rangle.X \wedge \langle B \to S\rangle.Y \wedge \ldots
\end{array}
$$

Al igual que $\widetilde{P}$, $\langle B \to S\rangle$ será una especificación, útil en metodologías de diseño de programas con especificaciones.

4.1. La sustitución como transformador de predicados. La sentencia de asignación

Como vimos en Ejemplo 3.7, la sustitución de una variable por una expresión captura un cambio de estado, al que debemos asociar una sentencia, que

se identifica con el propio transformador sustitución '$x := E$': sustituir todas las apariciones libres de x por la expresión E.

DEFINICIÓN 4.2 *Si Z es un predicado, $x := E.Z$ se define por inducción estructural sobre la estructura del predicado Z:*

$$\begin{array}{lcll} x := E.Falso & \doteq & Falso & \\ x := E.Cierto & \doteq & Cierto & \\ x := E.(A \vee B) & \doteq & x := E.A \vee x := E.B & \\ x := E.(A \wedge B) & \doteq & x := E.A \wedge x := E.B & \\ x := E.\neg A & \doteq & \neg(x := E.A) & \\ x := E.(\forall i :: X) & \doteq & \forall i :: x := E.X & \textit{si } x \not\equiv i \\ x := E.(\exists i :: X) & \doteq & \exists i :: x := E.X & \textit{si } x \not\equiv i \end{array}$$

Si A es una expresión sin conectivas lógicas se define $x := E.A$ por inducción estructural sobre la sintaxis de la expresión:

$$\begin{array}{lcl} x := E.(M \otimes N) & \doteq & x := E.M \otimes x := E.N \\ x := E.x & \doteq & E \\ x := E.y & \doteq & y, \textit{ si } y \not\equiv x \end{array}$$

donde $\otimes$ es cualquier operador.

NOTA 4.3 (Notaciones para expresar la *sustitución)* *La notación $x := E.P$ es básicamente la propuesta en [Dijkstra y Scholten, 1990]; hemos seleccionado ésta por ser una notación muy expresiva; $x := E.P$ se lee:*

el transformador de predicados $(x := E)$ actuando sobre P

y resulta interesante utilizar el operador $x := E$ en forma prefija. Otra razón para la elección de esta notación, más pragmática, está relacionada con la identificación del transformador sustitución *con la sentencia de asignación.*

En la literatura aparecen también otras notaciones. [Wirth y Hoare, 1973] utilizan la notación P^x_E, cuya principal desventaja es que no es lineal. Otra notación interesante es: $P(x\backslash E)$, que está justificada porque bajo ciertas condiciones se cumple la propiedad $P(x\backslash E)(E\backslash u) = P(x\backslash u)$ que recuerda la propiedad de la división.

Como se deduce de la Definición 4.2, el transformador $x := E$ es estricto, conjuntivo y disyuntivo; por tanto, podemos hacerle corresponder una sentencia determinista que se denota igualmente con $x := E$ y se denomina *sentencia de asignación*. Nótese que identificamos perfectamente la sentencia sintáctica con su transformador.

Algunos ejemplos de cálculos son, *ptle*

$$\begin{array}{lcl} a := a + 1.a > 10 & \equiv & a > 9 \\ a := 7.a = 7 & \equiv & C \\ a := 7.a = 6 & \equiv & F \end{array}$$

El último ejemplo ilustra que no existe ningún estado que conduzca a la poscondición $a = 6$ después de ejecutar la sentencia $a := 7$.

Asignación estricta La sustitución $x := E$ da problemas si la expresión lleva a un estado erróneo. Por ejemplo, tomando $P \equiv x\ es\ par$, entonces $(x := x + 2.P) \equiv (x + 2\ es\ par)$; pero es sabido que en ciertas implementaciones la expresión $x+2$ puede producir *overflow*. Una forma elegante de resolver el problema es considerar semánticas estrictas como en [Hehner, 1984]:130, donde se asocia a cada variable una *expresión de tipo*; por ejemplo, para la declaración

$$x, y :\in \mathbb{N}; z :\in \mathbb{Z}$$

la expresión $tp(x)$, que denota la expresión de tipo asociada a la variable x, es $'x :\in \mathbb{N}'$. Una expresión de tipo se transforma en un predicado si sustituimos la variable por una expresión y suprimimos los dos puntos; por ejemplo

$$[x := 8]'x :\in \mathbb{N}' \ \doteq\ 8 \in \mathbb{N}$$

Con esta notación, la semántica estricta de la asignación se puede escribir

$$\boxed{\langle\!\langle x := 8 \rangle\!\rangle.X \qquad \doteq \qquad x := E.tp(x) \wedge x := E.X}$$

(entrecomillamos en este caso la sentencia para distinguirla del transformador). Según esto, si $x :\in \mathbb{R}$, tendremos, $\langle\!\langle x := 1/0 \rangle\!\rangle = aborta$.

Asignaciones y efectos laterales En la asignación $x := E$ la expresión E no debe contener llamadas a funciones con efectos laterales; por ejemplo, si consideramos la función PASCAL

```
function dos : integer;
begin
    x := 5;
    dos := 2;
end;
```

la ejecución de la función *dos* provoca un efecto lateral sobre la variable x; tenemos, según la semántica de la asignación

$$\begin{array}{ll} & \{dos = 2 \wedge x < 1\} x := dos + x \{x < 3\} \\ = & \quad \because \text{definición de triplete} \\ & [\ dos = 2 \wedge x < 1 \ \Rightarrow\ x := dos + x.(x < 3)\ ] \\ = & \quad \because \text{semántica de la asignación} \\ & [\ dos = 2 \wedge x < 1 \ \Rightarrow\ dos + x < 3\ ] \\ = & \\ & Cierto \end{array}$$

pero operacionalmente tal triplete no es cierto ya que dos no es una variable sino la llamada a una función, cuya ejecución provoca un efecto lateral sobre x. Resaltamos que los efectos laterales en programación complican la definición semántica y la verificación.

Asignación paralela Una extensión natural de la sentencia de asignación es la *asignación paralela*: sustituir simultáneamente un conjunto de variables. Por ejemplo, para la sentencia:

$$x, y := M, N$$

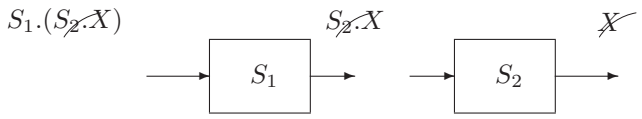


Figura 4.0: Composición secuencial de transformadores.

su transformador indica la sustitución simultánea y se define en forma análoga a la Definición 4.2:

$$a, b := a + 1, b - 2.(a + b > 10) \;\doteq\; a + 1 + b - 2 > 10 \;\equiv\; a + b > 11$$

Observemos que la sentencia $x, y := y, x$ permite el intercambio de los valores de las variables x e y:

$$\{x = a \wedge y = b\}x, y := y, x\{x = b \wedge y = a\}$$

En este punto tenemos definidas (caracterizadas semánticamente) las sentencias básicas del lenguaje (aquellas sentencias que no contienen sentencias en su código)

$$sentencia\_simple ::= nada \mid aborta \mid x := E \mid x, y := E, F \mid \ \ldots$$

4.2. Composición de sentencias

La forma más simple de obtener nuevos transformadores es por composición funcional; veremos que esta operación se corresponde con la composición secuencial de sentencias. Dadas dos sentencias S_1 y S_2 queremos definir una nueva sentencia cuyo significado sea *activar S_1, y una vez terminada, activar a continuación S_2*. En los lenguajes derivados de ALGOL la composición de dos mecanismos se escribe $begin\ S_1; S_2\ end$; nosotros no utilizaremos los *paréntesis* $begin/end$ y escribiremos simplemente '$S_1; S_2$'. El carácter ';' se interpreta como *secuencialidad*.

Tratemos de *calcular* $S_1; S_2.X$; es decir, el conjunto de estados más grande que garantiza la terminación de S_1 seguida de S_2, verificando finalmente X. Es obvio que S_2 debió terminar; luego debió comenzar en algún estado del mayor conjunto de estados que lo garantiza, que es $S_2.X$. Pero entonces, S_1 debió terminar en algún estado del conjunto $S_2.X$. ¿Cuál es el mayor? La respuesta es $S_1.(S_2.X)$. Así, definimos, $\forall X$, y *ptle*

$$\boxed{(S_1; S_2).X \;\doteq\; S_1.(S_2.X)}$$

La composición secuencial corresponde a la composición funcional de las funciones transformadores de predicados. Por consiguiente, conocidas S_1 y S_2, hemos caracterizado perfectamente la composición secuencial (su semántica). Es importante resaltar que el transformador de predicados actúa 'hacia atrás' como muestra la Figura 4.1.

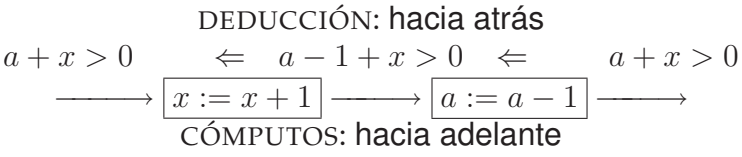


Figura 4.1: El mecanismo de deducción actúa en forma inversa.

EJEMPLO 4.4 $a := a + b; b := a * b.R(a, b)$

$=$ ⟨·⟩ semántica composición
$a := a + b.(b := a * b.R(a, b))$
$=$ ⟨·⟩ semántica de la asignación
$a := a + b.R(a, ab)$
$=$ ⟨·⟩ semántica de la asignación
$R(a + b, (a + b)b)$

Obsérvese que $a := a+b; b := a*b$ no tiene el mismo efecto que $a, b := a+b, a*b$:

$a, b := a + b, a * b.R(a, b)$
$=$
$R(a + b, ab)$ EJEMPLO

Es importante plantearnos si el operador composición ';' conserva la salubridad. Es decir: si S y T son sanos, ¿lo es la composición $S; T$? Esto puede probarse fácilmente, ya que *ptle*:

$S; T.(A \wedge B \wedge \dots)$
$=$ ⟨·⟩ semántica ;
$S.T.(A \wedge B \wedge \dots)$
$=$ ⟨·⟩ T conjuntivo
$S.(T.A \wedge T.B \wedge \dots)$
$=$ ⟨·⟩ S conjuntivo
$S.T.A \wedge S.T.B \wedge \dots$
$=$ ⟨·⟩ semántica ;
$S; T.A \wedge S; T.B \wedge \dots$

$S; T.F$
$=$ ⟨·⟩ semántica ;
$S.(T.F)$
$=$ ⟨·⟩ T estricto
$S.F$
$=$ ⟨·⟩ S estricto
F

de donde ';' conserva la conjuntividad (izquierda), y la *ley del milagro excluido* (derecha). También es fácil ver que la composición conserva el determinismo. Por otro lado, es un hecho conocido que la composición puede ir mal si va mal alguna de sus componentes; es decir, tenemos, *ptle*

$S; aborta.X$
$=$ ⟨·⟩ definición
$S.aborta.X$
$=$ ⟨·⟩ definición $aborta$
$S.F$
$=$ ⟨·⟩ S es sana
F

$aborta; S.X$
$=$ ⟨·⟩ definición
$aborta.(S.X)$
$=$ ⟨·⟩ definición $aborta$
F

de donde las tres sentencias siguientes tienen el mismo comportamiento

$$aborta; S \qquad S; aborta \qquad aborta$$

La composición de n (≥ 3) sentencias se define recurrentemente en la forma

$$S_1; S_2; \ldots; S_n \doteq S_1; (S_2; \ldots; S_n)$$

y la composición secuencial es *asociativa* ya que la composición de transformadores es asociativa.

EJEMPLO 4.5 (Intercambio de valores de dos variables) El programa

$$t := x; x := y; y := t$$

intercambia los contenidos de las variables x e y. En efecto:

$$\begin{array}{ll} & t := x; x := y; y := t.(x = a \wedge y = b) \\ = & \quad \because \text{semántica de la composición} \\ & t := x \,.\, x := y \,.\, y := t \,.(x = a \wedge y = b) \\ = & \quad \because \text{semántica de la asignación} \\ & t := x \,.\, x := y.(x = a \wedge t = b) \\ = & \quad \because \text{semántica de la asignación} \\ & t := x.(y = a \wedge t = b) \\ = & \quad \because \text{semántica de la asignación} \\ & (y = a \wedge x = b) \end{array}$$

(Intercambio de valores de dos variables (continuación)) El programa del Ejemplo 4.5 utiliza la variable auxiliar t; con la asignación paralela $x, y := y, x$ podemos intercambiar el contenido de las variables directamente (véase página 55) pero no es común que un lenguaje *real* disponga de la asignación paralela. Sorprendentemente, si el tipo asociado a las variables dispone de ciertos operadores es posible realizar el intercambio sin utilizar ninguna variable adicional; por ejemplo, para el tipo de los enteros, $x, y :\in Ent$, el siguiente programa realiza el intercambio sin variables adicionales:

$$x := x + y; y := x - y; x := x - y$$

En efecto:

$$\begin{array}{ll} & x := x + y; y := x - y; x := x - y \,.\, (x = b \wedge y = a) \\ = & \quad \because \text{semántica composición y asignación} \\ & x := x + y \,.\, y := x - y \,.\, (x - y = b \wedge y = a) \\ = & \quad \because \text{semántica composición y asignación} \\ & x := x + y \,.\, (x - (x - y) = b \wedge x - y = a) \\ = & \quad \because \text{simplificamos} \\ & x := x + y \,.\, (y = b \wedge x - y = a) \\ = & \quad \because \text{semántica} \\ & y = b \wedge (x + y) - y = a \\ = & \quad \because \text{simplificamos} \\ & (y = a \wedge x = b) \end{array}$$

EJEMPLO

¿Es válido el razonamiento anterior en todos los casos? Veamos que no. Al implementar el lenguaje, la sentencia $x := x + y$ puede dar lugar a *overflow* aritmético; por otro lado, si el tipo asociado a las variables es Flotante, la precisión de las operaciones puede dar lugar a incorrecciones; por consiguiente, las sentencias

$$\begin{array}{lcl} S & \doteq & x := x + y; y := x - y; x := x - y \\ S' & \doteq & x, y := y, x \end{array}$$

no son semánticamente equivalentes. Ahora bien, si las variables son de tipo Ent (donde Ent es el campo $\alpha..\beta$) y los valores iniciales a y b verifican el predicado:

$$R(a, b) \doteq \alpha \leq a, b, a + b \leq \beta$$

entonces se verifica:

$$\{R(a, b) \wedge x = a \wedge y = b\} S \{x = b \wedge y = a\}$$

En efecto; si utilizamos semántica estricta tendremos, en el contexto $x, y :\in \mathbb{Z}$,

$$\begin{array}{l} x := x + y; y := x - y; x := x - y \,.\, (x = b \wedge y = a) \\ = \\ x := x + y \,.\, y := x - y \,.\, (x - y = b \wedge y = a \wedge x - y \in \mathbb{Z}) \\ = \\ \quad \wedge \begin{array}{l} x := x + y \,.\, (x - (x - y) = b \wedge x - y = a \\ x - (x - y) \in \mathbb{Z} \wedge x - y \in \mathbb{Z}) \end{array} \\ = \\ \quad \wedge \begin{array}{l} (x + y - (x + y - y) = b \wedge x + y - y = a \\ x + y - (x + y - y) \in \mathbb{Z} \wedge x + y - y \in \mathbb{Z} \wedge x + y \in \mathbb{Z} \end{array} \end{array}$$

de donde

$$\begin{array}{l} a, b, a + b \in \mathbb{Z} \\ \Rightarrow \\ x := x + y; y := x - y; x := x - y.(x = b \wedge y = a) \equiv (x = a \wedge y = b) \end{array}$$

4.6 *[256] ¿Son equivalentes las sentencias* $t := x; x := y; y := t \qquad x, y := y, x$ *?.*

Lemas de sustitución

El siguiente resultado será de utilidad al trabajar con asignaciones.

LEMA 4.7 *El operador* sustitución *verifica las siguientes propiedades o* lemas de sustitución*:*

$$\begin{array}{lllll} (i) & x := E; x := F & = & x := (x := E.F) & \\ (ii) & y := E; x := F & = & x := (y := E.F); y := E, & \textit{si } x \not\equiv y, x \notin E \\ (iii) & y := E; x := F & = & x := F; y := E, & \textit{si } x \notin E, y \notin F \end{array}$$

donde $x \notin E$ significa que la variable x no aparece libre en E.

EJEMPLO 4.8 Tomando $E == 7$ y $F == x*n$ en el primer lema de sustitución, podemos probar la equivalencia:

$$x := 7; x := x * n \qquad = \qquad x := 7 * n$$

En forma parecida podemos probar $x := x + 1; x := x - 1 = nada$. El segundo lema de sustitución permite probar que la sentencia $y := n; x := x * y$ coincide con la sentencia $x := x * n; y := n$. EJEMPLO

Demostración.– (i) es consecuencia inmediata de la definición de composición y (iii) es consecuencia de (ii). Para probar (ii), demostraremos, por inducción sobre la estructura de Z,

$$\forall Z :: [\ y := E.(x := F.Z) \quad \equiv \quad x := (y := E.F).(y := E.Z)\ ]$$

Si Z es una variable, pueden darse tres casos:

— $Z \equiv x$; entonces, *ptle*

$$\begin{array}{ll} & y := E.(x := F.x) \quad \equiv \quad x := (y := E.F).(y := E.x) \\ = & \quad \because \text{definición sustitución, } y \not\equiv x \\ & y := E.F \quad \equiv \quad x := (y := E.F).x \\ = & \quad \because \text{definición sustitución} \\ & y := E.F \quad \equiv \quad y := E.F \end{array}$$

— $Z \equiv y$; entonces, *ptle*

$$\begin{array}{ll} & y := E.(x := F.y) \quad \equiv \quad x := (y := E.F).(y := E.y) \\ = & \quad \because \text{definición sustitución, } y \not\equiv x \\ & y := E.y \quad \equiv \quad x := (y := E.F).E \\ = & \quad \because \text{definición sustitución, } x \notin E \\ & E \quad \equiv \quad E \end{array}$$

— $Z \equiv z$, con $z \not\equiv y$, $z \not\equiv x$; entonces, *ptle*

$$\begin{array}{ll} & y := E.(x := F.z) \quad \equiv \quad x := (y := E.F).(y := E.z) \\ = & \quad \because \text{definición sustitución, } z \not\equiv y,\ z \not\equiv x \\ & y := E.z \quad \equiv \quad x := (y := E.F).z \\ = & \quad \because \text{definición sustitución, } z \not\equiv y,\ z \not\equiv x \\ & z \quad \equiv \quad z \end{array}$$

Si Z no es una variable, consideremos que puede descomponerse en la forma $Z_1 \otimes Z_2$, donde $\otimes$ es cualquier operador sustitutivo; entonces

$$\begin{array}{ll} \equiv & y := E.(x := F.(Z_1 \otimes Z_2)) \\ & x := (y := E.F).(y := E.(Z_1 \otimes Z_2)) \\ = & \quad \because \text{definición sustitución} \\ \equiv & y := E.(x := F.Z_1 \otimes x := F.Z_2) \\ & x := (y := E.F).(y := E.Z_1 \otimes y := E.Z_2) \\ = & \quad \because \text{definición sustitución} \\ \equiv & y := E.(x := F.Z_1) \otimes y := E.(x := F.Z_2) \\ & x := (y := E.F).(y := E.Z_1) \otimes x := (y := E.F).(y := E.Z_2) \\ \Leftarrow & \quad \because \otimes \text{ es sustitutivo} \\ \wedge & y := E.(x := F.Z_1) \quad \equiv \quad x := (y := E.F).(y := E.Z_1) \\ & y := E.(x := F.Z_2) \quad \equiv \quad x := (y := E.F).(y := E.Z_2) \\ = & \quad \because \text{HI} \\ & Cierto \end{array}$$

LEMA

4.9 *Probad que se verifica* $\{x < n, p = 2^x\} x := x + 1; p := p * 2 \{x \leq n, p = 2^x\}$.

4.10 *[256] Encontrad una expresión* E *verificando* $\{C\} a := a + 1; b := E\{a = b\}$.

4.11 *¿Es cierto* $[a := E; b := E.(a = b)]$*, siendo* E *una expresión arbitraria?*

4.12 *[256] Se define la sentencia* $inter(x, y)$ *como aquella que verifica, para toda variable* z *distinta de* x *e* y*:*

$$\{x = a \wedge y = b \wedge z = c\} inter(x, y) \{x = b \wedge y = a \wedge z = c\}.$$

Probad la equivalencia $inter(x, y) \ = \ (x, y := y, x)$.

4.13 *[257] Probad* $(S; T)^* = S^*; T^*$*, donde* S^* *es el conjugado de* S *(Definición 3.20).*

4.14 *Sin utilizar asignaciones paralelas, escribid un programa* S *formado como composición secuencial de sentencias de asignación para permutar los contenidos de 5 variables:*

$$\{(x, y, z, t, u) = (a, b, c, d, e)\} \ S \ \{(x, y, z, t, u) = (b, c, d, e, a)\}.$$

4.15 *[257]* **(Febrero, 96)** *Probad que la sentencia* $\langle b \to S \rangle$ *es sana, donde*

$$\forall X :: [\langle b \to S \rangle.X \ \doteq \ (b \ \Rightarrow \ S.X)].$$

4.16 *[257]* **(Febrero, 96)** *Encontrad una expresión* E *verificando:*

$$\{C\} b := N; a := E\{N > \text{máx}(a, b)\}.$$

4.17 *[257]* **(Febrero, 96)** *Probad que la sentencia* $\langle\langle b \to S \rangle\rangle$ *definida como*

$$\langle\langle b \to S \rangle\rangle.X \ \doteq \ X \vee b \wedge S.X,$$

se puede escribir en el lenguaje de Dijkstra y además es determinista si lo es S.

Sustitutividad de sentencias En la Sección 6.1 (página 98) trateremos con más rigor el concepto de sustitutividad. En líneas generales la propiedad de sustitutividad proporciona la siguiente propiedad

> al reemplazar en una sentencia S una subsentencia T por otra equivalente T' se obtiene otra sentencia S' equivalente a la primera

La propiedad anterior junto con la propiedad de *asociatividad* permite obtener resultados interesantes. A modo de ejemplo, para probar

$$x := x + 1; x := x - 1; S \qquad = \qquad S$$

tenemos el siguiente razonamiento:

$x := x + 1; x := x - 1; S$
$=$ $\quad \because$ asociatividad
$(x := x + 1; x := x - 1); S$
$=$ $\quad \because x := x + 1; x := x - 1 = nada$ (véase Ejemplo 4.8), y sustitutividad
$nada; S$
$=$ $\quad \because nada$ es la identidad
S

Intuitivamente, la siguiente composición de *cajas*

$$\longrightarrow \boxed{x := x + 1} \longrightarrow \boxed{\boxed{x := x - 1} \longrightarrow \boxed{S}} \longrightarrow$$

se puede redibujar en la forma

$$\longrightarrow \boxed{\boxed{x := x + 1} \longrightarrow \boxed{x := x - 1}} \longrightarrow \boxed{S} \longrightarrow$$

4.3. La sentencia selectiva

Hemos descrito un constructor para componer sentencias a partir de otras. Para obtener un lenguaje más rico debemos introducir la sentencia selectiva y el bucle. Dedicamos el resto de este capítulo a las selecciones. La construcción selectiva se define con la sintaxis:

$$\begin{array}{lcl} \textit{selectiva} & ::= & [\![\quad \textit{sentencia guardada} \\ & & \quad \{ \,\square\ \textit{sentencia guardada}\} \\ & &]\!] \\ \textit{sentencia guardada} & ::= & \textit{expr. booleana} \rightarrow \textit{sentencia} \end{array}$$

donde los símbolos ' $\square$ ' y '$\rightarrow$' son separadores. Un ejemplo de selección es:

$$[\![\, b > 0 \rightarrow b := b + 1 \,\square\, a > 0 \rightarrow a := a - 1 \,]\!] \tag{1}$$

Las expresiones booleanas $a > 0$ y $b > 0$ se llaman *guardas* y protegen la ejecución de la sentencia que aparece a la derecha del símbolo $\rightarrow$: la sentencia $b := b + 1$ está protegida por la guarda $b > 0$ y sólo se podrá ejecutar bajo la condición $b > 0$.

En la codificación de programas se acostumbra a escribir las estructuras selectivas en forma tabulada:

$$\begin{array}{lll} [\![& & b_1 \rightarrow S_1 \\ & \square & b_2 \rightarrow S_2 \\ & \ldots & \\ & \square & b_n \rightarrow S_n \,]\!] \end{array}$$

El significado 'informal' de esta sentencia es el siguiente:

(a) Se evalúan todas las guardas $b_1, b_2, \ldots, b_n$ (deben estar bien definidas).

(b) Entre las verdaderas se selecciona en forma indeterminista una de ellas y se ejecuta la correspondiente secuencia guardada.

(c) Si todas las guardas son falsas se *aborta* el programa.

Así, si ejecutamos la sentencia (1) a partir de un estado inicial verificando la precondición $P \doteq b > 0 \wedge a > 0$, al ser las dos guardas ciertas, se selecciona en forma indeterminista una de ellas. En definitiva, se eligirá en forma indeterminista, bien $b := b + 1$, o bien $a := a - 1$. Por consiguiente, el estado final no está completamente caracterizado por la precondición. Por ejemplo, partiendo del estado $(b = 6, a = 2)$ se puede llegar al estado $(b = 7, a = 2)$, o bien al estado $(b = 6, a = 1)$; es decir, se trata de un mecanismo indeterminista.

El mecanismo de selección anterior introduce una abstracción adicional sobre el mecanismo if-then-else de ALGOL o PASCAL: el *indeterminismo*. Este mecanismo está definido en el lenguaje ADA,

$$\begin{array}{ll} \textit{select} & \textit{when } b_1 \Rightarrow \ldots \\ & \textit{when } b_2 \Rightarrow \ldots \\ & \ldots \\ \textit{end select} & \end{array}$$

pero [Hoare, 1978] lo definió anteriormente para el lenguaje CSP (*Communicating Sequential Processes*); su uso es importante en programación concurrente. La notación original de Dijkstra para la sentencia selectiva es:

$$\textbf{if } b_1 \to S_1 \ldots \Box\, b_n \to S_n \textbf{ fi}$$

y en el lenguaje CSP se escribe igual que en el presente texto. También utilizaremos las notaciones

$$[\!]\ \Box\ : 1 \leq i \leq n : b_i \to S_i\ ]\!] \qquad\qquad [\!]\ \Box_{1\leq i\leq n} b_i \to S_i\ ]\!]$$

Ahora nuestro objetivo es definir, para cada predicado X, el predicado

$$[\!]\ \Box\ : 1 \leq i \leq n : b_i \to S_i\ ]\!].X$$

a partir de cada S_i. Según la interpretación informal antes descrita, en el estado inicial todas las guardas deben estar bien definidas (en otro caso añadiríamos, al igual que hicimos con la asignación, un predicado tal como

$$\forall i : 1 \leq i \leq n : b_i \in \mathcal{B}$$

y complicaríamos el transformador). Además, para no abortar la sentencia, debe verificarse el predicado OB (disyunción de las guardas) definido como:

$$OB \ \doteq\ b_1 \vee b_2 \vee b_3 \ldots \vee b_n (\equiv \exists i : 1 \leq i \leq n : b_i)$$

Luego de momento tenemos la descomposición, *ptle*,

$$SI.X \ \doteq\ OB \wedge \ \ldots$$

Si $OB = F$ para algún estado inicial, para ese estado se verificará $SI.X = F$, por lo que SI equivale a $aborta$ (para esos estados). Por otro lado, si en un estado ι se verifica la guarda b_i, la sentencia S_i puede ser ejecutada; en ese caso, para asegurar que se cumple X después de ejecutar S_i, ι debe cumplir $S_i.X$. Es decir, se cumplirá la implicación $b_i \ \Rightarrow\ S_i.X$, y esto debe darse para todos los estados y todos los índices. Por consiguiente, definimos, *ptle*:

$$\boxed{[\!]\ \Box_{1\leq i\leq n} b_i \to S_i\ ]\!].X \ \doteq\ OB \wedge (\forall i : 1 \leq i \leq n : b_i \ \Rightarrow\ S_i.X)}$$

Obsérvese que se captura el indeterminismo en forma simple y elegante: con una conjunción de implicaciones.

NOTA 4.18 *Si se pretende obtener solamente corrección parcial podemos introducir la* precondición libre *(weakest liberal precondition) en la forma:*

$$[\ wlp.SI.X \ \doteq\ (\forall i : 1 \leq i \leq n : b_i \ \Rightarrow\ wlp.S_i.X)\ ]$$

o con una notación más compacta: $[\ \widehat{SI}.X \ \doteq\ (\forall i : 1 \leq i \leq n : b_i \ \Rightarrow\ \widehat{S}_i.X)\ ]$*, en la cual observamos que para que sea válido el triplete* $\{Y\}\widehat{SI}\{X\}$ *no es necesario que* Y *asegure el predicado* OB*. Si la sentencia* SI *termina, es que alguna de las sentencias* S_i *lo hizo, y por ello, alguna de las guardas era cierta.*

4.19 *[257]* **(Febrero, 95)** *Probad que* $\mathcal{S}$ *es indeterminista y verifica* $\{Cierto\}\mathcal{S}\{x = 1\}$*, donde*

$$\mathcal{S} \ \doteq\ [\!]\ C \to y := 1 \,\Box\, C \to y := 0\ ]\!]\,; x := 1$$

EJEMPLO 4.20 (Máximo de dos valores) Estudiemos un mecanismo para encontrar el máximo m de dos valores x e y:

$$\{?\}S\{m = \text{máx}(x, y)\}$$

donde

$$m = \text{máx}(x, y) \doteq (m = x \vee m = y) \wedge (m \geq x \wedge m \geq y)$$

Si buscamos un mecanismo selectivo para el cómputo de m, parece evidente que dicho mecanismo deberá ejecutar $m := x$ o bien $m := y$. Pero:

$$\begin{array}{ll} & m := x.m = \text{máx}(x, y) \\ = & \quad \because \text{semántica asignación} \\ & x = max(x, y) \\ = & \quad \because \text{definición de máx y CP} \\ & x \geq y \end{array}$$

Luego la sentencia $m := x$ debe estar protegida por la guarda $x \geq y$, y simétricamente para la sentencia $m := y$. Estudiemos pues la precondición más débil $\boxed{?}$ para la validez del siguiente esquema

$$\begin{array}{l} \{?\} \\ \quad [\!] \quad x \geq y \quad \rightarrow \quad m := x \\ \quad \square \quad y \geq x \quad \rightarrow \quad m := y \,]\!] \\ \{m = \text{máx}(x, y)\} \end{array}$$

Únicamente es necesario un pequeño cálculo, *ptle*:

$$\begin{array}{ll} & [\![x \geq y \rightarrow m := x \, \square \, x \leq y \rightarrow m := y]\!] \, . \, m = \text{máx}(x, y) \\ = & \quad \because \text{semántica de la selectiva} \\ \wedge & (x \geq y \vee x \leq y) \wedge (x \geq y \;\Rightarrow\; m := x \, . \, m = \text{máx}(x, y)) \\ & (x \leq y \;\Rightarrow\; m := y \, . \, m = \text{máx}(x, y)) \\ = & \quad \because \text{cálculo de predicados y semántica} \\ & Cierto \wedge (x \geq y \;\Rightarrow\; x = \text{máx}(x, y)) \wedge (x \leq y \;\Rightarrow\; y = \text{máx}(x, y)) \\ = & \quad \because \text{cálculo de predicados} \\ & Cierto \wedge (x \geq y \;\Rightarrow\; x \geq y) \wedge (x \leq y \;\Rightarrow\; x \leq y) \\ = & \\ & Cierto \end{array}$$

Luego, el predicado desconocido $\boxed{?}$ se completa en la forma $\boxed{\text{C}}$. EJEMPLO

Una vez definida la semántica de la selectiva, podemos probar que tal sentencia no es determinista en muchos casos, según la Definición 3.17.

EJEMPLO 4.21 Sea $SI \doteq [\![C \rightarrow n := 0 \, \square \, C \rightarrow n := 1]\!]$. Veamos que se cumple

$$[\, SI.X \equiv n := 0.X \wedge n := 1.X \,] \tag{1}$$

En efecto; *ptle*,

$$\begin{array}{ll} & SI.X \\ = & \quad \because \text{semántica} \\ & C \wedge (C \;\Rightarrow\; n := 0.X) \wedge (C \;\Rightarrow\; n := 1.X) \\ = & \quad \because \text{CP} \end{array}$$

$n := 0.X \wedge n := 1.X$

Entonces, también *ptle*

$$
\begin{array}{ll}
& SI.n = 0 \vee SI.n = 1 \qquad \equiv \qquad SI.(n = 0 \vee n = 1) \\
= & \quad \because \text{por } (1) \\
& 0 = 0 \wedge 1 = 0 \vee 0 = 1 \wedge 1 = 1 \equiv (0 = 0 \vee 0 = 1) \wedge (0 = 0 \vee 1 = 1) \\
= & \quad \because \text{CP} \\
& Falso \equiv Cierto \\
= & \\
& Falso
\end{array}
$$

luego SI no es disyuntivo, y según la Definición 3.17, es indeterminista. Igualmente, el transformador $\mathcal{N}$ del Ejemplo 3.7 coincide con el transformador de la selectiva $[\![C \to x := cara \,\square\, C \to x := cruz]\!]$. EJEMPLO

4.22 *[257] Probad el triplete* $\{a < b\}\ SI\ \{m > a\}$, *donde* SI *es la sentencia*

$$
\begin{array}{lll}
[\![& a > b \to m := a & \\
\square & a < b \to m := b & \\
\square & a = b \to nada &]\!].
\end{array}
$$

4.23 *[258] Siendo* x *una variable entera, demostrad* $[OB \equiv SI.x \neq 0]$, *donde* SI *es la selectiva*

$$
\begin{array}{lllll}
[\![& x < 0 & \to & x := -x & \\
\square & x > 1 & \to & x := x - 1 & \\
\square & x > 2 & \to & x := x \div 2 &]\!]
\end{array}
$$

Mediante la aplicación del siguiente teorema podemos escribir mecanismos selectivos arbitrarios a partir de selectivas con dos guardas.

TEOREMA 4.24 *Las siguientes sentencias selectivas son equivalentes*

$$
\begin{array}{lll}
SI & \doteq & [\![b \to S \,\square\, b_1 \to S_1 \,\square\, \ldots \,\square\, b_n \to S_n]\!] \\
SI' & \doteq & [\![b \to S \,\square\, OB \to [\![b_1 \to S_1 \,\square\, \ldots \,\square\, b_n \to S_n]\!]]\!]
\end{array}
$$

donde $OB \doteq (\exists i : 1 \leq i \leq n : b_i)$.

Demostración.–

$$
\begin{array}{ll}
& SI'.X \\
= & \quad \because \text{semántica} \\
& (OB \vee b) \wedge (b \ \Rightarrow\ S.X) \wedge (OB \ \Rightarrow\ [\![\,\square\, : 1 \leq i \leq n : b_i \to S_i]\!].X) \\
= & \quad \because \text{semántica} \\
& (OB \vee b) \wedge (b \ \Rightarrow\ S.X) \wedge \\
& (OB \ \Rightarrow\ OB \wedge (\forall i : 1 \leq i \leq n : b_i \ \Rightarrow\ S_i.X)) \\
= & \quad \because \text{cálculo} \\
& (OB \vee b) \wedge (b \ \Rightarrow\ S.X) \wedge (\forall i : 1 \leq i \leq n : OB \ \Rightarrow\ (b_i \ \Rightarrow\ S_i.X)) \\
= & \quad \because \text{cálculo} \\
& (OB \vee b) \wedge (b \ \Rightarrow\ S.X) \wedge (\forall i : 1 \leq i \leq n : OB \wedge b_i \ \Rightarrow\ S_i.X) \\
= & \quad \because \text{regla de oro: } (b_i \ \Rightarrow\ OB) \equiv (OB \wedge b_i \equiv b_i) \\
& (OB \vee b) \wedge (b \ \Rightarrow\ S.X) \wedge (\forall i : 1 \leq i \leq n : b_i \ \Rightarrow\ S_i.X) \\
= & \quad \because \text{semántica} \\
& SI.X
\end{array}
$$

TEOREMA

EJEMPLO 4.25 El Teorema 4.24 tiene una aplicación esencial: escribir selectivas usando únicamente construcciones con dos guardas. Así, la sentencia

$$[\![\, a \to S \,\square\, b \to T \,\square\, c \to U \,\square\, d \to V \,]\!]$$

es equivalente a la selectiva *en cascada* siguiente

$$\begin{array}{llll}
[\![& a & \to & S \\
\square & b \vee c \vee d & \to & [\![\quad b \quad \to \quad T \\
 & & \square & c \vee d \quad \to \quad [\![\quad c \quad \to \quad U \\
 & & & \qquad\qquad\quad \square \quad d \quad \to \quad V \\
 & & & \qquad\qquad\quad]\!] \\
 & &]\!] & \\
]\!] & &
\end{array}$$

Operacionalmente, la segunda construcción podría realizar más comprobaciones que la primera (por ejemplo si la única guarda cierta es la última). EJEMPLO

Es importante comprobar que la construcción selectiva conserva la salubridad; para ello basta probar el siguiente teorema.

TEOREMA 4.26 (Salubridad de la construcción selectiva) *Para que la selectiva* $[\![\, \square : 1 \le i \le n : b_i \to S_i \,]\!]$ *sea sana, es suficiente con que lo sea cada sentencia* S_i*.*

Demostración.– Sea $OB \doteq (\exists i : 1 \le i \le n : b_i)$; entonces, *ptle*

$$\begin{array}{ll}
 & [\![\, \square : 1 \le i \le n : b_i \to S_i \,]\!].F \\
= & \quad \because \text{semántica} \\
 & OB \wedge (\forall i : 1 \le i \le n : b_i \;\Rightarrow\; S_i.F) \\
= & \quad \because S_i \in \mathcal{T}^{\perp} \text{ (ley del milagro excluido)} \\
 & OB \wedge (\forall i : 1 \le i \le n : b_i \;\Rightarrow\; F) \\
= & \quad \because \forall i : 1 \le i \le n : \neg b_i = \neg OB \\
 & OB \wedge \neg OB \\
= & \\
 & F
\end{array}$$

Sin pérdida de generalidad, probemos la conjuntividad para dos predicados; *ptle*

$$\begin{array}{ll}
 & [\![\, \square : 1 \le i \le n : b_i \to S_i \,]\!].(A \wedge B) \\
= & \quad \because \text{definición} \\
 & OB \wedge \forall i : 1 \le i \le n : b_i \;\Rightarrow\; S_i.(A \wedge B) \\
= & \quad \because \text{cada } S_i \text{ es conjuntiva} \\
 & OB \wedge \forall i : 1 \le i \le n : b_i \;\Rightarrow\; S_i.A \wedge S_i.B \\
= & \quad \because [(X \;\Rightarrow\; Y \wedge Z) \equiv (X \;\Rightarrow\; Y) \wedge (X \;\Rightarrow\; Z)] \\
 & OB \wedge \forall i : 1 \le i \le n : (b_i \;\Rightarrow\; S_i.A) \wedge (b_i \;\Rightarrow\; S_i.B) \\
= & \quad \because \forall \text{ es conjuntivo} \\
 & OB \wedge (\forall i : 1 \le i \le n : b_i \;\Rightarrow\; S_i.A) \wedge (\forall i : 1 \le i \le n : b_i \;\Rightarrow\; S_i.B) \\
= & \quad \because \text{idempotente} \\
\wedge & \begin{array}{l} OB \wedge (\forall i : 1 \le i \le n : b_i \;\Rightarrow\; S_i.A) \\ OB \wedge (\forall i : 1 \le i \le n : b_i \;\Rightarrow\; S_i.B) \end{array} \\
= & \quad \because \text{semántica} \\
 & [\![\, \square : 1 \le i \le n : b_i \to S_i \,]\!].A \wedge [\![\, \square : 1 \le i \le n : b_i \to S_i \,]\!].B
\end{array}$$

TEOREMA

Determinismo de la selectiva

Los programas forman un conjunto Bien Construido Sea $\mathcal{L}$ un lenguaje y $\mathcal{F}$ el conjunto de sus frases con la relación de orden parcial:

$$F < F' \doteq F \text{ es una subfrase propia de } F'$$

En particular, un programa S es menor que otro S' si S aparece como una subsentencia en S'. Ya que el número de subsentencias de una dada es finito, por aplicación inmediata del Teorema 1.23:23, el conjunto de programas $\mathcal{P}rog$ es un conjunto bien construido para dicha relación de orden. En consecuencia sobre tal lenguaje es válido el principio de inducción, que se llama en este caso inducción estructural o sobre la estructura del lenguaje.

De la propiedad de inducción para el conjunto $\mathcal{P}rog$ de programas tenemos el siguiente esquema de demostración de que cierta propiedad p ($: \mathcal{P}rog \to \mathcal{B}$) es válida para cada programa:

$\forall S : S \in \mathcal{P}rog : p.S$
$=$ $\quad \because$ Definición 1.21, en página 22
$\forall S : S \in \mathcal{P}rog : (\forall T : T \in \mathcal{P}rog \wedge T < S : p.T) \;\Rightarrow\; p.S$

Para probar lo anterior, podemos probar, para cada forma sintáctica S, el siguiente esquema de *inducción estructural*:

$$(\forall T : T \in \mathcal{P}rog \wedge T < S : p.T) \;\Rightarrow\; p.S \qquad (ie)$$

Una forma de probar (ie) es por casos, distinguiendo la forma sintáctica de S. Si S es una sentencia básica, en el antecedente de la implicación (ie) el campo $T \in \mathcal{P}rog \wedge T < S$ es falso (S no contiene sentencias propias); en estos casos (ie) toma la forma:

$(\forall T : T \in \mathcal{P}rog \wedge T < S : p.T) \;\Rightarrow\; p.S$
$=$ $\quad \because$ S no contiene sentencias propias
$(\forall T : Falso : p.T) \;\Rightarrow\; p.S$
$=$ $\quad \because$ definición cuantificador
$Cierto \;\Rightarrow\; p.S$
$=$ $\quad \because$ CP
$p.S$

Luego habría que demostrar directamente la propiedad $p.S$ para las sentencias simples o básicas. Para el resto de casos en que S no es básica usaremos $T < S$, siendo T una subfrase *inmediata* de S. Así, para demostrar que las sentencias de nuestro lenguaje imperativo con guardas verifican cierta propiedad p sería suficiente seguir (demostrar) el siguiente esquema esencial:

(a) CASOS BASE: p es cierta para las sentencias elementales; es decir,

$$p.aborta \qquad p.nada \qquad p.(x := E)$$

(b) PASOS INDUCTIVOS: la implicación (ie) para cada estructura compuesta

$p.S \wedge p.T$	$\Rightarrow$	$p.(S;T)$	— composición
$(\forall i :: p.S_i)$	$\Rightarrow$	$p.[\![\; \square : 1 \le i \le n : b_i \to S_i \;]\!]$	— selección
$p.S$	$\Rightarrow$	$*[\![\, b \to S \,]\!]$	— bucle

A modo de ejemplo, consideremos la propiedad de salubridad

$$p.S \doteq S \in \mathcal{T}^{\perp\wedge}$$

Podemos observar que las condiciones (b) del esquema anterior consisten en la conservación de la salubridad para los constructores del lenguaje, mientras que (a) afirma que las sentencias elementales son sanas. Luego hemos demostrado que las sentencias del lenguaje de Dijkstra (sin bucles) son sanas.

El determinismo de las sentencias componentes de una selección no es suficiente para el determinismo de ésta; sin embargo, el siguiente teorema muestra que esto ocurre cuando las guardas son excluyentes dos a dos. Como un importante corolario obtenemos que un lenguaje con selecciones donde las guardas son excluyentes entre sí será determinista. Este es el caso del lenguaje PASCAL.

TEOREMA 4.27 (Conservación del determinismo en la selección) *Si todas las sentencias S_i son deterministas $(1 \leq i \leq n)$ y las guardas son excluyentes dos a dos, entonces también es determinista la sentencia $SI \doteq [\![\Box : 1 \leq i \leq n : b_i \to S_i]\!]$.*

Demostración.– Por el Teorema 4.24 basta suponer dos guardas; sin embargo la dificultad en el caso de varias sentencias es parecida por lo que la demostración la hacemos para el caso general. Puesto que tenemos

$$[\, \forall j : A_j : M \;\Rightarrow\; N \;\equiv\; \forall j : A_j \wedge M : N \,]$$

(demuéstrese) podemos suponer el predicado $1 \leq j \leq n$ incluido en b_j:

$$[\, SI.R \doteq OB \wedge (\forall j : b_j : S_j.R) \,]$$

Tenemos entonces, para dos predicados cualesquiera X e Y, y *ptle*:

$$\begin{array}{ll} & SI.X \vee SI.Y \\ = & \quad \because \text{semántica} \\ & OB \wedge (\forall i : b_i : S_i.X) \vee OB \wedge (\forall j : b_j : S_j.Y) \\ = & \quad \because \text{distributiva: } [(A \;\Rightarrow\; B) \vee (A' \;\Rightarrow\; B') \equiv (A \wedge A' \;\Rightarrow\; B \vee B')] \\ & OB \wedge (\forall i, j : b_i \wedge b_j : S_i.X \vee S_j.Y) \\ = & \quad \because [(A \;\Rightarrow\; B) \equiv (A \wedge M \;\Rightarrow\; B) \wedge (A \wedge \neg M \;\Rightarrow\; B)] \\ & OB \wedge (\forall i, j : b_i \wedge b_j \wedge i = j : S_i.X \vee S_j.Y) \wedge \\ & (\forall i, j : b_i \wedge b_j \wedge i \neq j : S_i.X \vee S_j.Y) \\ = & \quad \because \text{las guardas son excluyentes} \\ & OB \wedge (\forall i : b_i : S_i.X \vee S_i.Y) \\ = & \quad \because \text{semántica, } S_i \text{ determinista} \\ & SI.(X \vee Y) \end{array}$$

TEOREMA

4.28 *[258] A partir de dos sentencias S y T la sentencia $S \nabla T$, cuya semántica informal es: elegir al azar una de ellas y ejecutarla. Dad su transformador, probar que es sano y dar ejemplos donde la construcción sea determinista o indeterminista.*

4.29 *[258] Sea $SI \doteq [\![b \to S \,\Box\, f \to T]\!]$. Poner ejemplos donde*

(A) S y T sean indeterministas, pero SI sea determinista.

(B) S y T sean deterministas, pero SI sea indeterminista.

(C) S, T y SI sean indeterministas.

(D) SI sea sana, pero T y S no lo sean.

La selección binaria tiene la forma $[\![b \to S \,\square\, \neg b \to T]\!]$, que también simplificaremos como $[\![b \to S \,\square\, T]\!]$; su transformador de predicados es, *ptle*:

$$
\begin{array}{ll}
 & SI.X \\
= & \quad \because \text{semántica de } SI \\
 & (b \vee \neg b) \wedge (b \;\Rightarrow\; S.X) \wedge (\neg b \;\Rightarrow\; T.X) \\
= & \quad \because \text{por el Lema 1.14, o directamente vía Lema 1.12}(iv)\text{:18} \\
 & b \wedge S.X \vee \neg b \wedge T.X
\end{array}
$$

Luego, *ptle*

$$[\![b \to S \,\square\, \neg b \to T]\!].X \;\equiv\; b \wedge S.X \vee \neg b \wedge T.X \qquad (1)$$

Obsérvese que en la selección binaria las guardas son excluyentes, y por el Teorema 4.27, ésta será determinista cuando lo sean las sentencias guardadas. En otros lenguajes (PASCAL, ADA, etc.) se suele escribir tal sentencia en la forma $if\ b\ then\ S\ else\ T$. Por otro lado, si tomamos en (1) $T == nada$, entonces tenemos:

$$(if\ b\ then\ S).X \equiv b \wedge S.X \vee \neg b \wedge X$$

También, como aplicación de (1) obtenemos:

$$[\![b \to S \,\square\, \neg b \to aborta]\!].X \equiv b \wedge S.X$$

que tiene una interpretación simple: puesto que si $\neg b$ es cierto la sentencia no termina, para asegurar la terminación de la selección debe darse el predicado b; por otro lado, para asegurar la terminación con la poscondición X deberá darse como precondición $S.X$; de aquí que deben darse ambos predicados: $b \wedge S.X$.

EJEMPLO 4.30 Como aplicación de (1) probemos, siendo x una variable entera,

$$\{x \neq 0\}[\![x < 0 \to x := -x \,\square\, x := x - 1]\!]\,\{x \geq 0\}$$

En efecto, *ptle*:

$$
\begin{array}{ll}
 & [\![x < 0 \to x := -x \,\square\, x := x - 1]\!]\,\{x \geq 0\} \\
= & \quad \because \text{selección binaria} \\
 & x < 0 \wedge x := -x.\, x \geq 0 \vee x \geq 0 \wedge x := x - 1.\, x \geq 0 \\
= & \quad \because \text{semántica asignación} \\
 & x < 0 \wedge\ -x \geq 0 \ \ \vee \ \ x \geq 0 \wedge x - 1 \geq 0 \\
= & \quad \because \text{cálculo} \\
 & x < 0 \vee x \geq 1 \\
\Leftarrow & \quad \because x \text{ es entero} \\
 & x \neq 0
\end{array}
$$

EJEMPLO

EJEMPLO 4.31 Apliquemos (1) en el caso

$$
\begin{array}{ll}
 & [\![i = 0 \to j := 0 \,\square\, i \neq 0 \to j := 1]\!].j = 1 \\
= & \\
 & i = 0 \wedge j := 0.j = 1 \vee i \neq 0 \wedge j := 1.j = 1 \\
= & \\
 & (i = 0 \wedge 0 = 1) \vee (i \neq 0 \wedge 1 = 1) \\
= & \\
 & i \neq 0
\end{array}
$$

Lo anterior tiene una interpretación fácil: para obtener $j = 1$ debemos asegurar que la primera guarda no es seleccionada.

EJEMPLO

EJEMPLO 4.32 Sea SI la estructura

$$\begin{array}{ll}[\![& j < k \to m := k \\ \square & j \geq k \to m := j \,]\!]\end{array}$$

y sea $R \equiv i, j, k \leq m$; vamos a calcular $SI.R$

$$\begin{array}{ll}= & \because \text{semántica de selección binaria} \\ & j < k \wedge m := k.R \vee j \geq k \wedge m := j.R \\ = & \because \text{semántica de asignación} \\ & j < k \wedge i \leq k \wedge j \leq k \vee j \geq k \wedge i \leq j \wedge k \leq j \\ = & \because \text{cálculo} \\ & i, j \leq k\end{array}$$

que tiene una interpretación fácil como el lector puede comprobar. EJEMPLO

Ejercicios

4.33 *[258] Definid una semántica para la estructura selectiva:*

$$\begin{array}{lll}\textit{si} & & b_1 \to S_1 \\ & \square & b_2 \to S_2 \\ & \ldots & \\ & \square & b_n \to S_n \\ & & \textit{en\_otro\_caso } S' \\ \textit{fin\_si} & & \end{array}$$

4.34 *Probad con un ejemplo que* $[\![b_1 \to S_1 \,\square\, b_2 \to S_2 \,]\!] \neq [\![b_1 \to S_1 \,\square\, b_2 \wedge \neg b_1 \to S_2 \,]\!]$.

4.35 *Si admitimos selectivas* infinitas *como* $Azar_x \doteq [\![C \to x := 1 \,\square\, C \to x := 2 \,\square\, \ldots]\!]$ *¿qué tipo de indeterminismo introducen? AYUDA.- Analice la siguiente propiedad: con la construcción* $Azar_x$ *no podemos prever ningún valor* N *de forma que se cumpla* $\{C\}Azar_x\{x < N\}$.

4.36 *[258]* **(Febrero, 96)** *Sea* S *una sentencia sana y determinista. Probad que la sentencia con transformador de predicados* $T.X \doteq b \wedge S.X$ *se puede escribir en el lenguaje de Dijkstra, y es determinista.*

4.37 *[258]* **(Diciembre, 96)** *¿Existe* S *verificando* $\forall X :: \{X\}S\{\neg X\}$ *?*

4.38 *[258]* **(Diciembre, 97)** *Probad que la sentencia* $SI \doteq [\![C \to x := 1 \,\square\, C \to x, y := 1, 0]\!]$ *verifica el triplete* $\{C\}SI\{x = 1\}$*, pero es indeterminista.*

4.39 *[259]* **(Diciembre, 97)** *Describid con transformadores de predicados y con tripletes, las siguientes frases, dando ejemplos de cada caso:*

- *(A) En el entorno del predicado* b*, la sentencia* S *nunca termina.*
- *(B)* X *es una precondición suficiente para que* S *termine verificando* $\neg X$.
- *(C) Si* S *termina, entonces termina verificando* Y.
- *(D)* S *puede no terminar si se ejecuta bajo la condición* b.

4.40 *[259]* **(Marzo, 94)** *Demostrad o refutad las siguientes afirmaciones*

- *(A) Si* $\mathcal{S}$ *es sana, entonces* $\exists X : X \in \mathcal{P} : [\mathcal{S}.X] \Longleftrightarrow \mathcal{S}$ *siempre termina.*
- *(B)* $[\![x > 1 \to nada \,\square\, x > 1 \to aborta]\!] = aborta.$
- *(C) Ningún estado inicial asegura que la sentencia* $[\![x > 0 \to x := 2 \,\square\, x > 1 \to x := 3]\!]$ *termina verificando* $x = 2$.
- *(D)* $[\![b \to \mathcal{S} \,\square\, \neg b \to \mathcal{S}']\!]$ *es determinista.*

4.41 *Sean* S, T *dos sentencias sanas y determinista. Probad que la sentencia con transformador de predicados* $U.X \doteq b \wedge S.X \wedge T.X$ *se puede escribir en el lenguaje de Dijkstra, y es determinista.*

Bibliografía

[Alagic y Arbib, 1978] Alagic, S. y Arbib, M. (1978). *The Design of Well-Structured and Correct Programs*. Springer-Verlag, New-York.

[ANSI-83, 1983] ANSI-83 (1983). Reference Manual for the Ada Programming Language. U.S. Government (Ada Joint Program Office). Reimpreso en [Horowitz, 1983].

[Apt, 1988] Apt, K. R. (1988). Proving Correctness of Concurrent Programs: A Quick Introduction. En Börger, E. (ed.), *Trends in Theoretical Computer Science*, pp. 305–345. Computer Science Press.

[Arsac, 1985] Arsac, J. (1985). Teaching Programming. En Griffiths, M. y Tagg, E. (eds.), *The role of programming in teaching Informatics. Proc. IFIP, TC3, Working Conference on Teaching Programming, París, 7–9 mayo'84*, pp. 3–6. Elsevier Science Pbl., Amsterdam.

[Babbage, 1864] Babbage, C. (1864). De la Máquina Analítica. En *Perspectives on Computer Revolution*. Prentice-Hall, New Jersey. Traducción al castellano, Alianza, Madrid (1975) de la del inglés (1970).

[Barendregt, 1984] Barendregt, H. P. (1984). *The Lambda Calculus, Its Syntax and Semantics*, volumen 103 de *Studies in Logic and the Foundations of Mathematics*. North-Holland, Amsterdam. Edición revisada de la primera (1981).

[Berg y o., 1982] Berg, H. y o. (1982). *Formal Methods of Program Verification and Specification*. Prentice-Hall, New Jersey.

[Bird y Wadler, 1988] Bird, R. y Wadler, P. (1988). *Introduction to Functional Programming*. Pretice–Hall.

[Cauchy, 1821] Cauchy, A. L. (1821). Cours d'analyse. En *Oeuvres Complétes (II^e Série)*, volumen 3. École Royale Polytechnique. Reeditado en forma facsimilar por SAEM Thales (1999).

[Dijkstra, 1981] Dijkstra, E. (1981). Why correctness must be a mathematical concern. En Boyer, R. y Moore, J. S. (eds.), *The correctness problem in computer science*. Academic Press, London.

[Dijkstra, 1982] Dijkstra, E. (1982). The equivalence of bounded nondeterminacy and continuity. En Dijkstra, E. (ed.), *Selected Writings on Computing: A personal Perspective*, pp. 358–359. Springer-Verlag.

[Dijkstra, 1990] Dijkstra, E. (ed.) (1990). *Formal Development of Programs and Proofs*. Addison-Wesley. The Year of Programming.

[Dijkstra y Feijen, 1984] Dijkstra, E. y Feijen, W. (1984). *Een methode van programmeren*. The Hague: Academic Service. traducido al inglés en [Dijkstra y Feijen, 1988].

[Dijkstra, 1976] Dijkstra, E. W. (1976). *A Discipline of Programming*. Prentice-Hall.

[Dijkstra y Feijen, 1988] Dijkstra, E. W. y Feijen, W. (1988). *A Method of Programming*. Addison-Wesley, Massachusetts.

[Dijkstra y Scholten, 1990] Dijkstra, E. W. y Scholten, C. S. (1990). *Predicate Calculus and Program Semantics*. Springer-Verlag, New York.

[Field y Harrison, 1988] Field, A. y Harrison, P. (1988). *Functional Programming*. Addison-Wesley.

[Floyd, 1967] Floyd, R. W. (1967). Assigning Meanings to Programs. En Schwartz, J. T. (ed.), *Mathematical Aspects of Computer Science*, volumen 19 de *Symposia in Applied Mathematics*, pp. 19–32. American Mathematical Society, Providence, RI.

[Gehani y McGettrick, 1988] Gehani, N. y McGettrick, A. (1988). *Concurrent Programming*. Addison-Wesley.

[Gries, 1981] Gries, D. (1981). *The Science of Programming*. Springer-Verlag, New-York.

[Hebenstreit, 1985] Hebenstreit, J. (1985). Teaching programming to everybody, why? to whom? what? En Griffiths, M. y Tagg, E. (eds.), *The role of programming in teaching Informatics, Proceed. IFIP, TC3, Working Conference on Teaching Programming, París, 7–9 mayo, 1984*, pp. 17–21. Elsevier Science Pbl., Amsterdam.

[Hehner, 1984] Hehner, E. (1984). *The Logic of Programming*. Prentice-Hall, New Jersey.

[Hennessy, 1990] Hennessy, M. (1990). *The Semantics of Programming Languajes; An Elementary Introduction using Structural Operational Semantics*. Wiley.

[Hoare, 1969] Hoare, C. (1969). An Axiomatic Basis for Computer Programming. *Communications of the ACM*, 12(10):576–580. Reimpreso en *C.ACM*, 26(1):53-56, 1983, y también en [Hoare y Jones, 1989]:45-58.

[Hoare, 1971] Hoare, C. (1971). Computer Science. *New Lectures Series*, 62. reimpreso en [Hoare y Jones, 1989]:89–101.

[Hoare, 1978] Hoare, C. (1978). Communicating Sequential Processes. *Communications of the ACM*, 21(8). Reimpreso en [Gehani y McGettrick, 1988]:278-308, y también en [Horowitz, 1983]:311-322.

[Hoare, 1985] Hoare, C. (1985). *Communicating Sequential Processes*. Prentice–Hall, New Jersey.

[Hoare y Jones, 1989] Hoare, C. y Jones, C. (1989). *Essays in Computing Science*. Prentice-Hall.

[Horowitz, 1983] Horowitz, E. (1983). *Programming Languages. A grand Tour*. Computer Science Press.

[Horowitz y Sahni, 1978] Horowitz, E. y Sahni, S. (1978). *Fundamentals of Computer Algorithms*. Comp. Science Press.

[Huet, 1990] Huet, G. P. (1990). A Uniform approach to Type Theory. En Huet, G. (ed.), *Logical Foundations of Functional Programming*, pp. 337–397. Addison-Wesley.

[Knuth, 1968] Knuth, D. E. (1968). *The Art of Computer Programming. Vol. 1: Fundamental Algorithms*. Addison-Wesley, Massachusetts. Segunda edición (1973). Traducido al castellano en Ed. Reverté, Barcelona.

[Kowalski, 1979] Kowalski, R. (1979). *Logic for Problem Solving*. Elsevier Sc. Publ. Co. Traducción al castellano en Díaz de Santos, Madrid (1986), con el título *Lógica, Programación e Inteligencia Artificial*.

[Liskov y Zilles, 1974] Liskov, B. y Zilles, S. (1974). Programming with abstract data types. En *Proc. ACM SIGPLAN Conference on Very High Level Languages*, volumen 9, 4, pp. 50–59.

[Manna, 1974] Manna, Z. (1974). *Mathematical Theory of Computation*. McGraw-Hill.

[Meyer, 1988] Meyer, B. (1988). *Object-Oriented Software Construction*. Prentice–Hall.

[Morris, 1990] Morris, J. (1990). Programs from Specifications. En Dijkstra, E. (ed.), *Formal Development of Programs and Proofs*, pp. 81–115. Addison-Wesley. The Year of Programming.

[Nielson y Nielson, 1992] Nielson, H. y Nielson, F. (1992). *Semantics with Applications*. Wiley.

[Popek y Horning, 1977] Popek, G. y Horning, J. (1977). Notes on the Design of Euclid. *ACM SIGPLAN Notices*, 12(3):11–19.

[Ruiz Jiménez et al., 2000] Ruiz Jiménez, B. C., Gutiérrez López, F., Guerrero García, P., y Gallardo Ruiz, J. E. (2000). *Razonando con Haskell. Una Introducción a la Programación Funcional*. José E. Gallardo Ruiz (editor).

[Schmidt, 1988] Schmidt, D. (1988). *Denotational Semantics*. Allyn and Bacon.

[Shapiro, 1987] Shapiro, E. (1987). *Concurrent Prolog. Collected Papers*. MIT Press, Cambridge. Dos volúmenes.

[Sperschneider y Antoniou, 1991] Sperschneider, V. y Antoniou, G. (1991). *LOGIC. A Foundation for Computer Science*. Addison Wesley.

[Ueda, 1985] Ueda, K. (1985). Guarded Horn Clauses. Informe Técnico núm. 103, ICOT, Tokyo. También en [Shapiro, 1987]:(Vol.1,140-156).

[van Gasteren, 1990] van Gasteren, A. (1990). On the Formal Derivation of a Proof of the Invariance Theorem. En Dijkstra, E. (ed.), *Formal Development of Programs and Proofs*, pp. 49–54. Addison-Wesley. The Year of Programming.

[Wegner, 1984] Wegner, P. (1984). Capital-intensive software technology. *IEEE Software*, pp. 7–45.

[Wirth, 1973] Wirth, N. (1973). *Systematic Programming*. Prentice-Hall, New Jersey. traducción al castellano en Ed. El Ateneo, Buenos Aires (1982).

[Wirth, 1976] Wirth, N. (1976). *Algorithms + Data Structures = Programs*. Prentice-Hall, New York. traducción al castellano en Ed. del Castillo, Madrid, 1980.

[Wirth, 1983] Wirth, N. (1983). On the Design of Programming Languages. En *IFIP, 1974*, pp. 386–393. North-Holland Pub. Comp. reimpreso en [Horowitz, 1983]:23–30.

[Wirth y Hoare, 1973] Wirth, N. y Hoare, C. (1973). An Axiomatic Definition of the Programming Language PASCAL. *Acta Informatica*, 2(4):335–355. Reimpreso en [Hoare y Jones, 1989]:153–169.