10
2ej.

UNIVERSIDAD NACIONAL AUTÓNOMA DE MÉXICO

FACULTAD DE CIENCIAS

LA PREESPECIFICACIÓN MÁS DÉBIL

T E S I S

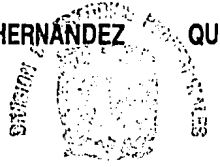
QUE PARA OBTENER EL TÍTULO DE

M A T E M Á T I C O

P R E S E N T A :

FRANCISCO HERNÁNDEZ QUIROZ

México, D.F. 1994



UNAM – Dirección General de Bibliotecas Tesis Digitales Restricciones de uso

UNIVERSIDAD NACIONAL AVPNOMA DE MEXICO

M. EN C. VIRGINIA ABRIN BATULE
Jefe de la División de Estudios Profesionales
Facultad de Ciencias
P r e s e n t e

Los abajo firmantes, comunicamos a Usted, que habiendo revisado el trabajo de Tesis que realiz(ó)ron el pasante(s) Francisco Hernández Quiroz

con número de cuenta 8006662-8 con el Título: La preespecificación más débil

Otorgamos nuestro Voto Aprobatorio y consideramos que a la brevedad deberá presentar su Examen Profesional para obtener el título de Matemático

GRADO	NOMBRE(S)	APELLIDOS COMPLETOS	FIRMA
Doctor	Felipe	Bracho Carpizo	
Director de Tesis M. en C.	José Alfredo	Amor Montaño	
Doctora	Hanna	Oktaba	
M. en C.	Elisa	Viso Gurovich	
Suplente M. en C.	Carlos	Torres Alcaraz	
Suplente			

INSTITUTO DE INVESTIGACIONES
EN MATEMATICAS APLICADAS
Y EN SISTEMAS
UNIVERSIDAD NACIONAL AUTONOMA DE MEXICO
APDO. POSTAL 20-726 ADMON. No. 20
DELEGACION DE ALVARO OBREGON
01000 MEXICO, D. F.
TELEX: 1764062 IIMASME
FAX: 5500047

México, D.F., a 8 de julio de 1994

Universidad Nacional Autónoma de México
Facultad de Ciencias
División de Estudios Profesionales
Presente

Por medio de la presente, quiero hacer constar que el pasante de la carrera de matemático Francisco Hernández Quiroz, con número de cuenta 8006662-8, ha concluido la elaboración de su tesis, titulada "La preespecificación más débil".

Sin otro particular por el momento, agradezco su amable atención.

Atentamente,

Dr. Felipe Bracho Carpizo

ÍNDICE

INTRODUCCIÓN

Cuando los programas se reducían a unas cuantas líneas de código que efectuaban unos pocos comandos no se apreciaba la importancia de demostrar que los programas hacían lo que deberían hacer. No obstante, conforme fue aumentando su complejidad y adquirieron dimensiones enormes (y de esto ya hace bastante tiempo) la necesidad de programar sobre un terreno más firme no sólo fue clara sino urgentísima, pues pronto quedó claro que depurar (*debug*) y probar con casos específicos los programas era insuficiente y poco seguro: no es raro que el número de casos sea, si no infinito, indefinible, y la depuración es una tarea ingrata, muy difícil y en sí misma se presta al error. Por otra parte, los errores en los programas de cómputo pueden ser costosos o incluso pueden poner en peligro vidas humanas: si un programa de control de cuentahabientes de un banco deposita más o menos dinero del debido en una cuenta puede mejorar o arruinar la economía de alguien; si el programa de control de aterrizajes en un aeropuerto asigna la misma pista a dos aviones distintos el resultado puede ser una tragedia. En consecuencia, asegurar la confiabilidad de los programas no es un objetivo sin importancia práctica: es una tarea prioritaria.

¿Cómo se puede hacer que los programas sean confiables? El primer paso es establecer claramente la naturaleza del problema. Considérese el siguiente programa (expresado en un lenguaje similar a Pascal):

```
Var
   a, i: integer;
   primo: boolean;
Begin
   ⋮
```

```
  primo := true;
  i := 2;
  while (i < a and primo = true) do
  begin
    if (a mod i) = 0 then
      primo := false;
    i := i + 1;
  end;
  if (primo = true) then
    writeln(a,' es primo');
End.
```

Desde luego, es un método poco eficiente para saber si *a* es primo. Pero ése no es su principal defecto: también es incorrecto. Sea *a* un número no primo menor que 0. En este caso, el ciclo indicado por "while" no se ejecuta ni una sola vez y, en consecuencia, la variable *primo* no se modifica. El resultado es un mensaje erróneo sobre *a*.

Aunque el programa es incorrecto, el compilador de "Pascal" no protestó a la hora de traducirlo al código de máquina, pues es correcto *sintácticamente*. El problema se encuentra en otro lado: la ejecución del programa no garantiza que se cumpla lo que se espera de él, pues las acciones que realiza son insuficientes (y aun equivocadas) para producir el resultado deseado. Es como si al querer expresar un hecho en inglés (por ejemplo, que está lloviendo en el este de Londres), se escribiera una oración que respetara la sintaxis de este idioma, pero cuyo significado fuera algo distinto a lo que se quiere comunicar (por ejemplo, "It's raining in Leeds"). En otras palabras, el problema se encuentra en el nivel *semántico* del programa.

Antes de proceder, conviene aclarar los términos "sintaxis" y "semántica" e introducir el de pragmática (sólo para completar la lista). Lucas [1982] da estas definiciones (tomadas a su vez del trabajo del filósofo estadounidense Charles Morris):

La *sintaxis* tiene que ver con cuáles combinaciones de símbolos son válidas (pertenecen a un lenguaje dado) sin considerar el significado o el contexto en el que se producen.

La *semántica* se ocupa del significado de los símbolos de un lenguaje, tanto aislados como en un contexto dado.

La *pragmática* de la relación de los símbolos con el contexto en el cual se producen: su origen, su uso y sus efectos en la conducta de emisores y receptores de símbolos.

No hay una frontera perfectamente definida entre estas áreas.[1] Pero la distinción es útil para el ejemplo anterior y para el caso general. El aspecto sintáctico corre por cuenta del compilador.[2] La semántica se ocupa de determinar si el significado del programa, tal como está formulado, es el deseado. La pragmática tiene que ver con el objetivo que se persigue al querer averiguar si *a* es primo o no (y con la utilidad que pueda tener este dato en el contexto general).

Para poder saber si un programa es confiable o no, es necesario que la confirmación de su corrección se realice a nivel de la semántica. Como ya se vio, un programa correcto sintácticamente no es necesariamente correcto en su significado. No obstante, la determinación de la corrección semántica es mucho más complicada que la del tipo sintáctico. El problema de la pragmática es aún más difícil, pues demostrar que un programa satisface ciertas necesidades (humanas, desde luego) es tan inalcanzable como querer especificar sin ambigüedades dichas necesidades. ¡Cuántos usuarios de computadoras gastan una buena parte de su tiempo en busca del programa ideal para sus fines personales!

El objetivo del presente trabajo es abordar el tema de la semántica de los lenguajes de programación. La gente que se dedica a este campo sostiene que por medio de la definición rigurosa del significado de las expresiones de un lenguaje es posible (o menos improbable) demostrar la corrección de un programa. Las matemáticas son el terreno de las demostraciones y las definiciones precisas por excelencia y la lógica es su instrumento privilegiado. Por tanto, parece conveniente "matematizar" el lenguaje en el que está escrito un programa de computadora

[1] Por ejemplo, se ha propuesto que, de hecho, el significado de los símbolos de un lenguaje se deriva de su uso en contextos específicos (lo que forma parte de la pragmática).

[2] Esto no es estrictamente cierto, pues el compilador también verifica algunos hechos que más bien forman parte de la semántica. Cómo se sabe, la mayor parte de un lenguaje de programación responde a una gramática libre del contexto. Pero en muchos casos el compilador verifica que, cuando se usa una variable, ésta se haya definido antes, o que el tipo de la variable corresponda a la operación que se realizará. Dependiendo del método que se haya elegido para definir el lenguaje, estas dos cuestiones forman parte en mayor o menor medida de la semántica.

para demostrar la corrección de éste, es decir, traducir a conceptos matemáticos los objetos que forman el lenguaje, como las variables, las secuencias de control de ejecución, las funciones, los procedimientos, etc., y demostrar que de las propiedades de estos objetos y su disposición en el programa se deduce el resultado esperado.

Sin embargo, de la necesidad de hacer que los programas sean confiables no se sigue inmediatamente que haya que demostrar lógicamente sus propiedades. Es perfectamente legítimo hacerse estas preguntas: ¿hace falta demostrar los programas? ¿No hay un método más fácil, como la depuración rigurosa o los ensayos con casos prototípicos (*testing*)?

Cualquiera que se haya enfrentado a la tarea de revisar y probar un programa de grandes dimensiones sabe la magnitud de las dificultades con las que se tropieza. La estrategia de definir unos cuantos casos prototípicos no siempre es fácil de desarrollar. Por ejemplo, ¿los casos prototípicos son los que de alguna manera representan límites posibles en la arquitectura de la máquina o en el compilador (por ejemplo, valores que produzcan enteros que rebasen el rango máximo o mínimo en la máquina específica)? O, por el contrario, ¿son los que pueden producir resultados matemáticamente inesperados? Es claro que con esta estrategia dependemos de qué tan acertada haya sido nuestra elección de los casos prototípicos. Además, como señala Edsger W. Dijkstra [1976], por este método sólo es posible demostrar (cuando mucho) que un programa tiene errores, pero nunca que *no* los tiene.

Por otra parte, intentar depurar un programa después de que éste falló con algún caso se parece mucho a la búsqueda de una aguja en un pajar: hay que tratar de aislar la sección del programa en la que ocurrió la falla; después hay que descubrir en ésta el punto exacto donde se encuentra el error; y, por último, hay que modificar el programa de forma que el error quede eliminado (lo cual, con frecuencia, no es trivial).

La estrategia de demostrar la corrección del programa sigue una orientación opuesta. El objetivo es construir un programa tal que se pueda estar seguro (en la medida en que las demostraciones sean correctas) de que cada sección realiza la tarea que deber realizar si se cumplen determinadas condiciones especificadas por el programador. Por otra parte, como lo señalan Dijkstra [1976] y David Gries [1981], la demostración de la corrección de un programa "abstracto" puede

servir como estructura básica para construir un programa real.[3] De esta forma, el programa resultante es lo más sólido posible y, si es que aún persiste algún error, la existencia de una demostración previa del programa, aunque sea imperfecta, permite atacar con más facilidad el problema.

Las siguientes son otras razones para recurrir a técnicas formales en lugar de métodos informales:

1) Hasta ahora, las mejores (y únicas) técnicas de demostración siguen siendo las formales, en particular las que se apoyan en la lógica.

2) Como se verá más adelante, la asignación de significado a un programa es un problema bastante complicado. La mejor forma de especificar lo que queremos que haga un programa es enunciar formalmente (*i.e.*, con toda precisión) las tareas que debe realizar. Con esta especificación formal es mucho más fácil demostrar que el programa hace lo que queremos.

3) Hay que recordar que la aplicación de técnicas formales a la sintaxis de los lenguajes ha hecho altamente confiable la compilación de programas. ¿Por qué no esperar beneficios análogos de la formalización de la semántica?

¿Por qué se debe usar la lógica para verificar los programas?

A finales del siglo pasado y principios de éste existía entre muchos matemáticos la preocupación por crear mecanismos formales para la demostración de teoremas matemáticos.[4] La idea era que, con estos mecanismos, se podría encomendar la demostración de los teoremas a las máquinas (en aquella época no existían, obviamente, computadoras, así que el término "máquina" era aún una abstracción).

Una parte de la lógica matemática moderna nació de los *formalismos* creados con este fin (es decir, de los lenguajes formales construidos para tratar de expresar simbólicamente los enunciados de la matemática).[5] Pero la inexistencia en

[3] Este último paso no se discutirá en el presente trabajo.

[4] En realidad, esta idea viene cuando menos desde Leibniz y su *calculus ratiocinator*. Leibniz habló de crear un lenguaje simbólico con el cual formalizar los argumentos. De este modo, sería posible verificar mecánicamente su corrección.

[5] En Lolli [1987] se expone parte de la historia de estos intentos y algunos argumentos en su contra.

aquellos tiempos de máquinas reales impidió llevar más adelante el objetivo de verificar automáticamente las demostraciones o, incluso, de demostrar teoremas no probados anteriormente.

Cuando surgen las computadoras modernas, nos encontramos ante el problema opuesto: tenemos mecanismos completamente formales que producen "fórmulas" (es decir el *output* de algún programa) que se derivan estrictamente de "axiomas", reglas de derivación e hipótesis (el *input* recibido: programas y datos) de manera rigurosa. En cierta forma, son la realización del sueño de automatizar las demostraciones y los razonamientos.

El problema ahora es establecer el significado de estas "demostraciones" y su relación con los objetivos que debe cumplir un programa dado. En síntesis, se trata de *demostrar* que las computadoras "demuestran" lo que se espera de ellas. Para lograr este objetivo, los mecanismos construidos por la lógica resultan un herramienta excelente, pues la lógica se preocupa sólo por las propiedades formales de los programas (lo que se ajusta muy bien al modo de actuar de una computadora) y por la relación entre estas propiedades y la especificación de lo que debe hacer el programa (el significado deseado).[6]

Un poco de historia

La idea de demostrar la corrección de los programas surgió ya hace varias décadas. Gries [1981] hace un breve recuento de cómo se originó la preocupación por la *metodología de la programación* a finales de los 60 y de algunos esfuerzos iniciales en ese sentido. Dicha preocupación generó una nueva área de trabajo en la ciencia de la computación, cuyo interés fundamental fue el encontrar técnicas para crear programas más confiables, comprensibles y manejables y, en especial, técnicas para demostrar la corrección de los programas. Lucas [1982], por su parte, hace una breve comparación entre los distintos puntos de vista en el área de semántica.

Floyd [1967], en un trabajo sobre el significado de los programas, sugirió que la especificación de las técnicas de demostración de un programa podría constituir

[6] Lolli [1987], cap. 4, también discute otras razones que llevaron a los científicos de la computación a buscar la verificación formal de sus programas por medio de la lógica y presenta su relación con los trabajos de los lógicos matemáticos, todo desde el ángulo de la filosofía de la matemática.

una definición adecuada del mismo, es decir, que al mismo tiempo que se escribe la demostración de un programa se está dando una definición formal de éste.

C.A.R. Hoare [1969] definió un lenguaje de programación con una base axiomática, con lo que llevó a cabo la sugerencia de Floyd. El enfoque de Hoare fue un giro revolucionario en la especificación de los programas: la semántica se puede construir axiomáticamente (como cualquier otra teoría formalizada) y, de este modo, nos acercamos realmente al objetivo de demostrar su corrección.

Dijkstra [1975] introdujo el concepto de *precondición más débil* (*weakest precondition*) para definir el comportamiento de un programa. A partir de este concepto, Dijkstra derivó un lenguaje de programación cuya semántica desarrolló rigurosamente.

El trabajo de Dijkstra se convirtió de inmediato en uno de los fundamentos de la nueva *disciplina de programación* (como Dijkstra [1976] llamó a uno de sus libros más célebres).

El enfoque utilizado por Dijkstra y otros investigadores, como Hoare, se conoce como semántica axiomática. La razón es obvia: el método de trabajo elegido para definir la semántica se basa en la enunciación de una serie de axiomas (el significado de los constructos del lenguaje) y reglas de inferencia que permiten derivar un programa a partir de los axiomas y, a la vez, probar su corrección.

Otros acercamientos a la semántica parten de ideas distintas. En la semántica operacional, el significado de los enunciados del lenguaje se define a partir de la especificación del *estado* de una máquina hipotética (o "intérprete definicional") antes de ejecutar una acción y el estado después de ejecutarla. Estas ideas se aplicaron en la notación conocida como VDL (*Vienna Definition Language*).

Un punto de vista totalmente distinto es el que se sigue en la semántica denotacional o matemática. Esta vertiente pone énfasis en la definición matemática de los objetos a los que hacen referencia (los objetos denotados por) los enunciados del lenguaje y se ha vuelto muy popular, sobre todo a través de la notación llamada VDM (*Vienna Definition Method*).

La preespecificación más débil

Siguiendo una orientación más cercana al trabajo de Dijkstra, Hoare y He Jifeng [1986] idearon otro concepto de carácter más general (y poderoso) que la

precondición más débil de Dijkstra: la *preespecificación más débil* (*weakest prespecification*), el cual es el tema central de esta tesis. El objetivo de mi escrito es exponer algunas de sus propiedades y explicar cómo se puede utilizar para construir un lenguaje de programación de gran poder (que incluye al lenguaje propuesto por Dijkstra). El plan es el siguiente:

El primer capítulo presenta algunas ideas de la teoría de conjuntos y se divide en cinco secciones. La primera y la segunda están dedicadas a definir tanto la notación como algunos conceptos muy básicos que se usarán más adelante. La tercera trata del cálculo de relaciones, mientras que la cuarta es sobre las funciones tradicionales y de orden superior y sobre la notación lambda. La última es sobre funciones recursivas y, junto con la tercera y cuarta, constituye el interés real del capítulo.

En el capítulo 2 se desarrolla el concepto de precondición más débil y sus propiedades, así como algunos constructos del lenguaje de Dijkstra basados en ella.

Finalmente, en el capítulo 3 se abordará la preespecificación más débil. En primer lugar, se hará una definición formal de esta noción y se demostrarán sus propiedades. Posteriormente, se presentará el lenguaje $\mathfrak{L}$ (basado en la preespecificación más débil), sus propiedades y sus constructos. Para concluir, se hará mención de cómo se puede hacer cierta equivalencia entre el lenguaje $\mathfrak{L}$ y el VDM. En todo este capítulo se sigue de cerca el artículo de Hoare y He Jifeng [1986].

Se incluye además un apéndice con un breve resumen de conceptos de la lógica elemental: proposiciones, predicados, cuantificadores y sustitución textual. El objetivo es mostrar la notación que se usa en la tesis, hacer explícitas las definiciones de algunos términos técnicos y enunciar algunos teoremas necesarios para las demostraciones de los tres capítulos. Por supuesto, no se pretendió hacer una exposición completa y rigurosa de estos temas. Si se desea profundizar en esta área, se recomienda la lectura de Mendelson [1964] o Enderton [1987].

Los teoremas y corolarios se numeraron consecutivamente en cada capítulo. Por ejemplo, 3.15 quiere decir el teorema (o corolario) 15 del capítulo 3, y A.3 es el teorema 3 del apéndice.

Agradecimientos

Aunque durante la redacción de este trabajo recibí el apoyo de mucha gente, hay varias personas que no puedo dejar de mencionar. La primera es el Dr. Felipe Bracho, mi asesor, cuya ayuda y comprensión rebasó con creces el sentido del deber. No sólo tengo que agradecer su supervisión sino, y sobre todo, el haberme iniciado en el estudio de este campo, al cual espero poder dedicarme de ahora en adelante. La segunda es Ena Lastra, cuyo aliento y estímulo fueron fundamentales. La tercera es Arturo González, quien leyó una versión anterior del apéndice y me hizo muchas sugerencias valiosas.

También deseo dar las gracias a los sinodales que revisaron la presente tesis: el M. en C. José Alfredo Amor Montaño, la Dra. Hanna Oktaba, el M. en C. Carlos Torres Alcaraz y la M. en C. Elisa Viso Gurovich.

Nota técnica

La tipografía de esta tesis se hizo con el programa TEX con macros del autor. La impresión se realizó en una impresora láser IBM 4029 PostScript con el programa dvipsone.

1. RELACIONES Y FUNCIONES

En las dos primeras secciones de este capítulo se expondrán algunos conceptos de la teoría de conjuntos con la intención de presentar la notación empleada en este trabajo. No se profundizará demasiado en ellos pues, seguramente, el lector ya los conoce. En caso contrario, se recomienda la lectura del primer capítulo de Devlin [1979].

En la sección 1.3 se presentará el cálculo de relaciones. La versión que se expone aquí está basada en el artículo clásico de Tarski [1941]. En su escrito, Tarski presenta dos métodos para construir el cálculo de relaciones. En el primero recurre ampliamente a los conceptos del cálculo de predicados, que son *externos* al cálculo de relaciones. En el segundo, en cambio, utiliza una serie de axiomas a partir del cual se puede construir el cálculo sin elementos externos. Aquí no se utilizará un enfoque axiomático, pero en el fondo se tratará del primer método y no se tocarán los problemas formales derivados de él.

La sección 1.4 trata sobre funciones, desde su definición más básica hasta una versión informal de la notación lambda. La sección 1.5 es la menos elemental de todas, pues aborda la definición de las funciones recursivas y un método para calcularlas. Por supuesto, no se pretendió hacer un desarrollo formal riguroso y se excluyeron todas las cuestiones relacionadas con el problema de la computabilidad y la decidibilidad de las funciones recursivas (lo cual es el tema verdadero de la teoría de funciones recursivas). Para estas cuestiones, se recomienda la lectura de Rogers [1987]. Aquí nos limitamos a exponer los problemas de cómo definir una función recursiva y de qué tipo de función es desde el punto de vista de la teoría de conjuntos. Además, se dan algunas condiciones suficientes para saber si una función recursiva tiene un punto fijo mínimo (un referente adecuado para una función recursiva).

1.1 Operaciones fundamentales

Como es tradicional, cuando se hable explícitamente de los elementos de un conjunto, éstos se indicarán encerrados entre llaves. Por ejemplo, los números naturales: $\{1,2,3,\ldots\}$. A veces, se denotará un conjunto por medio de una definición: $\{x \mid x \text{ es par y } 2 < x\}$.

Las operaciones y las relaciones básicas entre conjuntos, a saber, intersección, unión, complemento e inclusión, se simbolizan de este modo:

Sean A, B dos conjuntos.

1. $A \cup B$ es la *unión* de los dos conjuntos, es decir, $\{x \mid x \in A \vee x \in B\}$.

2. $A \cap B$ es la *intersección* de los dos conjuntos, *i.e.*, $\{x \mid x \in A \wedge x \in B\}$.

3. $A - B$ es el *complemento* de B con respecto a A, o sea, $\{x \mid x \in A \wedge x \notin B\}$.

4. $\bar{A}$ es el *complemento* de A respecto de un conjunto universal. El contexto en el que aparezca determinará de que conjunto universal se trata.

5. Se dice que A es un *subconjunto* de B si $\forall x(x \in A \Rightarrow x \in B)$, lo cual se denotará $A \subseteq B$. Si $\exists x(x \in B \wedge x \notin A)$, entonces se dice que A es un subconjunto *propio* de B.

6. El *conjunto vacío*, $\emptyset$, es el conjunto que no tiene elementos. Dada la definición anterior de subconjunto, es claro que $\emptyset \subseteq A$ para todo A.

7. El *conjunto potencia* de A, simbolizado $\mathcal{P}(A)$, es el conjunto de todos los subconjuntos de A.

8. Una familia indexada[1] $H = \{H_n\}_{n \in \eta}$ de conjuntos H_n con índices en η contiene un conjunto por cada elemento en η. En este trabajo sólo se usarán familias indexadas por $\mathbb{N}$ y, por esto, se usará indistintamente $H = \{H_n\}_{n \in \mathbb{N}}$, $H = \{H_n\}_{0 \leq n}$ o, simplemente, $H = \{H_n\}$.

9. Sea $A = \{A_i\}$ una familia de conjuntos. Se pueden realizar las operaciones de unión e intersección sobre todos los elementos de la familia y se denotarán de la siguiente forma:

[1] Tal vez sea más correcto traducir la palabra en inglés *indexed* por *indizada*. Sin embargo, casi nadie usa esta última en los textos de matemáticas.

a) $\bigcup_{i \in \mathbb{N}} A_i$ es la unión de todos los A_i, con $i \in \mathbb{N}$.

b) $\bigcap_{i \in \mathbb{N}} A_i$ es la intersección de todos los A_i, con $i \in \mathbb{N}$.

c) Cuando aparezca simplemente $\bigcap_i$ o $\bigcup_i$ se dará por hecho que $i \in \mathbb{N}$. También se puede indicar este caso con $\bigcap_{0 \leq i}$ y $\bigcup_{0 \leq i}$.

Ahora se demostrarán algunos teoremas de la teoría de conjuntos (los más obvios sólo se enunciarán):

Teorema 1.1. $A \subseteq B$ si y sólo si $A \cap B = A$.

Demostración. Sea $x \in A$. Por la definiciones de $\cap$ y de $\subseteq$, es claro que

$$\forall x((x \in A \wedge x \in B) \Rightarrow x \in A).$$

Por el teorema A.8 y *modus ponens*, se sigue que $x \in B \Rightarrow x \in A$. También por hipótesis, $x \in A \Rightarrow x \in B$ y, entonces, $x \in A \wedge x \in B$, es decir, $x \in A \cap B$ y $A \subseteq A \cap B$. La contención contraria es parte de la definición de $\cap$ y la implicación inversa es análoga. ■

Teorema 1.2. $A \subseteq B$ si y sólo si $A \cup B = B$.

Demostración. Sea $A \subseteq B$. Entonces, si $x \in A \cup B$, $x \in A \vee x \in B$. Por el teorema A.7, $x \notin A \Rightarrow x \in B$. Pero $x \in A \Rightarrow x \in B$ por hipótesis y, por A.4, $x \in B$ y $A \cup B \subseteq B$. La contención inversa es parte de la definición de $\cup$ y la implicación contraria es análoga. ■

Teorema 1.3. $A \cap B = \emptyset \Rightarrow A \subseteq \bar{B}$.

Teorema 1.4. $(A \cap B) \subseteq C \Leftrightarrow A \subseteq (\bar{B} \cup C)$.

Demostración. Supongamos que $(A \cap B) \subseteq C$. Entonces,

$$\forall x((x \in A \wedge x \in B) \Rightarrow x \in C). \qquad (1)$$

Por los teoremas A.8 y A.7, (1) equivale a

$$\forall x(x \in A \Rightarrow (x \notin B \vee x \in C)),$$

es decir, $A \subseteq (\bar{B} \cup C)$. La implicación inversa es análoga. ■

Teorema 1.5. $(A \cap B) \cup C = (A \cup C) \cap (B \cup C)$ y $(A \cup B) \cap C = (A \cap C) \cup (B \cap C)$.

El tema siguiente es un tipo especial de conjuntos: los productos cartesianos.

1.2 Producto cartesiano

Se dice que dos conjuntos A y B son iguales si y sólo si A y B contienen los mismos elementos. Por ejemplo, si $A = \{1, 2\}$ y $B = \{2, 1\}$, entonces A y B son iguales. Como se ve, no importa el orden en que se escriban los elementos.

No obstante, en muchas ocasiones es necesario recurrir a objetos formados por dos elementos cuyo orden es pertinente. Es el caso de la representación de los puntos del plano cartesiano por medio de pares ordenados de números reales. Si $\mathbf{x} = (1, 0)$ y $\mathbf{y} = (0, 1)$, se tiene que $\mathbf{x} \neq \mathbf{y}$. En este caso, sí es importante el orden de los elementos del par. Por este motivo, cuando se usen *pares ordenados* de la forma (x_1, x_2), el orden es fundamental.

¿Es posible definir un par ordenado sólo en términos de conjuntos? La respuesta es sí. Considérense los pares ordenados (x_1, y_1) y (x_2, y_2). ¿Qué propiedad de estos pares nos interesa en especial? Pues que $(x_1, y_1) = (x_2, y_2)$ si y sólo si $x_1 = x_2$ y $y_1 = y_2$. Si se define el par ordenado (x, y) como una abreviatura del conjunto $\{\{x\}, \{x, y\}\}$, se conservará la propiedad deseada. Sin embargo, en el presente trabajo no se demostrará este hecho.

Los pares ordenados se pueden generalizar a n-adas ordenadas si se considera la n-ada $(x_1, x_2, \ldots, x_n)$ un par ordenado de la forma $((x_1, x_2, \ldots, x_{n-1}), x_n)$, en el cual el primer elemento es una $(n-1)$-ada ordenada. El proceso se continúa hasta llegar a un par ordenado simple.

Definición. El *producto cartesiano* de los conjuntos A, B, que se indicará así: $A \times B$, es el conjunto $\{(a, b) \mid a \in A \wedge b \in B\}$.

También se puede generalizar el producto cartesiano de $A_1 \times A_2 \times \cdots \times A_n$ como $\{(a_1, a_2, \ldots, a_n) \mid a_i \in A_i\}$.

Puesto que en un par ordenado la ubicación de los elementos no es irrelevante, $\times$ no es un operador conmutativo.

1.3 Relaciones

1.3.1 *Definición y propiedades básicas*

Sean A, B, dos conjuntos. Sea $R \subseteq A \times B$. Entonces, R es una *relación* con dominio A (denotado $\mathfrak{D}(R)$) y contradominio B (que se simbolizara $\mathfrak{R}(R)$). Si para todo $x \in A$ existe $y \in B$ tal que $(x,y) \in R$, se dice que R es una *relación total*. En caso contrario, R es una *relación parcial*. Sea $C \subseteq A$; entonces $R(C) = \{y \mid \exists x(x \in C \wedge (x,y) \in R)\}$; $R(C)$ se llamará la imagen de C bajo la relación R. Si $D \subseteq B$, entonces $R^{-1}(D) = \{x \mid y \in D \wedge (x,y) \in R\}$; $R^{-1}(D)$ se llamará la imagen inversa de D bajo R. Si una relación es total, $R^{-1}(B) = A$.

En la definición anterior se habla de relaciones *binarias*, es decir, formadas por pares ordenados. Si se generaliza del mismo modo que el producto cartesiano, se pueden tener relaciones n-arias, formadas por n-adas ordenadas.

Además de las operaciones que se pueden realizar con conjuntos, existen otras propias de las relaciones. Sean R y Q dos relaciones:

1) Composición: $R ; Q = \{(x,x') \mid \exists y(x,y) \in R \wedge (y,x') \in Q\}$.

2) Complemento: $\bar{R} = \{(x,y) \mid (x,y) \notin R\}$.

3) Conversa (o recíproca): $\check{R} = \{(x,y) \mid (y,x) \in R\}$.

Por otro lado, las siguientes relaciones son importantes:

a) Vacía: $\emptyset$.

b) Universal: $U = A \times B$.

c) Identidad: $I = \{(x,y) \mid x = y\}$.

Las relaciones pueden ser de tipos muy variados. No obstante, hay tres especialmente interesantes:

i) Simétricas: R es simétrica si para todo $(x,y) \in R$, se tiene que $(y,x) \in R$.

ii) Reflexivas: R es reflexiva si para todo $x \in \mathfrak{D}(R)$ se tiene que $(x,x) \in R$.

iii) Transitivas: R es transitiva si para todos los pares ordenados $(x,y),(y,z) \in R$ se tiene que $(x,z) \in R$.

La relación identidad cumple estas tres condiciones. En cambio, $\leq$ sólo cumple las dos últimas.

A continuación se demostrarán algunos teoremas:

Teorema 1.6. $R ; I = I ; R$.

Demostración. Sea $(x, y) \in R$. Entonces, (x, y) está en $R ; I$, puesto que $(y, y) \in I$. Pero como (x, x) también está en I, se tiene que $(x, y) \in I ; R$ y $R ; I \subseteq I ; R$. La inversa es evidente. ■

Teorema 1.7. $R ; \emptyset = \emptyset ; R = \emptyset$.

Demostración. Sea R una relación. Es claro que $(x, y) \in R ; \emptyset$ si y sólo si $\exists z\,(x, z) \in R \wedge (z, y) \in \emptyset$. Pero no hay z que cumpla esta condición. Por tanto, $R ; \emptyset = \emptyset$. $\emptyset ; R$ es obvia. ■

Teorema 1.8. $P \cap U = P$.

Demostración. Por definición, $P \subseteq U$. Entonces, por el teorema 1.1, $P \cap U = P$. ■

Teorema 1.9. $(P ; Q) ; R = P ; (Q ; R)$.

Demostración. Sea $(w, z) \in (P ; Q) ; R$. Entonces, existen x, y tales que $(w, x) \in P \wedge (x, y) \in Q \wedge (y, z) \in R$. Pero, por estas mismas razones, $(x, z) \in Q ; R$ y, en consecuencia, $(w, z) \in P ; (Q ; R)$ y $(P ; Q) ; R \subseteq P ; (Q ; R)$. La contención inversa es análoga. ■

Teorema 1.10. $U ; U = U$.

Demostración. Claramente, $\forall x \forall y ((x, y), (y, y) \in U)$. Entonces $(x, y) \in U ; U$ y $U ; U \subseteq U$. La contraria es evidente. ■

Teorema 1.11. $(P \cup Q) ; R = (P ; R) \cup (Q ; R)$.

Demostración. $(x, y) \in (P \cup Q) ; R$ si y sólo si $\exists w (x, w) \in (P \cup Q) \wedge (w, y) \in R$. Por otra parte, $(x, w) \in (P \cup Q)$ si y sólo si $(x, w) \in P \vee (x, w) \in Q$. Pero, entonces, $(x, y) \in (P ; R) \vee (x, y) \in (P ; R)$, por lo que $(P \cup Q) ; R \subseteq (P ; R) \cup (Q ; R)$. La contención inversa es similar. ■

Teorema 1.12. $R;(P \cup Q) = (R;P) \cup (R;Q)$.

Demostración. $(x,y) \in R;(P \cup Q)$ si y sólo si $\exists w(x,w) \in R \wedge ((w,y) \in P \vee (w,y) \in Q)$. De este modo, $(x,y) \in R;P$ o bien $(x,y) \in R;Q$. Por tanto, $(x,y) \in (R;P) \cup (R;Q)$ y $R;(P \cup Q) \subseteq (R;P) \cup (R;Q)$. La inversa se demuestra del mismo modo. ■

Teorema 1.13. Si $P \subseteq Q$ y $R \subseteq S$, entonces $(P;R \subseteq Q;S)$.

Demostración. Sea $(x,z) \in P;R$. Entonces, $\exists y((x,y) \in P \wedge (y,z) \in R)$. Por la hipótesis, $(x,y) \in Q \wedge (y,z) \in S$. Por tanto, $(x,y) \in Q;S$. ■

Teorema 1.14. $P \subseteq P;U$.

Demostración. Considérese el par $(x,y) \in P$. Por definición, $(y,y) \in U$. Por tanto, por definición de composición, $(x,y) \in P;U$. ■

Teorema 1.15. $P \subseteq Q \Rightarrow \breve{P} \subseteq \breve{Q}$.

Demostración. Sea $P \subseteq Q$ y sea $(x,y) \in P$. Por tanto, $(y,x) \in \breve{P}$. Por hipótesis, $(x,y) \in Q$ y, entonces, $(y,x) \in \breve{Q}$, y se concluye que $\breve{P} \subseteq \breve{Q}$. ■

Teorema 1.16. $\breve{P;Q} = \breve{Q};\breve{P}$.

Demostración. Sea $(x,z) \in \breve{P;Q}$, por lo que $(z,x) \in P;Q$ y

$$\exists y((z,y) \in P \wedge (y,x) \in Q).$$

Por la definición del converso, $(y,z) \in \breve{P}$ y $(x,y) \in \breve{Q}$ y, en consecuencia, $(x,z) \in \breve{Q};\breve{P}$ y $\breve{P;Q} \subseteq \breve{Q};\breve{P}$. La contención inversa es análoga. ■

Teorema 1.17. $\breve{\breve{P}} = P$.

Demostración. Sea $(x,y) \in P$. Entonces, $(y,x) \in \breve{P}$ y $(x,y) \in \breve{\breve{P}}$, *i.e.*, $P \subseteq \breve{\breve{P}}$. La contención inversa es similar. ■

Teorema 1.18. $\bar{\breve{P}} = \breve{\bar{P}}$.

Demostración. Supongamos que $(x,y) \in \bar{\breve{P}}$, de donde se deduce que $(y,x) \in \bar{P}$ y $(y,x) \notin P$. Entonces, $(x,y) \notin \breve{P}$, *i.e.*, $(x,y) \in \breve{\bar{P}}$ y $\bar{\breve{P}} \subseteq \breve{\bar{P}}$. Siguiendo el camino inverso se llega a la otra contención. ■

Teorema 1.19. $(P ; Q) \cap \check{R} = \emptyset \Leftrightarrow (Q ; R) \cap \check{P} = \emptyset$.

Demostración. Sea $(P ; Q) \cap \check{R} = \emptyset$. Por tanto,

$$\forall (x,y)\,(x,y) \in (P ; Q) \Rightarrow ((x,y) \notin \check{R} \wedge (y,x) \notin R). \tag{2}$$

Por definición,

$$(x,y) \in P ; Q \Rightarrow (\exists z\,(x,z) \in P \wedge (z,y) \in Q). \tag{3}$$

Pero $(x,z) \in P$ equivale a $(z,x) \in \check{P}$. Si este es el caso, de (2) y (3) se deduce que $(z,x) \notin Q ; R$. En conclusión

$$\check{P} \cap (Q ; R) = \emptyset.$$

La conclusión inversa es análoga. ∎

1.3.2 *Relaciones finitarias*

El problema con una relación parcial es que no hay garantía de que para cada elemento x del dominio exista una y tal que el par ordenado (x,y) pertenezca a la relación. Para los casos en que se necesiten relaciones totales y, al mismo tiempo, no se quiera "agregar información" innecesaria a una relación parcial, se puede recurrir al siguiente "truco".

Definición. Sea $R \subseteq A \times B$ una relación estrictamente parcial. $R^{\perp}$, la *extensión* de R, se define del siguiente modo:

$$R^{\perp} = \{(x,y) \mid ((x,y) \in R) \vee (x \notin R^{-1}(B) \wedge y = \perp)\},$$

donde $\perp$ designa un elemento especial que *no pertenece* a B. Sea $\Sigma = B \cup \{\perp\}$. $R^{\perp}$ es total, por supuesto.

Con $\perp$ se está definiendo un nuevo contradominio, Σ, y una nueva relación $R^{\perp}$ que es igual a R, excepto en el caso en que R sea estrictamente parcial, pues $R^{\perp}$ incluirá entonces todos los pares de la forma $(x,\perp)$ tales que x no está en la imagen inversa de R. El tener un elemento distinto, como $\perp$, permite controlar los elementos que se agregan a R para completarla. En el capítulo 3 se verá la utilidad de este método. Por ahora, tenemos otra definición:

Definición. Sea $C \subseteq \Sigma$ y $R \subseteq \Sigma \times \Sigma$. La imagen de C bajo R se denotará así $C \uparrow R$, que es igual al conjunto $\{y \mid \exists x(x \in C \wedge (x,y) \in R)\}$.

$C \uparrow R$ no es más que otra forma de decir $R(C)$ cuando se presupone que $R \subseteq \Sigma \times \Sigma$. Aunque por ahora parezca que pierde parte de su sentido la extensión de B por medio de $\perp$, más adelante se verá que este elemento distinto de Σ desempeña un papel importante en la expresión de los programas de computadora por medio de relaciones.

Definición. Sea $R \subseteq \Sigma \times \Sigma$. R es una relación *total finitaria* si, $\forall s \in \Sigma$, se tiene que $\{s\} \uparrow R$ es un conjunto finito no vacío o es el conjunto universal Σ y, además, $\{\perp\} \uparrow R$ es Σ.

Todas las relaciones que aparezcan en los teoremas siguientes son subconjuntos de $\Sigma \times \Sigma$.

Teorema 1.20. Si R es total finitaria, $\Sigma \uparrow R = \Sigma$.

Demostración. Como R es finitaria, $\{\perp\} \uparrow R = \Sigma$. Pero $\{\perp\} \subseteq \Sigma$ y, entonces, $\{\perp\} \uparrow R \subseteq \Sigma \uparrow R$. Como Σ es el conjunto universal, lo anterior equivale al resultado deseado. ■

Teorema 1.21. $B \uparrow (P \cup Q) = (B \uparrow P) \cup (B \uparrow Q)$.

Demostración. Por definición, $B \uparrow (P \cup Q) = \{y \mid \exists x(x \in B \wedge (x,y) \in P) \vee \exists x'(x' \in B \wedge (x',y) \in Q)\}$. De aquí se sigue que $y \in (B \uparrow P) \vee y \in (B \uparrow Q)$, o sea, $y \in (B \uparrow P) \cup (B \uparrow Q)$. En conclusión, $B \uparrow (P \cup Q) \subseteq (B \uparrow P) \cup (B \uparrow Q)$. La contención inversa es similar. ■

Teorema 1.22. $B \uparrow (P ; Q) = (B \uparrow P) \uparrow Q$.

Demostración. Por definición

$$B \uparrow (P ; Q) = \{y \mid \exists x(x \in B \wedge (x,y) \in P ; Q)\}. \tag{4}$$

por definición de ";", (4) es igual a

$$= \{y \mid \exists x \exists z(x \in B \wedge ((x,z) \in P \wedge (z,y) \in Q))\}. \tag{5}$$

Por la asociatividad de la conjunción,

$$= \{y \mid \exists x \exists z(x \in B \wedge (x,z) \in P) \wedge (z,y) \in Q\}.$$

Finalmente, por definición de $\uparrow$ se obtiene el resultado deseado. ■

Teorema 1.23. $\{s\} \uparrow P = \overline{(\{s\} \uparrow \bar{P})}$.

Demostración. Por definición

$$\{s\} \uparrow P = \{y \mid (s, y) \in P\}. \tag{6}$$

Pero $(s, y) \in P$ implica que $(s, y) \notin \bar{P}$, es decir, (6) equivale a

$$= \{y \mid (s, y) \notin \bar{P}\}.$$

Por su parte, $\{s\} \uparrow \bar{P} = \{y \mid (s, y) \in \bar{P}\}$, por lo que se puede concluir que $\{s\} \uparrow P = \overline{\{s\} \uparrow \bar{P}}$. ■

Teorema 1.24. $\{s\} \uparrow (\cap_i P_i) = \cap_i (\{s\} \uparrow P_i)$.

Demostración. Aplicando la definición

$$\{s\} \uparrow \bigcap_i P_i = \{y \mid (s, y) \in \bigcap_i P_i\},$$

que no es más que otra forma de decir

$$\bigcap_i \{y \mid (s, y) \in P_i\},$$

lo cual equivale a $\cap_i (\{s\} \uparrow P_i)$. ■

Las relaciones totales finitarias serán la base del lenguaje de programación de la sección 4.4.

1.4 Funciones

Las funciones se pueden definir a partir de las relaciones de manera directa.

Definición. Sean R una relación y $(x, y), (x, y') \in R$. Si lo anterior implica que $y = y'$, entonces R es una función.

En términos intuitivos, se tiene que R es una función si y sólo si para cada elemento en el dominio se tiene, a lo más, un par ordenado en R.

1.4.1 *Tipos de funciones*

Las funciones pueden ser, al igual que las relaciones, totales o parciales. Además, las funciones también pueden pertenecer a una de las siguientes categorías (o a ninguna de ellas):

1) Inyectivas: sea F una función. Si $(x,y),(z,y) \in F$ implica que $x = z$, entonces F es inyectiva.

2) Sobreyectivas: sea F una función. Si para todo $y \in \mathcal{R}(F)$ existe $x \in \mathcal{D}(F)$ tal que $(x,y) \in F$, entonces F es sobreyectiva.

3) Biyectivas: se dice que F es biyectiva si y sólo si es inyectiva y sobreyectiva.

1.4.2 *Funciones de orden superior*

Hasta ahora no se han puesto restricciones al tipo de argumentos que puede tomar una función. En esta sección se considerarán funciones cuyos argumentos son, a su vez, funciones.

Sea $\mathcal{F} = \{f : A \to A\}$, a saber, el conjunto de funciones de A en A. Sean $f \in \mathcal{F}$ y $F : \mathcal{F} \to \mathcal{F}$. Entonces, F es una *función de orden superior* o *funcional*. Por ejemplo, considérese la función $F(f)(x) = (f(x))^2$. Si $f(x) = 2x$, entonces $F(f)(x) = 4x^2$.

La introducción de funcionales permite redefinir las operaciones entre funciones como funciones de orden superior. Supongamos que $\mathcal{F} = \{h : \mathbb{N} \to \mathbb{N}\}$ y g, $f \in \mathcal{F}$, y sean H_1, H_2 y $H_3 : \mathcal{F} \times \mathcal{F} \to \mathcal{F}$ las siguiente funciones:

$$H_1(f,g) = f + g,$$
$$H_2(f,g) = fg,$$
$$H_3(f,g) = f \,;\, g.$$

H_1 es la suma de las funciones f y g, H_2 es su producto y H_3 es su composición.

1.4.3 *Funciones monótonas, continuas y co-continuas*

Ya que se han presentado los funcionales, es el momento de añadir algunos conceptos que les atañen. El primero de ellos se aplica también a funciones y a conjuntos en general:

Definición. Sea $\mathcal{F} = \{f_n\}$ una familia numerable de funciones tal que, para toda $n \geq 0$, $f_n \subseteq f_{n+1}$. Entonces $\mathcal{F}$ es una *cadena ascendente*.

Definición. Sea $\mathcal{F} = \{f_n\}$ una familia numerable de conjuntos con la propiedad de que, para toda $n \geq 0$, $f_{n+1} \subseteq f_n$. $\mathcal{F}$ es una *cadena descendente*.

La monotonía es una propiedad que pueden tener los funcionales:

Definición. Sean f, g funciones. Un funcional F es monótono si $f \subseteq g$ implica que $F(f) \subseteq F(g)$.

Una propiedad mucho más fuerte que la monotonía es la continuidad:

Definición. Sea $\mathcal{F} = \{f_n\}$ una cadena ascendente de funciones. Sea F un funcional tal que $\mathcal{F} \subseteq \mathcal{D}(F)$. F es continuo si

$$F(\bigcup_{0 \leq n} f_n) = \bigcup_{0 \leq n} F(f_n).$$

Como ya ha ocurrido antes, cuando una definición o propiedad incluye el operador $\cup$, existe una propiedad dual con el operador $\cap$:

Definición. Sea $\mathcal{F} = \{f_n\}$ una cadena descendente de funciones. Sea F un funcional tal que $\mathcal{F} \subseteq \mathcal{D}(F)$. F es co-continuo si

$$F(\bigcap_{0 \leq n} f_n) = \bigcap_{0 \leq n} F(f_n).$$

Del mismo modo que la continuidad, la co-continuidad es una propiedad más fuerte que la monotonía:

Teorema 1.25. Sea F un funcional. Entonces, si F es continuo o co-continuo, entonces F es monótono.

Demostración. Supongamos que $f \subseteq g$ y sea F continua. Sea ahora $\mathcal{F} = \{h_n\}$ con $h_0 = f$ y $h_1 = h_2 = \cdots = h_n = \cdots = g$. Es claro que $\mathcal{F}$ es una cadena ascendente y

$\bigcup_{0\leq n} h_n = f \cup g$. Como F es continua:

$$F(\bigcup_{0\leq n} h_n) = \bigcup_{0\leq n} F(h_n) \text{ y}$$
$$\therefore F(f \cup g) = F(f) \cup F(g).$$

Pero $f \subseteq g$ implica que $f \cup g = g$ y, entonces,

$$F(g) = F(f) \cup F(g).$$

Por el teorema 1.2, $F(f) \subseteq F(g)$.

Supongamos ahora que F es co-continua. Ahora se define $\mathcal{F}' = \{h_n'\}$ así: $h_0' = g$, $h_1' = h_2' = \cdots = h_n' = \cdots = f$. $\mathcal{F}'$ es una cadena descendente y, como F es co-continua,

$$F(\bigcap_{0\leq n} h_n') = \bigcap_{0\leq n} F(h_n').$$

Ahora bien, $f = \bigcap_{0\leq n} h_n = f \cap g = f$. Entonces,

$$F(f) = F(f) \cap F(g).$$

Según el teorema 1.1, se concluye que $F(f) \subseteq F(g)$. ■

Se puede extender fácilmente el uso de funcionales a funciones cuyos argumentos y valores no son funciones, sino relaciones. No es necesario usar un nombre distinto para esta clase de funcionales.

Antes de continuar con otros conceptos relacionados con funcionales conviene presentar una nueva notación.

1.4.4 *Notación lambda*

La notación tradicional para funciones tiene la desventaja de que se presta a confusión entre distintos niveles. Por ejemplo, cuando se habla de $f(x)$, ¿se habla de la función f y la variable x simplemente indica el tipo de valores del dominio de f? ¿O se habla del valor de la función f en x? Por otro lado, si se decidiera que las funciones se deben indicar por letras solas, como f, g, etc., ¿cómo sabemos cuáles y cuántas son las variables que requieren dichas funciones? La introducción de los funcionales hace más necesario aún establecer las diferencias entre estos usos distintos de la misma notación.

Veamos la función identidad. Una forma de indicarla es $I(x) = x$, pero así no queda claro que se trata de la identidad, pues x (considerado como un valor) podría ser un punto fijo de la función I. La identidad también puede indicarse de esta forma:

$$\forall x(I(x) = x). \tag{7}$$

Aquí ya no hay confusión. Pero (7) es un enunciado, no un objeto del tipo de las funciones. ¿Se puede aplicar un funcional a un enunciado? ¿Se aplica el funcional al "referente" del enunciado, a saber, el conjunto $\{(x, x) \mid x = x\}$? No es fácil aplicar un funcional cuando una función se define de esta forma. Aunque es posible obviar el problema en la práctica (a costa de ciertas complicaciones), lo ideal sería tener una notación para funciones que nos permita manejarlas del mismo modo en que se manejan otros objetos. La notación lambda, que se presenta a continuación, es una solución ideal a estos problemas.

En el ejemplo de la función identidad, queremos establecer con claridad el significado de la identidad y, al mismo tiempo, tener una expresión manipulable como un solo objeto. En notación lambda se puede hacer del siguiente modo:

$$I = \lambda x \bullet x,$$

I, desde luego, es el objeto definido;[2] el símbolo λ cumple el papel de un cuantificador aplicado a la variable x, y el significado es que x es la variable a la que se aplicará las operaciones indicadas a continuación del signo $\bullet$. Desde luego, es posible poner otra variable en lugar de x (aunque las reglas de sustitución tienen restricciones equivalentes a las de otros cuantificadores) y la expresión $\lambda y \bullet y$ es igualmente buena como definición de la identidad. La función de la variable cuantificada por λ es la de ser una *variable aparente* (en inglés se llama *dummy variable*), es decir, un indicador de las operaciones que la función definida aplicará a los *argumentos* reales que se quiera.

En términos más generales, aunque sin el rigor de una definición en la notación de Backus-Naur, una función indicada por la notación lambda tiene esta forma:

$$\lambda x \bullet \textit{expresión},$$

[2] No se hará por ahora ninguna distinción entre la identidad marcada por el símbolo '=' y la definición de un nuevo objeto por medio de este mismo símbolo. En el primer caso, '=' indica un afirmación, un hecho, mientras que en el segundo se trata de la asignación de un *nombre* a un objeto dado. En la mayor parte de los casos, la diferencia quedará clara por el contexto.

donde x es un identificador cualquiera, y *expresión* puede ser un identificador, una función u otra expresión lambda. He aquí algunos ejemplos de la notación lambda:

$$\begin{aligned} f(x) = x^2 \quad &\text{se define así} \quad \lambda x \bullet x^2, \\ f(x,y) = x + y \quad &\text{se define así} \quad \lambda x \lambda y \bullet x + y \\ f(x) = c \quad &\text{se define así} \quad \lambda x \bullet c. \end{aligned}$$

La extensión a funcionales no requiere de ninguna particularidad. Sea H un funcional que corresponde a la composición de una función consigo misma y cuyo dominio es $\mathcal{F} = \{f : \mathbb{R} \to \mathbb{R}\}$ (las funciones con dominio y contradominio en los reales). En notación lambda se tiene:

$$H = \lambda f \lambda x \bullet f(f(x)).$$

Para los casos en que se tengan funciones con reglas distintas según el valor del argumento se utilizará una notación prestada de los lenguajes de programación. Sea f la siguiente función (en notación tradicional):

$$f(x) = \begin{cases} 0 & \text{si } 0 < x; \\ 1 & \text{si } x = 0; \\ 2 & \text{si } 0 > x. \end{cases}$$

En notación lambda, se tiene

$$f = \lambda x \bullet \textbf{if } x < 0 \textbf{ then } 0 \textbf{ else if } x = 0 \textbf{ then } 1 \textbf{ else } 2,$$

donde las expresiones **if**, **else** y **else if** se usan del mismo modo que en el lenguaje de programación C.

Las expresiones lambda, como funciones, se pueden aplicar a argumentos (valores) dados. Sean f, H_1, H_2 las siguientes expresiones lambda

$$f = \lambda x \bullet x^2, \quad H_1 = \lambda g \lambda x \bullet g(x) + g(g(x)) \quad \text{y} \quad H_2 = \lambda g \lambda h \lambda x \bullet 2f(x) + h(x).$$

Veamos que resulta si se aplica f a 3, H_1 a f y H_2 a f y $H_1(f)$:

$$\begin{aligned} f(3) &= 3^2 = 9, \\ H_1(f) &= \lambda x \bullet f(x) + f(f(x)) = \lambda x \bullet x^2 + x^4 \\ H_2(f, H_1(f)) &= \lambda x \bullet 2x^2 + x^2 + x^4 \end{aligned}$$

La regla es muy simple: se sustituye uniformemente la primera variable cuantificada por λ por el primer valor proporcionado, después la segunda variable por el segundo valor, etcétera. Por supuesto, en todos los casos se espera que el argumento sea del tipo adecuado para realizar la operación (un entero no puede ser un argumento válido para un funcional, por ejemplo).

1.5 Funciones recursivas

1.5.1 *Funciones recursivas y puntos fijos*

Considérese el ejemplo clásico de función recursiva:

$$0! = 1$$
$$n! = (n-1)!n.$$

En computación, las funciones recursivas surgen con frecuencia como la solución más fácil para diversos problemas.[3] Por ejemplo, la instrumentación de la función anterior en lenguaje Pascal es facilísima:

```
Function Fact(n: integer):integer;
begin
        if n = 0 then Fact:= 1;
            else Fact:=Fact(n - 1) * n;
end;
```

Para aclarar más la naturaleza de las funciones recursivas, convendrá verlas en términos de funcionales y del cálculo lambda. Sea f la siguiente función:

$$f(n) = \begin{cases} 1 & \text{si } n = 0; \\ f(n-1)n & \text{si } n > 0. \end{cases}$$

[3] Existe otra razón, mucho más poderosa, por la que las funciones recursivas son importantes. En teoría de autómatas se ha demostrado que la clase de procedimientos de decisión ejecutados por máquinas de Turing es equivalente a la clase de relaciones recursivas. A su vez, las máquinas de Turing parecen reflejar la naturaleza de las computadoras digitales y, por tanto, los problemas que se pueden solucionar con máquinas de Turing corresponderían a los que se pueden resolver con el uso de computadoras. Véase Mendelson [1964], cap. 5, Enderton [1987], cap. 3 y Brookshear [1989].

que, reformulada en la notación lambda, es:

$$f = \lambda n.\, \textbf{if}\, n = 0\, \textbf{then}\, 1\, \textbf{else}\, f(n-1)n.$$

Esta expresión no es una definición en el sentido en que lo son las definiciones que se han dado anteriormente, pues el *definiendum* aparece en el *definiens*. Por otro lado, una función que se define en términos de otra función (en este caso, de sí misma) nos hace pensar que tal vez sea mejor usar un funcional (con lo que de paso se evita la definición circular):

$$F = \lambda f.\lambda n.\, \textbf{if}\, n = 0\, \textbf{then}\, 1\, \textbf{else}\, f(n-1)n,$$

es decir, F es un funcional con un parámetro señalado por f, que a su vez debe ser una función con un parámetro, señalado por n. ¿F es la definición adecuada de la función factorial? Por supuesto que no, pues un funcional tiene como dominio y contradominio conjuntos de funciones. Si el funcional F no es una función constante, entonces el valor de $F(f)$ será una función diferente dependiendo del valor de f. La función factorial original es, desde luego, única.

Sea $g(n) = \lambda n.2n$. Aplíquese el funcional F a g:

$$F(g) = \lambda n.\, \textbf{if}\, n = 0\, \textbf{then}\, 1\, \textbf{else}\, 2(n-1)n.$$

Es claro que $F(g)$ no es la función factorial. En la definición original de factorial, la función factorial aparece en el cuerpo de su propia definición. Esto sugiere la conveniencia de una función a la cual, al aplicársele el funcional F, vuelva a dar como resultado ella misma, un punto fijo de F, es decir, $F(f) = f$. Supongamos que f es un punto fijo de F. Entonces,

$$f = F(f) = \lambda n.\, \textbf{if}\, n = 0\, \textbf{then}\, 1\, \textbf{else}\, f(n-1)n,$$

o, en términos más tradicionales,

$$f(n) = \begin{cases} 1 & \text{si } n = 0; \\ f(n-1)n & \text{si } n > 0. \end{cases}$$

Ahora sí, f *parece* cumplir con la definición de factorial. Pero, ¿en realidad $f = F(f)$ es la solución deseada? Y, si es así, ¿es la única solución? El siguiente ejemplo (tomado de Bjørner y Jones [1982]) nos dará una justificación de la segunda pregunta.

Sea $G = \lambda g.\lambda n.$ **if** $n = 0$ **then** 1 **else if** $n = 1$ **then** $g(3)$ **else** $g(n-2)$. Nos interesa una g tal que $G(g) = g$. Regresando a la notación tradicional, sean

$$g_1(n) = 1$$

$$g_2(n) = \begin{cases} 1 & \text{si } n \text{ es par;} \\ & \text{indefinida para cualquier otro valor.} \end{cases}$$

dos funciones con dominio en los enteros. Tanto g_1 como g_2 son funciones constantes; sin embargo, g_2 es una función parcial. Aplíquese el funcional G a g_1 y g_2:

$G(g_1) = \lambda n.$ **if** $n = 0$ **then** 1 **else if** $n = 1$ **then** 1 **else** 1.

$G(g_2) = \lambda n.$ **if** $n = 0$ **then** 1 **else if** $n = 1$ **then** indefinida **else** $g_2(n-2)$.

Ambas funciones son puntos fijos de G, pues es fácil darse cuenta que $G(g_1) = g_1$ y $G(g_2) = g_2$, a la vez que las dos satisfacen lo que se esperaba de ellas, pues $g_1(1) = g_1(3) = 1$, $g_1(n) = g_1(n-2) = 1$ para $n > 1$, y, por su parte, $g_2(1) = g_2(3) =$ indefinida y $g_2(n) = g_2(n-2)$ para $n > 1$, ya que si n es par si sólo si $n - 2$ es par cuando $n > 1$. No obstante, es obvio que $g_1 \neq g_2$. La conclusión es que no siempre hay un solo punto fijo y la nueva pregunta es cuál de todos los puntos fijos es la definición adecuada de una función recursiva.

1.5.2 *Puntos fijos mínimos y máximos*

Además de tener que escoger un punto fijo como la función definida por recursión, está el problema de encontrar un método para calcular dicha función. El método que se expone aquí está tomado en parte de Meyer [1990] y guarda semejanzas formales con los métodos iterativos que se ocupan en análisis numérico para hallar soluciones aproximadas a algunas ecuaciones.

En el ejemplo del funcional F, derivado de la definición "informal" de factorial, se intentará calcular el valor de un punto fijo por medio de un método iterativo. El primer paso consiste en encontrar una función de la cual partir. Consideremos la función $\emptyset$:

$$F(\emptyset) = \lambda n.\ \textbf{if } n = 0 \textbf{ then } 1 \textbf{ else } \emptyset(n-1)n.$$

Por definición, $\forall x \forall y((x,y) \notin \emptyset)$ y, por tanto, $F(\emptyset)$ está indefinida para valores de $n > 0$. Sin embargo, $F(\emptyset)(0) = 1$, lo cual es una función (aunque muy parcial). Sean $f_0 = \emptyset$ y $f_1 = F(\emptyset)$; f_2 se calcula de la siguiente forma:

$$f_2 = F(f_1) = \lambda n.\ \textbf{if } n = 0 \textbf{ then } 1 \textbf{ else } f_1(n-1)n.$$

El único valor de f_1 que conocemos es $f_1(0)$. Pero $n - 1 = 0$ cuando $n = 1$ y, entonces, $f_2(1) = f_1(1 - 1) = f_1(0) = 1$, por lo que f_2 está definida en dos puntos: 0 y 1. Si se reitera este procedimiento, se tendrá que

$$f_n = F(f_{n-1}) \qquad n \geq 1.$$

Por otra parte, como $f_1 = \{(0, 1))\}$ y $f_2 = \{(0, 1), (1, 1)\}$, $f_1 \subseteq f_2$. A simple vista, parecería que $\forall n(f_{n-1} \subseteq f_n)$ (aún no se demuestra). Si esto es cierto, estamos ante una definición *inductiva* para calcular los valores de la función factorial, la cual resultaría ser

$$f = \bigcup_{0 \leq n} F(f_n),$$

en caso de que se pueda demostrar que f es un punto fijo de F, es decir

$$f = F(f) = F(\bigcup_{0 \leq n} F(f_n)) = \bigcup_{0 \leq n} F(f_n).$$

La demostración no se hará para la F factorial en particular, sino en general, y para esto hay que utilizar algunos conceptos de la sección 1.4.3. Se comenzará con el siguiente teorema:

Teorema 1.26. Sea F un funcional continuo. La función f_∞ se define de la siguiente forma:

$$f_0 = \emptyset$$
$$f_{n+1} = F(f_n)$$
$$f_\infty = \bigcup_{0 \leq n} f_n.$$

Entonces f_∞ es un punto fijo de F.

Demostración. Se quiere demostrar que $F(f_\infty) = f_\infty$. Por la forma en que se definió f_∞, esto es igual a

$$F(\bigcup_{0 \leq n} f_n) = \bigcup_{0 \leq n} f_n,$$

y el primer paso será demostrar que $\{f_n\}$ es una cadena ascendente de funciones. La demostración se hará por inducción.

Sea $i = 0$. Es obvio que $f_0 \subseteq f_1$, pues $f_0 = \emptyset$.

Supongamos ahora que $\forall 0 < k \leq i$ vale que $f_{k-1} \subseteq f_k$. Consideremos f_{i+1}. Por hipótesis de la inducción, $f_{i-1} \subseteq f_i$. Como F es monótona,

$$F(f_{i-1}) \subseteq F(f_i),$$

y, por definición de f_n, se obtiene la conclusión deseada, a saber, que $f_i \subseteq f_{i+1}$.

El resto es aún más fácil. Como $\{f_n\}$ es una cadena ascendente y F es continua, entonces:

$$F(\bigcup_{0\leq n} f_n) = \bigcup_{0\leq n} F(f_n) = \bigcup_{0\leq n} f_{n+1}.$$

Al correr el índice del último elemento de la igualdad anterior, queda $\bigcup_{1\leq n} f_n$. Pero $f_0 \subseteq f_1$ y, por el teorema 1.2,

$$\bigcup_{1\leq n} f_n = f_0 \cup (\bigcup_{1\leq n} f_n) = \bigcup_{0\leq n} f_n.$$ ■

El teorema anterior nos dice que basta con que un funcional sea continuo para que tenga un punto fijo, a saber, f_∞. La existencia de este punto fijo nos garantiza, además, que podemos utilizar su definición inductiva para calcular por iteración el valor de la función recursiva original. Por todo esto, f_∞ merece un nombre:

Definición. Sea ϕ un funcional continuo. El punto fijo mínimo de ϕ es ϕ_∞.

La razón del nombre se encuentra en el siguiente teorema:

Teorema 1.27. Sea F un funcional continuo y sea f un punto fijo de F. Entonces $f_\infty \subseteq f$.

Demostración. Nuevamente, se recurrirá a la inducción. Sea $i = 0$. Desde luego, $f_0 = \emptyset \subseteq f$.

Sea ahora $i > 0$ y para toda $k \leq i$, se tiene que $f_k \subseteq f$. Hay que demostrar que $f_{i+1} \subseteq f$. Por hipótesis de la inducción, $f_i \subseteq f$. Como F es monótona,

$$F(f_i) \subseteq F(f),$$

y, por definición de f_n y de punto fijo, $f_{i+1} \subseteq f$.

Se ha demostrado que $\forall i \geq 0(f_i \subseteq f)$. Por tanto, $f_\infty = \bigcup_{0\leq n} f_n \subseteq f$. ■

La ventaja del punto fijo mínimo es que es la opción menos comprometedora como referente de una función recursiva, pues se limita a cumplir con las propiedades mínimas para ser un punto fijo del funcional. Sin embargo, no hay ninguna garantía de que f_∞ sea una función total. Si lo que se quiere es un punto fijo que, de ser posible, sea una función total, hay que recurrir al punto fijo máximo:

Definición. Sea F un funcional co-continuo. El conjunto f^∞ se define de la siguiente forma:

$$f^0 = U;$$
$$f^{n+1} = F(f^n);$$
$$f^\infty = \bigcap_{0 \le n} f^n.$$

Hay dos supuestos arbitrarios en la definición anterior. El primero es que f^∞ es una función y no simplemente una relación. Para el lenguaje que se desarrolla en el capítulo 3, no es necesaria esta presuposición y, por este motivo, se puede eliminar tranquilamente. El segundo es que f^∞ es un punto fijo de F ("el punto fijo máximo"), pero el teorema 1.28 lo justifica

Teorema 1.28. Sea F un funcional co-continuo. Entonces f^∞ es un punto fijo de F.

Demostración. Se quiere demostrar que $f^\infty = F(f^\infty)$. El primer paso consistirá en mostrar que $\{f^n\}$ es una cadena descendente de relaciones.

Sea $i = 0$. Entonces, $f^0 = U$ y, por definición, $f^1 \subseteq U$. Sea ahora $i > 0$ y concédase que $\forall k \le i$ es verdad que $f^k \subseteq f^{k-1}$. Como F es monótona, se puede decir que

$$F(f^i) \subseteq F(f^{i-1}),$$

de lo que se sigue automáticamente el resultado deseado.

Puesto que $\{f^n\}$ es una cadena descendente y F es co-continua, entonces

$$F(f^\infty) = F(\bigcap_{0 \le n} f^n) = \bigcap_{0 \le n} F(f^n). \tag{8}$$

Por definición de f^n, (8) equivale a

$$F(f^\infty) = \bigcap_{0 \le n} f^{n+1}.$$

Hacemos otra vez un recorrido del índice n y queda

$$F(f^\infty) = \bigcap_{1 \le n} f^n.$$

Pero, por definición, $f_1 \subseteq U = f^0$ y, por el teorema 1.1,

$$F(f^\infty) = \bigcap_{1 \le n} f^n = U \cap (\bigcap_{1 \le n} f^n) = \bigcap_{0 \le n} f^n = f^\infty. \qquad \blacksquare$$

Por último, se necesita justificar el adjetivo máximo:

Teorema 1.29. Sea F un funcional continuo y sea f un punto fijo de F. Entonces, $f \subseteq f^{\infty}$.

Demostración. Primero hay que justificar la afirmación $\forall i \geq 0 (f \subseteq f^i)$. Recurramos, una vez más, a la inducción. Sea $i = 0$; entonces $f^0 = U$ y, por supuesto, $f \subseteq U$.

La hipótesis ahora es que $\forall k \leq i, f \subseteq f^k$. Por hipótesis $f \subseteq f^i$ y, dado que F es monótona, $F(f) \subseteq F(f^i)$. Finalmente, por definición de f^n y puesto que f es un punto fijo, $f \subseteq f^{i+1}$. ■

El punto fijo máximo es una función que también cumple con las propiedades de la función recursiva expresada por el funcional F, aunque no es tan fácil construirlo ni es tan intuitivo. No obstante, en el capítulo 3 se empleará el punto fijo máximo, y no el mínimo, para el desarrollo de un lenguaje de programación.[4]

[4] La existencia de puntos fijos máximos o mínimos gracias a la continuidad o la co-continuidad no agota el problema de las funciones recursivas. Aunque las condiciones de continuidad y co-continuidad nos garantizan la corrección de una función recursiva, no se puede verificar la corrección de todas las funciones recursivas con este método. Sin embargo, esta cuestión está más allá del alcance del presente trabajo.

2. LA PRECONDICIÓN MÁS DÉBIL DE DIJKSTRA

Después de hablar de funciones y relaciones en el capítulo anterior, es hora de regresar a los lenguajes de programación. Ahora se abordará el concepto de precondición más débil (wp). Se tratará de describir la semántica de un programa, es decir, se trata de especificar exactamente qué tareas realiza un programa (o parte de éste) y en qué condiciones puede efectuar adecuadamente estas tareas.

Sin embargo, el poder de la wp va más allá: a partir de ella se puede definir un lenguaje de programación cuya semántica estará formalizada desde el principio.

2.1 Variables y estados

Aunque no se ha definido el significado formal de programa, por ahora se puede continuar con el significado intuitivo. Del mismo modo, podemos seguir usando el significado no formal de variable.

Supongamos que tenemos un programa en el que aparecen dos variables, x_1 y x_2, ambas de tipo entero. El programa realiza una tarea muy sencilla: asigna a x_1 el valor de $2x_2$. Sea $x_1 = 3$ y $x_2 = 2$ antes de la ejecución del programa. Después de la ejecución, se tiene que $x_1 = 4$ (no sabemos que pasa con x_2; un programa "sensato" no debería modificar su valor). Haciendo abstracción de cualesquier otros datos presentes en la memoria de la computadora en que se ejecutó el programa, podemos decir que el estado en que se encontraba antes de ejecutar el programa era ($x_1 = 3, x_2 = 2$) y, después, ($x_1 = 4, x_2 = ?$). Si suponemos que el orden en que se da el valor de las variables siempre es x_1, x_2, entonces, podemos decir que el estado *inicial* era (3, 2) y el *final* (4, ?). En términos más formales:

Definición. Sean $x_1, x_2, \ldots, x_n$ las variables del programa, y sean $a_1 \in A_1, a_2 \in A_2, \ldots, a_n \in A_n$ constantes (donde A_i es el conjunto de valores posibles de x_i). Si en determinado momento, $x_1 = a_1$, $x_2 = a_2, \ldots, x_n = a_n$, el *estado de las variables del programa* es $(a_1, a_2, \ldots, a_n)$. El *estado de la variable* x_i es el valor a_i. El producto cartesiano $A_1 \times A_2 \times \cdots A_n$ es el *espacio de estados* de las variables del programa. Un estado *inicial* es el estado inmediatamente anterior a la ejecución de un programa; un estado *final* es el inmediatamente posterior.

En el apéndice se define a los predicados como expresiones no cuantificadas del tipo $(x_1 < x_2 \wedge x_1^3 \geq x_3)$. Supongamos que en el estado actual $x_1 = 2$ y $x_2 = 3$, es decir, el estado es $(2, 3)$ y que P es el predicado anterior. Si se sustituyen x_1 y x_2 en P por estos valores, se tiene una proposición verdadera. En cambio, si el estado es $(1, 2)$ el resultado es una proposición falsa.

Definición. Sean $x_1, x_2, \ldots, x_n$ las variables de un programa, sea P un predicado que incluye sólo variables del programa (posiblemente no todas) y sea $s = (a_1, a_2, \ldots, a_n)$ un estado. Se dice que s *satisface* a P si y sólo si, al sustituir en P las variables por los valores correspondientes en s, se obtiene una proposición verdadera.

En adelante, los estados se representarán en forma completa (como n-adas ordenadas) o por medio de las variables s, s_0, etcétera. A veces, cuando se escoja la representación explícita, no se indicarán todos los valores para las variables, sino sólo los pertinentes o los que no hayan cambiado desde el estado anterior.

Definición. Sea S el conjunto de estados que satisfacen el predicado P. Se dirá entonces que P representa a S. El conjunto de estados representados por P se puede denotar por S_P.

Por ejemplo, si $P = (x < 0)$, $S = \{(x) \mid x \in \mathbb{N}^-\}$ es el conjunto de estados que satisfacen a P o, según la definición anterior, P representa a S.

Como ya se dijo en el capítulo 1, un predicado es una expresión booleana porque puede tener el valor de verdadero o falso (agreguemos ahora que en función del estado en que se evalúe el predicado). Sin embargo, hay predicados que no pueden ser satisfechos por ningún estado, *v.g.*, $P = (x < 0 \wedge x > 0)$. Estos predicados representan al conjunto $\emptyset$, pero también se puede convenir que representan al valor de verdad F y se escribirá $P = F$. Un predicado Q que es satisfecho por cualquier

estado es *válido* (por ejemplo, $x = x$) y representa al espacio de estados. También por convención, se dirá que Q representa a V y a veces se escribirá $Q = V$.

En el capítulo 1 ya se han empleado los predicados en el lugar de proposiciones para construir oraciones compuestas por medio de las conectivas lógicas. Una vez relacionados los predicados y los estados, se impone una nueva interpretación de este uso. Sean P y Q dos predicados:

1. $\neg Q$ es verdadero si Q es falso, y falso en caso contrario. Pero Q es verdadero si es satisfecho por alguno de los estados que representa. Entonces $\neg Q$ representa el complemento de los estados representados por Q, es decir $\neg Q = \bar{S}_Q$.

2. $P \wedge Q$ es verdadero si P y Q se satisfacen a la vez. Por esto, $P \wedge Q$ representa el conjunto de estados que satisfacen a P y Q a la vez: $S_P \cap S_Q$.

3. Como era de esperarse, $P \vee Q$ equivale a $S_P \cup S_Q$.

4. $P \Rightarrow Q$ quiere decir que si P es satisfecha, Q también lo es. En pocas palabras, $S_P \subseteq S_Q$.

5. $P \Leftrightarrow Q$ quiere decir que P se satisface si y sólo si Q se satisface, es decir, $S_P = S_Q$.

2.2 Precondiciones y postcondiciones

La tarea de definir qué debe hacer un programa o un segmento de éste se puede ver de la siguiente manera: dado un *estado* (es decir una asignación de valores a las variables del programa) anterior a la ejecución de un programa, el resultado es otro estado (es decir, una nueva asignación de valores a las variables).

Como ya se dijo, los conjuntos de estados están representados por predicados. Entonces, se puede *especificar* con mayor precisión la tarea que debe realizar un programa: sean S un programa y Q y R predicados. $\{Q\}S\{R\}$ significa que si la ejecución de S comienza en un estado que satisface Q entonces S terminará en tiempo finito en un estado que satisfará R.

En otras palabras, S realiza su tarea si, dada la *precondición* Q, después de terminar su ejecución en un tiempo finito, se cumple la *postcondición* R.

Pero, ¿qué pasa si S se ejecuta en un estado que *no* satisface Q? El resultado puede satisfacer o no a R. Si lo que nos interesa es que R se cumpla, sería preferible usar, en lugar de Q, un predicado más débil, es decir, un predicado Q' que represente un conjunto de estados que contenga como subconjunto al conjunto de estados representados por Q y que, al mismo tiempo, siga siendo cierta la afirmación $\{Q'\}S\{R\}$. De hecho, nos interesa el predicado más débil que cumpla la condición anterior como la mínima o más débil precondición para R.

Definición. El predicado que representa el conjunto de *todos* los estados iniciales tales que la ejecución del programa S terminará en un tiempo finito en un estado que satisface el predicado R, se llamará la *precondición más débil* para S y R y se denotará así: wp(S, R).

A partir de la wp se pueden definir los comandos de un lenguaje de programación. Pero, antes de hacerlo, se verán algunas propiedades de la *wp*.

2.3 Propiedades de la precondición

Teorema 2.1. Sean S un programa y Q y R predicados cualesquiera. Entonces:

$$\text{wp}(S, F) = F.$$

Esta propiedad nos dice que ningún estado inicial (recuérdese que el predicado F representa el conjunto vacío) permitirá que el programa S produzca un estado que satisfaga el predicado F (es decir, el predicado que es falso para todos los estados y que, por tanto, no puede satisfacerse) después de ejecutarse. Esta propiedad se conoce como *la ley de la exclusión de un milagro*[1] (pues sería un milagro que se satisficiera un predicado que por definición no puede satisfacerse). Pero antes hay que demostrar que 2.1 es verdadera:

Demostración. Reductio ad Absurdum. Supóngase que la propiedad 1 es falsa, es decir, que (puesto que F representa a $\emptyset$) existe un estado Q que satisface wp(S, F). Entonces, si se ejecuta S en el estado Q, entonces se llegará a un estado que satisfaga F. Pero esto es imposible, pues F no puede ser satisfecho por ningún estado. Por tanto, wp$(S, F) = F$ vale. ■

[1] *Law of excluded miracle*, en inglés, tal y como la llaman en los textos especializados.

Teorema 2.2. Supóngase que $Q \Rightarrow R$. Entonces,

$$\text{wp}(S, Q) \Rightarrow \text{wp}(S, R).$$

Esta propiedad se conoce como la *ley de la monotonía.*

Demostración. Si $Q \Rightarrow R$, entonces cualquier estado que satisfaga Q satisfará R. Entonces, si S se ejecuta en un estado que satisfaga wp(S, Q) el resultado, por definición, será un estado que satisfaga Q y, por tanto, R. Por tanto, 2.2 vale. ■

Considérese ahora la propiedad siguiente:

Teorema 2.3. Distribución de la conjunción:

$$(\text{wp}(S, Q) \wedge \text{wp}(S, R)) \equiv \text{wp}(S, Q \wedge R).$$

Demostración. Sea s un estado inicial que satisface wp(S, Q) y wp(S, R). Por tanto, si se ejecuta S en s, el resultado satisfará Q y R y entonces, por definición, s satisface wp$(S, Q \wedge R)$.

A la inversa, si s es un estado que satisface wp$(S, Q \wedge R)$, entonces si se ejecuta S en s el resultado satisfará $Q \wedge R$ por definición. Pero entonces este resultado satisface también Q y R. Por tanto, s satisface wp(S, Q) y wp(S, R). En conclusión, 2.3 es verdadera. ■

Existe una propiedad similar a 2.3 para el operador lógico $\vee$, llamada *distributividad de la disyunción*:[2]

Teorema 2.4. $(\text{wp}(S, Q) \vee \text{wp}(S, R)) \Rightarrow \text{wp}(S, Q \vee R)$.

Demostración. Sabemos que $Q \Rightarrow (Q \vee R)$ y $R \Rightarrow (Q \vee R)$. Entonces, por 2.2 se tiene que wp$(S, Q) \Rightarrow$ wp$(S, Q \vee R)$ y wp$(S, R) \Rightarrow$ wp$(S, Q \vee R)$. Por el teorema A.12, se puede concluir que $(\text{wp}(S, Q) \vee \text{wp}(S, R)) \Rightarrow \text{wp}(S, Q \vee R)$. ■

Las propiedades 2.1, 2.3 y 2.4 se conocen también como *condiciones de salud* de Dijkstra, pues su cumplimiento por parte de los comandos del lenguaje que se definirá en la siguiente sección garantiza que éstos se comporten adecuadamente.

[2] La propiedad correspondiente a $\vee$ es más débil que la de $\wedge$, pero puede hacerse igual de fuerte que la propiedad 2.3 si se consideran máquinas deterministas exclusivamente. Aquí no se abordará la diferencia entre una máquina determinista y una no determinista. Para este concepto (así como para una versión más fuerte de 2.4), pueden consultarse Gries [1981], cap. 2; Dijkstra [1976], cap. 3 y Backhouse [1986], cap. 3.

2.4 Un lenguaje de programación basado en la precondición más débil

Se construirá ahora un lenguaje de programación cuyas instrucciones estarán definidas a partir de la wp.

2.3.1 *Skip*

El primer enunciado que se definirá es el de *skip*.

Definición. Sea R una postcondición cualquiera. Entonces $\text{wp}(skip, R) = R$.

En términos intuitivos, el comando *skip* se puede ver como una instrucción nula: el programa no afecta las variables cuando se invoca este comando y, en consecuencia, deja inalterado el estado actual después de ejecutarse. Por esto mismo, el mayor conjunto de estados tal que todos sus elementos satisfacen R después de que se ejecuta un comando que no altera las variables es R mismo.

2.3.2 *Abort*

El siguiente enunciado definido es *abort*:

Definición. Sea R una postcondición. Entonces $\text{wp}(abort, R) = F$.

En este caso, después de ejecutar *abort* en algún estado que satisficiera F, deberíamos tener un estado final que satisficiera R. Pero ningún estado satisface F. Por tanto, *abort* es un signo de que ha ocurrido un error en el programa. Dijkstra [1976] señala que *abort* nos tiene que llevar a una situación en la que no se alcanza un estado final.

Con estos dos enunciados, sin embargo, no se puede hacer gran cosa. El siguiente comando es, en cambio, mucho más útil.

2.3.3 *Composición secuencial*

En un programa con frecuencia se tienen enunciados que se ejecutan uno inmediatamente después de otro (y el estado de las variables en que se ejecuta el segundo depende, desde luego, del resultado de la ejecución del primero). El objetivo del comando de composición secuencial es representar este caso:

Definición. Sean S_1 y S_2 comandos de nuestro lenguaje. Entonces, la composición secuencial de estos enunciados, simbolizada por ';', cumple lo siguiente: $wp("S_1 ; S_2", R) = wp(S_1, wp(S_2, R))$.

Según se desprende de la definición, ejecutar un comando después de otro equivale a ejecutar un comando en un estado inicial tal que el resultado es un estado inicial adecuado para que la ejecución del segundo comando satisfaga la postcondición deseada. Por su similitud con la composición de funciones, este comando también se conoce como composición funcional.

2.3.4 *Asignación*

El enunciado de asignación (representado por ":=") es mucho más rico que todos los anteriores, pues nos permite definir ya una alteración a los valores de las variables:

Definición. Sea x una variable, e una expresión y R una postcondición.[3] Entonces, $wp("x := e", R) = R^x_e$.

¿Esta definición corresponde a la noción intuitiva de asignación de un valor a una variable? Considérese el siguiente ejemplo:

Ejemplo. Sea x una variable a la que queremos asignar el valor '5', es decir, que después del enunciado de asignación se debe satisfacer el predicado '$x = 5$'. En este predicado, si sustituimos las apariciones libres de x por la expresión '5' obtenemos '$5 = 5$'. Entonces nuestra definición del comando de asignación nos indica que $wp("x := 5", x = 5) = (5 = 5)$. Pero $(5 = 5)$ es V. Entonces $wp("x := 5", x = 5) = V$. En resumen, después del enunciado de asignación será cierto que la variable x tiene el valor asignado si y sólo si ocurre la asignación.

[3] Gries [1981] da una definición ligeramente distinta: $wp("x := e", R) = R^x_e$ *cand domain(e)*, donde *cand* es una variante de la conjunción lógica que toma en consideración, además de los valores de verdad F y V, el valor indefinido U (indefinido). Por su parte, el predicado *domain(e)* describe el conjunto de todos los estados en los cuales se puede evaluar e. Esta definición es más rigurosa, aunque no es esencialmente distinta a la de Dijkstra, que es la que se presenta en esta tesis.

2.3.5 *Estructuras de control de flujo*

Un lenguaje de programación que no admite estructuras de control de flujo que permitan la ramificación del programa es extremadamente pobre. Por esta razón es necesario extender el lenguaje en esta dirección.

Se comenzará con una estructura condicional. La idea es tener enunciados de la siguiente forma:

if $B_1 \rightarrow S_1$
[] $B_2 \rightarrow S_2$
...
[] $B_n \rightarrow S_n$
fi

donde B_i es una expresión booleana y S_i un comando que se ejecutará si la expresión booleana es verdadera. El símbolo [] sirve para separar las distintas condiciones y comandos asociados a cada una de ellas (como el enunciado "else if" en C). Cada enunciado de la forma $B_i \rightarrow S_i$ se conoce como "comando custodiado" (*guarded command*) en el que B_i es el "custodio" o guardia para la ejecución de S_i. El significado del constructo **if**... **fi** es el siguiente: durante la ejecución del comando, se verifican las expresiones booleanas (no hay un orden implícito) y si alguna es verdadera, se ejecuta el comando asociado.

Nótese que hasta ahora no se ha impuesto ninguna restricción sobre las expresiones booleanas, pero es necesario aclarar que no debe ejecutarse más de un comando si hay más de una expresión verdadera. En el momento de definir formalmente el comando **if**... **fi** se pondrán algunas restricciones para evitar cualquier error. La primera de estas restricciones es que al menos *una* de las expresiones booleanas sea verdadera, como se desprende de la siguiente definición formal de **if**... **fi** (representado de ahora en adelante así: IF) en términos de la wp:

Definición. $\text{wp}(\text{IF}, R) = (B_1 \vee B_2 \vee \ldots \vee B_n) \wedge (B_1 \Rightarrow \text{wp}(S_1, R)) \wedge (B_2 \Rightarrow \text{wp}(S_2, R)) \wedge \ldots \wedge (B_n \Rightarrow \text{wp}(S_n, R))$.[4]

[4] Dijkstra [1976], p. 35, señala que esta definición de **if**... **fi** da libertad de elección sobre qué comando ejecutar cuando más de una condición es verdadera, pues *todas* las implicaciones $B_i \Rightarrow \text{wp}(S_i, R)$ deben ser verdaderas.

Si se observa con cuidado la definición anterior, se verá cuál es la segunda restricción en el comando IF: todos los enunciados $B_i \Rightarrow \text{wp}(S_i, R)$ deben ser verdaderos.

En programación es frecuente que no sólo se desee que un comando se ejecute una vez en caso de que se cumpla una condición (como con el comando IF), sino que una serie de comandos se ejecuten mientras se cumpla una condición o hasta que dicha condición sea verdadera. Los comandos de esta forma se conocen como ciclos (*e.g.*, 'loop', 'while... do...', 'repeat... until...', 'for', etcétera, en muchos lenguajes de alto nivel). Con la notación para comandos custodiados, se puede agregar fácilmente un nuevo constructo del lenguaje que estamos definiendo:

$$
\begin{array}{l}
do\ \ B_1 \rightarrow S_1 \\
\quad [\!]\ B_2 \rightarrow S_2 \\
\quad \ldots \\
\quad [\!]\ B_n \rightarrow S_n \\
od
\end{array}
$$

La interpretación del comando es la siguiente: mientras alguna de las expresiones booleanas B_i sea verdadera se ejecutará su comando asociado S_i.[5] El ciclo sólo se detendrá cuando *todas* las condiciones B_i sean falsas.

Este enunciado es más general que los *loops* de la mayoría de los lenguajes de programación, en los cuales hay una sola expresión booleana y una sola lista de comandos:

$$do\ B \Rightarrow S\ od.$$

Antes de escribir el enunciado DO (como se llamará de ahora en adelante al nuevo constructo) en términos de la wp, veamos el siguiente predicado, definido recursivamente:

$$H_0(R) = \neg(B_1 \vee B_2 \vee \ldots \vee B_n) \wedge R$$
$$H_k(R) = H_0(R) \vee \text{wp}(IF, H_{k-1}(R))$$

$H_k(R)$ es el conjunto de estados tales que el constructo DO terminará después de, cuando mucho, k iteraciones en un estado final que satisface la postcondición R. Es claro que si $k = 0$, todas las condiciones son falsas y no se ejecuta ningún comando del ciclo.

[5] Nuevamente, se está ante un mecanismo no determinista, pues se otorga libertad a la "máquina" para ejecutar cualquiera (aunque sólo uno) de los comandos cuya condición sea verdadera.

Ahora ya se puede definir DO en términos de la wp:

Definición. $\text{wp}(\text{DO}, R) = \exists k((0 \leq k) \wedge H_k(R))$.

Dijkstra [1976] llama *constructo alternativo* al comando IF y *constructo iterativo* a DO y presenta estos dos teoremas:

Teorema 2.5. (Teorema básico para el comando alternativo.). Sean Q y R dos predicados tales que

$$Q \Rightarrow (B_1 \vee B_2 \vee \ldots \vee B_n)$$

y

$$\forall i((1 \leq i \leq n) \wedge ((Q \wedge B_i) \Rightarrow \text{wp}(S_i, R)))$$

son verdaderas en todos los estados. Entonces, y sólo entonces,

$$Q \Rightarrow \text{wp}(\text{IF}, R).$$

Demostración. Se tiene que $\forall i((Q \wedge B_i) \Rightarrow \text{wp}(S_i, R))$ es equivalente a

$$\forall i(\neg(Q \wedge B_i) \vee \text{wp}(S_i, R)),$$

por la regla de equivalencia entre la implicación y la disyunción. A su vez, esta fórmula equivale a

$$\forall i(\neg Q \vee \neg B_i \vee \text{wp}(S_i, R)),$$

por las leyes de Morgan. Como puede observarse, Q no depende de la variable cuantificada i y, entonces, se puede sacar de su alcance:

$$\neg Q \vee \forall i(\neg B_i \vee \text{wp}(S_i, R)).$$

Una vez más, por la equivalencia entre condicional y disyunción,

$$Q \Rightarrow \forall i(B_i \Rightarrow \text{wp}(S_i, R)).$$

De esta forma, las hipótesis del teorema quedan así:

$$(Q \Rightarrow (B_1 \vee B_2 \vee \ldots \vee B_n)) \wedge (Q \Rightarrow \forall i(B_i \Rightarrow \text{wp}(S_i, R)),$$

lo cual es equivalente a

$$Q \Rightarrow ((B_1 \vee B_2 \vee \ldots \vee B_n) \wedge \forall i(B_i \Rightarrow \text{wp}(S_i, R))),$$

y, por la definición de IF,

$$Q \Rightarrow \text{wp}(\text{IF}, R). \qquad \blacksquare$$

Este teorema es importante porque con frecuencia es muy difícil encontrar la precondición más débil para un comando alternativo, pero es mucho más fácil encontrar un predicado que implique la precondición buscada si se satisfacen las dos condiciones del teorema. Además, este teorema sirve como lema para el siguiente.

Teorema 2.6. (Teorema básico para el comando iterativo.)[6] Sea P un predicado que satisface las siguientes condiciones (con $1 \leq i \leq n$):

$$\forall i((P \wedge B_i) \Rightarrow \text{wp}(S_i, P)) \tag{1}$$

$$(P \wedge (B_1 \vee B_2 \vee \ldots \vee B_n)) \Rightarrow (t > 0) \tag{2}$$

$$\forall i((P \wedge B_i) \Rightarrow \text{wp}(\text{"}t_1 := t\,;\, S_i\text{"}, t < t_1)), \tag{3}$$

donde t es una función entera y t_1 una variable del mismo tipo que no aparece en P, B_i o S_i alguna. Entonces

$$P \Rightarrow \text{wp}(\text{DO}, P \wedge \neg(B_1 \vee B_2 \vee \ldots \vee B_n)).$$

Demostración. En el teorema anterior se demostró que (1) es equivalente a

$$P \Rightarrow \forall i(B_i \Rightarrow \text{wp}(S_i, P)). \tag{4}$$

Supóngase ahora que $P \wedge (B_1 \vee B_2 \vee \ldots \vee B_n)$ vale. Entonces, P y $(B_1 \vee B_2 \vee \ldots \vee B_n)$ valen por separado (teorema A.16). P y (4) implican que

$$\forall i(B_i \Rightarrow \text{wp}(S_i, P)).$$

Por A.15, se tiene que

$$(B_1 \vee B_2 \vee \ldots \vee B_n) \wedge \forall i(B_i \Rightarrow \text{wp}(S_i, P)),$$

[6] En realidad, esta versión del teorema, dada por Gries [1981], es una combinación de dos teoremas de Dijkstra, uno de los cuales sí es el básico para el comando iterativo. Sin embargo, esta versión es más útil para la definición de *invariantes*, que se usan mucho en la verificación de programas. La demostración se tomó, con variaciones, de Gries [1981].

es decir, por la definición del comando IF

$$wp(\text{IF}, P).$$

En conclusión

$$(P \wedge (B_1 \vee B_2 \vee \ldots \vee B_n)) \Rightarrow \text{wp}(\text{IF}, P). \tag{5}$$

Por otra parte, (3) implica, por el teorema A.8,

$$P \Rightarrow \forall i(B_i \Rightarrow \text{wp}(\text{“}t_1 := t\,;\, S_i\text{”}, t < t_1)),$$

dado que ninguna variable de P está ligada por el cuantificador $\forall$. De forma análoga que en (5), se concluye que

$$(P \wedge (B_1 \vee B_2 \vee \ldots \vee B_n)) \Rightarrow \text{wp}(\text{“}t_1 := t\,;\, \text{IF”}, t < t_1),$$

lo cual equivale, por la definición de composición secuencial,

$$(P \wedge (B_1 \vee B_2 \vee \ldots \vee B_n)) \Rightarrow \text{wp}(\text{“}t_1 := t\text{”}\,;\, \text{wp}(\text{IF}, t < t_1)).$$

Pero IF no contiene la variable t_1, por lo que se puede aplicar la definición de asignación sin problema:

$$(P \wedge (B_1 \vee B_2 \vee \ldots \vee B_n)) \Rightarrow \text{wp}(\text{IF}, t < t_1)^{t}_{t_1}. \tag{6}$$

Por otra parte, dado que t y t_1 son enteros, (6) es igual a

$$(P \wedge (B_1 \vee B_2 \vee \ldots \vee B_n)) \Rightarrow \text{wp}(\text{IF}, t \leq t_1 - 1)^{t}_{t_1}. \tag{7}$$

Sea t_0 una nueva variable que no haya aparecido hasta ahora. Por el teorema A.3, (7) equivale a

$$(P \wedge (B_1 \vee B_2 \vee \ldots \vee B_n) \wedge (t \leq t_0 + 1)) \Rightarrow (\text{wp}(\text{IF}, t \leq t_1 - 1)^{t}_{t_1} \wedge (t \leq t_0 + 1)). \tag{8}$$

Nótese que t_1 no aparece en $(t \leq t_0 + 1)$, por lo que la expresión $(t \leq t_0 + 1)^{t}_{t_1}$ es satisfecha por el mismo conjunto de estados que $(t \leq t_0 + 1)$. Si se sustituye esta última por $(t \leq t_0 + 1)^{t}_{t_1}$ en (8) se tiene

$$(P \wedge (B_1 \vee B_2 \vee \ldots \vee B_n) \wedge (t \leq t_0 + 1)) \Rightarrow (\text{wp}(\text{IF}, t \leq t_1 - 1)^{t}_{t_1} \wedge (t \leq t_0 + 1)^{t}_{t_1}),$$

lo que implica, por distributividad de la sustitución textual (teorema A.32)

$$(P \wedge (B_1 \vee B_2 \vee \ldots \vee B_n) \wedge (t \leq t_0 + 1)) \Rightarrow (\text{wp}(\text{IF}, t \leq t_1 - 1) \wedge (t \leq t_0 + 1))_t^{t_1}. \quad (9)$$

Pero IF no contiene a t_1 ni a t_0 y, entonces, (9) nos da

$$(P \wedge (B_1 \vee B_2 \vee \ldots \vee B_n) \wedge (t \leq t_0 + 1)) \Rightarrow (\text{wp}(\text{IF}, t \leq t_1 - 1) \wedge \text{wp}(\text{IF}, t_1 \leq t_0 + 1))_t^{t_1}, \quad (10)$$

y (10) lleva a

$$(P \wedge (B_1 \vee B_2 \vee \ldots \vee B_n) \wedge (t \leq t_0 + 1)) \Rightarrow \text{wp}(\text{IF}, (t \leq t_1 - 1) \wedge (t_1 \leq t_0 + 1))_t^{t_1}, \quad (11)$$

por la distributividad de la conjunción (2.3) de la wp. Por otra parte, $(t \leq t_1 - 1) \wedge (t_1 \leq t_0 + 1)$ implica que $t \leq t_0$ y, entonces, (11) da

$$(P \wedge (B_1 \vee B_2 \vee \ldots \vee B_n) \wedge (t \leq t_0 + 1)) \Rightarrow \text{wp}(\text{IF}, t \leq t_0)_t^{t_1}.$$

Por definición de la asignación, de lo anterior resulta

$$(P \wedge (B_1 \vee B_2 \vee \ldots \vee B_n) \wedge (t \leq t_0 + 1)) \Rightarrow \text{wp}("t_1 := t, \text{IF}", t \leq t_0).$$

Pero t_1 no aparece en IF ni en la postcondición $t \leq t_0$, por lo que la asignación $t_1 := t$ es irrelevante. En consecuencia, la proposición anterior es equivalente a

$$(P \wedge (B_1 \vee B_2 \vee \ldots \vee B_n) \wedge (t \leq t_0 + 1)) \Rightarrow \text{wp}(\text{IF}, t \leq t_0). \quad (12)$$

Ahora, dada la condición (2), se demostrará por inducción que lo siguiente es cierto,

$$(P \wedge t \leq k) \Rightarrow H_k(P \wedge \neg(B_1 \vee B_2 \vee \ldots \vee B_n)). \quad (13)$$

Base: $k = 0$. Hemos supuesto que $P \wedge (B_1 \vee B_2 \vee \ldots \vee B_n) \Rightarrow t > 0$. Por A.8, (2) es equivalente a

$$P \Rightarrow ((B_1 \vee B_2 \vee \ldots \vee B_n) \Rightarrow (t > 0)).$$

Por A.13, se tiene ahora

$$P \Rightarrow (\neg(t > 0) \Rightarrow \neg(B_1 \vee B_2 \vee \ldots \vee B_n)).$$

Nuevamente, por A.8, se llega a

$$(P \wedge \neg(t > 0)) \Rightarrow \neg(B_1 \vee B_2 \vee \ldots \vee B_n).$$

Por el teorema A.9, lo anterior es equivalente a

$$(P \wedge t \leq 0) \Rightarrow (P \wedge \neg(B_1 \vee B_2 \vee \ldots \vee B_n)).$$

Si ahora aplicamos la definición de H_0, tenemos

$$(P \wedge t \leq 0) \Rightarrow H_0(P). \tag{14}$$

Paso inductivo. Sea cierto (13) para $k = m$ y se demostrará que vale para $k = m + 1$. Si sustituimos en (12) t_0 por m (lo cual es posible, puesto que t_0 es libre en dicha proposición) nos queda

$$(P \wedge (B_1 \vee B_2 \vee \ldots \vee B_n) \wedge (t \leq m + 1)) \Rightarrow \text{wp}(\text{IF}, P \wedge t \leq m),$$

y, por hipótesis de inducción,

$$(P \wedge (B_1 \vee B_2 \vee \ldots \vee B_n) \wedge (t \leq m+1)) \Rightarrow \text{wp}(\text{IF}, H_m(P \wedge \neg(B_1 \vee B_2 \vee \ldots \vee B_n)). \tag{15}$$

Por otra parte, al aplicar a

$$P \wedge \neg(B_1 \vee B_2 \vee \ldots \vee B_n) \wedge t \leq m + 1,$$

la simplificación nos queda

$$P \wedge \neg(B_1 \vee B_2 \vee \ldots \vee B_n),$$

que, junto con (14), por *modus ponens* nos permite llegar a

$$H_0(P \wedge \neg(B_1 \vee B_2 \vee \ldots \vee B_n)). \tag{16}$$

De (15) y (16) obtenemos, por adición,

$$P \wedge t \leq m+1 \Rightarrow H_0(P \wedge \neg(B_1 \vee B_2 \vee \ldots \vee B_n)) \vee \text{wp}(\text{IF}, P \wedge \neg(B_1 \vee B_2 \vee \ldots \vee B_n)),$$

lo cual equivale a

$$P \wedge t \leq m + 1 \Rightarrow H_{m+1}(P \wedge \neg(B_1 \vee B_2 \vee \ldots \vee B_n)),$$

lo cual completa la prueba inductiva de (13).

La interpretación del lema (13) es que, si P es cierto y existe una k tal que la función $t \leq k$ (lo cual siempre es cierto, pues t toma valores enteros), entonces el

ciclo definido por DO terminará, cuando mucho, en k pasos, con el resultado de que P sigue siendo verdadero, pero todos los guardas de los comandos custodiados de la iteración son falsos. Es decir,

$$P \Rightarrow \exists k H_k(P \wedge \neg(B_1 \vee B_2 \vee \ldots \vee B_n)). \tag{17}$$

El consecuente de este condicional es, como se recordará, la definición del comando iterativo y, entonces, (17) equivale a

$$P \Rightarrow \text{wp}(\text{DO}, P \wedge \neg(B_1 \vee B_2 \vee \ldots \vee B_n)). \quad \blacksquare$$

A primera vista, resulta difícil ver la utilidad de un teorema tan abstruso. Sin embargo, éste es uno de los resultados fundamentales para la construcción y verificación de programas. Considérese primero el predicado P, al que se llamará de ahora en adelante *invariante del ciclo*. Este predicado se satisface antes de entrar al cuerpo del ciclo, después de ejecutar cualquiera de los comandos contenidos en el DO y al final de la ejecución del ciclo. El ciclo, por su parte, terminará cuando ya no sea cierto ninguno de los guardas B_i del los comandos custodiados del ciclo. Además, la existencia de la función t nos garantiza que, mientras valga algún guarda B_i, el ciclo continuará ejecutándose, pero, como t es entera y decreciente monótona, el ciclo terminará en, cuando mucho, k pasos para alguna k, pues $t \leq 0$ sólo si todos los guardas B_i son falsos.

Si regresamos a la notación de precondiciones y postcondiciones anterior, a saber, $\{Q\}S\{R\}$ (donde S es un ciclo), para demostrar que un ciclo cumple con una tarea dada sólo debemos verificar estos hechos: a) que P es cierta antes de la ejecución del ciclo (es decir, que $Q \Rightarrow P$); b) que, para todo i, vale que $\{P \wedge B_i\}S_i\{P\}$; c) que $P \wedge \neg(B_1 \vee B_2 \vee \ldots \vee B_n) \Rightarrow R$; d) que existe la función t tal que $P \wedge (B_1 \vee B_2 \vee \ldots \vee B_n) \Rightarrow (t > 0)$, es decir, que t tiene una cota mínima inferior mientras sea cierto algún guarda B_i; e) que $\forall i\{P \wedge B_i\}t_1 := t\,; S_i\{t < t_1\}$, es decir, que la función es monótona decreciente durante la ejecución del ciclo. En pocas palabras, hay que encontrar una condición invariante para un ciclo y una cota superior para el número de ejecuciones para demostrar que el ciclo termina y que se cumple la postcondición al final si la invariante y la negación de los guardas implican la postcondición.

2.3.6 *Un pequeño ejemplo*

Con las herramientas anteriores, se puede abordar el primer ejemplo de un programa *real*.[7]

Se trata de un problema muy simple: hallar el máximo de dos números dados. Sean x y y los dos números en cuestión y sea z una variable que contendrá el valor del mayor de ambos. Sean Q y R la precondición y la postcondición que queremos satisfacer y sea S el programa que lo hará. El problema se puede denotar entonces $\{Q\}S\{R\}$. Pero, ¿cuál es el significado concreto de Q, R y S?

Es fácil definir R como la asignación final a z del valor máximo entre x y y: R es equivalente a $z \geq x \wedge z \geq y \wedge (z = x \vee z = y)$. Por su parte Q debe ser un predicado satisfecho por cualquier valor inicial de x, y y z, pues los valores de las dos primeras variables son enteramente arbitrarios, mientras que a z se le asignará el valor deseado por medio de S, es decir, no importa su valor inicial. De este modo, se puede definir Q por medio del predicado universal V. Entonces, el programa y sus pre- y postcondiciones se pueden expresar ahora así:

$$\{V\}S\{z \geq x \wedge z \geq y \wedge (z = x \vee z = y)\}. \tag{18}$$

Aún queda pendiente la parte más importante, a saber, ¿qué comando o serie de comandos S establece la verdad de (18)?

Intuitivamente, queremos que se asigne a z el valor de x o de y, según cuál de estas variables tiene un valor mayor. Esto nos lleva a un comando alternativo:

if $x \geq y \rightarrow z := x$
[] $y \geq x \rightarrow z := y$
fi

El teorema básico sobre el comando alternativo imponía que, para que la precondición implicara la wp del comando alternativo aquélla debía implicar la disyunción de las guardias de los comandos custodiados del IF. Pero es claro que $y \geq x \vee x \geq y$ es cierto en todos los estados y, por tanto, cualquier predicado la implica (en este caso, el predicado es V). Entonces, por el teorema básico para el comando alternativo, tenemos que es cierto que

$$V \Rightarrow \text{wp}(\text{IF}, z \geq x \wedge z \geq y \wedge (z = x \vee z = y))$$

[7] Este ejemplo está tomado de Gries [1981].

En pocas palabras, es verdadera la siguiente expresión

$$\{V\}\ \textbf{if}\ y \geq x \rightarrow z := y \,[]\, x \geq y \rightarrow z := x\ \textbf{fi}\{z \geq x \wedge z \geq y \wedge (z = x \vee z = y)\},$$

por lo que el programa hace lo que se espera de él.[8]

[8] Las guardias de este comando no son excluyentes, pues si $y = x$ se cumple tanto $y \geq x$ como $x \geq y$. Nuevamente, se trata de un caso no determinista y dependerá de la instrumentación concreta del comando alternativo cuál de las asignaciones sea la que se realice.

3. LA PREESPECIFICIÓN MÁS DÉBIL

3.1 Estados iniciales/estados finales

La definición de las instrucciones de un lenguaje de programación a partir de la wp ha permitido especificar con claridad a partir de qué estados se puede alcanzar cierto resultado cuando se aplica una instrucción dada. Sin embargo, existen otros métodos para definir la semántica de un lenguaje en los que hay más transparencia con respecto a la relación entre el estado inicial y final de las variables, pues es precisamente esta relación lo que constituye la definición de un programa. En la precondición más débil, esta relación está inmersa en la relación entre precondiciones y postcondiciones del programa y conviene hacerla explícita pues, entre otras ventajas, si se recurre a las relaciones entre estados, será más fácil utilizar las herramientas del cálculo de relaciones, que, como se vio en el capítulo 1, son de gran poder y universalidad (*v.g.*, la inclusión de la recursión general es muy fácil). De hecho, a veces este punto de vista parece más natural, pues en las descripciones informales de los lenguajes de programación imperativos se explica el efecto de una instrucción por medio de los cambios que produce en las variables del programa, lo que establece una relación entre su estado antes y después de la instrucción. La *preespecificación más débil* es una forma de definir estas ideas intuitivas.

Sea s_0 un estado de las variables de un programa dado. Sea P un conjunto de instrucciones (puede ser una instrucción simple, una serie de instrucciones ejecutadas secuencialmente o un procedimiento, por ejemplo) y sea s_1 el estado de las variables del programa después de que se ha ejecutado P en el estado s_0. Considérese ahora el conjunto de pares $R = \{(s_0, s_1) \mid s_0$ es un estado inicial y s_1 es un estado final después de ejecutar $P\}$, y se tendrá una relación con dominio y contradominio en el conjunto de estados posibles para las variables de un programa.

Las relaciones como R pueden servir para definir tanto relaciones entre estados (que de ahora en adelante se llamarán *especificaciones*) como *programas*, los cuales establecen determinados valores para las variables después de ejecutar ciertas instrucciones en ciertos estados iniciales. Dicho de manera más informal: las especificaciones expresan lo que se espera de un programa, un programa es un conjunto de instrucciones concretas que hace realidad esas esperanzas. Hoare [1986] señala que, de este modo, la relación de inclusión entre las relaciones R y S, por ejemplo, se puede interpretar de tres formas:

1) Sea R un programa y S una especificación. $R \subseteq S$ significa que el programa R *cumple o satisface* la especificación S, es decir, que los cambios en los valores de las variables realizados por R satisfacen la relación (especificación) S.

2) Sean R y S dos programas. $R \subseteq S$ significa que R es más determinista que S, o sea, que cualquier cosa que haga R también la hace S (pues los cambios de estado realizados por R son un subconjunto de los realizados por S), pero S puede realizar otras cosas también. De acuerdo con la interpretación 1), además, se tiene que cualquier especificación satisfecha por S también es satisfecha por R.

3) Sean R y S dos especificaciones. $R \subseteq S$ significa que la especificación S es más *débil* o *general* que R, es decir, que cualquier programa que satisfaga R también satisface S.

Por otra parte, si los programas se definen por medio de relaciones, la ejecución secuencial de dos programas es la composición de las relaciones que los definen. Por ejemplo, P; Q quiere decir que primero se ejecuta el programa P y después el programa Q.

Ahora se puede definir la preespecificación más débil combinando las tres interpretaciones de la relación de inclusión.

3.2 La preespecificación más débil: definición

En la sección 2.3.6 se presentó un programa cuyo objetivo era encontrar el máximo de dos números dados. Visto como una relación, el programa debe satisfacer la siguiente especificación R:

$$R = \{((x,y,z),(x',y',z')) \mid x = x' ; \quad y = y' \quad y \quad z' = \max(x,y)\}.$$

El programa puede dividirse en dos pasos, el segundo de los cuales consiste en asignar a z el valor del máximo entre x y y. El primero, como es obvio, es encontrar el máximo de x y y. Sea Q la especificación del segundo paso. Hay muchas especificaciones posibles para el primer paso (y en consecuencia, muchos programas que las cumplen), y la satisfacción de cualquiera de estas especificaciones es una condición *suficiente* para que la ejecución sucesiva de los pasos 1 y 2 cumpla R. Si se toma la especificación más general (más débil) del paso 1, se tiene que ésta es una condición *necesaria* para que la ejecución sucesiva de 1 y 2 satisfaga R, pues cualquier otra especificación del paso 1 es un subconjunto de la más general. Se simbolizará la especificación más general de 1 de la siguiente forma: $Q \setminus R$.

Si ahora se generaliza la situación anterior a cualesquiera especificaciones R y Q, $Q \setminus R$ es la especificación de un programa P tal que, al ejecutarse P ; Q, se satisface la especificación R. En términos de la relación de inclusión, se quiere que P cumpla lo siguiente:

$$P \subseteq (Q \setminus R). \tag{1}$$

y que

$$(P ; Q) \subseteq R. \tag{2}$$

Además, se desea que $Q \setminus R$ sea la especificación más general que permita lo anterior o, en otras palabras, la especificación que sea una condición necesaria y suficiente para que se satisfagan (1) y (2), es decir:

$$P \subseteq (Q \setminus R) \Leftrightarrow (P ; Q) \subseteq R. \tag{3}$$

Hoare [1986] llama a (3) la *ley básica de la preespecificación*. ¿Cómo se debe definir la relación $Q \setminus R$ para que cumpla la ley básica? He aquí una propuesta:

Definición. Sean s_0, s_f y s estados de las variables de un programa, y Q y R especificaciones de programas. Entonces:

$$(Q \setminus R) = \{(s_0, s) \mid \forall s_f \ ((s, s_f) \in Q \Rightarrow (s_0, s_f) \in R)\}.$$

Para parafrasear la definición anterior, la ejecución de un programa que satisfaga $Q \setminus R$ en un estado inicial s_0 en el dominio de R resultará en un estado en el dominio de Q que garantiza que Q dará un estado final s_f tal que $(s_0, s_f) \in R$.

Sin embargo, en términos intuitivos lo que se pretende con la preespecificación más débil es algo más simple: $Q \setminus R$ es lo mínimo que se debió haber hecho antes de Q para que la ejecución de Q satisfaga R.

La definición cumple con la ley básica, como lo demuestra el siguiente teorema.

Teorema 3.1. Ley básica de la preespecificación. La definición de la preespecificación más débil cumple que $P \subseteq (Q \setminus R) \Leftrightarrow P ; Q \subseteq R$.

Más aún, $Q \setminus R$ es la única relación que la cumple.

Demostración. Primero se demostrará que la definición de la preespecificación cumple la ley básica.

1) Supongamos que $P \subseteq Q \setminus R$. Ahora se mostrará que $P ; Q \subseteq R$.

Sea $(s_0, s_f) \in P ; Q$. Entonces, por la definición de composición,

$$\exists s((s_0, s) \in P \wedge (s, s_f) \in Q).$$

Por otra parte, como $P \subseteq Q \setminus R$, se tiene que $(s_0, s) \in Q \setminus R$ y, entonces,

$$\forall s_f((s, s_f) \in Q \Rightarrow (s_0, s_f) \in R).$$

En este caso, ya se sabe que $(s, s_f) \in Q$ y, por *modus ponens*, $(s_0, s_f) \in R$ y

$$\therefore \quad P ; Q \subseteq R \tag{4}$$

2) Supongamos ahora que $P ; Q \subseteq R$. Por demostrar que $P \subseteq Q \setminus R$.

Sea $(s_0, s_f) \in P ; Q$. Entonces $(s_0, s_f) \in R$. En términos del cálculo de predicados, lo anterior queda así:

$$\forall s_0 \forall s_f((s_0, s_f) \in P ; Q \Rightarrow (s_0, s_f) \in R)$$

Pero $(s_0, s_f) \in P ; Q$ si y sólo si

$$\exists s((s_0, s) \in P \wedge (s, s_f) \in Q),$$

y, por tanto,

$$\forall s_0 \forall s_f(\exists s((s_0, s) \in P \wedge (s, s_f) \in Q) \Rightarrow (s_0, s_f) \in R).$$

Si se aplican sucesivamente a la fórmula anterior los teoremas A.7, A.22, nuevamente A.7 y A.25 se obtendrán las siguientes equivalencias

$$\Leftrightarrow \forall s_0 \forall s_f(\neg \exists s((s_0, s) \in P \wedge (s, s_f) \in Q) \vee (s_0, s_f) \in R)$$
$$\Leftrightarrow \forall s_0 \forall s_f \forall s(\neg((s_0, s) \in P \wedge (s, s_f) \in Q) \vee (s_0, s_f) \in R)$$
$$\Leftrightarrow \forall s_0 \forall s_f \forall s(((s_0, s) \in P \wedge (s, s_f) \in Q) \Rightarrow (s_0, s_f) \in R).$$

Ahora se aplica el teorema A.8 y se obtiene

$$\forall s_0 \forall s_f \forall s ((s_0, s) \in P \Rightarrow ((s, s_f) \in Q \Rightarrow (s_0, s_f) \in R)).$$

Sea $(s_0, s) \in P$, entonces, por *modus ponens*,

$$\forall s_0 \forall s_f \forall s ((s, s_f) \in Q \Rightarrow (s_0, s_f) \in R),$$

lo cual quiere decir que $(s_0, s) \in P \Rightarrow (s_0, s) \in Q \setminus R$, y, por tanto,

$$P \subseteq Q \setminus R. \tag{5}$$

De (4) y (5) se deduce la ley básica.

3) Por último, se quiere demostrar que si X es tal que

$$P \subseteq X \Leftrightarrow P ; Q \subseteq R. \tag{6}$$

entonces $X = Q \setminus R$.

La proposición (6) implica que cualquier subconjunto de X es tal que la relación resultante de la composición con Q es un subconjunto de R. Pero $X \subseteq X$, por lo que (6) implica

$$X ; Q \subseteq R.$$

Por la ley básica se deduce que

$$X ; Q \subseteq R \Rightarrow X \subseteq Q \setminus R,$$

y por *modus ponens* se tiene que

$$X \subseteq Q \setminus R.$$

Además, (6) también implica que cualquier relación que al componerse con Q sea un subconjunto de R es también un subconjunto de X. Pero, por definición $Q \setminus R$ cumple esta condición. Por tanto

$$Q \setminus R \subseteq X.$$

Finalmente, concluimos que $X = Q \setminus R$. ■

Se tiene ahora el siguiente teorema:

Teorema 3.2. $Q \setminus R = \cup\{X \mid X ; Q \subseteq R\}$

Demostración. Es obvio que $(Q \setminus R) \subseteq (Q \setminus R)$ y, por la ley básica,

$$(Q \setminus R) ; Q \subseteq R.$$

Por tanto,

$$Q \setminus R \subseteq \cup\{X | X ; Q \subseteq R\}.$$

La inversa es aún más clara, pues

$$\cup\{X | X ; Q \subseteq R\} \subseteq \cup\{X | X ; Q \subseteq R\},$$

y, en consecuencia,

$$(\cup\{X | X ; Q \subseteq R\} ; Q) \subseteq R.$$

Por la ley básica, $\cup\{X | X ; Q \subseteq R\} \subseteq Q \setminus R$. ■

Ahora se verán algunas propiedades del nuevo concepto.

3.3 Propiedades de la preespecificación más débil

Del mismo modo que con la precondición más débil, se puede definir la semántica de un lenguaje (y el lenguaje mismo) en términos de la preespecificación. No obstante, antes de definir los constructos del lenguaje conviene demostrar algunas propiedades de la preespecificación y empezaremos por la siguiente observación:

Enunciado 3.3. $Q \setminus R$ cumple la siguiente ecuación: $X ; Q \subseteq R$.

Demostración. Como es obvio, $Q \setminus R \subseteq Q \setminus R$. Entonces, por la ley básica:

$$(Q \setminus R) ; Q \subseteq R,$$

con lo que se cumple la ecuación. ■

Explicado sin símbolos, el teorema dice que la preespecificación más débil es la especificación de un programa tal que, al ejecutarse, da como resultado un estado de las variables apropiado para que Q cumpla con su parte, a saber, dar un resultado que satisfaga R.

Teorema 3.4. $U \setminus R \subseteq R$.

Demostración. Por el teorema 3.3 se sabe que

$$(U \setminus R) ; U \subseteq R$$

y, por el teorema 1.14,

$$U \setminus R \subseteq (U \setminus R) ; U \subseteq R,$$

que es el resultado deseado. ∎

En otras palabras, si se desea cumplir la especificación R finalizando con un programa que da cualquier resultado posible (a saber, U), se debió aplicar antes un programa que ya satisfacía R de antemano. ¿Cuál es la naturaleza del programa $U \setminus R$? Por definición, $U \setminus R = \{(s_0, s) | \forall s_f (s, s_f) \in U \Rightarrow (s_0, s_f) \in R\}$. Pero $(s, s_f) \in U$ siempre y, entonces, $U \setminus R$ es el conjunto $\{(x, y) | R(\{x\}) = U\}$.

Teorema 3.5. $(U \setminus R) ; U = U \setminus R$.

Demostración. De nuevo, por el teorema 3.3,

$$(U \setminus R) ; U \subseteq R.$$

Pero $U ; U = U$ y, sustituyendo

$$(U \setminus R) ; (U ; U) \subseteq R,$$

lo cual nos da, por el teorema 1.9 y la ley básica,

$$(U \setminus R) ; U \subseteq (U \setminus R).$$

Por otro lado, por el teorema 1.14,

$$(U \setminus R) \subseteq (U \setminus R) ; U.$$ ∎

En este caso, se tiene una versión más fuerte del teorema 1.14: $P \subseteq P ; U$, pues $U \setminus R = \{(x, y) | R(\{x\}) = U\}$ y esto nos garantiza la igualdad y no sólo la contención.

Teorema 3.6. $P \subseteq Q \setminus (P ; Q)$.

Demostración. Es obvio que $P;Q \subseteq P;Q$ y, por la ley básica, $P \subseteq Q \setminus (P;Q)$. ■

Esto significa que, si se desea cumplir con la especificación combinada de P seguido de Q, P ya cumple con dicha preespecificación.

Teorema 3.7. $I \subseteq Q \setminus R \Leftrightarrow Q \subseteq R$.

Demostración. Ya se sabe que $I \subseteq Q \setminus R \Leftrightarrow I;Q \subseteq R$. Además, $I;Q = Q$ y, por tanto,

$$I \subseteq Q \setminus R \Leftrightarrow Q \subseteq R.$$ ■

En otras palabras, si antes de hacer Q para cumplir R se puede "no hacer nada" (o sea, I), entonces Q ya cumple R.

Teorema 3.8. $I \subseteq Q \setminus Q$.

Demostración. Como $I;Q \subseteq Q$, por ley básica se tiene $I \subseteq Q \setminus Q$. ■

Una paráfrasis del teorema anterior es la siguiente: "no se necesita hacer nada para que al ejecutar Q se cumpla Q".

Por otra parte, es interesante preguntarse cuál es el conjunto $(Q \setminus Q) - I$, es decir, qué elementos de $Q \setminus Q$ impiden que el teorema anterior sea una igualdad. Por definición, $Q \setminus Q = \{(s_0,s) | (s,s_f) \in Q \Rightarrow (s_0,s_f) \in Q\}$. Entonces, $Q \setminus Q = \{(x,y) | Q(\{y\}) \subseteq Q(\{x\})\}$. Es obvio que si $x = y$, $(x,y) \in Q \setminus Q$. Pero $(Q \setminus Q) - I$ son los pares $(x,y) \in I$ tales que la imagen de y es un subconjunto de la de x. Este conjunto es vacío si Q es una función.

Teorema 3.9. $P = (I \setminus P)$.

Demostración. Es evidente que $P;I \subseteq P$. Por la ley básica, se deduce inmediatamente $P \subseteq (I \setminus P)$.

Por otra parte, $(I \setminus P) \subseteq (I \setminus P)$ y, por tanto,

$$(I \setminus P);I \subseteq P,$$

de lo cual se deduce directamente la otra contención y el teorema. ■

Dicho en un lenguaje más llano, lo mínimo que se debe hacer antes de "no hacer nada" para que se cumpla P es P mismo.

Teorema 3.10. $Q \setminus (R \cap S) = (Q \setminus R) \cap (Q \setminus S)$.

Demostración. Claramente, $(Q \setminus R) \cap (Q \setminus S) \subseteq (Q \setminus R)$. Por la ley básica

$$((Q \setminus R) \cap (Q \setminus S)) ; Q \subseteq R.$$

También es cierto que $(Q \setminus R) \cap (Q \setminus S) \subseteq (Q \setminus S)$ y

$$((Q \setminus R) \cap (Q \setminus S)) ; Q \subseteq S.$$

Por tanto,

$$((Q \setminus R) \cap (Q \setminus S)) ; Q \subseteq (R \cap S).$$

De nuevo por la ley básica,

$$((Q \setminus R) \cap (Q \setminus S)) \subset Q \setminus (R \cap S).$$

Para la contención inversa hay que considerar que $Q \setminus (R \cap S) \subseteq Q \setminus (R \cap S)$ y, entonces, $(Q \setminus (R \cap S)) ; Q \subseteq (R \cap S)$. Por la definición de $\cap$,

$$(Q \setminus (R \cap S)) ; Q \subseteq R$$
$$(Q \setminus (R \cap S)) ; Q \subseteq S.$$

De la ley básica se deduce que

$$(Q \setminus (R \cap S)) \subseteq (Q \setminus R)$$
$$(Q \setminus (R \cap S)) \subseteq (Q \setminus S).$$

En conclusión

$$(Q \setminus (R \cap S)) \subseteq (Q \setminus R) \cap (Q \setminus S). \qquad \blacksquare$$

Se puede interpretar este teorema diciendo que lo *mínimo* que se debe hacer antes de ejecutar Q para satisfacer tanto R como S es la intersección de lo que habría que hacer para satisfacer R y S por separado.

Corolario 3.11. $Q \setminus (\cap_n R_n) = \cap_n (Q \setminus R_n)$.

Demostración. Por la definición de $\cap$

$$\forall i \in \mathbb{N} \left(\bigcap_n (Q \setminus R_n) \subseteq (Q \setminus R_i)\right).$$

y de ahí se obtienen las siguientes afirmaciones

$$\forall i((\bigcap_n (Q \setminus R_n) ; Q) \subseteq R_i)$$

$$(\bigcap_n (Q \setminus R_n) ; Q) \subseteq \bigcap_n R_n$$

$$\bigcap_n (Q \setminus R_n) \subseteq (Q \setminus \bigcap_n R_n).$$

La inversa es similar. ■

Por supuesto, como lo indica la demostración, 3.11 es una generalización de 3.10.

Teorema 3.12. $(P \cup Q) \setminus R = (P \setminus R) \cap (Q \setminus R)$.

Demostración. Sea $(s_0, s) \in (P \cup Q) \setminus R$. En consecuencia,

$$\forall s_f (((s, s_f) \in P \vee (s, s_f) \in Q) \Rightarrow (s_0, s_f) \in R).$$

Por el teorema A.12, la expresión anterior equivale a

$$\forall s_f (((s, s_f) \in P \Rightarrow (s_0, s_f) \in R) \wedge ((s, s_f) \in Q \Rightarrow (s_0, s_f) \in R)).$$

Esto es,

$$(s_0, s) \in ((P \setminus R) \cap (Q \setminus R)),$$

y, por tanto,

$$(P \cup Q) \setminus R \subseteq (P \setminus R) \cap (Q \setminus R).$$

La contención inversa es análoga. ■

En términos intuitivos, si se desea cumplir la especificación R a partir de P o de Q (por ejemplo, cuando se trabaja con un mecanismo no determinista que puede elegir cualquiera de los dos), entonces la especificación del programa anterior a P o Q debe satisfacer cada una de las preespecificaciones separadas.

Corolario 3.13. $(\bigcup_n P_n) \setminus R = \bigcap_n (P_n \setminus R)$.

La demostración es análoga a la del teorema 3.12.

Teorema 3.14. $(P ; Q) \setminus R = P \setminus (Q \setminus R)$.

Demostración. Sea $X \subseteq P \setminus (Q \setminus R)$. Entonces, por la ley básica,

$$(X ; P) \subseteq (Q \setminus R).$$

Se aplica nuevamente la ley básica,

$$(X ; P) ; Q \subseteq R,$$

y, por la asociatividad de la composición y la ley básica

$$X \subseteq (P ; Q) \setminus R.$$

por lo que $P \setminus (Q \setminus R) \subseteq (P ; Q) \setminus R$.

Sea ahora $X \subseteq (P ; Q) \setminus R$. Ahora la ley básica nos da

$$X ; (P ; Q) \subseteq R,$$

por medio de la asociatividad de la composición y la ley básica se obtiene

$$(X ; P) \subseteq (Q \setminus R).$$

Una nueva aplicación de la ley básica produce el resultado deseado

$$X \subseteq P \setminus (Q \setminus R).$$

En conclusión, $P \setminus (Q \setminus R) = (P ; Q) \setminus R$. ■

Dicho en palabras comunes, lo que se debió haber hecho antes de $P ; Q$ para que se cumpla R es lo mismo que se debió haber hecho antes de P para que se cumpla $Q \setminus R$.

Teorema 3.15. $\breve{Q} = \breve{Q} \setminus I$.

Demostración. La hipótesis inicial será, ahora, $(s_0, s) \in \breve{Q} \setminus I$, es decir,

$$\forall s_f ((s, s_f) \notin Q \Rightarrow (s_0, s_f) \notin I).$$

Al aplicar el teorema A.13 se obtiene

$$\forall s_f ((s_0, s_f) \in I \Rightarrow (s, s_f) \in Q).$$

Por la definición de '\', $(s, s_0) \in I \setminus \breve{Q}$. Por el teorema 3.9, $I \setminus Q = Q$ y, de este modo,

$$(s_0, s) \in \breve{Q} \quad \text{y} \quad \breve{Q} \setminus I \subseteq \breve{Q}.$$

Invirtiendo el sentido de la demostración se obtiene el otro caso. ■

Siguiendo a Tarski [1941], en el capítulo 1 se introdujo como una operación primitiva la conversa de una relación. El teorema 3.15 permite definirla a partir de la preespecificación más débil.

Corolario 3.16. $Q \setminus R = \overline{\breve{R} ; \bar{Q}}$.

Demostración. Sea $(s_0, s) \in \overline{\breve{R} ; \bar{Q}}$, es decir,

$$\neg\exists s' ((s_0, s') \in \breve{R} \wedge (s', s) \in \bar{Q}).$$

Por los teoremas A.22, A.10 y A.7 lo anterior equivale a

$$\forall s' ((s', s) \in \bar{Q} \Rightarrow (s_0, s') \in R).$$

Por la definición de relación conversa, $(s', s) \in \bar{Q} \Leftrightarrow (s, s') \in Q$:

$$\forall s' ((s, s') \in Q \Rightarrow (s_0, s') \in R).$$

En conclusión, $(s_0, s) \in Q \setminus R$ y $\overline{\breve{R} ; \bar{Q}} \subseteq Q \setminus R$. Puesto que en todos los casos se trata de equivalencias, la contención inversa es inmediata. ■

Este es el resultado inverso del teorema anterior: la preespecificación se puede definir a partir de la operación de relación conversa.

Teorema 3.17. $\overline{Q \setminus R} = (\breve{R} \setminus \bar{Q}) \setminus \bar{1}$.

Demostración. Sea $(s_0, s) \in (\breve{R} \setminus \bar{Q}) \setminus \bar{1}$, lo cual, por el teorema 3.15, equivale a

$$(s, s_0) \notin \breve{R} \setminus \bar{Q},$$

es decir,

$$\neg\forall s_f ((s_0, s_f) \notin R \Rightarrow (s, s_f) \notin Q),$$

y, por definición de la preespecificación más débil y el teorema A.13,

$$(s_0, s) \notin Q \setminus R,$$

o, en otras palabras,

$$(s_0, s) \in \overline{Q \setminus R} \quad \text{y} \quad (\breve{R} \setminus \bar{Q}) \setminus \bar{1} \subseteq \overline{Q \setminus R}.$$

Y como sólo se usaron equivalencias la contención inversa es análoga. ■

Definición. Se llamará condición a la relación b si $b ; U = b$. De ahora en adelante, todas las relaciones representadas con las letras b, c, etc., serán condiciones.

Aunque no lo parezca, esta definición coincide con la noción intuitiva de condición. Una expresión booleana (véase el apéndice) es aquella que al evaluarse en cierto estado da el valor de verdadero o falso. Sea V el conjunto de estados en los que la expresión da el valor de verdadero. Entonces, si se quiere representar esta expresión como una condición b, $\mathfrak{D}(b) = V$. Como las condiciones se usarán como guardas para comandos, al cumplirse una condición (es decir, cuando se está en algún estado que pertenece a su dominio) se debe quedar en posición de ejecutar el comando (la relación) que guarda sin restricciones, o sea, partiendo de cualquier estado posible. Por tanto, si $x \in \mathfrak{D}(b)$, entonces $b(\{x\}) = \Sigma$, donde Σ representa el espacio de estados. Es claro entonces que b cumple con $b ; U = U$.

Teorema 3.18. b es una condición si y sólo si $\bar{b} = U ; \bar{b}$. Si b es una condición, $\bar{b}$ es una condición.

Demostración. Por el teorema 3.15, $\bar{b} = \bar{\bar{b}} \setminus I$, y por la definición de una condición, $\bar{b} = \overbar{b ; U} \setminus I$, es decir, $\bar{b} = b \,\widetilde{;}\, U$. Por el teorema 1.16, $\bar{b} = \check{U} ; \check{b}$. Y como $\check{U} = U$, se llega a la conclusión deseada. La implicación inversa es análoga. ■

Además, se quiere demostrar que si $b = b ; U$, entonces $\bar{b} = \bar{b} ; U$. Es claro que

$$\emptyset = b \cap \bar{b} = (b ; U) \cap \bar{b},$$

por lo que

$$\emptyset = (b ; U) \cap (\bar{b} \setminus I) \setminus I,$$

por 1.17 y 3.15. Se aplica ahora 1.19 y se tiene:

$$\emptyset = U ; \overbar{(\bar{b} \setminus I)} \cap \bar{b} \setminus I.$$

Del teorema 1.3 se deduce

$$U ; \check{\bar{b}} \subseteq \check{b}.$$

Pero el teorema 1.14 dice que $\check{\bar{b}} \subseteq \check{\bar{b}} ; U$ y se concluye que

$$U ; \check{\bar{b}} = \check{\bar{b}}.$$ ■

Teorema 3.19. Sea b una condición. Entonces $\bar{b} \setminus I = \overbar{\bar{b} \setminus \emptyset} = \bar{b} \setminus \emptyset$.

Demostración. Como se recordará, por 3.15 $\breve{b} = \bar{b} \setminus \bar{1}$. Por el teorema 3.18, $\breve{b} = U ; \breve{b}$ y, entonces

$$\bar{b} \setminus \bar{1} = U ; \breve{b} = U ; (\bar{b} \setminus \bar{1}).$$

Los teoremas 1.17 y 3.15 establecen que

$$U ; (\bar{b} \setminus \bar{1}) = ((U ; (\bar{b} \setminus \bar{1})) \setminus \bar{1}) \setminus \bar{1}.$$

Ahora recurrimos al teorema 3.14 y se obtiene

$$(U \setminus ((\bar{b} \setminus \bar{1}) \setminus \bar{1})) \setminus \bar{1}.$$

o, de forma más simple, $(U \setminus \breve{\bar{b}}) \setminus \bar{1} = (U \setminus b) \setminus \bar{1}$, gracias a 3.15 y 1.17. Por último, se utiliza 3.17 para concluir

$$\bar{b} \setminus \bar{1} = (U \setminus b) \setminus \bar{1} = \overline{b \setminus \emptyset}. \tag{7}$$

Ahora bien, $\bar{b} \setminus \bar{1} = \breve{b} = \breve{\bar{b}} = \bar{\breve{b}} = \overline{\breve{b} \setminus \bar{1}}$ por 1.18. Como b es una condición, $\bar{b}$ también lo es (3.18) y, entonces, se tiene

$$\bar{b} \setminus \bar{1} = \overline{\overline{b \setminus \emptyset}},$$

como ya se demostró en (7), en este mismo teorema. Se aplica la doble negación y se obtiene el resultado deseado. ■

Teorema 3.20. $P ; (b \cap Q) = (P \cap \overline{b \setminus \emptyset}) ; Q$.

Demostración. Sea $(x, z) \in P ; (b \cap Q)$. Entonces, y sólo entonces,

$$\exists y((x, y) \in P \wedge (y, z) \in b \cap Q).$$

A su vez, $(y, z) \in b \cap Q$ si y sólo si

$$(y, z) \in b \wedge (y, z) \in Q. \tag{8}$$

Pero $b = b ; U$ o, en otras palabras,

$$\forall x (y, x) \in b.$$

Ya se sabía que $(x, y) \in P$ y ahora se tiene que $(x, y) \in \breve{b}$. La conclusión es

$$(x, y) \in (P \cap \breve{b}). \tag{9}$$

(8) y (9) equivalen a

$$(x, y) \in (P \cap \check{b}) ; Q \quad \text{y} \quad P ; (b \cap Q) \subseteq (P \cap \check{b}) ; Q.$$

Finalmente, por 3.19, $\check{b} = \overline{b \setminus \emptyset}$. Como se usaron sólo equivalencias, la contención inversa es inmediata. ■

Corolario 3.21. $X ; ((P ; U) \cap Q) = (X \cap \overline{P \setminus \emptyset}) ; Q.$

Demostración. Claramente, $P ; U = (P ; U) ; U$, por lo que $P ; U$ es una condición. Se aplica el teorema anterior:

$$X ; ((P ; U) \cap Q) = (X \cap \overline{(P ; U) \setminus \emptyset}) ; Q. \tag{10}$$

El teorema 3.14 hace que (10) equivalga a

$$= (X \cap \overline{P \setminus (U \setminus \emptyset)}) ; Q.$$

Por 3.4, $U \setminus \emptyset \subseteq \emptyset$ y, entonces, $U \setminus \emptyset = \emptyset$. Así, obtenemos la última equivalencia:

$$= (X \cap \overline{P \setminus \emptyset}) ; Q. \quad ■$$

Teorema 3.22. $((P ; U) \cap Q) \setminus R = (P \setminus \emptyset) \cup (Q \setminus R)$.

Demostración. Sea $X \subseteq ((P ; U) \cap Q) \setminus R$. Por la ley básica, esto equivale a

$$X ; ((P ; U) \cap Q) \subseteq R. \tag{11}$$

(11) equivale a

$$(X \cap \overline{(P \setminus \emptyset)}) ; Q \subseteq R. \tag{12}$$

gracias a 3.21. Nuevamente, la ley básica permite transformar (12) en

$$(X \cap \overline{(P \setminus \emptyset)}) \subseteq Q \setminus R. \tag{13}$$

Por último, el teorema 1.4 hace que (13) sea

$$X \subseteq (P \setminus \emptyset) \cup (Q \setminus R). \quad ■$$

Corolario 3.23. $(b \cap Q) \setminus R = (b \setminus \emptyset) \cup (Q \setminus R)$.

Demostración. Dado que b es una condición,

$$(b \cap Q) \setminus R = ((b ; U) \cap Q) \setminus R,$$

y, por el teorema anterior, esto es igual a $(b \setminus \emptyset) \cup (Q \setminus R)$. ■

Teorema 3.24. $\overline{P ; U} \setminus R = \overline{(P \setminus \emptyset)} \cup (U \setminus R)$.

Demostración. El teorema 1.8 nos permite afirmar que $\overline{P ; U} \setminus R = (\overline{P ; U} \cap U) \setminus R$. Por el teorema 3.20 se puede deducir entonces

$$(\overline{P ; U} \setminus \emptyset) \cup (U \setminus R).$$

$P ; U$ es una condición; entonces, se puede aplicar 3.19 y entonces se tiene

$$\overline{(P ; U) \setminus \emptyset} \cup (U \setminus R).$$

Por 3.14, ahora se tiene

$$\overline{P \setminus (U \setminus \emptyset)} \cup (U \setminus R),$$

y, finalmente, por 3.4,

$$\overline{(P \setminus \emptyset)} \cup (U \setminus R).$$ ■

Definición. Si P es una relación parcial, entonces

$$P^+ = P \cup \overline{P ; U}.$$

Es claro que P^+ es una extensión de P a una relación total.

Teorema 3.25. $P^+ \setminus R = P \setminus R \cap (\overline{P \setminus \emptyset} \cup U \setminus R)$.

Demostración. Por definición $P^+ \setminus R = (P \cup \overline{P ; U}) \setminus R$. Por 3.12, se tiene

$$P^+ \setminus R = (P \setminus R) \cap (\overline{P ; U} \setminus R).$$

3.21 permite concluir que

$$P^+ \setminus R = P \setminus R \cap (\overline{P \setminus \emptyset} \cup U \setminus R).$$ ■

Teorema 3.26. $\overline{((b \cap P) \cup (c \cap Q)) ; U} = (b \cup \overline{P ; U}) \cap (c \cup \overline{Q ; U})$

Demostración. Por 1.11

$$\overline{((b \cap P) \cup (c \cap Q)) ; U} = \overline{(b \cap P) ; U} \cup \overline{(c \cap Q) ; U}.$$

Por 1.6 y la definición de condición, se llega a

$$= \overline{(I ; ((b ; U) \cap P) ; U)} \cup \overline{(I ; ((c ; U) \cap Q) ; U)}.$$

Al aplicar 3.21 se obtiene

$$= \overline{((I \cap \overline{b \setminus \emptyset}) ; (P ; U))} \cup \overline{((I \cap \overline{c \setminus \emptyset}) ; (Q ; U))}.$$

Por 3.20 esto es igual a

$$= \overline{(b \cap (P ; U))} \cup \overline{(c \cap (Q ; U))}.$$

y por A.11:

$$= \overline{b \cap (P ; U)} \cap \overline{c \cap (Q ; U)}$$
$$= (\bar{b} \cup \overline{P ; U}) \cap (\bar{c} \cup \overline{Q ; U}).$$ ■

Corolario 3.27. Sean P y Q relaciones totales. Entonces, $\overline{((b \cap P) \cup (c \cap Q)) ; U} = \bar{b} \cap \bar{c}$.

Demostración. Por 3.26

$$\overline{((b \cap P) \cup (c \cap Q)) ; U} = (\bar{b} \cup \overline{P ; U}) \cap (\bar{c} \cup \overline{Q ; U}).$$

Pero, como P y Q son totales, $P ; U = U = Q ; U$ y

$$\therefore \quad = \bar{b} \cap \bar{c}.$$ ■

Teorema 3.28. Si P y Q son relaciones totales, entonces $\overline{((b \cap P) \cup (c \cap Q)) ; U} \setminus R = (\bar{b} \setminus \emptyset) \cup (\bar{c} \setminus \emptyset) \cup (U \setminus R)$.

Demostración. El corolario 3.27 dice que

$$\overline{((b \cap P) \cup (c \cap Q)) ; U} \setminus R = (\bar{b} \cap \bar{c}) \setminus R. \tag{14}$$

Por A.11, (14) es igual a

$$\overline{(b \cup c)} \setminus R. \tag{15}$$

Pero como b y c son condiciones, $b \cup c = (b \cup c) ; U$ y, por 3.24, (15) es

$$\overline{(b \cup c) \setminus \emptyset} \cup (U \setminus R). \tag{16}$$

Ahora, pro 3.12, (16) es

$$\overline{(b \setminus \emptyset) \cap (c \setminus \emptyset)} \cup (U \setminus R). \tag{17}$$

Nuevamente, por A.11, (17) es

$$\overline{(b \setminus \emptyset)} \cup \overline{(c \setminus \emptyset)} \cup (U \setminus R),$$

y, por 3.19, se tiene que

$$= (\bar{b} \setminus \emptyset) \cup (\bar{c} \setminus \emptyset) \cup (U \setminus R). \quad \blacksquare$$

Teorema 3.29. $((P ; U) \cap Q) \setminus R = (P \setminus \emptyset) \cup (Q \setminus R)$.

Demostración. Sea $X \subseteq ((P ; U) \cap Q) \setminus R$. Por la ley básica

$$X ; ((P ; U) \cap Q) \subseteq R. \tag{18}$$

Por 3.21, (18) equivale a

$$(X \cap \overline{(P \setminus \emptyset)}) ; Q \subseteq R.$$

Y, de nuevo por la ley básica, se tiene

$$X \cap \overline{(P \setminus \emptyset)} \subseteq Q \setminus R. \tag{19}$$

Por 1.4, (19) equivale a

$$X \subseteq (P \setminus \emptyset) \cup (Q \setminus R).$$

Por tanto, $((P ; U) \cap Q) \setminus R \subseteq (P \setminus \emptyset) \cup (Q \setminus R)$. La contención inversa es inmediata, pues sólo se usaron equivalencias. $\blacksquare$

Corolario 3.30. $(b \cap Q) \setminus R = (b \setminus \emptyset) \cup (Q \setminus R)$.

Demostración. Como b es una condición, $(b \cap Q) \setminus R = ((b ; U) \cap Q) \setminus R$. Por 3.29, $(b \cap Q) \setminus R = (b \setminus \emptyset) \cup (Q \setminus R)$. $\blacksquare$

Teorema 3.31. Si P y Q son relaciones totales, entonces

$$((b \cap P) \cup (c \cap Q))^* \setminus R = (b \setminus \emptyset \cup P \setminus R) \cap (c \setminus \emptyset \cup Q \setminus R) \cap (\tilde{b} \setminus \emptyset \cup \tilde{c} \setminus \emptyset \cup U \setminus R).$$

Demostración. Por la definición de +, se tiene

$$((b \cap P) \cup (c \cap Q))^* \setminus R = (((b \cap P) \cup (c \cap Q)) \cup \overline{(((b \cap P) \cup (c \cap Q)) ; U)}) \setminus R.$$

Gracias a 3.12, se tiene ahora

$$= ((b \cap P) \setminus R \cap (c \cap Q) \setminus R) \cap (((\overline{(b \cap P) \cup (c \cap Q)) ; U}) \setminus R).$$

Del corolario 3.30 se tiene

$$= (b \setminus \emptyset \cup P \setminus R) \cap (c \setminus \emptyset \cup Q \setminus R) \cap (((\overline{(b \cap P) \cup (c \cap Q)) ; U}) \setminus R),$$

y, por último, 3.28 da el resultado

$$= (b \setminus \emptyset \cup P \setminus R) \cap (c \setminus \emptyset \cup Q \setminus R) \cap (\tilde{b} \setminus \emptyset \cup \tilde{c} \setminus \emptyset \cup U \setminus R). \qquad \blacksquare$$

Teorema 3.32. Sea $P ; U$. Se tiene entonces que $P \setminus \emptyset = \emptyset$.

Demostración. Es evidente que $\emptyset = U = \tilde{U}$. Por 3.15 y 3.14, entonces:

$$(P ; U) \setminus 1 = P \setminus (U \setminus 1) = P \setminus \tilde{U} = P \setminus \emptyset. \tag{20}$$

Por otro lado, por hipótesis:

$$(P ; U) \setminus 1 = U \setminus 1. = \emptyset \tag{21}$$

De (20) y (21) se deduce que $P \setminus \emptyset = \emptyset$. $\blacksquare$

Teorema 3.33. $P \setminus R \subseteq (R ; (\tilde{P} \setminus 1)) \cup (P \setminus \emptyset)$.

Demostración. Sea $X \subseteq P \setminus R$. Entonces, por la ley básica

$$X ; P \subseteq R. \tag{22}$$

Como ';' es un operador monótono, (22) equivale a

$$X ; P ; (\tilde{P} \setminus 1) \subseteq R ; (\tilde{P} \setminus 1). \tag{23}$$

Por 3.15, (23) es igual a:

$$X ; P ; \breve{P} \subseteq R ; (\bar{P} \setminus \bar{I}). \tag{24}$$

Como $(P;U) \cap I$ es la relación I restringida al dominio de P, es claro que $(P;U) \cap I \subseteq P ; \breve{P}$. En consecuencia, y por la monotonía de ';', (24) implica

$$X ; ((P ; U) \cap I) \subseteq R ; (\bar{P} \setminus \bar{I}). \tag{25}$$

Es claro que $P ; U$ es una condición y, entonces, se aplica el teorema 3.20 a (25):

$$X \cap \overline{P \setminus \emptyset} \subseteq R ; (\bar{P} \setminus \bar{I}). \tag{26}$$

El teorema 1.4 convierte (26) en:

$$X \subseteq P ; (\bar{P} \setminus \bar{I}) \cup (P \setminus \emptyset),$$

con lo que se concluye que $P \setminus R \subseteq R ; (\bar{P} \setminus \bar{I}) \cup (P \setminus \emptyset)$. ∎

Teorema 3.34. Sea P una función parcial. Entonces

$$P \setminus R ; (\bar{P} \setminus \bar{I}) \cup (P \setminus \emptyset).$$

Demostración. Una contención se demostró ya en el teorema anterior. Veamos ahora la otra. Consideremos el conjunto

$$((R ; \breve{P}) \cup (P \setminus \emptyset)) ; P.$$

La aplicación de 1.11 nos da

$$(R ; \breve{P}) ; P \cup (P \setminus \emptyset) ; P. \tag{27}$$

Como $\bar{P} \setminus \bar{I} \subseteq (P ; \bar{I}) \setminus \bar{I} = P \setminus (\bar{I} \setminus \bar{I}) = P \setminus I$, entonces $\bar{P} \setminus \bar{I} \subseteq P \setminus I$. Gracias al teorema 1.15, (27) es subconjunto de

$$R ; (P \setminus I) ; P \cup (P \setminus \emptyset) ; P. \tag{28}$$

Por otro lado, por la definición de '\' y de $\emptyset$, $P \setminus \emptyset = \{(x,y) \mid \forall z (y,z) \notin P\}$, es decir, y no pertenece a P^{-1}. Por tanto, $(P \setminus \emptyset) ; P = \emptyset$. De este modo, (28) se transforma en

$$R ; (P \setminus I) ; P. \tag{29}$$

A su vez,

$$P \setminus I = \{(x,y) \mid \forall z\,(y,z) \in P \Rightarrow x = z\}. \tag{30}$$

y, por su parte,

$$(P \setminus I) ; P = \{(a,c) \mid \exists b\,(a,b) \in P \setminus I \wedge (b,c) \in P\}. \tag{31}$$

(30) y (31) nos dicen que $(P \setminus I) ; P$ es P restringida al subconjunto en el que equivale a I. Por esto,

$$R ; (P \setminus I) ; P \subseteq R. \tag{32}$$

Por la ley básica, de (32) se concluye que

$$(R ; (\breve{P} \setminus \breve{I}) \cup (P \setminus emptyset)) \subseteq P \setminus R. \qquad \blacksquare$$

Corolario 3.35. Sea P una función parcial. Entonces,

$$P^+ \setminus R = R ; (\breve{P} \setminus \breve{I}) \cup ((P \setminus \emptyset) \cap (U \setminus R)).$$

Demostración. Por el teorema 3.25:

$$P^+ \setminus R = P \setminus R \cap (\overline{P \setminus \emptyset} \cup \overline{U \setminus R}),$$

y, por el teorema anterior y 1.5, se obtiene la conclusión deseada. $\blacksquare$

Teorema 3.36. Si F es una función total, entonces

$$F \setminus R = R ; (F \setminus I).$$

Demostración. El teorema 1.9 nos dice que

$$(R ; (P \setminus I)) ; P = R ; ((P \setminus I) ; P). \tag{33}$$

En la demostración de 3.34, (30) y (31) nos permitieron concluir que $R ; ((P \setminus I) ; P) \subseteq R$. Por tanto, de (33) se concluye que

$$P \setminus R \subseteq R,$$

lo cual, por ley básica, nos da

$$R ; (P \setminus I) \subseteq P \setminus R.$$

Para la contención inversa hay que observar que, como P es una función total

$$P \setminus \emptyset = \emptyset,$$

pues $P \setminus \emptyset = \{(x, y) \mid \forall z\,(y, z) \notin P\}$ y se tiene que P es total. Por tanto

$$\emptyset = P \setminus \emptyset = P \setminus (R \cap \check{R}).$$

3.10 transforma lo anterior en:

$$\emptyset = (P \setminus R) \cap (P \setminus \check{R}).$$

Aplicando 1.3, se concluye que

$$P \setminus R \subseteq \overline{P \setminus \check{R}}.$$

Por 3.16 y 3.17, lo anterior equivale a

$$P \setminus R \subseteq R ; \check{Q},$$

que, gracias a 3.15 se obtiene el resultado deseado. ■

3.4 El lenguaje $\mathfrak{L}$

Antes de definir un lenguaje basado en la preespecificación más débil, se harán algunas consideraciones generales.

3.4.1 *¿Qué clase de lenguaje se necesita?*

La manera más fácil de definir un lenguaje con la preespecificación más débil es hacerlo en "abstracto", es decir, sin poner restricciones al tipo de relaciones que se usarán para construir los programas. Sin embargo, de esta forma no se toman en cuenta algunos puntos cruciales:

a) ¿Los programas definidos son computables? En otras palabras, si se define una instrucción en el lenguaje (por ejemplo, la asignación), ¿es posible llevarla a cabo en un tiempo finito?

b) En caso de que se desee incluir dentro del lenguaje procedimientos recursivos para aumentar su capacidad de expresión, es indispensable saber si las funciones recursivas están bien definidas en el lenguaje (que den un resultado).

c) ¿Es posible construir ("implementar") de modo eficiente las operaciones del lenguaje?[1]

Como se verá más adelante, para cumplir con a) y b) se necesita que los operadores del lenguaje sean monótonos y co-continuos.[2] Para el punto c), hará falta limitar la clase de operadores relaciones válidos en el lenguaje en un sentido más fuerte y habrá que restringir los programas válidos a las relaciones totales.

3.4.2 *Recursión y puntos fijos*

En la sección 1.5 se vio que el mínimo punto fijo del funcional F se puede calcular de la siguiente manera:

$$f_\infty = \bigcup_{0 \le n} f_n,$$

donde

$$f_0 = \emptyset \qquad \text{y} \qquad f_{n+1} = F(f_n),$$

si F es continuo (la continuidad es condición suficiente, pero necesaria). Por esta razón, es necesario limitar los operadores del lenguaje a los continuos.

Ahora es posible definir la preespecificación más débil de los procedimientos recursivos puesto que, por 3.13:

$$\bigcup_{0 \le n} f_n \setminus R = \bigcap_{0 \ge n} (f_n \setminus R),$$

y, por la definición de f_∞,

$$f_\infty \setminus R = \bigcap_{0 \le n} (f_n \setminus R).$$

[1] "Eficiente" no es un término vago, pues en este caso se refiere al significado técnico de la palabra en teoría de la computación (calculable en tiempo polinomial). En relación con este tema, puede consultarse Garey y Johnson [1979].

[2] Véase el capítulo 1 para estos conceptos.

No obstante, como se vio en 1.5, el punto fijo mínimo no es necesariamente una relación total. La alternativa es el uso del punto fijo máximo. Cuando se aborde el tema de la eficiencia, se verá que esta opción es la mejor.

Hay que recordar que el punto fijo máximo se define de la siguiente forma:

$$f^{\infty} = \bigcap_{0 \leq n} f^n,$$

con $f^0 = U$ y $f^{n+1} = F(f^n)$, si es que F es co-continuo (también una condición suficiente).

En este caso, en lugar de exigir continuidad a los operadores del lenguaje, se necesita la co-continuidad para asegurarse de que los programas definidos recursivamente terminan.

El requisito de la monotonía para los operadores del lenguaje puede parecer excesivo. No obstante, los riesgos de eliminarlo no son despreciables (además de que es necesaria para la demostración de existencia de puntos fijos). Un de estos riesgos es el de introducir expresiones sin sentido al lenguaje, como en el caso de la negación, para la cual no existen ni $\bar{X}_{\infty}$ ni $\bar{X}^{\infty}$ y, de hecho, no existe ningún punto fijo, o sea, ninguna X tal que $X = \bar{X}$.

Ahora se verá el problema de la eficiencia en la construcción de los operadores.

3.4.3 *Operadores eficientes*

El operador $\cap$ puede ser un operador potentísimo, pues permite combinar de manera muy simple dos programas y obtener un resultado valioso. *V.g.*, considérese un arreglo de números enteros positivos. Sea P un programa que asigna al arreglo una secuencia creciente de números y sea Q un programa que efectúa una permutación en los elementos del arreglo. $P \cap Q$, entonces, es un programa que ordena el arreglo en orden creciente.

Pero, a pesar de su simplicidad, el programa anterior presenta una seria desventaja: para encontrar $P \cap Q$ hay que probar todas las ejecuciones posibles de P y de Q y ver cuáles son comunes. En el caso de P, la exploración de todas las posibilidades tomaría un tiempo $O(m - n + 1)$, donde n es la longitud del arreglo y m el valor máximo asignable a cada entrada del arreglo (necesariamente, $m \geq n + 1$). Pero para Q, el tiempo sería $O(n!) = O(c^n)$, lo cual es, evidentemente de orden exponencial. ¿Cómo se puede evitar que una "implementación" proceda de esta forma? No se puede: si existe una solución al problema (esto es, si

$P \cap Q \neq \emptyset$) la "implementación" está obligada a encontrarla y esto puede implicar la exploración de todas las posibilidades (pues dar una directriz del orden más adecuado para explorar las opciones, además de ser muy difícil de hacer, violaría el no determinismo de los constructos del lenguaje).

La alternativa es, entonces, no incluir $\cap$ entre los operadores válidos del lenguaje.

Por su parte $\cup$ presenta otro problema. Sea $F(X) = X$. El mínimo punto fijo de F es:

$$f_{\mathbf{x}} = \bigcup_{0 \leq n} f_n.$$

Pero $f_1(\emptyset) = \emptyset = f_0$ y, en realidad, $f_n = \emptyset$ para toda n. En otras palabras, la relación vacía es el mínimo punto fijo de la identidad. ¡Cómo se puede "interpretar" una relación vacía"! Como una recursión que no termina (en este caso, una definición recursiva de la igualdad no termina, como era de esperarse). Entonces, si F es tal que $f_{\mathbf{x}} = \emptyset$, el programa $f_{\mathbf{x}} \cup P$ puede no terminar, dependiendo de cuál de las dos alternativas se ejecute primero (si es que alguna se ejecuta primero) P o $f_{\mathbf{x}}$.

Nuevamente, la solución está en restringir las relaciones que pertenecen al lenguaje. En este caso, si se incluyen sólo relaciones totales, no se puede tener a $\emptyset$ entre las elementos del lenguaje y, por tanto, no hay procedimientos recursivos que no terminen.

En principio, con estas restricciones, y las de la subsección 3.4.2, se puede construir un lenguaje eficiente y de gran capacidad expresiva.

3.4.4 *Un conjunto de relaciones para el lenguaje* $\mathfrak{L}$

Casi es el momento de comenzar la definición del lenguaje basado en la preespecificación más débil. De ahora en adelante, nos referiremos a dicho lenguaje con el símbolo $\mathfrak{L}$. Sólo se harán algunas consideraciones más antes de comenzar.

La restricción de los programas válidos en $\mathfrak{L}$ a las relaciones totales se hará extendiendo las relaciones parciales a relaciones totales de la siguiente forma:

Sea P una relación no total. Si se ejecuta P en un estado $s \in \mathfrak{D}(P)$, la ejecución terminará en un estado s' tal que $(s,s') \in R$. Si en cambio, $s \notin \mathfrak{D}(P)$, entonces el programa P no termina (pues no existe un estado al cual se llegue después de ejecutar P). Agréguese $\perp$, un nuevo estado, al conjunto de estados en el dominio y en el rango de las relaciones posibles. Se puede construir ahora una nueva relación

P':

$$P' = \{(s,s') \mid (s,s') \in P \vee (s \notin \mathfrak{D}(P) \wedge s' = \bot)\}.$$

En resumen, $\bot$ representa el "estado" de no terminación. Por supuesto, P' es una relación total. Este truco se presentó en la subsección 1.3.2. Σ denota a los nuevos dominio y contradominio ampliados con $\{\bot\}$.

Hay un problema más. El mínimo punto fijo de una relación total no necesariamente es total, por lo que la recursión podría introducir elementos ajenos al lenguaje. El punto fijo máximo de una relación total, en cambio, sí es total. Los operadores del lenguaje se restringirán entonces a los co-continuos.

La definición del conjunto de relaciones que formarán parte de $\mathfrak{L}$ se hará de manera inductiva:

1. $U \in \mathfrak{L}$.
2. Si P es una función total (de cuyos dominio y contradominio excluimos solamente $\bot$), entonces $P^* \in \mathfrak{L}$.
3. Si $P, Q \in \mathfrak{L}$, y b y c son condiciones, entonces también pertenecen a $\mathfrak{L}$:
   $P ; Q$;
   $((b \cap P) \cup (c \cap Q))^*$.
4. Si $P_i \in \mathfrak{L}$ y $\forall i\, P_i \supseteq P_{i+1}$, entonces
$$\bigcap_{i \geq 0} P_i \in \mathfrak{L}.$$

Pero de las condiciones anteriores no se sigue que las relaciones que las cumplen sean totales y que todos los operadores sean co-continuos (al menos no de manera evidente). Si agregamos a la hipótesis 1 que P sea además finitaria[3] entonces se tendrá que todas las relaciones de $\mathfrak{L}$ son totales y finitarias y que todos los operadores son co-continuos. Los teoremas 3.37–3.45 fundamentan esta afirmación.

Teorema 3.37. La relación U es total y finitaria.

Demostración. Es evidente que U es total. Sea $s \in \Sigma$. Entonces, es obvio que $\{s\} \uparrow U = \Sigma$. ∎

[3] Véase la sección 1.3.2.

Teorema 3.38. Si P es una función total, $P \cup \{\bot\} \times \Sigma$ es total finitaria.

Demostración. Si $s \neq \bot$, 1.21 nos dice que

$$\{s\} \upharpoonright (P \cup \{\bot\} \times \Sigma) = (\{s\} \upharpoonright P) \cup (\{s\} \upharpoonright \{\bot\} \times \Sigma).$$

Además, $\{s\} \upharpoonright \{\bot\} \times \Sigma = \emptyset$, porque ningún miembro de $\mathfrak{D}(P)$ está relacionado con $\bot$. Por tanto,

$$\{s\} \upharpoonright (P \cup \{\bot\} \times \Sigma) = \{s\} \upharpoonright P,$$

y $\{s\} \upharpoonright P$, tiene un solo elemento pues P es una función.

Por otra parte, si $s = \bot$, entonces

$$\begin{aligned} \{\bot\} \upharpoonright (P \cup \{\bot\} \times \Sigma) &= \{\bot\} \upharpoonright P \cup \{\bot\} \upharpoonright (\{\bot\} \times \Sigma) \\ &= \{\bot\} \upharpoonright (\{\bot\} \times \Sigma) \\ &= \Sigma. \end{aligned}$$

En consecuencia $P \cup \{\bot\} \times \Sigma$ es total y finitaria. ■

Teorema 3.39.. Si P y Q son relaciones totales y finitarias, $P ; Q$ es total finitaria.

Demostración. Dado que P es total finitaria, tenemos dos posibilidades: $\{s\} \upharpoonright P = \Sigma$ o $\{s\} \upharpoonright P = \{s_1, s_2, \ldots, s_m\}$, para cualquier s arbitraria. Supongamos que vale el primer caso y, por el teorema 1.22,

$$\{s\} \upharpoonright (P ; Q) = (\{s\} \upharpoonright P) \upharpoonright Q = \Sigma \upharpoonright Q = \Sigma.$$

Si, en cambio, es cierta la segunda opción,

$$\begin{aligned} \{s\} \upharpoonright (P ; Q) &= (\{s\} \upharpoonright P) \upharpoonright Q \\ &= \{s_1, s_2, \ldots, s_m\} \upharpoonright Q \\ &= \bigcup_{i \leq m} (\{s_i\} \upharpoonright Q). \end{aligned}$$

Como Q es total finitaria, $\{s_i\} \upharpoonright Q$ es finito o es igual a Σ y, en consecuencia, $\bigcup_{i \leq m} (\{s_i\} \upharpoonright Q)$ es finita o igual a Σ también, con lo que concluye la demostración. ■

Teorema 3.40.. Sean P, Q relaciones totales finitarias y sean b y c condiciones, entonces:

$$(b \cap P \cup c \cap Q)^*$$

también es finitaria.

Demostración. Sea s un estado arbitrario. Por la definición de $^+$,

$$(b \cap P \cup c \cap Q)^+ = ((b \cap P \cup c \cap Q) \cup \overline{(b \cap P \cup c \cap Q)} ; U),$$

y, por el corolario 3.27,

$$= (b \cap P \cup c \cap Q) \cup (\bar{b} \cap \bar{c}).$$

Si se aplica el teorema 1.21, se obtiene

$$\{s\} \uparrow ((b \cap P) \cup (c \cap Q))^+ = (\{s\} \uparrow (b \cap P)) \cup (\{s\} \uparrow (c \cap Q)) \cup (\{s\} \uparrow (\bar{b} \cap \bar{c})).$$

Por otra parte, como P y Q son totales y finitarios, $\{s\} \uparrow (b \cap P) \cup \{s\} \uparrow (c \cap Q) \cup \{s\} \uparrow (\bar{b} \cap \bar{c})$ es finito o igual a Σ. ■

Teorema 3.41.. Supongamos que $\forall i$, P_i es total finitaria y $P_i \supseteq P_{i+1}$. Entonces,

$$\bigcap_i P_i$$

es total finitaria.

Demostración. Por el teorema 1.24,

$$\{s\} \uparrow (\bigcap_i P_i) = \bigcap_i (\{s\} \uparrow P_i).$$

Hay dos casos posibles: a) $\forall i (\{s\} \uparrow P_i) = \Sigma$, y b) $\exists j (\{s\} \uparrow P_j = \{s_1, s_2, \ldots, s_n\})$. En el caso a), es claro que

$$\bigcap_i (\{s\} \uparrow P_i) = \Sigma.$$

Para el caso b) hay que considerar que $P_{i+1} \subseteq P_i$, por lo que $\{s\} \uparrow P_{i+1} \subseteq \{s\} \uparrow P_i$. En consecuencia,

$$\bigcap_i (\{s\} \uparrow P) \subseteq \{s_1, s_2, \ldots, s_n\}.$$

Como ninguno de los P_i es vacío y todos son relaciones totales, la intersección de sus imágenes no es vacía. En consecuencia la imagen de $\cap_i P_i$ es finita y no vacía. ■

Corolario 3.42.. Sea $\{P_i\}$ una cadena decreciente de relaciones totales finitarias. Entonces, para todo s, existe m tal que

$$\{s\} \uparrow P_n = \{s\} \uparrow P_m \qquad \text{si } n \geq m.$$

Demostración. La demostración se vale del hecho de que $\{s\} \uparrow P_{i+1} \subseteq \{s\} \uparrow P_i$ y de que, en caso de que la imagen de $\{s\}$ sea infinita, es igual a Σ. ■

Teorema 3.43.. Sean Q total finitaria, $\{P_i\}$ una cadena decreciente de relaciones totales finitarias, b y c condiciones. Entonces se tiene que

$$((b \cap (\bigcap_i P_i)) \cup (c \cap Q))^+ = \bigcap_i ((b \cap P_i) \cup (c \cap Q))^+.$$

Demostración. Se aplica el corolario 3.27 y se obtiene

$$((b \cap (\bigcap_i P_i)) \cup (c \cap Q))^+ = (b \cap (\bigcap_i P_i)) \cup (c \cap Q) \cup (\bar{b} \cap \bar{c}).$$

Aplicando la propiedad de distribución para la unión y la intersección, lo anterior queda

$$= \bigcap_i ((b \cap P_i) \cup (c \cap Q) \cup (\bar{b} \cap \bar{c})).$$

Y, nuevamente por el corolario 3.27, el resultado es

$$= \bigcap_i ((b \cap P_i) \cup (c \cap Q) \cup \overline{(b \cap P) \cup (c \cap Q)} ; U).$$

Por último, por la definición de $^+$, se concluye

$$\bigcap_i ((b \cap P_i) \cup (c \cap Q))^+ \qquad \blacksquare$$

Teorema 3.44.. Sea $\{P_i\}$ una cadena descendente de relaciones totales finitarias. Entonces:

$$(\bigcap_i P_i) ; Q = \bigcap_i (P_i ; Q).$$

Demostración. Existen dos casos: a) $\forall i (\{s\} \uparrow P_i = \Sigma)$, y b) $\exists i (\{s\} \uparrow P_i = \{s_1, s_2, \ldots, s_m\})$.

a) Por el teorema 1.22

$$\{s\} \upharpoonright ((\bigcap_i P_i) ; Q) = (\{s\} \upharpoonright (\bigcap_i P_i)) \upharpoonright Q.$$

Utilizando la hipótesis se obtiene

$$= (\bigcap_i \Sigma) \upharpoonright Q = \Sigma \upharpoonright Q.$$

Es obvio que $\Sigma \upharpoonright Q = \cap_i(\Sigma \upharpoonright Q)$ y, por tanto, es igual a

$$\bigcap_i((\{s\} \upharpoonright P_i) \upharpoonright Q).$$

que, por los teoremas 1.22 y 1.24, equivale a

$$\{s\} \upharpoonright \bigcap_i(P_i ; Q).$$

b) Para el segundo caso se recurre al corolario 3.40, a saber, que existe m tal que:

$$\{s\} \upharpoonright P_n = \{s\} \upharpoonright P_m,$$

para todo $n \geq m$. Entonces, por el teorema 1.22,

$$\{s\} \upharpoonright ((\bigcap_i P_i) ; Q) = \{s\} \upharpoonright (P_m ; Q) = \{s\} \upharpoonright (P_n ; Q),$$

si $n \geq m$. Pero, como $\{s\} \upharpoonright P_{i+1} \subseteq \{s\} \upharpoonright P_i$, lo anterior es igual a

$$\bigcap_i(\{s\} \upharpoonright (P_i ; Q)),$$

y, aplicando el teorema 1.24,

$$= \{s\} \upharpoonright \bigcap_i(P_i ; Q).$$

En los dos casos anteriores se concluyó que $\forall s(\{s\} \upharpoonright ((\cap_i P_i) ; Q) = \{s\} \upharpoonright \cap_i(P_i ; Q))$, es decir, que $(\cap_i P_i) ; Q$ y $\cap_i(P_i ; Q)$ son la misma relación. ■

Teorema 3.45.. Sean P una relación total finitaria y $\{Q_i\}$ una cadena decreciente de relaciones totales finitarias. Entonces

$$P ; (\bigcap_i Q_i) = \bigcap_i(P ; Q_i).$$

Demostración. Supongamos que $\{s\} \uparrow P = \Sigma$. Entonces, por el teorema 1.22,

$$\{s\} \uparrow (P ; \bigcap_i Q_i) = \Sigma \uparrow \bigcap_i Q_i.$$

Se distribuye la operación $\uparrow$ y queda

$$\bigcap_i (\Sigma \uparrow Q_i),$$

y, por hipótesis y el teorema 1.22, esto es

$$\bigcap_i \{s\} \uparrow (P ; Q_i).$$

Se utiliza ahora el teorema 1.24 y se logra el resultado deseado:

$$\{s\} \uparrow \bigcap_i (P ; Q_i).$$

Supongamos ahora que $\{s\} \uparrow P = \{s_1, s_2, \ldots, s_m\}$. Por el teorema 1.22:

$$\{s\} \uparrow (P ; \bigcap_i Q_i) = (\{s\} \uparrow P) \uparrow (\bigcap_i Q_i).$$

Como $\bigcup_{j \leq m} \{s_j\} = \{s_1, s_2, \ldots, s_m\}$,

$$(\{s\} \uparrow P) \uparrow (\bigcap_i Q_i) = \bigcup_{j \leq m} \{s_j\} \uparrow (\bigcap_i Q_i).$$

Por 1.24:

$$= \bigcap_i (\bigcup_{j \leq m} \{s_j\} \uparrow Q_i)$$

y, por hipótesis,

$$= \bigcap_i ((\{s\} \uparrow P) \uparrow Q_i.$$

Finalmente, por el teorema 1.22 y 1.24,

$$= \{s\} \uparrow \bigcap_i (P ; Q_i).$$

En esta ocasión, también se tiene que $\{s\} \uparrow (P ; \bigcap_i Q_i) = \{s\} \uparrow \bigcap_i (P ; Q_i)$ para todo s. En consecuencia, $P ; (\bigcap_i Q_i)$ y $\bigcap_i (P ; Q_i)$ son la misma relación. ■

Teorema 3.46. Si $\{P_i\}$ es una cadena descendente de relaciones totales finitarias, entonces

$$(\bigcap_i P_i) \setminus R = \bigcup_i (P_i \setminus R).$$

Demostración. Por los teoremas 1.18 y 3.15

$$\overline{(\bigcap_i P_i) \setminus R} = \overline{(\bigcap_i P_i) ; R \setminus \bar{1}} \setminus 1.$$

Con 4.44 se obtiene

$$\overline{\bigcap_i (P_i ; (R \setminus \bar{1}))} \setminus 1.$$

Por A.11 y 4.13 se obtiene

$$\bigcap_i \overline{(P_i ; (R \setminus \bar{1}))} \setminus 1.$$

Finalmente, 1.18, 3.15 nos permiten concluir

$$\bigcap_i \overline{(P_i \setminus R)}.$$

Es obvio que de aquí se deriva la afirmación deseada. ∎

Ahora se ha demostrado que todas las relaciones pertenecientes a $\mathfrak{L}$ son totales finitarias y todos los operadores co-continuos (lo que garantiza funciones recursivas bien construidas). Llegó el momento de presentar los constructos del lenguaje.

3.4.5 *Operadores del lenguaje* $\mathfrak{L}$

Para que $\mathfrak{L}$ tenga algún interés, es necesario que contenga al menos los mismos constructos que se definieron para el lenguaje de la precondición más débil: *abort*, *skip*, IF y DO.

Definición. **skip** $= \{1 - \{(\bot, \bot)\}\}^*$.

La elección de la relación de identidad para el comando **skip** es más o menos obvia: **skip** es un comando que no hace nada, que deja intacto el valor de las variables del programa. Sin embargo, como no se espera que el comando **skip** (¡ni

ningún otro!) se ejecute en el estado $\perp$, eliminamos este caso de la definición. Por otra parte, la preespecificación más débil del comando **skip** aparece en el siguiente teorema.

Teorema 3.47.. Sea R una especificación dada. Entonces

$$\textbf{skip} \setminus R = (((\{\perp\} \times \Sigma) \setminus \emptyset \cap R) \cup ((\overline{\{\perp\} \times \Sigma} \setminus \emptyset) \cap (U \setminus R)).$$

Demostración. Antes que nada, hay que observar que $\overline{\{\perp\} \times \Sigma} \cap I = I - \{(\perp, \perp)\}$. Por tanto

$$\textbf{skip} \setminus R = (\overline{\{\perp\} \times \Sigma} \cap I)^* \setminus R.$$

Y, por 3.31,

$$= (\overline{\{\perp\} \times \Sigma} \setminus \emptyset \cup I \setminus R) \cap (((\{\perp\} \times \Sigma) \setminus \emptyset \cup U \setminus R).$$

Pero $I \setminus P = P$ (teorema 3.9). Si se usan además las leyes de distribución de la unión y la intersección se llega al siguiente resultado

$$\begin{aligned}(((\overline{\{\perp\} \times \Sigma} \setminus \emptyset \cap (\{\perp\} \times \Sigma) \setminus \emptyset) \cup (((\{\perp\} \times \Sigma) \setminus \emptyset \cap R)) \cup \\ (((\overline{\{\perp\} \times \Sigma} \setminus \emptyset \cap U \setminus R) \cup (U \setminus R \cap R)).\end{aligned}$$

Dado que $\overline{\{\perp\} \times \Sigma}$ es una condición, el teorema 3.19 nos dice que $\overline{\{\perp\} \times \Sigma} \setminus \emptyset = \overline{\{\perp\} \times \Sigma} \setminus \emptyset$ y, en consecuencia, $\overline{\{\perp\} \times \Sigma} \setminus \emptyset \cap U \setminus R = \emptyset$. Además, el teorema 3.4 dice que $U \setminus R \subseteq R$ y, entonces, $R \cap U \setminus R = U \setminus R$. En conclusión,

$$(((\{\perp\} \times \Sigma) \setminus \emptyset \cap R) \cup ((\overline{\{\perp\} \times \Sigma} \setminus \emptyset) \cap (U \setminus R)) \quad \blacksquare$$

Corolario 3.48. Si se supone además que $\forall s_0 ((s_0, \perp) \in R \Rightarrow \{s_0\} \uparrow R = \Sigma)$, entonces **skip** $\setminus R = R$.

Demostración. Por 3.9 ya se sabe que $U \setminus R \subseteq R$: se trata de todos los estados tales que R los relaciona con cualquier otro estado. Pero la hipótesis adicional del corolario pide que R cumpla esto para todo $s \in \Sigma$. Por tanto, $U \setminus R = R$. Entonces

$$\begin{aligned}&(((\{\perp\} \times \Sigma) \setminus \emptyset \cap R) \cup \\ &(((\overline{\{\perp\} \times \Sigma} \setminus \emptyset) \cap (U \setminus R)) &&= (((\{\perp\} \times \Sigma) \setminus \emptyset \cap R) \cup (\overline{\{\perp\} \times \Sigma} \setminus \emptyset \cap R) \\ &&&= R \cup (((\{\perp\} \times \Sigma) \setminus \emptyset \cap \overline{\{\perp\} \times \Sigma} \setminus \emptyset) \\ &&&= R. \qquad \text{por 3.31} \quad \blacksquare\end{aligned}$$

Continuemos ahora con el comando **abort**.

Definición. **abort** = U.

Ahora se presenta la preespecificación más débil de **abort**.

Teorema 3.49. **abort** $\setminus R = \{(s_0, s) \mid \forall s' (s_0, s') \in R\}$.

Demostración. Por definición,

$$\textbf{abort} \setminus R = U \setminus R = \{(s_0, s) \mid \forall s_f ((s, s_f) \in U \Rightarrow (s_0, s_f) \in R)\},$$

es decir, todos los estados que R relaciona con Σ relacionados a su vez con Σ también:

$$= \{(s_0, s) \mid \forall s_f ((s_0, s_f) \in R)\}. \quad \blacksquare$$

Definición. Composición secuencial: $P ; Q = P ; Q$.

En este caso, además, la siguiente afirmación sobre la preespecificación de la composición secuencial ya se demostró en el teorema 3.14: $(P ; Q) \setminus R = P \setminus (Q \setminus R)$.

Toca el turno de la asignación:

Definición. Asignación: $(s := f(s)) = \{(s_0, s) \mid s = f(s_0) \wedge s_0 \neq \bot \wedge s \neq \bot\}^+$.

Antes de presentar la preespecificación más débil de la asignación, hay que notar que, en realidad, el conjunto $\{(s_0, s) \mid s = f(s_0) \wedge s_0 \neq \bot \wedge s \neq \bot\}$ es la función f restringida a $\mathfrak{D}(f) - \bot$. Sin embargo, desde el principio se esperaba que f no estuviera definida en $\bot$.

Teorema 3.50. $(s := f(s)) \setminus R = R ; (\tilde{f} \setminus \bar{1}) \cup (f \setminus \emptyset \cap U \setminus R)$.

Demostración. Aplicando la definición de := y la observación anterior

$$(s := f(s)) \setminus R = f^+ \setminus R.$$

Por otro lado, el colorario 3.35, nos dice que lo anterior es

$$= (R ; (\tilde{f} \setminus \bar{1})) \cup ((f \setminus \emptyset) \cap (U \setminus R)),$$

que es el resultado deseado. $\blacksquare$

El siguiente corolario nos da una preespecificación más cercana en la forma a la precondición más débil de Dijkstra:

Corolario 3.51. Si además $U \setminus R = \emptyset$ o f es una función total, entonces

$$(s := f(s)) \setminus R = R ; (\bar{f} \setminus \bar{I}) = \{(s_0, s) \mid (s_0, f(s)) \in R\}.$$

Demostración. Si $U \setminus R$ la afirmación es evidente. Si f es total, entonces $f \setminus \emptyset = \emptyset$ por el teorema 3.32. ■

Definición. $\mathbf{if}\ b \to P \,[]\, c \to Q\ \mathbf{fi} = (b \cap P \cup c \cap Q)^*$.

Teorema 3.52.. $\mathbf{if}\ b \to P \,[]\, c \to Q\ \mathbf{fi} \setminus R = (b \setminus \emptyset \cup P \setminus R) \cap (c \setminus \emptyset \cup Q \setminus R) \cap (\bar{b} \setminus \emptyset \cup \bar{c} \setminus \emptyset \cup U \setminus R)$.

Demostración. Por definición

$$\mathbf{if}\ b \to P \,[]\, c \to Q\ \mathbf{fi} \setminus r = (b \cap P \cup c \cap Q)^*,$$

El teorema 3.31 nos da el resultado deseado. ■

Definición. Sea F el funcional $\lambda X.\, \mathbf{if}\ b \to (P ; X) \,[]\, \neg b \to \mathbf{skip}\ \mathbf{fi}$ y sea f^∞ su máximo punto fijo. Entonces $\mathbf{do}\ b \to P\ \mathbf{od} = f^\infty$.

Aquí b se puede ver como una expresión booleana (cuyo dominio está restringido). $\neg b$ es igual a b, excepto que es falsa en los casos en que b es verdadera y viceversa.

En la mayoría de los lenguajes de programación, siempre es aconsejable transformar las funciones recursivas en cálculos iterativos, por problemas de memoria. Sin embargo, una iteración se puede ver como un caso de recursión, en la que el cuerpo del ciclo se ejecuta cada vez que se el ciclo se llama a sí mismo (en otras palabras, mientras se cumple una condición dada) y se para hasta que se deja de cumplir la condición. Por esta razón, la definición de ciclo en $\mathfrak{L}$ invierte las cosas: un ciclo es un caso especial de recursión.

Teorema 3.53.. Sean b una condición, R una relación en el lenguaje, $H_0 = U \setminus R$ y $H_{n+1} = (b \setminus \emptyset \cup P \setminus H_n) \cap (\neg b \setminus \emptyset \cup \mathbf{skip} \setminus R) \cap (\overline{b \setminus \emptyset} \cup \overline{\neg b \setminus \emptyset} \cup U \setminus R)$. Entonces

$$\mathbf{do}\ b \to P\ \mathbf{od} \setminus R = \bigcup_{n \geq 0} H_n.$$

Demostración. Sea f^{∞} el punto fijo máximo de $F = \lambda X.\,\mathbf{if}\, b \rightarrow (P\,;X)\,[]\,\neg b \rightarrow \mathbf{skip}\,\mathbf{fi}$. Por la definición de $do \ldots od$ y de f^{∞}, f^{∞},

$$(do\, b \rightarrow p\, od) \setminus R = f^{\infty} \setminus R = (\bigcap_{0 \leq n} f^n) \setminus R.$$

Por el teorema 3.46

$$f^{\infty} \setminus R = \bigcup_{0 \leq n} (f^n \setminus R).$$

Basta ahora con probar que $f^m \setminus R = H_m$, lo que se hará por inducción a partir de m.

Sea $m = 0$. Entonces $f^m \setminus R = U \setminus R = H_0$. Supongamos que $f^i \setminus R = H_i$ para $i \leq m - 1$. Por definicición

$$f^m \setminus R = F(f^{m-1}) \setminus R.$$

A su vez, por la definición de F, se tiene que

$$f^m \setminus R = (\mathbf{if}\, b \rightarrow (P\,; f^{m-1})\,[]\,\neg b \rightarrow \mathbf{skip}) \setminus R.$$

Gracias a 3.52, se llega a

$$(b \setminus \emptyset \cup (P\,; f^{m-1}) \setminus R) \cap (\neg b \setminus \emptyset \cup \mathbf{skip} \setminus R) \cap (\overline{b} \setminus \emptyset \cup \overline{\neg b}\emptyset \cup U \setminus R).$$

Por 3.19, esto es igual a

$$(b\emptyset \cup P \setminus H_{m-1}) \cap (\neg b \setminus \emptyset \cup \mathbf{skip} \setminus R) \cap (\overline{b \setminus \emptyset} \cup \overline{\neg b \setminus \emptyset} \cup U \setminus R),$$

lo que es igual a H_m por definición. ∎

3.5 El método de definición de Viena

Hay otros formas de definir la semántica de un lenguaje de tal forma que quede clara la relación entre los estados inicial y final cuando se ejecuta una instrucción. Uno de estas, el *método de definición de Viena* (VDM), tiene la enorme ventaja de que no hace mención del "estado" de no terminación $\perp$, que en realidad es un parche al conjunto de estados. Desde luego, con este método no podemos especificar explícitamente los programas que no terminan, pero ¿para qué queremos

especificar un programa que no da ningún resultado? Para comenzar, veamos una definición *tentativa*:

Definición. Una *tarea de programación* (*programming task*) en VDM es un par de la forma (b, R), donde b es una condición y R es una relación entre estados iniciales y finales *reales*, *i.e.*, $\perp$ *no* se encuentra entre ellos.

Esta definición tiene el defecto de que no está hecha con base en términos ya conocidos. No puede decirse que sea una definición; más bien es un ideal. Lo que pretende es postular que la especificación de una tarea de programación se debe hacer a partir de una relación R que define ciertos estados inicial y final, y que el programa que cumple dicha especificación se debe ejecutar cuando se satisface la condición b (es decir, cuando el estado inicial de las variables está en el dominio de b). Tanto b como R excluyen el estado $\perp$.

El cumplimiento de la restricción fundamental de que b debe cumplirse cuando se ejecuta la parte del programa especificado por R está a cargo del resto del programa. ¿Qué pasa si no se cumple esto? Podría ocurrir cualquier cosa, lo que sugiere una definición nueva y mejor de (b, R):

$$(b, R) = \bar{b} \cup R.$$

En palabras, se trata de R o de cualquier otra cosa si no se cumple b.

La primera pregunta que surge ahora es: ¿se pueden expresar todas las especificaciones posibles de este modo? Desafortunadamente, no. Por ejemplo, $\{(\perp, \perp)\}$ no se puede poner en términos de VDM pues, obviamente, se quiere excluir cualquier referencia a $\perp$. Aunque no fuera así, de todas formas la especificación anterior crea varios problemas serios. Pero si excluimos todos los casos "patológicos" (en los que aparece $\perp$), todas las especificaciones y programas se pueden expresar en VDM, como se demostrará más adelante. Por suerte, los casos patológicos no se necesitan en la vida real.

Hablando con más precisión, se demostrará que a) todos los programas de $\mathfrak{P}$ pueden escribirse en VDM, y que b) dada una relación cualquiera, hay una relación *equivalente* que se puede escribir en VDM. "Equivalencia" quiere decir esto:

Definición. Sean R y S relaciones. Si $\forall P(P \in \mathfrak{P} \Rightarrow (P \subseteq R \Leftrightarrow P \subseteq S))$, entonces R y S son equivalentes.

Para cumplir con a) hay que mostrar que todas las instrucciones de $\mathcal{L}$ pueden ponerse en VDM. Dado que los elementos de $\mathcal{L}$ se definieron inductivamente, basta expresar los constructos básicos de $\mathcal{L}$ en VDM para que la inducción demuestre lo que queremos. Los dos teoremas que siguen se deducen inmediatamente de definiciones anteriores.

Teorema 3.54. $U = (\emptyset, R)$.

Teorema 3.55. $P^+ = ((P\,;U), P)$.

Teorema 3.56. $(b,P)\,;(c,Q) = (b \cap \overline{P\,;\bar{c}}, P\,;Q)$.

Demostración. Por definición

$$(b,P)\,;(c,Q) = (\bar{b} \cup P)\,;(\bar{c} \cup Q),$$

y, por 1.11 y 1.12,

$$= (b\,;(\bar{c} \cup Q)) \cup (P\,;\bar{c}) \cup (P\,;Q).$$

Como b es una condición, $\bar{b}$ también lo es y $\bar{b}\,;(\bar{c} \cup Q) = \bar{b}$. Por tanto,

$$(b,P)\,;(c,Q) = \bar{b} \cup (P\,;\bar{c}) \cup (P\,;Q).$$

Por definición de $\cap$ y $\cup$ y A.11:

$$= \overline{(b \cap \overline{P\,;\bar{c}})} \cup (P\,;Q).$$

$\overline{P\,;\bar{c}}$ es una condición, pues $(P\,;\bar{c})\,;U = P\,;(\bar{c}\,;U) = P\,;\bar{c}$. De esta forma, se llega a la conclusión deseada. ■

Teorema 3.57. $((b \cap (d,P)) \cup (c \cap (e,Q)))^* = ((b \cup c) \cap (\bar{b} \cup d) \cap (\bar{c} \cup e), (b \cap P) \cup (c \cap Q))$.

Demostración. Por definición, se tiene esta igualdad:

$$((b \cap (d,P)) \cup (c \cap (e,Q)))^* = ((b \cap (d,P)) \cup (c \cap (e,Q))) \cup \overline{((b \cap (d,P)) \cup (c \cap (e,Q)))\,;U}.$$

Es claro que (d,P) y (e,Q) son relaciones totales. Entonces, por 3.27 lo anterior es igual a

$$((b \cap (d,P)) \cup (c \cap (e,Q))) \cup (\bar{b} \cap \bar{c}).$$

Por definición, $b \cap (d, P) = b \cap (\bar{d} \cup P)$ y $c \cap (e, Q)$. Después, se recurre a A.10 y A.11 para obtener

$$\overline{(b \cup c) \cap (\overline{b \cup d}) \cap (\overline{c \cup e})} \cup (b \cap P) \cup (c \cap Q).$$

Finalmente, A.11 y la definición de VDM nos dan

$$((b \cup c) \cap (\bar{b} \cup d) \cap (\bar{c} \cup e), (b \cap P) \cup (c \cap Q)). \quad \blacksquare$$

Teorema 3.58. Sea $\{(b_i, P_i)\} \in \mathfrak{Q}$ una cadena descendente. Entonces $\bar{b}_{i+1} \subseteq \bar{b}_i$ y $b_i \cap P_{i+1} \subseteq b_i \cap P_i$.

Demostración. Al ser $\bar{b}_{i+1}$ una condición, $\bar{b}_{i+1} = \bar{b}_{i+1} ; (\{\perp\} \times \Sigma)$. Por su parte, es obvio que $\{\perp\} \times \Sigma = \{\perp\} \times \Sigma \cap U$. El teorema 3.20 establece que

$$\bar{b}_{i+1} = (\bar{b}_{i+1} \cap \overline{\{\perp\} \times \Sigma \setminus \emptyset}) ; U. \tag{34}$$

Por 4.16, $\overline{\{\perp\} \times \Sigma \setminus \emptyset} = \Sigma \times \{\perp\}$. Como $\forall x ((x, y) \in P_{i+1} \Rightarrow y \neq \perp)$, (34) es igual a

$$((\bar{b}_{i+1} \cup P_{i+1}) \cap (\Sigma \times \{\perp\})) ; U. \tag{35}$$

La hipótesis nos dice que

$$(35) \subseteq ((\bar{b}_i \cup P_i) \cap (\Sigma \times \{\perp\})). \tag{36}$$

Por hipótesis también, $P_i \cap (\Sigma \times \{\perp\}) = \emptyset$ y $\bar{b}_i \cap (\Sigma \times \{\perp\}) = \bar{b}_i$. (36) se convierte entonces en

$$\bar{b}_i ; U = \bar{b}_i. \qquad \blacksquare$$

Teorema 3.59. Sea $\{(b_i, P_i)\} \in \mathfrak{Q}$ una cadena descendente. Por tanto, $\bigcap_i (b_i, P_i) = (\bigcup_i b_i, \bigcup_{0 \leq i} \bigcap_{i \leq j} (b_j \cap P_j))$.

Demostración. Utilizando la definición de la especificación en VDM se obtienen las siguientes igualdades:

$$\bigcap_i (b_i, P_i) = \bigcap_i (\bar{b}_i \cup P_i). \tag{37}$$

$$(\bigcup_i b_i, \bigcup_{0 \leq i} \bigcap_{i \leq j} (b_j \cap P_j)) = \bigcap_i \bar{b}_i \cup (\bigcup_{0 \leq i} \bigcap_{i \leq j} (b_j \cap P_j)). \tag{38}$$

Se considerará primero la inclusión de izquierda a derecha. Sea $(s,s') \in \bigcap_i (b_i, P_i)$. Considerando la igualdad (37), hay dos casos.

Caso 1a. $\forall i\,(s,s') \in \tilde{b}_i$. Por tanto, $(s,s') \in \bigcap \tilde{b}_i$ y, por (38),

$$(s,s') \in (\bigcup_i b_i, \bigcup_{0 \le i} \bigcap_{i \le j} (b_j \cap P_j)).$$

Dicho de otra forma,

$$\bigcap_i (b_i, P_i) \subseteq (\bigcup_i b_i, \bigcup_{0 \le i} \bigcap_{i \le j} (b_j \cap P_j)).$$

Caso 2a. $\exists k\,(s,s') \notin \tilde{b}_k$, o sea, $(s,s') \in b_k$. Por 3.58, $(s,s') \in \bigcap_{i \le k} b_i \cap P_i$ y, por (38),

$$(s,s') \in (\bigcup_i b_i, \bigcup_{0 \le i} \bigcap_{i \le j} (b_j \cap P_j)).$$

Ahora veamos la inclusión opuesta. Sea $(s,s') \in (\bigcup_i b_i, \bigcup_{0 \le i} \bigcap_{i \le j} (b_j \cap P_j))$.

Caso 1b. Supongamos que $\forall i\,(s,s') \in \tilde{b}_i$. En este caso, (37) hace obvio que $(\bigcup_i b_i, \bigcup_{0 \le i} \bigcap_{i \le j} (b_j \cap P_j)) \subseteq \bigcap_i (b_i, P_i)$.

Caso 2b. $\exists k\,(s,s') \notin \tilde{b}_k$. En consecuencia, $(s,s') \notin \bigcap_i \tilde{b}_i$. Por la hipótesis y por (38) se deduce que

$$(s,s') \in \bigcup_{0 \le i} \bigcap_{i \le j} (b_j \cap P_j).$$

En este caso, debe existir una m tal que $(s,s') \in (b_j \cap P_j)$ si $m \le j$. Por inducción a partir de m se obtendrá la inclusión de derecha a izquierda del teorema.

Sea $m = 0$. Entonces

$$(s,s') \in \bigcap_{0 \le j} (b_j \cap P_j) \subseteq \bigcap_{0 \le j} P_j.$$

Por (37),

$$(\bigcup_i b_i, \bigcup_{0 \le i} \bigcap_{i \le j} (b_j \cap P_j)) \subseteq \bigcap_i (b_i, P_i).$$

Sigue el caso de $m-1$, que tiene dos posibilidades: $(s,s') \notin b_{m-1}$ y el contrario. En el primero, se tiene que $(s,s') \in \tilde{b}_{m-1}$. Por 3.58, $(s,s') \in \bigcap_{i \le m-1} \tilde{b}_i$. De este hecho se sigue que

$$(s,s') \in \bigcap_{i \le m-1} (b_i, P_i).$$

Pero 3.58 dice que $b_{m-1} \cap P_m \subseteq b_{m-1} P_{m-1}$ y $b_{m-1} \subseteq b_m$. Así, por (37) se concluye que

$$(s,s') \in \bigcap_{i \leq m} (b_i, P_i).$$

Supogamos ahora que $(s,s') \in b_{m-1}$. Entonces, $(s,s') \in \bigcap_{i \leq m-1}$. Por hipótesis y por (37), se obtiene

$$(s,s') \in \bigcup_{0 \leq i \leq j} \bigcap (b_j \cap P_j).$$

En otras palabras,

$$(s,s') \in \bigcap_{i \leq m-1} (b_i \cap P_i).$$

Pero, por 3.58, $b_{m-1} \cap P_m \subseteq b_{m-1} \cap P_{m-1}$ y $b_{m-1} \subseteq b_m$. De este modo,

$$(s,s') \in \bigcap_{i \leq m} (b_i \cap P_i),$$

con lo que se completa la inducción y se demuestra que

$$(\bigcup_i b_i, \bigcup_{0 \leq i \leq j} \bigcap (b_j \cap P_j)) \subseteq \bigcap_i (b_i, P_i)$$ ■

Teorema 3.60. Sea (b,P) un programa de $\mathfrak{D}$. Entonces, $b \subseteq P ; U$.

Demostración. Se sabe que (b,P) es una relación total y, entonces, $(b,P) ; U = U$. Ahora, por definición y 1.11,

$$(\bar{b} \cup U) ; U = (\bar{b} ; U) \cup (P ; U).$$

Como $\bar{b}$ es una condición, se obtiene

$$\bar{b} \cup (P ; U).$$

Y por 1.4 se llega a la conclusión deseada. ■

Los teoremas 3.54–3.55 demuestran que todos los elementos de $\mathfrak{D}$, definidos en los incisos 1–4 de la subsección 3.4.4, se pueden definir dentro de VDM. El teorema 3.60 establece que la condición b de una especificación (b,P) está en el dominio de P: así se evita la situación indeseable de que en algunos casos se cumpla la condición b y P no sepa qué hacer. 3.61, a su vez, nos dice cuál es la precondición más débil de las especificaciones de VDM:

Teorema 3.61. $(b,P) \setminus (c,Q) = (\overline{(c,Q)} : U, \overline{b \setminus \emptyset} \cup (p \setminus (c,Q)))$.

Demostración. ∎

Teorema 3.62. Existe una especificación equivalente en VDM para toda especificación posible.

Demostración. Sea S una especificación. Se quiere demostrar que existen b y R tales que $\forall P(P \in \mathfrak{Q} \Rightarrow (P \subseteq (b,R) \Leftrightarrow P \subseteq S))$.

Definamos el conjunto

$$\hat{\mathfrak{Q}}(S) = \{(c,P) \mid (c,P) \in \mathfrak{Q} \wedge (c,P) \subseteq S\}.$$

A su vez, sean $b = \bigcap\{c \mid (c,P) \in \hat{\mathfrak{Q}}(S)\}$ y $R = \bigcup\{P \mid (c,P) \in \hat{\mathfrak{Q}}(S)\}$. Evidentemente,

$$(b,R) = \bigcup\{(c,P) \mid (c,P) \in \hat{\mathfrak{Q}}(S)\} \subseteq S,$$

y, entonces, si $Q \subseteq (b,P)$ entonces $Q \subseteq S$.

Ahora, sea $Q \in \mathfrak{Q}$ y $Q \subseteq S$. Entonces $Q \in \mathfrak{Q}(S)$ y $Q \subseteq (b,R)$. ∎

De esta forma, la "traducción" de $\mathfrak{Q}$ a VDM evita las referencias explícitas al estado de no terminación, cuya naturaleza es bastante artificial. Asimismo, los instrumentos desarrollados en VDM podrían retomarse para los programas que se escriban en $\mathfrak{Q}$.[4]

3.6 Conclusiones

Algunas de las virtudes de la preespecificación más débil han quedado claras a lo largo de este capítulo. Entre sus cualidades están la claridad y la gran capacidad para varias expresar ideas subyacentes en la semántica de los lenguajes de programación. En buena medida, esto se debe a su estrecha relación con el cálculo de relaciones, que se distingue precisamente por su simplicidad y poder. Pero, además, la integración casi directa de las nociones de recursión contribuye a ampliar el alcance del lenguaje $\mathfrak{Q}$.

[4] El trabajo realizado alrededor de VDM ha llegado a un punto de desarrollo muy avanzado. Con respecto a VDM, puede consultarse Bjørner y Jones [1982].

Por otra parte, la preespecificación más débil también es lo suficientemente flexible para que, a partir de ella, se tienda un puente a otros puntos de vista sobre la semántica. En la sección anterior se expuso como se pueden "transformar" las relaciones pertenecientes a $\mathfrak{Q}$ en especificaciones de VDM.

Aunque no se dio ningún ejemplo al respecto, el lector podría estar de acuerdo en que no parece muy difícil utilizar la preespecificación más débil como una base para construir programas correctos y demostrarlos. Además, el tratamiento matemáticamente homogéneo de las especificaciones y los programas permite que se facilite el subdividir las tareas durante la elaboración de un programa: una tarea puede ser un programa particular ya elaborado o una especificación de un programa no escrito aún. En ambos casos, los "objetos" matemáticos considerados son los mismos, a saber, relaciones.

En resumen, se trata de una herramienta de gran poder, claridad y utilidad práctica en la construcción de lenguajes y programas confiables.

APÉNDICE:
ALGUNOS CONCEPTOS DE LA LÓGICA

A.1 Proposiciones

La parte más elemental de la lógica es el cálculo de proposiciones. Una *proposición* es una variable de tipo booleano (o una expresión construida a partir de variables booleanas), es decir, una variable que puede tener el valor verdadero (*V*) o falso (*F*).[1] Los valores *V* y *F* se conocen genéricamente como valores de verdad.

Al igual que en los lenguajes de programación, las proposiciones, al ser variables, estarán representadas por identificadores. Se usarán las letras *p*, *q* y *r* para identificadores de proposiciones y nos referiremos a ellas indistintamente como identificadores o proposiciones.

La lógica no se interesa por la verdad de los hechos a los que se refieren las proposiones. Por ejemplo, para la lógica es irrelevante que la variable *p* represente la oración "La nieve es blanca" o la oración "La luz viaja a 300 mil km por hora". La lógica se preocupa más bien por cómo afecta la estructura de las oraciones su valor de verdad.

Las proposiciones se pueden combinar por medio de *conectivas lógicas* para formar proposiciones más complejas. Las conectivas lógicas empleadas son: *no* (negación), *y* (conjunción), *o* (disyunción), *si... entonces* (condicional), ... *si y sólo si...* (bicondicional), simbolizadas por $\neg$, $\wedge$, $\vee$, $\Rightarrow$ y $\Leftrightarrow$, respectivamente. El

[1] En este trabajo se usarán los términos "proposición" y "oración" como sinónimos, aunque en algunos textos de lógica (especialmente los relacionados con problemas filosóficos) no lo son. "Enunciado" se reservará para las construcciones propias de un lenguaje de programación.

uso de estas conectivas (y el conjunto de expresiones de este lenguaje) se puede establecer de manera recursiva:

1. Los identificadores $p, q, r, p_1, q_1, r_1, \ldots$, son proposiciones.

2. Si P es una proposición, entonces $\neg P$ es una proposición.

3. Si P y Q son proposiciones, entonces $P \wedge Q$, $P \vee Q$, $P \Rightarrow Q$ y $P \Leftrightarrow Q$ son proposiciones.

4. Una expresión es una proposición sólo si puede demostrarse que lo es por medio de las reglas 1–3.

Una proposición en la que aparece un sólo identificador sin ninguna conectiva lógica es una *proposición simple*. Una proposición en la que aparecen conectivas es una *proposición compuesta*.

Se pueden construir oraciones aún más complejas si se utilizan varias conectivas; por ejemplo, $(p \wedge q) \Rightarrow r$ significa "Si p y q, entonces r". Los paréntesis se usan para eliminar las ambigüedades en la interpretación: a diferencia del ejemplo anterior, $p \wedge (q \Rightarrow r)$ significa "p y si q, entonces r".

El valor de verdad de una proposición compuesta depende de los valores de verdad de sus componentes. Como una oración puede tener dos valores de verdad, si una oración compuesta tiene n componentes simples, hay 2^n combinaciones posibles de valores de verdad de los componentes. Para evaluar una proposición compuesta se puede recurrir a una *tabla de verdad*. En una tabla de verdad hay una columna por cada proposición simple que aparece en la proposición evaluada y una por cada conectiva. Cada renglón indica una combinación distinta de valores de verdad de los componentes. Las tablas de verdad de las oraciones formadas con las conectivas mencionadas anteriormente son las siguientes:

p	q	$p \wedge q$
V	V	V
V	F	F
F	V	F
F	F	F

p	q	$p \vee q$
V	V	V
V	F	V
F	V	V
F	F	F

p	q	$p \Rightarrow q$
V	V	V
V	F	F
F	V	V
F	F	V

p	q	$p \Leftrightarrow q$
V	V	V
V	F	F
F	V	F
F	F	V

p	$\neg p$
V	F
F	V

Para la lógica, estas tablas definen por completo el comportamiento de las conectivas y, en consecuencia, se puede prescindir tranquilamente de cualquier interpretación intuitiva asociada a éstas.

Cuando se tiene una oración con más de una conectiva se calcula primero el valor de los componentes más simples y se continúa con el de los más complejos, hasta llegar al valor de toda la oración.

Una *tautología* es una oración cuyo único valor posible es V. Por ejemplo, la tabla de verdad de la tautología $\neg p \vee p$ es:

p	$\neg$	p	$\vee$	p
V	F		V	
F	V		V	

Una *contradicción* es una oración cuyo único valor de verdad posible es F. Considérese ahora $\neg p \wedge p$:

p	$\neg$	p	$\wedge$	p
V	F		F	
F	V		F	

Un *argumento* es un conjunto de oraciones divido en premisas y conclusión. Un argumento típico es:

a) Todos los hombres son mortales.

b) Sócrates era hombre.

c) Sócrates era mortal.

Aquí a) y b) son las premisas y c) la conclusión. Un argumento es *válido* cuando no puede ocurrir simultáneamente que las premisas sean verdaderas y la conclusión falsa. Por ejemplo, el argumento con premisas $p \Rightarrow q$ y p y conclusión q es válido y esto se representa así: $p \Rightarrow q, p \vdash q$.

Un método mecáncio de verificar que un argumento es válido es éste: sean $p_1, p_2, \ldots, p_n$ las premisas del argumento y sea q su conclusión. El argumento es válido si y sólo si la oración $(p_1 \wedge p_2 \wedge \cdots \wedge p_n) \Rightarrow q$ es una tautología.

Todas las tautología, al igual que todos los argumentos válidos, son teoremas en lógica. Ya se vio un método para verificar las tautologías: las tablas de verdad. En los textos de lógica se dice que este método es de naturaleza semántica,[2] pues recurre a los conceptos de verdad y falsedad. El otro método, en cambio, procede de esta forma. Hay un conjunto finito de tautologías llamadas *axiomas*. A partir de estos axiomas y de ciertas *reglas de deducción* se producen otras oraciones. Estas oraciones son también teoremas (y, por añadidura, tautologías). La regla de deducción más usada es el *modus ponens*, que permite deducir la oración q de las oraciones $p \Rightarrow q$ y p.

Para demostrar que un argumento es válido, se toman como *hipótesis* sus premisas y, entonces, se debe poder deducir la conclusión del argumento. El teorema que se demuestra en este caso, como ya se dijo, tiene la forma $(p_1 \wedge p_2 \wedge \cdots \wedge p_n) \Rightarrow q$.

Este segundo método se conoce como método axiomático y se dice que es de naturaleza sintáctica, pues no se hace mención de los valores de verdad de las oraciones.[3] Para efectos prácticos, ambos métodos son equivalentes.

A continuación se presenta, sin demostración,[4] una lista de teoremas:

1. *Tautologías varias*

A.1 *Ley del tercio excluso*: $p \vee \neg p$.

A.2 *Ley de no contradicción*: $\neg(p \wedge \neg p)$.

A.3 [A.4] $(p \Rightarrow r) \Rightarrow ((p \wedge q) \Rightarrow (r \wedge q))$.

A.4 [A.15] $((p \Rightarrow q) \wedge (\neg p \Rightarrow q)) \Rightarrow q$.

[2] En este contexto, *semántica* tiene un significado muy distinto al que se expuso en la Introducción.

[3] Aquí el término sintaxis tiene un significado más amplio que en la Introducción.

[4] No hay texto de lógica que no los demuestre.

2. *Equivalencias*

La importancia de las equivalencias es que si, por ejemplo, $a \Leftrightarrow b$ es una tautología, al sustitur a por b en una oración como $(a \vee p) \Rightarrow r$ la oración resultante, a saber $(b \vee p) \Rightarrow r$, tiene el mismo valor de verdad que la original.

A.5 *Doble negación*: $\neg\neg p \Leftrightarrow p$.

A.6 *Leyes de idempotencia*: $(p \vee p) \Leftrightarrow p$ y $(p \wedge p) \Leftrightarrow p$.

A.7 [A.1] *Equivalencia entre condicional y disyunción*: $(p \Rightarrow q) \Leftrightarrow (\neg p \vee q)$.

A.8 *Ley de exportación*: [A.3] $((p \wedge q) \Rightarrow r) \Leftrightarrow (p \Rightarrow (q \Rightarrow r))$.

A.9 [A.5] $((p \wedge q) \Rightarrow (p \wedge r)) \Leftrightarrow ((p \wedge q) \Rightarrow r)$.

A.10 [A.8] *Leyes de distributividad*: $(p \wedge (q \vee r)) \Leftrightarrow ((p \wedge q) \vee (p \wedge r))$ y $(p \vee (q \wedge r)) \Leftrightarrow ((p \vee q) \wedge (p \vee r))$.

A.11 [A.16] *Leyes de Morgan*: $\neg(p \wedge q) \Leftrightarrow (\neg p \vee \neg q)$ y $\neg(p \vee q) \Leftrightarrow (\neg p \wedge \neg q)$.

A.12 [A.9] $((p \vee q) \Rightarrow r) \Leftrightarrow ((p \Rightarrow r) \wedge (q \Rightarrow r))$.

A.13 [A.11] *Ley de trasposición*: $(p \Rightarrow q) \Leftrightarrow (\neg q \Rightarrow \neg p)$.

3. *Argumentos*

Recuérdese que si $p_1, p_2, \ldots, p_n \vdash q$, entonces $(p_1 \wedge p_2 \wedge \cdots \wedge p_n) \Rightarrow q$ es una tautología, y viceversa.

A.14 [A.12] *Silogismo disyuntivo*: $(p \vee q), \neg p \vdash q$

A.15 [A.14] *Conjunción*: $p, q \vdash p \wedge q$.

A.16 [A.13] *Separación*: $p \wedge q \vdash p$.

A.17 *Adición*: $p \vdash p \vee q$.

A.2 Cálculo de predicados[5]

El cálculo de proposiciones es una teoría que carece del poder suficiente para demostrar argumentos tan elementales como el que se presentó en la sección anterior sobre Sócrates. La razón es que dicho argumento se basa en la relación

[5] La primera parte de esta sección se basa parcialmente en el capítulo 2 de Mendelson [1964].

entre las *partes* de las oraciones que lo forman y el cálculo de proposiciones solamente toma en cuenta la relación entre las oraciones. El *cálculo de predicados* permite sortear esta dificultad.

Un predicado típico es la relación '<'. En la expresión '$x < y$', '<' relaciona las variables x y y. Los predicados pueden relacionar una variable o n variables. Sea $P^3(x, y, z)$ una fórmula que representa la expresión $x + y < z$. P^3 es un predicado triádico. En general, sea P^n un predicado n-ádico.

Las *variables individuales* son las letras que aparecen en los predicados, como x, y, z, x_1, y_2, etc. Las *constantes* individuales son las letras como $a, a_1, a_2, \ldots$, que tienen un valor que no cambia. f^n es una función con n argumentos cuyo valor es una constante individual. Un término es una de las siguientes expresiones:

a) Las variables y las constantes individuales son términos.

b) Sean f^n una función y $t_1, t_2, \ldots, t_n$ términos. Entonces, $f^n(t_1, t_2, \ldots, t_n)$ es un término.

Las fórmulas del cálculo de predicados, como las del cálculo de proposiciones, se pueden definir también de manera recursiva:

1. Sean P^n un predicado y $t_1, t_2, \ldots, t_n$ términos. Entonces, $P^n(t_1, t_2, \ldots, t_n)$ es una fórmula *atómica*.
2. Las fórmulas atómicas son fórmulas del cálculo de predicados.
3. Si P y Q son fórmulas, $\neg P$, $P \vee Q$, $P \wedge Q$, $P \Rightarrow Q$ y $P \Leftrightarrow Q$ son fórmulas.
4. Si P es una fórmula, $\forall x(P)$ y $\exists x(P)$ son fórmulas. Los símbolos $\forall$ y $\exists$ se llaman *cuantificador universal* y *cuantificador existencial*, respectivamente.
5. Una expresión es una fórmula si es posible construirla a partir de las reglas 1–4.

La interpretación de los cuantificadores es muy simple. En la fórmula $\forall x(P)$ se afirma que, para todo x, vale que P. En la fórmula $\exists x(P)$ se dice que existe al menos un valor de x tal que P. Los cuantificadores se pueden definir uno en función de otro y la regla de equivalencia es la siguiente: $\forall x(P) \Leftrightarrow \neg(\exists x(\neg P))$.

Sea $\forall x(P)$ una fórmula. El *alcance* del cuantificador de la variable x en esta expresión es la fórmula P. La aparición de una variable x en una fórmula está *ligada* si y sólo si x es la variable de un cuantificador o x está dentro del alcance

de un cuantificador que la califica. Si la aparición de una variable no está ligada, entonces es *libre*. Por ejemplo, en $(x < y) \wedge \forall x \exists y (x + z = 5)$, la primera aparición de x es libre, mientras que la segunda y la tercera están ligadas, y aparece libre primero y después ligada. Los términos también pueden estar libres o ligados para una variable. Por ejemplo, sea t el término $x + 1$ y sea P la fórmula $\forall x(x < y)$. En este caso, t *no* es libre para y en P, pues la sustitución de y por t nos da la expresión $\forall x(x < x + 1)$, cuyo significado es radicalmente distinto del de la expresión original. En el caso general, un término t es libre para la variable y en la expresión P si ninguna aparición libre de y en P está en el alcance de un cuantificador de la forma $\forall x$ o $\exists x$, donde x es cualquier variable que aparezca en t.

Sean E una fórmula, t un término y x una variable individual. La notación E_t^x significa que se han sustituido todas las ocurrencias libres de x por el término t. Este proceso se conoce como *sustitución textual*. Por ejemplo, $(x < y)_{(z-a)}^{x} = (z - a) < y$, pues en la expresión $x < y$ la variable x aparece libre y, entonces, se debe sustituir por el término $z - a$.

También se pueden hacer cambios de variables ligadas por otras variables individuales. En este caso, se cambian todas las apariciones de una variable ligada por *una sola* variable nueva, es decir, una variable que no aparezca en el alcance del cuantificador de la variable que se está sustituyendo.

La fórmula $x < y$ no es verdadera ni falsa, pues su valor de verdad depende del valor que tomen las variables x y y. Algunos valores la hacen verdadera si las variables x y y se sustituyen por ellos, como cuando x se cambia por 1 y y por 2. En este caso, se dice que 1 y 2 satisfacen la expresión $x < y$. Una fórmula del cálculo de predicados es verdadera cuando la satisfacen todos los valores posibles para sus variables. Si una fórmula no es satisfecha por ningún valor posible es falsa. Por ejemplo, $x \cdot 0 = 0$ es verdadera, ya que todos los valores posibles de x la satisfacen. En cambio, $x \cdot 0 = 1$ es falsa. (Como se puede apreciar, una gran cantidad de fórmulas del cálculo de predicados no son ni verdaderas ni falsas.)

Es imposible demostrar que una fórmula es verdadera por inspección de los valores que pueden tener sus variables. Por tanto, hay que desechar las demostraciones semánticas (por tablas de verdad) para el cálculo de predicados y quedarse sólo con las demostraciones sintácticas. No obstante, las reglas de deducción del cálculo de proposiciones y sus teoremas siguen siendo válidos. Los siguientes

son algunas reglas de deducción y algunos axiomas particulares del cálculo de predicados.[6]

A.18 Sean $P(x)$ es una fórmula y t un término libre para x en $P(x)$. Entonces, $(\forall x P(x)) \Rightarrow P(t)$.

A.19 $(\forall x(P \Rightarrow Q)) \Rightarrow (P \Rightarrow \forall x Q)$, siempre y cuando P no contenga apariciones libres de x.

A.20 *Regla de generalización*. De la fórmula P se puede deducir $\forall x(P)$.

A.21 *Regla de particularización*. Si t está libre para x en $P(x)$, entonces $\forall x P(x) \vdash P(t)$.

A.22 *Regla de generalización existencial*. Si t está libre para x en $P(x)$, entonces $P(t) \vdash \exists x\, P(x)$. Además, de $P(x)$ se deduce que $\exists x\, P(x)$.

El siguiente enunciado se tomó como una definición posible de $\exists$ en función de $\forall$:

A.23 $\forall x(P) \Leftrightarrow \neg(\exists x(\neg P))$

Nuevamente, ésta es una lista de teoremas sin demostrar:[7]

A.24 $\forall x \forall y(P) \Rightarrow \forall y \forall x(P)$.

A.25 [A.10] $P, (\forall x\, P) \Rightarrow Q \vdash (\forall x Q)$.

A.26 $(\forall x(P \Rightarrow Q)) \Leftrightarrow (P \Rightarrow (\forall x\, Q))$.

A.27 $(\forall x(P \Rightarrow Q)) \Leftrightarrow ((\exists x P) \Rightarrow Q)$.

A.28 $(\forall x(P \Rightarrow Q)) \Rightarrow (\forall x\, P \Rightarrow \forall x\, Q)$.

A.29 $(\forall x(P \Rightarrow Q)) \Rightarrow (\exists x\, P \Rightarrow \exists x\, Q)$.

A.30 $(\forall x(P \wedge Q)) \Leftrightarrow (\forall x\, P \wedge \forall x\, Q)$.

A.31 $(\forall x\, P) \Rightarrow (\exists x\, P)$.

A.32 [A.6] *Distributividad de la sustitución textual*: Sean P y Q fórmulas y t un término. Entonces $(P_t^x \wedge Q_t^x) \Leftrightarrow (P \wedge Q)_t^x$.

[6] En sentido estricto, son axiomas y reglas de deducción en *una versión* del calculo de predicados.

[7] Para las demostraciones de casi todos puede consultarse el capítulo 2 de Mendelson [1964].

BIBLIOGRAFÍA

Backhouse, Roland C., 1986. *Program Construction and Verification*, Series in Computer Science, Prentice-Hall (UK), Londres.

Bjørner, Dines y Cliff B. Jones, 1982. *Formal Specification and Software Development*, Series in Computer Science, Prentice-Hall (Estados Unidos).

Brookshear, J. Glenn, 1989, *Theory of Computation. Formal Languages, Automota and Complexity*, The Benjamin/Cumming Publishing Company, Nueva York.

Devlin, 1979, *Fundamentals of Set Theory*, Springer Verlag, Nueva York.

Dijkstra, Edsger W., 1976, *A Discipline of Programming*, Series in Automatic Computation, Prentice-Hall, Englewood Cliffs, New Jersey.

Enderton, H.B., 1987, *Una introducción matemática a la lógica*, Universidad Nacional Autónoma de México-Instituto de Investigaciones Filosóficas, México.

Floyd, R., 1967, "Assigning meaning to programs", en *Mathematical Aspects of Computer Science*, XIX American Mathematical Society, pp. 19–32. [Citado por Gries, 1981.]

Garey, Michael R. y David S. Johnson, 1979, *Computers and Intractability. Guide to the Theory of NP-Completeness*, W.H. Freeman and Company, Nueva York.

Gries, David, 1981, *The Science of Programming*, Texts and Monographs in Computer Science, Springer-Verlag, Nueva York.

Hoare, C.A.R. y He Jifeng, 1986, "The Weakest Prespecification", *Fundamenta Informaticae*, IX, no. 1 y 2, pp. 51–84 y 217–252.

Lolli, Gabriele, 1987, *La máquina y las demostraciones*, Alianza Editorial, Madrid.

Lucas, 1982, "Main Approaches to Formal Specifications", en Bjørner y Jones [1982].

Mendelson, Elliot, 1964, *Introduction to Mathematical Logic*, D. Van Nostrand, Nueva Jersey.

Meyer, Bertrand, 1990, *Introduction to the Theory of Programming Languages*, Series in Computer Science, Prentice-Hall, Nueva York.

Rogers, Hartley, *Theory of Recursive Functions and Effective Computability*.

Tarski, Alfred, 1941, "On the Calculus of Relations", *The Journal of Symbolic Logic*, vol. 6, no. 3, septiembre.