2.2 Lógica de Hoare.

La lógica de Hoare es una extensión de la lógica de predicados de primer orden para razonar sobre la corrección de programas imperativos construidos sobre una signatura (S, Σ). Esta extensión se obtiene introduciendo un lenguaje de comandos con el que se construyen los programas y una relación especial para expresar el comportamiento de un programa, así como un conjunto de reglas de cálculo para la manipulación de las expresiones de comportamiento.

2.2.1 Universo de programas

Sintaxis

Cada programa se construye a partir de una signatura (S, Σ) correspondiente a los tipos de datos con los que opera, un lenguaje de comandos y una familia de variables V o *variables de programa*. Generalmente se supone que la signatura consta de los universos $BOOL$ y NAT, correspondientes a los valores lógicos $\{V, F\}$ y a los números naturales junto con las operaciones propias de estos universos, más otros posibles universos correspondientes a tipos de datos definidos por el programador. En todos supondremos definida una relación de igualdad '=' que nos permite establecer una equivalencia entre términos del tipo correspondiente a dicho universo, con unos axiomas y unas reglas de cálculo.

El *lenguaje de comandos* que vamos a utilizar para construir programas es un lenguaje elemental que consta de unos comandos simples y unos comandos compuestos con los cuales se construyen los términos de un universo especial: el *universo de programas* PRG. Los *comandos simples* son:

- $\text{SKIP}: \rightarrow PRG$, o comando *vacío*, y
- $(\cdot :=_s \cdot): V_s\ s \rightarrow PRG$, uno por cada universo $s \in S$, o comando de *asignación* de valor a variables del tipo s, donde el primer argumento es una variable de programa del tipo s y el segundo una expresión del mismo tipo. En adelante supondremos que las asignaciones se hacen con los tipos correctos y prescindiremos del subíndice s;

y los *comandos compuestos*:

- $(\text{IF} \cdot \text{THEN} \cdot \text{ELSE} \cdot \text{END}): BOOL\ PRG\ PRG \rightarrow PRG$, o comando de *selección simple*, donde el primer argumento es un término del tipo $BOOL$ y los otros dos son programas;
- $(\text{WHILE} \cdot \text{DO} \cdot \text{END}): BOOL\ PRG \rightarrow PRG$, o comando de *iteración*, donde el primer argumento es un término del tipo $BOOL$ mientras que el segundo es un programa, y

- $(\cdot\ ;\ \cdot)$: $PRG\ PRG \to PRG$, o comando de *composición secuencial* de programas.

Ejemplo 2.4 *Con estos comandos y una signatura correspondiente al álgebra de los números naturales con las operaciones aritméticas y las relaciones usuales se pueden construir términos (programas) como el siguiente:*

```
X:=1;
WHILE Z>0 DO
  IF (Z mod 2)=0 THEN
    Y:=(Y*Y); Z:=(Z div 2)
  ELSE
    X:=(Y*X); Z:=Z-1
  END
END
```

donde se han observado las normas habituales de sangrado con objeto de facilitar su legibilidad.

Semántica

Los términos del universo de programas no se valoran de la misma forma que los términos correspondientes a los universos de datos, es decir, no se valoran sobre un simple dominio en una interpretación A de la signatura correspondiente, sino que toman valores sobre un dominio especial: el conjunto $[STA_A(V) \not\to STA_A(V)]$ de los transformadores parciales de estados definidos sobre A. Así, cada programa α se valora como un transformador de estados, escrito $\cdot\,[\alpha]_A\,\cdot$, según su estructura, de la siguiente forma:

- $\cdot\,[\mathrm{SKIP}]_A\,\cdot$, transforma cada estado *sta* en sí mismo, es el transformador identidad;
- $\cdot\,[X := t]_A\,\cdot$, transforma cada estado *sta* en el estado $sta(X/val_{A,sta}(t))$; es decir, modifica el estado en la variable X asignándole el valor de t en el estado anterior;
- $\cdot\,[\mathrm{IF}\,t_B\,\mathrm{THEN}\,\alpha\,\mathrm{ELSE}\,\beta\,\mathrm{END}]_A\,\cdot$, para cada estado *sta* se comporta de acuerdo con el criterio siguiente: si $A \models_{sta} t_B = V$ lo transforma del mismo modo que lo hace $\cdot\,[\alpha]_A\,\cdot$ y si $A \not\models_{sta} t_B = V$ lo transforma del mismo modo que lo hace $\cdot\,[\beta]_A\,\cdot$;
- $\cdot\,[\mathrm{WHILE}\,t_B\,\mathrm{DO}\,\alpha\,\mathrm{END}]_A\,\cdot$, transforma un estado *sta* en otro sta' sólo cuando existe una secuencia finita de estados $sta_0, \cdots, sta_p$ tales que para $0 \le i < p$, $A \models_{sta_i} t_B = V$ y $sta_i[\alpha]_A sta_{i+1}$, mientras que $A \not\models_{sta_p} t_B = V$; en tal caso será $sta' = sta_p$;

- $\cdot\ [\alpha\ ;\ \beta]_A\ \cdot$, transforma un estado *sta* en otro sta' sólo cuando existe un estado sta'' tal que $sta[\alpha]_A sta''$ y $sta''[\beta]_A sta'$.

Como vemos, puede haber programas a los que les corresponda un transformador parcial; es decir un transformador que no produzca imágenes para algunos estados; serán programas en los que aparezcan comandos de iteración. Cuando para un programa α y una interpretación A se cumple $sta[\alpha]_A sta'$ se dice que dicho programa cuando parte del estado *sta* termina y alcanza el estado sta' en la interpretación A.

Ejercicio 2.2 *Determinar los transformadores que corresponden, en el álgebra* $\boldsymbol{N}$ *de los números naturales, a los programas siguientes:*

- X:=1; Y:=(X+1),
- IF (X > 0) THEN X:=(X-1) ELSE SKIP END
- WHEN (X <> 0) DO X:=(X-1) END

2.2.2 Sentencias de corrección parcial

En un programa real, aparecen declaraciones de tipo y declaraciones de variables así como implementaciones de funciones (pueden ser implementaciones incorporadas al sistema o hechas por el usuario) que determinan la signatura del programa, además el segundo argumento de un comando de asignación puede ser cualquier expresión del correspondiente tipo s construida con constantes y funciones implementadas, y el primer argumento de un comando de selección o de iteración puede ser cualquier expresión del tipo *BOOL*. En nuestros programas, aunque no vamos a considerar declaraciones de tipo ni de variables, supondremos que existe una declaración de signatura donde aparecen los tipos y las constantes y funciones necesarias con propiedades conocidas, así como una asignación de tipo a cada variable de programa.

Para expresar el comportamiento de un programa normalmente se recurre a una signatura (S, Σ) más amplia que la que se deriva del propio programa, incorporando tipos y funciones auxiliares que no aparecen en el programa (pero que deben estar debidamente especificados), y a una familia de variables V, también más amplia que la familia de variables de programa incorporando variables nuevas o *variables lógicas* que tampoco aparecen en el programa, y se introduce una relación especial

$$\{\cdot\}\cdot\{\cdot\}\colon F_\Sigma(V)\ PRG\ F_\Sigma(V)$$

con la que se expresan *fórmulas o sentencias de corrección parcial*. Al primer argumento se le llama *precondición* y al tercero *postcondición*; estos argumentos están sujetos a una restricción sobre las variables de programa: dichas variables sólo deben tener apariciones libres o, dicho de otro modo, las variables de programa no deben aparecer cuantificadas.

La validez de estas fórmulas se define del siguiente modo: $\{P\}\alpha\{Q\}$ es válida en una interpretación A, para un estado *sta*, o $A \models_{sta} \{P\}\alpha\{Q\}$[2], si y sólo si

$$A \models_{sta} P \text{ y } sta[\alpha]_A sta' \text{ implica } A \models_{sta'} Q$$

Esto significa que $\{P\}\alpha\{Q\}$ será válida cuando la validez de la precondición P en el estado *sta* y el hecho de que cuando α parte del estado *sta* termina en un cierto estado sta' implique que la postcondición Q es válida para dicho estado final sta'. De esta definición se deduce que una fórmula $\{P\}\alpha\{Q\}$ puede ser válida en una interpretación A, para un estado *sta* por alguna de la circunstancias siguientes:

- porque $A \models_{sta} P$ y α cuando parte del estado *sta* termina en un estado sta' tal que $A \models_{sta'} Q$;
- porque $A \models_{sta} P$, pero cuando α parte del estado *sta* no termina (el transformador $\cdot[\alpha]_A\cdot$ no está definido en *sta*), o
- porque $A \not\models_{sta} P$;

lo que significa que la validez de una fórmula de corrección parcial no implica la terminación del correspondiente programa, y, en algunos casos, ni siquiera la validez de la precondición para el estado que se considera. $\{P\}\alpha\{Q\}$ se dice que es válida en la interpretación A cuando lo es para todos los estados definibles sobre A, en cuyo caso se escribe $A \models \{P\}\alpha\{Q\}$ y se dice que el programa α es *parcialmente correcto* con respecto a la especificación (P, Q). También se dice que (P, Q) es una *especificación parcial* de α.

Ejercicio 2.3 *Demuéstrese que para cualquier programa α y cualesquiera fórmulas P y Q, las fórmulas de corrección parcial $\{\boldsymbol{F}\}\,\alpha\,\{Q\}$ y $\{P\}\,\alpha\,\{\boldsymbol{V}\}$, son siempre válidas, siendo $\boldsymbol{V}$ un átomo que siempre es válido en todas las interpretaciones y $\boldsymbol{F}$ un átomo que siempre es falso.*

Ejercicio 2.4 *Demuéstrese que si en un álgebra A son válidas las fórmulas $P' \Rightarrow P$ y $\{P\}\alpha\{Q\}$, entonces también es válida la fórmula $\{P'\}\alpha\{Q\}$. Demuéstrese igualmente que si son válidas las fórmulas $Q \Rightarrow Q'$ y $\{P\}\alpha\{Q\}$, entonces también es válida la fórmula $\{P\}\alpha\{Q'\}$.*

Ejercicio 2.5 *Demuéstrese que si en un álgebra A son válidas las fórmulas de corrección parcial $\{P\}\alpha\{Q_i\}$, para i=1,2, también lo será la fórmula $\{P\}\alpha\{Q_1 \wedge Q_2\}$.*

Ejercicio 2.6 *Demuéstrese que si en un álgebra A son válidas las fórmulas de corrección parcial $\{P_i\}\alpha\{Q\}$, para i=1,2, también lo será la fórmula $\{P_1 \vee P_2\}\alpha\{Q\}$.*

[2]en adelante utilizaremos letras mayúsculas para representar pre y postcondiciones y letras griegas para representar programas

2.2.3 Diseño de especificaciones de programas

Cada fórmula $P \in F_\Sigma(V)$ caracteriza en cada interpretación A a un conjunto de estados: el conjunto $\{sta \mid A \models_{sta} P\}$ formado por todos aquellos estados en los que la fórmula P es válida, y la implicación $P \Rightarrow Q$ entre fórmulas se traduce en una relación de contenido

$$\{sta \mid A \models_{sta} P\} \subseteq \{sta \mid A \models_{sta} Q\}$$

entre los conjuntos de estados caracterizados por cada fórmula.

La validez de una fórmula de corrección parcial $\{P\}\alpha\{Q\}$ nos asegura que si el programa α termina, partiendo de un estado caracterizado por P, alcanza un estado caracterizado por Q. Pensando en esta interpretación, podemos utilizar especificaciones parciales (P, Q) para expresar comportamientos de programas, dando una precondición P que caracterice a todos los estados (y sólo a ellos) que el programa puede transformar correctamente y una postcondición Q que caracterice a los estados transformados, y la corrección de un programa α respecto a una especificación (P, Q) se enunciará mediante la correspondiente fórmula de corrección parcial $\{P\}\alpha\{Q\}$.

Las especificaciones parciales, también conocidas como especificaciones e/s (de la entrada y la salida), se suelen aplicar tanto a programas construidos, con objeto de sintetizar su comportamiento, como a programas sin construir, para conducir su construcción. En cualquiera de los casos, el comportamiento (real o esperado) del programa se debe precisar al máximo dando la precondición más amplia posible —que refleje el máximo rango de aplicabilidad del programa— y la postcondición más restrictiva —para determinar al máximo la tarea del programa— aunque sin comprometerse con una forma concreta de cálculo; sobre todo cuando la especificación se utiliza para construir el programa, pues así permite distintas implementaciones equivalentes.

A la hora de construir una especificación para un programa α, se procede del siguiente modo:

1. Se identifica el cálculo que debe realizar α, expresándolo como un predicado Q sobre el estado (los valores de las variables) en el que termina dicho programa, ayudándose posiblemente de variables lógicas y funciones auxiliares (no implementadas) de manera que Q refleje exactamente los estados que se quiere alcanzar. En la postcondición Q se suelen incluir valores finales de algunas variables (p.e. en los cálculos de operaciones), relaciones entre los valores finales y los valores iniciales de ciertas variables (p.e. en los intercambios de valores, ordenaciones, ...) o el estado final de alguna estructura (p.e. estructuras ordenadas o invariantes).

2. Se fijan, mediante otro predicado P, los estados de los que puede partir el programa (es decir, las condiciones de aplicabilidad) para que se cumpla Q en el estado final producido por la transformación definida por el programa. En la precondición P se suelen incluir los tipos asociados a las variables de programa

(que deben permanecer invariantes en la transformación asociada al programa y sólo aparecen en la precondición), los valores asociados a dichas variables, dados mediante variables lógicas para facilitar la expresión de relaciones entre valores iniciales y finales de dichas variables, límites sobre algunos valores, p.e. el tamaño mínimo para el que se resuelve un problema (los tamaños máximos dependen de las máquinas concretas y no se suelen fijar en programas generales), o el estado de alguna estructura (p.e. la ordenación de un vector en un programa de búsqueda).

Veamos algunos ejemplos de diseño de especificaciones:

Ejemplo 2.5 *En el álgebra* $\boldsymbol{N}$ *de los números naturales,*

- *un programa para calcular el producto de dos números tendría una especificación de la forma:*

$$(\{a, b, c : NAT, a = X, b = Y\}, \{c = X * Y\}),$$

donde a*,* b *y* c *son variables de programa mientras que* X *e* Y *son variables lógicas que representan los valores iniciales de* a *y* b *respectivamente;*

- *un programa para intercambiar los valores de dos variables tendría la especificación:*

$$(\{a, b : NAT, a = X, b = Y\}, \{a = Y, b = X\});$$

- *y un programa para calcular la raíz cuadrada entera de un número:*

$$(\{a, r : NAT, a = X\}, \{r * r \leq X < (r + 1) * (r + 1)\}).$$

En estas especificaciones aparece clara la necesidad de introducir variables lógicas (con letras mayúsculas)[3] para representar los valores iniciales de las variables de programa y poder expresar su relación con los valores finales (razonar qué ocurriría si no se estableciese esta relación); además se utiliza la ',' en lugar de la conjunción $\wedge$ con objeto de hacer más legibles las fórmulas.

Para facilitar la expresión de pre y postcondiciones se suelen utilizar cuantificadores, no sólo los ya conocidos $\forall$ y $\exists$, sino otros derivados de la generalización de cualquier operación binaria $\oplus$ asociativa, conmutativa y con elemento neutro ϵ, para la que el resultado de su aplicación reiterada sobre los resultados de aplicar una expresión $e(X)$ a todos los valores $X \in I$ se escribe de manera sintetizada en la forma $\bigoplus_{X \in I} e(X)$, representando la expresión $e(t_1) \oplus \cdots \oplus e(t_n)$ cuando $I = \{t_1, \cdots, t_n\}$, o ϵ cuando

[3] en adelante usaremos letras mayúsculas para representar variables lógicas y minúsculas para las variables de programa

$I = \emptyset$. Cuando I es un intervalo $[a, b]$ de un conjunto bien ordenado se suele escribir $\bigoplus_{X=a}^{b} e(X)$, y si el intervalo es numérico y no vacío se cumple:

$$\bigoplus_{X=a}^{b} e(X) = (\bigoplus_{X=a}^{b-1} e(X)) \oplus e(b) = e(a) \oplus (\bigoplus_{X=a+1}^{b} e(X)).$$

Como generalización de las operaciones con predicados se usan las cuantificaciones

- $\forall X \in I \cdot P(X)$, generalización de la operación $\wedge$ —también se suele escribir $\bigwedge_{X \in I} P(X)$—, equivalente a la expresión $\forall X \cdot (X \in I \Rightarrow P(X))$, que cuando $I = \emptyset$ equivale al átomo **V** siempre cierto;

- $\exists X \in I \cdot P(X)$, generalización de la operación $\vee$ —también se suele escribir $\bigvee_{X \in I} P(X)$—, equivalente a la expresión $\exists X \cdot (X \in I \wedge P(X))$, que cuando $I = \emptyset$ equivale al átomo **F** siempre falso.

Ejemplo 2.6 *En el álgebra* $\boldsymbol{N}$ *de los números naturales ampliada con vectores de números,*

- *un programa para buscar un número dentro de un vector, con una variable para recoger el resultado, tendría la siguiente especificación:*

$$\begin{array}{ll} (& \{\ X, N\colon NAT,\ esta\colon BOOL,\ b, B\colon ARRAY\,[0..N]\ OF\ NAT,\ b = B\ \}, \\ & \{\ esta = \exists Y \in [0..N] \cdot b[Y] = X,\ b = B\ \}\ \ ); \end{array}$$

- *y un programa para ordenar todas las componentes de un vector necesitaría una función auxiliar permutacion*(A, B) *—no necesariamente implementada, aunque sí debidamente especificada— para indicar que los vectores* A *y* B *son permutación uno del otro y así poder expresar adecuadamente la postcondición de la especificación correspondiente:*

$$\begin{array}{ll} (& \{\ N\colon NAT,\ b, B\colon ARRAY\,[0..N]\ OF\ NAT,\ b = B\ \}, \\ & \{\ \forall X \in [1..N] \cdot b[X-1] = b[X],\ permutacion(b, B)\ \}\ \ ). \end{array}$$

También se suelen generalizar operaciones aritméticas como la suma, el producto, el máximo o el mínimo, así,

- $\sum_{X \in I} e(X)$, o $\sum\{e(X) \mid X \in I\}$, es la generalización de la suma, equivale a $e(t_1) + \cdots + e(t_n)$ cuando $I = \{t_1, \cdots, t_n\}$ y a 0 cuando $I = \emptyset$;

- $\prod_{X \in I} e(X)$, o $\prod\{e(X) \mid X \in I\}$, es la generalización del producto, equivale a $e(t_1) * \cdots * e(t_n)$ cuando $I = \{t_1, \cdots, t_n\}$ y a 1 cuando $I = \emptyset$;

- $Max_{X \in I} e(X)$, o $Max\{e(X) \mid X \in I\}$ es la generalización del máximo de dos números, equivale a la expresión $max(t_1, max(\cdots max(t_{n-1}, t_n) \cdots))$ cuando $I = \{t_1, \cdots, t_{n-1}, t_n\}$ y representa el mínimo valor del dominio, $\bot$, cuando $I = \emptyset$;

- $Min_{X \in I} e(X)$, o $Min\{e(X) \mid X \in I\}$, es la generalización del mínimo de dos números, equivale a la expresión $min(t_1, min(\cdots min(t_{n-1}, t_n) \cdots))$ cuando $I = \{t_1, \cdots, t_{n-1}, t_n\}$ y representa el máximo valor del dominio, $\top$, cuando $I = \emptyset$;

Ejemplo 2.7 *En el álgebra del ejemplo anterior,*

- *un programa para sumar todas las componentes de un vector, sin alterarlas, se especificaría:*

$$\begin{array}{ll} (& \{\ s, N\colon NAT,\ b, B\colon ARRAY\ [0..N]\ OF\ NAT,\ b = B\ \}, \\ & \{\ s = \sum_{X=0}^{N} b[X],\ b = B\ \}\ ); \end{array}$$

- *y un programa para calcular el máximo valor de un vector:*

$$\begin{array}{ll} (& \{\ max, N\colon NAT,\ b, B\colon ARRAY\ [0..N]\ OF\ NAT,\ b = B\ \}, \\ & \{\ max = Max_{X=0}^{N} b[X],\ b = B\ \}\ ). \end{array}$$

2.2.4 Cálculo de Hoare

Para demostrar la validez de las fórmulas de corrección de programas se recurre a la ampliación del cálculo para la deducción natural —apto para tratar sólo con fórmulas del cálculo de predicados de primer orden— incorporando un conjunto de reglas para la deducción de la validez de fórmulas de corrección parcial conocido como *cálculo de Hoare*. Este cálculo consta de una regla general independiente de la estructura del programa que aparece en la fórmula de corrección, que refleja el significado de las fórmulas de corrección parcial, y una serie de reglas, una por cada comando, ligadas a la estructura del programa. Con estas últimas reglas se intenta plasmar el efecto de cada comando del lenguaje de programación, por lo que constituyen una auténtica semántica para dicho lenguaje conocida como *semántica axiomática*. Las regla independiente es:

- Regla de refinamiento:

$$\frac{\vdash\ P \Rightarrow P';\ \vdash\ \{P'\}\alpha\{Q'\};\ \vdash\ Q' \Rightarrow Q}{\vdash\ \{P\}\alpha\{Q\}}$$

  Esta regla refleja realmente la semántica de las fórmulas de corrección parcial (véase el ejercicio 2.4) y nos viene a decir que siempre se puede reforzar la precondición y/o debilitar la postcondición de una fórmula de corrección válida.

- Axioma para el programa vacío:

$$\frac{}{\vdash \{P\}\text{SKIP}\{P\}}$$

Este axioma refleja la semántica del programa vacío como transformador identidad.

- Axioma para la asignación:

$$\frac{}{\vdash \{P\{X/t\}\}X := t\{P\}}$$

cuando la sustitución $P\{X/t\}$ sea admisible y el cálculo de la expresión t termine. Este axioma refleja el hecho de que después de ejecutar un comando $X := t$, el valor de la variable X se iguala al valor de t, luego si se quiere que la fórmula P sea válida después de la asignación, antes debe ser válida la misma fórmula para t en lugar de X.

- Regla para la selección:

$$\frac{\vdash \{P \wedge B\}\beta\{Q\};\ \vdash \{P \wedge \neg B\}\gamma\{Q\}}{\vdash \{P\}\,\text{IF}\,B\,\text{THEN}\,\beta\,\text{ELSE}\,\gamma\,\text{END}\,\{Q\}}$$

Esta regla refleja la semántica del comando correspondiente y nos dice que para que una construcción con este comando sea correcta respecto a una especificación (P, Q), lo debe ser cada uno de los programas alternativos β y γ respecto a las especificaciones obtenidas reforzando la precondición con las condiciones necesarias para que se ejecuten dichos programas.

- Regla para la repetición:

$$\frac{\vdash \{P \wedge B\}\beta\{P\}}{\vdash \{P\}\,\text{WHILE}\,B\,\text{DO}\,\beta\ \text{END}\,\{P \wedge \neg B\}}$$

Esta regla nos dice que si P es una fórmula que se mantiene cada vez que se ejecuta el cuerpo del ciclo con las condiciones adicionales dadas por B, entonces P se mantendrá invariante para cualquier número de repeticiones del ciclo y se cumplirá al final junto con la condición de terminación del ciclo. A estas fórmulas P se les llama *invariantes* del ciclo. La aplicación de la regla exige que se determine un invariante P conveniente para la corrección que se quiera deducir.

- Regla para la composición:

$$\frac{\vdash \{P\}\alpha\{R\};\ \vdash \{R\}\beta\{Q\}}{\vdash \{P\}\alpha;\beta\{Q\}}$$

Esta regla refleja también la semántica de la composición de programas. Su aplicación para deducir una corrección exige que se determine una fórmula R para la que se cumplan las premisas.

Las reglas de este cálculo se utilizan para derivar sentencias de corrección parcial $\{P\}\alpha\{Q\}$ a partir de una teoría axiomática Ax (conjunto de fórmulas cerrado bajo consecuencia lógica) relativa a los universos de datos. Cuando dicha derivación es posible, se escribe

$$Ax \vdash_{CH} \{P\}\alpha\{Q\}.$$

Ejemplo 2.8 *En el álgebra* $\boldsymbol{N}$ *de los números naturales podemos considerar la fórmula de corrección parcial* $\{x = N\}x := x + 1\{x = N + 1\}$ *donde* x *y* N *son variables de tipo NAT y* N *es una variable lógica que representa un valor; su deducción se puede hacer de la siguiente forma, aplicando la regla de refinamiento y el axioma para la asignación,*

$$\frac{\vdash \{x = N\}x := x + 1\{x = N + 1\}}{\vdash x = N \Rightarrow x + 1 = N + 1; \qquad \vdash \{x + 1 = N + 1\}x := x + 1\{x = N + 1\}}$$

donde la deducción se leería de abajo hacia arriba, teniendo en cuenta que $x = N \Rightarrow x + 1 = N + 1$ *es deducible en la teoría correspondiente a* $\boldsymbol{N}$*.*

También podemos considerar la fórmula $\{\boldsymbol{V}\}x := N\{x = N\}$ *cuya deducción sería*

$$\frac{\vdash \{\boldsymbol{V}\}x := N\{x = N\}}{\dfrac{\vdash \boldsymbol{V} \Rightarrow N = N; \qquad \boldsymbol{V} \vdash N = N}{\vdash N = N} \qquad \vdash \{N = N\}x := N\{x = N\}}$$

En esta fórmula se comprueba que cuando a una variable se le asigna un valor independiente de su valor anterior la postcondición sólo depende del nuevo valor.

La deducción de la fórmula de corrección $\{x = X \wedge y = Y\}\, r := x;\, x := y;\, y := r\, \{x = Y \wedge y = X\}$ *donde aparece una secuencia de asignaciones se haría del siguiente modo*

$$\frac{\vdash \{x = X \wedge y = Y\}\, r := x;\, x := y;\, y := r\, \{x = Y \wedge y = X\}}{\dfrac{\vdash \{x=X\wedge y=Y\}\, r:=x;x:=y\, \{x=Y\wedge r=X\}; \qquad \vdash \{x=Y\wedge r=X\}\, y:=r\, \{x=Y\wedge y=X\}}{\vdash \{x=X\wedge y=Y\}\, r:=x\, \{y=Y\wedge r=X\}; \quad \vdash \{y=Y\wedge r=X\}\, x:=y\, \{x=Y\wedge r=X\}}}.$$

De aquí se deduce que para deducir una fórmula $\{P\}x_1 := X_1; \cdots ; x_k := X_k\{Q\}$ *sobre una secuencia de asignaciones, basta con deducir* $P \Rightarrow Q\{x_k/X_k\} \cdots \{x_1/X_1\}$*.*

Para la siguiente fórmula relativa a un comando se selección

$$\{\boldsymbol{V}\}\,\text{IF } X \geq Y \,\text{THEN}\, m := X \,\text{ELSE}\, m := Y \,\text{END}\, \{m = max(X, Y)\}$$

su deducción sería

$$\frac{\vdash \{\boldsymbol{V}\}\,\text{IF } X \geq Y \,\text{THEN}\, m := X \,\text{ELSE}\, m := Y \,\text{END}\, \{m = max(X, Y)\}}{\dfrac{\vdash \{X \geq Y\}\, m := X\, \{m = max(X, Y)\}; \qquad \vdash X \geq Y \Rightarrow X = max(X, Y)}{} \qquad \dfrac{\vdash \{X < Y\}\, m := X\, \{m = max(X, Y)\}}{\vdash X < Y \Rightarrow Y = max(X, Y)}}$$

que terminaría en dos implicaciones deducibles en la teoría de los números naturales.

Este cálculo es consistente con la semántica dada para las fórmulas de corrección parcial, e.d. que toda fórmula deducible mediante el cálculo, a partir de la teoría axiomática correspondiente a un álgebra, es válida en dicha álgebra. Sin embargo, la completitud sólo se cumple para las álgebras expresivas (aquéllas para las que cualquier aserto intermedio necesario para una derivación en el cálculo de Hoare siempre se puede expresar en la lógica de predicados de primer orden).

Generalmente se suelen añadir al cálculo dos reglas independientes de la estructura de los programas cuyo interés reside en facilitar las demostraciones de una determinada forma. Son las siguientes

- Regla para las demostraciones incrementales:

$$\frac{\vdash \{P\}\alpha\{Q_1\};\ \vdash \{P\}\alpha\{Q_2\}}{\vdash \{P\}\alpha\{Q_1 \wedge Q_2\}}$$

  Esta regla permite, a la hora de tener que demostrar una fórmula de corrección en la que la postcondición sea una conjunción de dos (o más) fórmulas, proceder de forma incremental demostrando la corrección para cada una de las fórmulas que aparecen en la postcondición.

- Regla para las demostraciones por casos:

$$\frac{\vdash \{P_1\}\alpha\{Q\};\ \vdash \{P_2\}\alpha\{Q\}}{\vdash \{P_1 \vee P_2\}\alpha\{Q\}}$$

  Esta regla es aplicable a los casos en los que la precondición se puede separar en dos (o más) casos, cada uno de los cuales puede requerir una demostración diferente.

2.2.5 Corrección total

Como hemos visto, la validez de una sentencia de corrección parcial no implica que el transformador de estados correspondiente al programa que aparece en la sentencia esté definido para todos los estados denotados por la precondición. Esta indefinición, a nivel teórico, sólo puede deberse a la aparición de ciclos tales que la aplicación reiterada del transformador asociado al cuerpo del ciclo a estados en los que es válida la condición de control nunca alcanza un estado en el que no se cumpla dicha condición. En estos casos se dice que el programa *no termina*. El lenguaje de la lógica de Hoare se podría ampliar con fórmulas de *corrección total* de la forma $[P]\alpha[Q]$ cuya validez en una interpretación A, para un estado sta, se puede establecer de la forma siguiente: $A \models_{sta} [P]\alpha[Q]$, si y sólo si

$$A \models_{sta} P \ \text{ implica } \ sta[\alpha]_A sta' \ \text{ y } \ A \models_{sta'} Q;$$

y su validez en A se cumplirá cuando sea válida para todos los estados definibles sobre A; lo que significa que la validez de una fórmula $[P]\alpha[Q]$ implica la terminación del

programa α cuando parte de un estado caracterizado por P y la validez de Q en el estado que se alcanza. Sin embargo, el cálculo de Hoare no se amplía para tratar sentencias de corrección total, sino que la corrección parcial y la terminación se suelen tratar por separado[4].

Aunque el lenguaje de sentencias de Hoare no contempla la posibilidad de expresar la terminación normal de programas, en muchos casos se puede recurrir a un método parcialmente informal derivado del siguiente razonamiento: la posibilidad de no terminación de un programa se debe normalmente a la aparición de ciclos[5] y para que pueda terminar un ciclo es necesario que su cuerpo produzca algún tipo de evolución o progreso de los estados hacia situaciones en las que no se cumpla la condición de control. Esta idea la podemos concretar expresando la evolución del ciclo mediante una *función de progreso* f, con valor entero[6], tal que deba ser no negativa para que se ejecute el ciclo y disminuya estrictamente después de cada repetición; como el decrecimiento de la función en estas condiciones no puede ser perpetuo, resulta obvio —al menos intuitivamente— que el ciclo deberá terminar. Esta idea se formaliza, para un ciclo

$$\text{WHILE}\, t_B \,\text{DO}\, \alpha \,\text{END}$$

con invariante I, buscando una función f para la que se pueda demostrar lo siguiente

1. $\vdash\ I \wedge t_B \ \Rightarrow\ f \geq 0$,
2. $\vdash\ \{I \wedge t_B \wedge f = A\}\, \alpha\, \{f < A\}$.

2.3 Verificación de Programas.

Como hemos dicho, las reglas del cálculo de Hoare se utilizan para demostrar la validez de fórmulas de corrección parcial o también, dicho de otro modo, para demostrar la corrección de un programa α respecto a una especificación parcial (P, Q), aplicando dichas reglas de acuerdo con la estructura del programa. Esta tarea se conoce con el nombre de *verificación de programas*.

El cálculo de Hoare se puede utilizar como un *cálculo deductivo* de forma ascendente, con encadenamientos hacia adelante, partiendo del conjunto de fórmulas de la teoría de base más los axiomas del cálculo y generando deducciones por aplicación de las reglas hasta llegar a la sentencia buscada, o como un *cálculo regresivo* de forma descendente, con encadenamientos hacia atrás, partiendo de la fórmula cuya validez se quiere demostrar y reduciéndola, progresivamente, a sentencias más simples con ayuda

[4]Esto se debe a que la corrección parcial es una propiedad de primer orden, e.d., que se mantiene en las álgebras en las que son válidas las mismas fórmulas cerradas, mientras que la terminación no.

[5]Después se verá que también influyen las llamadas a procedimientos recursivos.

[6]En general, sirve cualquier valor dentro de un dominio bien fundado, donde las secuencias estrictamente decrecientes deban ser finitas

de las reglas hasta alcanzar los axiomas del cálculo o las fórmulas de la teoría de base. Esta última forma de uso es la más extendida debido a la facilidad que ofrecen ciertas reglas, en particular el axioma de la asignación que parece especialmente concebido para un cálculo regresivo.

Por combinación de las reglas básicas se pueden obtener reglas derivadas que contribuyen a simplificar las demostraciones; así, del axioma de asignación, de la regla de la composición y de la regla de refinamiento se obtiene la siguiente regla

$$\frac{\vdash\ P \Rightarrow Q\{X_n/t_n\}\ldots\{X_1/t_1\}}{\vdash\ \{P\}\ X_1 := t_1; \ldots; X_n := t_n\ \{Q\}}$$

aplicable a una secuencia de asignaciones siempre y cuando las sustituciones que aparecen en $Q\{X_n/t_n\}\ldots\{X_1/t_1\}$ sean admisibles.

Veamos a continuación algunos ejemplos de verificación con idea de sistematizar en la medida de lo posible este tipo de demostraciones.

Ejemplo 2.9 *Para verificar que la siguiente expresión de corrección parcial*

$$\{x, y, t, A, B : NAT, x = A, y = B\}\ \texttt{t:=x; x:=y; y:=t}\ \{x = B, y = A\}$$

es una consecuencia lógica de la teoría característica del álgebra $\boldsymbol{N}$ *de los números naturales, podemos aplicar directamente la regla que acabamos de enunciar para secuencias de asignaciones, con lo que la verificación se reduce a demostrar a partir de la teoría de los números naturales que*

$$\vdash (x, y, t, A, B : NAT, x = A, y = B)\ \Rightarrow\ (x = B, y = A)\{y/t\}\{x/y\}\{t/x\}$$

o también, que

$$\vdash (x, y, t, A, B : NAT, x = A, y = B)\ \Rightarrow\ y = B, x = A,$$

lo que resulta trivialmente cierto.

Ejemplo 2.10 *Para verificar en* $\boldsymbol{N}$ *esta otra expresión de corrección parcial*

$\{x, y, A, B : NAT, A > 0, B > 0\}$
```
    x:=A; y:=B;
    WHILE x<>y DO
        IF x<y THEN y:=y-x ELSE x:=x-y END
    END
```
$\{x\ =\ mcd(A, B)\}$

debemos tener en cuenta que el programa es una secuencia compuesta por dos asignaciones seguidas de un comando de iteración. Como disponemos de reglas para derivar la validez de expresiones de corrección parcial relativas a secuencias de asignaciones y a comandos de iteración, según la regla de la composición nos bastará con determinar un predicado intermedio I *tal que las fórmulas*

$\{x, y, A, B : NAT, A > 0, B > 0\}$ $\{I\}$
`x:=A; y:=B` WHILE `x<>y` DO
$\{I\}$ IF `x<y` THEN `y:=y-x` ELSE `x:=x-y` END
END
$\{x = mcd(A, B)\}$

sean derivables. Para demostrar la derivabilidad de la primera fórmula bastará demostrar la derivabilidad de

$$A > 0, B > 0 \Rightarrow I\{y/B\}\{x/A\} \tag{2.1}$$

a partir de la teoría de los números naturales. Teniendo en cuenta la regla para la repetición, podemos pensar en I como un invariante del ciclo, con lo que la derivabilidad de la segunda fórmula se reducirá a demostrar la derivabilidad de

$\{I, x \neq y\}$ IF `x<y` THEN `y:=y-x` ELSE `x:=x-y` END $\{I\}$

—es decir, a demostrar que I es efectivamente un invariante del ciclo— y la derivabilidad de

$$I, x = y \Rightarrow x = mcd(A, B) \tag{2.2}$$

también a partir de la teoría de los números naturales. Y, teniendo en cuenta la regla de derivación para la selección, para demostrar la derivabilidad de la fórmula anterior correspondiente a un comando de selección bastará con demostrar la derivabilidad de las siguientes fórmulas

$$I, x \neq y, x < y \Rightarrow I\{y/y - x\} \tag{2.3}$$

$$I, x \neq y, x \geq y \Rightarrow I\{x/x - y\} \tag{2.4}$$

a partir de la teoría de los números naturales. Con lo que, resumiendo, la derivabilidad de la fórmula inicial de corrección parcial se traduce finalmente en la búsqueda de un invariante I y en la derivabilidad de las fórmulas 2.1, 2.2, 2.3, 2.4 a partir de la teoría de los números naturales.

Para determinar el invariante podemos recurrir a la postcondición y al análisis de las transformaciones que se producen en el cuerpo del ciclo teniendo en cuenta que de la teoría de los números naturales se deducen las siguientes consecuencias

$$0 < x < y \quad \Rightarrow \quad mcd(x, y) = mcd(x, y - x) \tag{2.5}$$

$$0 < y < x \quad \Rightarrow \quad mcd(x, y) = mcd(x - y, y) \tag{2.6}$$

$$0 < x = y \quad \Rightarrow \quad mcd(x, y) = x. \tag{2.7}$$

A partir de esto, podemos conjeturar un invariante de la forma

$$I \equiv mcd(x, y) = mcd(A, B),\ 0 < x,\ 0 < y$$

y abordar las demostraciones de la derivabilidad de las fórmulas 2.1, 2.2, 2.3, 2.4.

En general, siempre que se plantee la verificación de un programa respecto a una especificación se comenzará estableciendo un invariante para cada uno de los comandos de iteración que aparezcan en el programa, lo que permitirá establecer anotaciones (predicados) antes, a la entrada y después de los correspondientes ciclos de la forma siguiente

$\{I\}$
WHILE `b` DO
$\{I, b\}\ \alpha\ \{I\}$
END
$\{I, \neg b\}$

y a partir de ellos, desde abajo hacia arriba, aplicando las reglas de las secuencias de asignaciones y de la selección simple se van derivando nuevas anotaciones en la forma

$\{Q\{xn/tn\}\cdots\{x1/t1\}\}$
`x1:=t1; ...;xn:=tn;`
$\{Q\}$

$\{P\}$
IF `b` THEN
$\{P, b\}\ \alpha\ \{Q\}$
ELSE
$\{P, \neg b\}\ \beta\ \{Q\}$
END
$\{Q\}$

hasta alcanzar la precondición, con lo que tendremos un *programa anotado*. En ese momento veremos que en determinados puntos del programa aparecen dos anotaciones: una previa posiblemente obtenida a partir de los invariantes o la propia precondición y otra posterior resultado del proceso ascendente que hemos indicado; y, por aplicación de la regla de refinamiento, habrá que demostrar la derivabilidad de las implicaciones de las anotaciones previas a las anotaciones posteriores en cada uno de estos puntos. Estas implicaciones —correspondientes al cálculo de predicados— constituyen las denominadas *condiciones de verificación* a las que se reducen en última instancia todos los ejercicios de verificación.

Ejemplo 2.11 *Otro aspecto que se debe tratar en el programa del ejemplo anterior es el de la terminación debido a que en él aparece un comando de iteración o ciclo. Para probar que el programa termina bastará probar que termina el ciclo, y para esto, debemos buscar una función f de las variables que se modifican en el cuerpo del ciclo que cumpla las condiciones 1 y 2 enunciadas en la sección 2.2.5. Examinando lo que ocurre dentro del ciclo se ve que unas veces disminuye la variable* `x` *y otras la variable* `y`*, luego en cualquier caso siempre disminuye la suma de ambas variables por lo que podemos considerar la función $f(x, y) = x + y$ como posible función de progreso. Con esta función la condición 1 queda*

$$\vdash I, x \neq y \Rightarrow x + y \geq 0,$$

fácilmente deducible a partir de la información $0 < x, 0 < y$ *contenida en el invariante que se propuso; en cuanto a la condición 2, estableciendo las oportunas anotaciones da lugar al siguiente par de condiciones dentro del cálculo de predicados*

$$\vdash I, x \neq y, x + y = A, x < y \Rightarrow x + (y - x) < A$$
$$\vdash I, x \neq y, x + y = A, x \geq y \Rightarrow (x - y) + y < A,$$

también deducible fácilmente a partir de $0 < x, 0 < y$.

Ejercicio 2.7 *Verifíquense en* $\boldsymbol{N}$ *cada una de las siguientes expresiones de corrección parcial y pruébese que los ciclos terminan.*

$\{u, v, w, X : NAT\}$
```
   u:=0; v:=1; w:=1;
   WHILE w≤X DO
       u:=u+1; v:=v+2; w:=w+v
   END
```
$\{u * u \leq X < (u + 1) * (u + 1)\}$

$\{x, y, z, A, B : NAT,\ \ A > 0\}$
```
   x:=A; y:=B; z:=1;
   WHILE y>0 DO
      IF y mod 2 = 0 THEN
         y:=y div 2; x:=x*x
      ELSE
         y:=y-1; z:=z*x
      END
   END
```
$\{\ z = A\hat{}B\ \}$

Ejercicio 2.8 *Verifíquense en el álgebra* ***Array[0..L]****, extensión de* $\boldsymbol{N}$ *con el dominio de los vectores de números naturales con* $L+1$ *componentes, las siguientes expresiones de corrección parcial, y pruébese que los ciclos terminan*

$\{i, X, L : NAT,\ \ found : BOOLEAN,\ \ a : ARRAY\ [0..L]\ OF\ NAT\}$
```
    i:=0; found:=FALSE;
    WHILE (i≤L) and (not found) DO
       IF a[i]=X THEN found:=TRUE END;
       i:=i+1
    END
```
$\{found = TRUE \ \Leftrightarrow\ \exists\beta \cdot (0 \leq \beta \leq L \ \wedge\ a[\beta] = X)\}$

$\{i, max, L : NAT,\ \ a : ARRAY\ [0..L]\ OF\ NAT\}$

```
    i:=0; max:=a[0];
    WHILE i<L DO
       IF a[i+1]>max THEN max:=a[i+1] END;
       i:=i+1
    END
```

$\{max = Max\{\, a[\alpha] \,|\, 0 \leq \alpha \leq L\}\,\}$

2.4 Ampliación del universo de programas.

En este apartado estudiaremos algunos comandos que incorporan habitualmente los lenguajes de programación, son dos formas de alternativas múltiples, los ciclos controlados al final del cuerpo y los ciclos con un número fijo de repeticiones, así como construcciones para la estructuración de los programas como los procedimientos (procedimientos puros y procedimientos funcionales), y veremos cómo se pueden incorporar a nuestro lenguaje de comandos.

2.4.1 Ampliación del lenguaje de comandos

Las alternativas múltiples del tipo (IF · THEN · ELSIF · THEN · · · ELSE · END), controladas por expresiones booleanas, que incorporan lenguajes como p.e. MODULA-2, se pueden traducir fácilmente en un anidamiento de comandos (IF · THEN · ELSE · END) sobre la rama ELSE, de acuerdo con el siguiente esquema

IF	b_1	THEN	α_1	$\rightsquigarrow$	IF		b_1	THEN	α_1	
ELSIF	b_2	THEN	α_2		ELSE	IF	b_2	THEN	α_2	
$\cdots$					$\cdots$					
ELSE			α_{m+1}		ELSE					α_{m+1}
END					$\cdots$					
						END				
					END					

dado para m expresiones booleanas $b_1, b_2, \ldots, b_m$, con $m \geq 2$, por lo que da lugar a la siguiente regla derivada

$$\frac{\begin{array}{l}\vdash\ \{P \wedge b_1\}\ \alpha_1\ \{Q\};\\ \vdash\ \{P \wedge \neg b_1 \wedge b_2\}\ \alpha_2\ \{Q\};\\ \cdots\\ \vdash\ \{P \wedge \neg b_1 \wedge \neg b_2 \wedge \cdots \wedge \neg b_m\}\ \alpha_{m+1}\ \{Q\}\end{array}}{\vdash\ \{P\}\ \text{IF}\ b_1\ \text{THEN}\ \alpha_1\ \text{ELSIF}\ b_2\ \text{THEN}\ \alpha_2\ \cdots\ \text{ELSE}\ \alpha_{m+1}\ \text{END}\ \{Q\}}$$

Ejercicio 2.9 *Verifíquese en un álgebra extensión de* $\mathbf{N}$ *con los dominios de los vectores de números naturales con* N *y con* M *componentes, la siguiente expresión de corrección parcial, y pruébese que el ciclo termina*

$\{i, j, k, N, M : NAT,\ \ N > 0,\ \ M > 0,$
$a : ARRAY\,[1..N]\,OF\,NAT, \forall \alpha, \beta \cdot (1 \leq \alpha < \beta \leq N \ \Rightarrow\ a[\alpha] < a[\beta]),$
$b : ARRAY\,[1..M]\,OF\,NAT, \forall \alpha, \beta \cdot (1 \leq \alpha < \beta \leq M \ \Rightarrow\ b[\alpha] < b[\beta])\}$

```
k:=0; i:=1; j:=1;
WHILE (i≤N) and (j≤M) DO
  IF    a[i]<b[j] THEN i:=i+1
  ELSIF a[i]>b[j] THEN j:=j+1
  ELSE  k:=k+1; i:=i+1; j:=j+1
  END
END
```

$\{k \ = \ card\{(\alpha, \beta) \mid 1 \leq \alpha \leq N,\ \ 1 \leq \beta \leq M,\ \ a[\alpha] = b[\beta]\}\}$

Las alternativas múltiples del tipo CASE e OF $v_1 : \alpha_1 \mid v_2 : \alpha_2 \mid \cdots \mid v_m : \alpha_m$ ELSE v_{m+1} END, controladas por una expresión e de un tipo arbitrario T, en las que cada programa $\alpha_1, \alpha_2, \ldots, \alpha_m$ se ejecuta cuando la expresión e toma el correspondiente valor $v_1, v_2, \ldots, v_m$ y cuando no toma ninguno de estos valores se ejecuta α_{m+1}, también se pueden traducir en un anidamiento de comandos (IF · THEN · ELSE · END) sobre la rama ELSE, ahora de acuerdo con el esquema

CASE e	OF	$v_1 : \alpha_1$	$\rightsquigarrow$	IF	$e = v_1$	THEN	α_1
	$\mid$	$v_2 : \alpha_2$		ELSIF	$e = v_2$	THEN	α_2
		$\cdots$		$\cdots$			
	$\mid$	$v_m : \alpha_m$		ELSIF	$e = v_m$	THEN	α_m
	ELSE	α_{m+1}		ELSE			α_{m+1}
END				END			

dado para m valores distintos $v_1, \ldots, v_m$, con $m \geq 1$, donde se utiliza la alternativa múltiple del párrafo anterior únicamente con el objeto de simplificar la traducción. Para este comandolo la regla derivada que se obtiene es la siguiente

$$\frac{\begin{array}{l} \vdash\ \{P \wedge e = v_1\}\ \alpha_1\ \{Q\}; \\ \vdash\ \{P \wedge e = v_2\}\ \alpha_2\ \{Q\}; \\ \cdots \\ \vdash\ \{P \wedge e \neq v_1 \wedge e \neq v_2 \wedge \cdots \wedge e \neq v_m\}\ \alpha_{m+1}\ \{Q\} \end{array}}{\vdash\ \{P\}\ \text{CASE } e \text{ OF } v_1 : \alpha_1 \mid v_2 : \alpha_2\ \cdots\ \text{ELSE } v_{m+1}\ \text{END}\ \{Q\}}$$

aplicable cuando los valores $v_1, \ldots, v_m$ son todos distintos, aunque del mismo tipo.

Los ciclos controlados a la salida (REPEAT · UNTIL·) se pueden traducir a ciclos WHILE con arreglo al siguiente esquema

$$\text{REPEAT } \alpha \text{ UNTIL } b \quad \rightsquigarrow \quad \alpha;\ \text{WHILE } \textit{not}\ b \text{ DO } \alpha \text{ END}$$

Ejercicio 2.10 *Verifíquese en* $\boldsymbol{N}$ *la siguiente expresión de corrección parcial aplicando el esquema de traducción anterior.*

$\{u, z, A, B : NAT,\ \ A > 0\}$
```
   u:=A; z:=0;
   REPEAT
     z:=z+B; u:=u-1
   UNTIL u = 0
```
$\{z \ = \ A * B\}$

Pruébese también que el ciclo termina.

Los ciclos con un número fijo de repeticiones (FOR · TO · DO · END) se pueden traducir también a programas en términos de ciclos WHILE mediante el esquema siguiente

$$\text{FOR } i := t_1 \text{ TO } t_2 \text{ DO } \alpha \text{ END} \quad \rightsquigarrow \quad i := t_1;\ \text{WHILE } i \leq t_2 \text{ DO } \alpha \text{ END}$$

Ejercicio 2.11 *Verifíquese en el álgebra* ***Array[0..L]*** *la expresión de corrección parcial,*

$\{\, k, \mathit{min}, \mathit{suma}, L : NAT,\ \ L \geq 0,\ \ a : ARRAY\,[0..L]\,OF\,NAT\,\}$
```
   min:=a[0]; suma:=a[0];
   FOR k:=1 TO L DO
      suma:=suma + a[k];
      IF min>a[k] THEN min:=a[k] END
   END
```
$\{\, \mathit{min} = Min\{\, a[\alpha] \,|\, 0 \leq \alpha \leq L\,\},\ \ \mathit{suma} = \sum\{\, a[\alpha] \,|\, 0 \leq \alpha \leq L\,\}\,\}$

aplicando el esquema de traducción de los ciclos FOR*, y pruébese que el ciclo termina.*

2.4.2 Procedimientos

La incorporación de procedimientos a nuestro lenguaje de programación supone la extensión del lenguaje con las declaraciones y las invocaciones de dichos procedimientos. La *declaración* de un procedimiento se hará de la forma

PROCEDURE $p(x_1 : s_1; \cdots ; x_n : s_n;$ VAR $y_1 : s'_1; \cdots ; y_m : s'_m);$
VAR $z_1 : s''_1; \cdots ; z_r : s''_r;$
BEGIN $\ \alpha\ $ END

donde p es el *nombre* del procedimiento, $x_1, \cdots, x_n$ son los *parámetros de entrada*, $y_1, \cdots, y_m$ los *parámetros de salida*, $z_1, \cdots, z_r$ son variables de programa denominadas *variables locales* y α es un programa, el *cuerpo* del procedimiento, en el que sólo aparecen las variables x_i, y_j y z_k, no aparecen asignaciones a las variables x_i y se instancian todas las variables y_j. La *invocación* de procedimientos será un nuevo comando (CALL $\cdot$); en concreto, una invocación del procedimiento p declarado anteriormente tendrá la forma

$$\text{CALL } p(t_1, \cdots, t_n, u_1, \cdots, u_m)$$

donde cada t_i es un término del tipo s_i y cada u_j una variable de programa del tipo s'_j, además todas las variables u_j son distintas y no aparecen en los términos t_i.

Los procedimientos se introducen normalmente para realizar unas tareas concretas, estas tareas deben reflejarse mediante especificaciones parciales dadas en función de los parámetros formales que aparecen en la declaración de cada procedimiento, así, si el procedimiento p se construye con la intención de que realice una tarea dada por la especificación (P, Q) escribiremos

$$\begin{array}{l}\text{PROCEDURE } p(x_1 : s_1; \cdots; x_n : s_n; \text{VAR } y_1 : s'_1; \cdots; y_m : s'_m);\\ \text{VAR } z_1 : s''_1; \cdots; z_r : s''_r;\\ \text{BEGIN } \{P\}\ \alpha\ \{Q\} \text{ END}\end{array}$$

con lo que tendremos un procedimiento especificado. El procedimiento p se dice que está correctamente especificado sobre un álgebra A cuando la fórmula de corrección parcial $\{P\}\,\alpha\,\{Q\}$ puede derivarse a partir de la teoría correspondiente a A.

Las declaraciones de los procedimientos que se utilizan en un programa constituyen lo que se denomina un *entorno* de procedimientos, y estos entornos deben citarse —como parte de las precondiciones— en las demostraciones de corrección de los programas que los utilicen como parte de dichos programas. Normalmente los procedimientos deberán aparecer correctamente especificados y durante la verificación de los programas se hará uso de estas especificaciones.

La *semántica de una invocación* de un procedimiento se da a partir de la semántica del cuerpo de dicho procedimiento con las oportunas sustituciones de los parámetros formales de la declaración por los términos y variables que aparecen en la invocación mediante el siguiente esquema de traducción dado para el caso del procedimiento p

$$\begin{array}{lll}\text{CALL}\, p(t_1, \cdots, t_n, u_1, \cdots, u_m) & \rightsquigarrow & x_1 := t_1; \cdots; x_n := t_n;\\ & & y_1 := u_1; \cdots; y_m := u_m;\\ & & \alpha;\\ & & u_1 := y_1; \cdots; u_m := y_m\end{array}$$

siempre que no se produzcan conflictos de nombres entre los parámetros formales x_i, y_j o variables locales z_k y las variables del programa donde aparezca la llamada; si

se produjese algún conflicto bastaría con cambiar los nombres de los parámetros o variables locales.

Según esto, a las invocaciones de procedimientos le corresponde, en el cálculo de Hoare, una regla derivada de la forma

$$\frac{\begin{array}{l}\vdash\ P^* \Rightarrow P\{x_1/t_1, \cdots, x_n/t_n, y_1/u_1, \cdots, y_m/u_m\} \\ \vdash\ \{P\}\,\alpha\,\{Q\} \\ \vdash\ \exists u_1, \cdots, u_m \cdot P^* \wedge Q\{y_1/u_1, \cdots, y_m/u_m\} \Rightarrow Q^*\end{array}}{\vdash\ \{P^*\}\,\mathrm{CALL}\,p(t_1, \cdots, t_n, u_1, \cdots, u_m)\,\{Q^*\}}$$

donde α es el cuerpo del procedimiento p. Esta regla nos dice que para que la fórmula de corrección $\{P^*\}\,\mathrm{CALL}\,p(t_1, \cdots, t_n, u_1, \cdots, u_m)\,\{Q^*\}$ sea derivable, para un procedimiento cuyo comportamiento viene dado por la especificación (P, Q), P^* debe implicar la precondición P del procedimiento con los parámetros formales cambiados por los actuales con los que se hace la llamada, y la postcondición Q, también con los parámetros formales cambiados por los actuales, junto con la información que se tenga sobre el estado correspondiente a las variables que no modifica la llamada, debe implicar Q^*[7].

La *corrección del cuerpo de un procedimiento* respecto a su especificación se hace exactamente igual que la de cualquier comando o programa y si aparecen invocaciones a otros procedimientos se aplica la regla de acabamos de ver. En el caso de que aparezcan invocaciones recursivas al mismo procedimiento que se está definiendo, la aplicación de la regla de la invocación supone en realidad la aplicación de un principio de inducción: admitir la corrección de las invocaciones recursivas y, a partir de esta hipótesis, demostrar la corrección del cuerpo del procedimiento.

Ejemplo 2.12 *Consideremos el siguiente procedimiento especificado en el álgebra de los números enteros*

```
PROCEDURE McCarthy(x:INT; VAR y:INT);
VAR u:INT;
BEGIN
{x, y, u : INT}
  IF x>100 THEN
      y:=x-10
  ELSE
      CALL McCarthy(x+11,u);
      CALL McCarthy(u,y)
```

[7] Téngase en cuenta que en el enunciado general no podemos separar la información que tiene P^* relativa a las variables u_j de la relativa a las demás variables, por lo que lo único que podemos hacer para anularla es cuantificar existencialmente P^* en las variables u_j, con lo que dicha información ya no se interpreta como relativa a las variables de programa u_j.

END
$\{(x > 100 \;\Rightarrow\; y = x - 10), (x \leq 100 \;\Rightarrow\; y = 91)\}$
END

aplicando el análisis que vimos en la sección 2.3 obtenemos la siguiente condición de verificación para la rama THEN *del comando de selección simple*

$$\vdash\; x, y, u : INT, x > 100 \;\Rightarrow\; (x > 100 \Rightarrow x - 10 = x - 10), (x \leq 100 \Rightarrow x - 10 = 91);$$

al comienzo de la rama ELSE *tenemos el predicado,* $x \leq 100$*, y para poder aplicar la regla correspondiente a la invocación de procedimientos necesitamos que se cumpla la precondición de la primera invocación recursiva, es decir, que*

$$\vdash\; x, y, u : INT, x \leq 100 \;\Rightarrow\; x + 11, u : INT,$$

con lo que inmediatamente después de la primera invocación recursiva tendremos la postcondición particularizada para esta llamada más la información referente al valor de x*; todo esto deberá implicar la precondición de la segunda invocación recursiva*

$$\vdash\; x, y, u : INT,\; \exists u \cdot (x \leq 100),\; (x > 89 \Rightarrow u = x + 1), (x \leq 89 \Rightarrow u = 91) \;\Rightarrow\; u, y : INT$$

para que después de dicha invocación tengamos su postcondición más la información anterior referente a los valores de las variables que no se han modificado; todo esto deberá implicar la postcondición del procedimiento, es decir

$$\begin{array}{l}\vdash\; x, y, u : INT, \\ \quad \exists y \cdot (\exists u \cdot (x \leq 100),\; (x > 89 \Rightarrow u = x + 1), (x \leq 89 \Rightarrow u = 91)), \\ \quad (u > 100 \Rightarrow y = u - 10), (u \leq 100 \Rightarrow y = 91) \\ \quad \Rightarrow\; (x > 100 \Rightarrow y = x - 10), (x \leq 100 \Rightarrow y = 91).\end{array}$$

Las implicaciones de las precondiciones de las invocaciones recursivas son inmediatas pues las precondiciones se reducen a que los valores de los parámetros cumplan unas condiciones de tipo (en general, cuando planteemos condiciones de verificación prescindiremos de las condiciones de tipo, salvo cuando afecten a los resultados de operaciones parciales que no estén definidas para ciertos valores de sus argumentos) y la última condición de verificación se puede escribir

$$\begin{array}{l}\vdash\; (x \leq 100),\; (x > 89 \Rightarrow u = x + 1), (x \leq 89 \Rightarrow u = 91), \\ \quad (u > 100 \Rightarrow y = u - 10), (u \leq 100 \Rightarrow y = 91) \\ \quad \Rightarrow\; (x > 100 \Rightarrow y = x - 10), (x \leq 100 \Rightarrow y = 91).\end{array}$$

prescindiendo de las condiciones de tipo y de las cuantificaciones existenciales ya que afectan a expresiones donde no aparecen las variables cuantificadas. La implicación

$$(x \leq 100) \;\Rightarrow\; (x > 100 \Rightarrow y = x - 10)$$

es deducible por ser $x \leq 100$ *y* $x > 100$ *predicados contrarios, también*

$$\vdash\ (x \leq 100),\ (x > 89 \Rightarrow u = x + 1), (x \leq 89 \Rightarrow u = 91)\ \Rightarrow\ 90 < u \leq 101;$$

luego sólo queda demostrar que

$$\vdash\ (90 < u \leq 101), (u > 100 \Rightarrow y = u - 10), (u \leq 100 \Rightarrow y = 91)\ \Rightarrow\ y = 91;$$

pero esta implicación se deduce fácilmente razonando sobre la posibilidad de que sea $x > 100$ *o* $x \leq 100$*, en cualquiera de los casos se obtiene* $y = 91$*, y la última condición de verificación será deducible.*

Con los procedimientos recursivos se presenta en mismo problema de terminación que con los ciclos: el número de invocaciones recursivas podría ser infinito, con lo que al aplicar la semántica de la invocación nos encontaríamos con el problema de tener que realizar un número infinito de reemplazamientos de invocaciones por el cuerpo del procedimiento —con las oportunas asignaciones— con lo que nunca terminaríamos. La *terminación de un procedimiento recursivo* se puede abordar de manera parecida a como se ha hecho con los ciclos: determinar una función de los parámetros del procedimiento, o *función de progreso* o *de cota*, f, con valor entero, tal que la precondición del procedimiento implique que deba ser no negativa y que disminuya estrictamente para los parámetros de las invocaciones recursivas. Expresado de una manera formal, f debe ser tal que

1. $\vdash\ P \Rightarrow f(x_1, \cdots, x_n, y_1, \cdots, y_m) \geq 0$

2. $\vdash\ f(x_1, \cdots, x_n, y_1, \cdots, y_m) = A,\ P^* \Rightarrow f(t_1, \cdots, t_n, v_1, \cdots, v_m) < A$, para cada invocación recursiva que aparezca con valores t_i y v_j, siendo P^* la caracterización del estado en el momento de la invocación.

Ejemplo 2.13 *Para demostrar que el procedimiento del ejemplo anterior termina tenemos que buscar una función de cota. Para determinar dicha función podemos contar el número de invocaciones recursivas que se producen para distintos valores del primer argumento y así obtenemos la tabla siguiente*

x	*invocaciones*
> 100	0
100	2
99	4
98	6
97	8
76	10

que nos permite conjeturar que el número de invocaciones recursivas que se producen en una invocación de la forma CALL `McCarthy(t,u)` *vendrá dado por la función*

$$i(t) = \begin{cases} 0, & si\ t > 100 \\ 2 * (101 - t), & si\ t \leq 100 \end{cases}$$

lo que nos permite tomar como función de cota $f(x, y) = i(x)$ *o* $f(x, y) = i(x)\, div\, 2$ *que tiene una expresión algo más simple. Esta función cumple obviamente la primera condición. Respecto a la segunda condición teniendo en cuenta que las invocaciones recursivas se producen para* $x \leq 100$*, tendremos* $f(x, y) = 101 - x$ *para la invocación principal y para la primera invocación recursiva habrá que demostrar*

$$\vdash\ x \leq 100\ \Rightarrow\ f(x + 11, u) < 101 - x,$$

que por la definición de f *se traduce en las dos condiciones siguientes*

$$\vdash\ 89 < x \leq 100\ \Rightarrow\ 0 < 101 - x,$$
$$\vdash\ x \leq 89\ \Rightarrow\ 101 - x - 11 < 101 - x,$$

que se cumplen evidentemente. Para la segunda invocación recursiva tendremos que probar

$$\vdash\ (x \leq 100), (x > 89 \Rightarrow u = x + 1), (x \leq 89 \Rightarrow u = 91)\ \Rightarrow\ f(u, v) < 101 - x,$$

que se traduce en las tres condiciones siguientes

$$\vdash\ x = 100\ \Rightarrow\ 0 < 101 - x,$$
$$\vdash\ 89 < x < 100\ \Rightarrow\ 101 - x - 1 < 101 - x,$$
$$\vdash\ x \leq 89\ \Rightarrow\ 101 - 91 < 101 - x,$$

que también son evidentes.

2.4.3 Procedimientos funcionales

Son procedimientos cuyas invocaciones dentro de un programa se reemplazan por valores de un cierto tipo, por lo que se utilizan —como las funciones del álgebra sobre la que se interpreta el programa— en expresiones donde se requiera un valor del tipo que reemplaza al procedimiento (en la parte derecha de una asignación, en parámetros de entrada de una invocación de procedimiento, en comparaciones,...). Nuestro lenguaje de programación se puede ampliar con la incorporación de declaraciones y de invocaciones de procedimientos funcionales.

La *declaración* de un procedimiento funcional pf se hará en la forma

PROCEDURE $pf(x_1 : s_1; \cdots ; x_n : s_n) : s;$
VAR $z_1 : s'_1; \cdots ; z_r : s'_r;$
BEGIN α END

donde pf es el *nombre* del procedimiento, los parámetros $x_1, \cdots, x_n$ son todos de entrada, $z_1, \cdots, z_r$ son variables de programa denominadas *variables locales* y α es un programa, el *cuerpo* del procedimiento, en el que sólo aparecen las variables x_i y z_k, —y no aparecen asignaciones a las variables x_i— y una variable especial pf del tipo s, con el nombre del procedimiento, cuyo valor debe quedar instanciado al final de la ejecución de α y es el que sustituye a la invocación del procedimiento funcional.

La *invocación* de un procedimiento funcional pf será un término de la forma

$$pf(t_1, \cdots, t_n),$$

donde cada t_i es un término del tipo s_i, que puede aparecer dentro de un programa en cualquier expresión donde se requiera un término del tipo s.

Los procedimientos funcionales se especifican igual que los procedimientos normales enunciando una precondición P y una postcondición Q para el cuerpo α en las cuales las únicas variables de programa que pueden aparecer son las x_i y en la postcondición deberá aparecer además pf que representa el valor que calcula el procedimiento funcional. Un procedimiento funcional especificado tendrá el siguiente aspecto

$$\begin{array}{l}\text{PROCEDURE } pf(x_1 : s_1; \cdots ; x_n : s_n) : s; \\ \text{VAR } z_1 : s'_1; \cdots ; z_r : s'_r; \\ \text{BEGIN } \ \{P\}\alpha\ \{Q\} \text{ END}\end{array}$$

con las restricciones que acabamos de enunciar para P y Q.

La *semántica de una invocación* de un procedimiento funcional se da teniendo en cuenta que desencadena un cálculo, debido al cuerpo del procedimiento, que produce un valor (y ninguna alteración en el estado del programa) que debe atenerse a la especificación (postcondición) de dicho cuerpo. Así, para un comando COM donde aparezca una invocación $pf(t_1, \cdots, t_n)$ del procedimiento funcional pf el significado de la ejecución de dicho comando será equivalente a la ejecución de la secuencia

$$\begin{array}{l}x_1 := t_1; \cdots ; x_n := t_n; \\ \alpha; \\ \text{COM}\{pf(t_1, \cdots, t_n)/pf\};\end{array}$$

siempre que no haya conflicto de nombres entre los parámetros formales x_i y las variables del programa donde aparezca el comando COM (de haberlos se cambiarían los nombres de los parámetros formales).

Según esto, a las invocaciones de procedimientos funcionales se les puede asociar una regla derivada en el cálculo de Hoare de la forma

$$\frac{\{P\}\alpha\{Q\}}{\forall x_1, \cdots, x_n \cdot (P \Rightarrow Q\{pf/pf(x_1, \cdots, x_n)\})}$$

que indica que si en el momento de la invocación se cumple la precondición del cuerpo del procedimiento, entonces se implica la postcondición (cualesquiera que sean los valores de los parámetros x_i en ese momento) cambiando la variable local pf por la llamada $pf(x_1, \cdots, x_n)$.

La *corrección del cuerpo* de un procedimiento funcional se realiza de forma parecida a la corrección del cuerpo de un procedimiento normal aplicando la regla anterior si aparecen invocaciones a procedimientos funcionales, lo que supone la aplicación de un principio de inducción cuando aparecen invocaciones recursivas al propio procedimiento.

Ejemplo 2.14 *Consideremos el siguiente procedimiento funcional especificado en el álgebra de los números naturales*

PROCEDURE `sumadigitos(n:NAT):NAT;`
VAR `sumadigitos:NAT;`
BEGIN $\{TRUE\}$
  IF `n<10` THEN
    `sumadigitos:=n;`
  ELSE
    `sumadigitos:=sumadigitos(n div 10)+(n mod 10);`
  END
END $\{sumadigitos = \sum_{\alpha=0}^{\lfloor \log_{10}(n) \rfloor} (n \, div \, 10^{\alpha}) \, mod \, 10\}$

Para verificar la corrección de este procedimiento tendremos que verificar el cuerpo del procedimiento respecto a la especificación dada. Analizando la estructura de dicho cuerpo se obtienen las siguientes condiciones de verificación

Con los procedimientos funcionales se presenta también el problema de la terminación debido a la aparición de invocaciones recursivas. Este problema se trata de manera parecida a como se hace con los procedimientos normales buscando una función de cota f, que dependa únicamente de los parámetros del procedimiento, con las mismas condiciones.