F. Orejas

Facultad d'Informática
Univeristat Politècnica
Barcelona

Tipos abstractos de datos y programación concurrente

1. Introducción

En los últimos años, el concepto de tipos abstractos de datos se ha ido introduciendo con éxito en varios campos de la Informática, tanto a nivel teórico como práctico, especialmente en aquellos más relacionados con la programación. En concreto, los tipos abstractos de datos han encontrado aplicación en el diseño, especificación, implementación y verificación de estructuras de datos [1, 19, 20, 21], diseño, especificación y verificación de programas [3, 17, 20, 25], síntesis de programas a partir de especificaciones [16, 20, 29] semántica de lenguajes de programación [7, 15, 18, 33], construcción y verificación de compiladores [2, 15, 29], diseño de bases de datos [6, 13, 26], representación del conocimiento [6, 24], etc.

Todo ello es consecuencia de una progresiva algebraización de la informática (a fin de cuentas, un tipo abstracto de datos es simplemente un álgebra). Algebraización que a veces es explícita, como en la propia teoría de tipos abstractos de datos, en la semántica denotacional o en la teoría de bases de datos relacionales, pero que a veces es implícita, manteniéndose a un nivel más subconsciente. Por ejemplo, es comunmente admitido que un buen programador debe de tener una sólida formación matemática (formación más en el sentido de hábitos de pensamiento, que de conocimientos concretos) [12]. Pues bien, si se concreta aún más el tipo de formación matemática adecuada, surgen siempre dos áreas: el álgebra y la lógica (y no olvidemos que, hoy día, la lógica es en gran parte álgebra). El motivo es claro, las herramientas mentales con las que ha de trabajar un programador son, fundamentalmente, la abstracción, la generalización, la conceptualización y (a un primer nivel de aprendizaje) la facilidad de expresión en un lenguaje formal, herramientas inherentes al álgebra y a la lógica, quizás más que a ninguna otra rama de las matemáticas.

En este artículo, se presenta (a un nivel muy intuitivo), un modelo de concurrencia en términos de tipos abstractos de datos (una presentación algo más formal puede verse en [31], la versión definitiva podrá verse (si el tiempo no lo impide y la autoridad lo permite) en [32]). El modelo es utilizado, a continuación, para el estudio de algunos de los modelos de concurrencia más conocidos. El artículo, por tanto, es de alguna forma complementario al que pesenta A. Llamosí en este mismo número, en el sentido de que estudian las mismas ideas desde otro punto de vista: en concreto, el análisis comparativo que se hace aquí está, quizás más preocupado por cuestiones conceptuales y metodológicas y menos (en realidad, nada en absoluto) por cuestiones de eficiencia en la implementación en arquitecturas actuales.

En la primera sección, se introducirá el concepto de tipo abstractos de datos y se discutirán algunas aplicaciones a la programación secuencial. A continuación, se define un proceso como un tipo abstracto de datos, con algunas características particulares: el modo de interacción con otros procesos. Finalmente, se aplica el modelo al estudio de los modelos y construcciones más conocidas: en la sección 3 se estudian los monitores y en la 4 como consecuencia de las deficiencias observadas en los monitores, los enfoques más modernos, i.e. DP (distributed processes), CSP (communicating sequential processes), tasking de ADA y CCS (calculus of communicating systems).

La descripción de las distintas construcciones o modelos será muy breve, para evitar la reiteración con respecto del artículo de A. Llamosí.

1.1. Advertencia final

A pesar de que este artículo presenta un nuevo enfoque para el estudio de la concurrencia, más aún, realizándose comparaciones entre otros ya conocidos, *en ningún momento* se utilizará el ejemplo de los filósofos que comen spaghetti de Dijkstra (en realidad, casi no aparece ningún ejemplo). El motivo es la repugnancia que causa al autor la extraordinaria falta de educación gastronómica que constituye el hecho de pensar en comer spaghetti con dos (!) tenedores.

2. Tipos abstractos de datos

El término «tipo» tiene su origen en la lógica matemática. Hizo su aparición para resolver ciertas paradojas que afectaban a los fundamentos de la Matemática planteadas a finales del siglo pasado y comienzos de éste. Básicamente, un tipo caracterizaba el conjunto de objetos que podían servir de dominio de alguna familia de predicados o funciones. En concreto, lo que se pretendía era evitar contradicciones debidas a la aplicación, intuitivamente inadecuada, de un cierto predicado o función sobre objetos de una clase distinta a la que, de alguna manera, se había previsto originalmente.

El término «tipo» aparece en informática con el Algol 60. Sirve para diferenciar las clases de datos que aparecen en el lenguaje: enteros, reales y booleanos, y para impedir que operaciones de un «tipo» se apliquen sobre otros «tipos» (por ejemplo, que una operación booleana se aplique sobre números reales). Como se puede ver, este concepto de tipo está íntimamente relacionado con el original. Como los tipos designan clases de datos, pasan a ser conocidos como «tipos de datos». Pronto el término se vulgariza y se extiende: la vulgarización ocasiona que el término tipo de datos se asocie, exclusivamente, con conjuntos significativos de datos básicos (las operaciones se olvidan) con posible distinta representación interna. La extensión ocasiona que se aplique también a clases de datos de otros lenguajes (Fortran, PL/1, etc.). Puntos importantes a señalar: 1) en estos lenguajes se consideran tipos

de datos, *exclusivamente*, a las clases de datos «básicos» proporcionados por el lenguaje; y 2) un tipo es *exclusivamente* un conjunto de datos. Más adelante, Pascal eliminaría la 1.ª característica: en efecto, Pascal considera tipos de datos no solamente los básicos (integer, real, bool y char) sino también los tipos «estructurados» proporcionados por el lenguaje (array, record, etc.) así como nuevos tipos que puede definir el usuario. En cambio, Pascal mantendría la 2.ª característica, esto es, en Pascal los tipos siguen siendo conjuntos.

Sin embargo, ya antes de la aparición de Pascal aparecía la idea de que hablar de tipos sin hablar de operaciones tenía poco sentido. Más aún, se comenzó a ver que lo caracterizaba a un tipo, como tal, no eran sus objetos, sino las operaciones que generaban y manipulaban dichos objetos. Esto es así por varias razones:

- En informática no tiene sentido hablar de los objetos de un tipo si estos no pueden ser generados por el ordenador, o dicho de otra forma, obtenidos con las operaciones de que se disponen, considerando como operaciones las constantes (esto es, el 0 sería una operación que produce siempre el valor cero). En efecto, supongamos que en un tipo T existe un valor **v** que no puede ser generado de ninguna manera, ni nos podemos referir a él porque no existe ninguna constante que denote este valor, entonces **v** no tendría ninguna utilidad práctica ya que nunca podría ser tratado o «procesado» y no olvidemos que la función de los ordenadores es «procesar datos».
  Las propiedades de un tipo de datos son en realidad las propiedades de sus operaciones. Por ejemplo, lo que caracteriza a los enteros no son los valores 0,1, -1, 2, -2,... ya que cualquier otro conjunto infinito numerable, como los naturales o los racionales sería entonces idéntico a los enteros. Lo que realmente distingue a los enteros de los naturales es su diferente estructura: los enteros son un anillo conmutativo con elemento unidad y los naturales un semianillo conmutativo con elemento unidad, o dicho de otra forma, los enteros son un conjunto con operaciones de suma y producto, conmutativas, asociativas, distributivas (del producto con respecto a la suma) con elementos neutros y con inversos respecto a la suma, mientras que los naturales tienen las mismas características salvo la existencia de inversos respecto a la suma.

De aquí surge el nuevo concepto de tipo de datos: un tipo de datos es un conjunto de objetos y una familia de operaciones que generan y manipulan dichos objetos; entre las operaciones se encuentran las constantes que son funciones sin parámetros que generan siempre el mismo objeto. Por ejemplo, el tipo booleano podría definirse como un conjunto de dos objetos {t,f} y la familia de operaciones: *true*, constante que genera el objeto t, *false* constante que genera el objeto f; *and* y *or* funciones de dos parámetros con la definición habitual y *not* función de un parámetro, también con la definición usual.

Queda ya sólo por ver que es un tipo *abstracto* de datos. La programación clásica, partía de la base de que programar consistía en diseñar un algoritmo, más o menos complejo, que trabajase sobre tipos de datos básicos, con la ayuda de estructuras de datos más o menos sofisticadas. Los primeros enfoques de programación «estructurada» [8, 32] intentaron sistematizar el diseño del algoritmo por medio de una descomposición sucesiva del problema en subproblemas, de tal forma que se pudiera abordar dicho diseño a distintos niveles de abstracción. La herramienta básica utilizada en este tipo de enfoque era la abstracción funcional encarnada en los procedimientos y funciones. Sin embargo, este enfoque adolece de dos defectos graves:

1. El diseño de las estructuras de datos sobre las que trabajará el algoritmo es, a menudo, tan complicado como el diseño del propio algoritmo, pero ni este método, ni los lenguajes de programación clásicos proponían ningún mecanismo de abstracción que simplificara o intentara sistematizar el diseño de estructuras de datos.
2. El diseño de un algoritmo por descomposición en sucesivos niveles de abstracción exclusivamente funcionales, presenta un desequilibrio claro: es difícil describir una acción de un nivel de abstracción alto en términos de objetos de un nivel de abstracción bajo: los tipos de datos básicos.

Estos dos problemas fueron resueltos con la introducción de los tipos «abstractos» de datos. Un tipo abstracto de datos (t.a.d) es, simplemente, un tipo de datos que no necesariamente se encuentra implementado en la máquina o lenguaje de programación utilizado, y que, en consecuencia, a menudo habrá que implementar. Veamos como esta noción sirve para resolver los dos problemas apuntados más arribas:

1. Los t.a.d.'s constituyen la herramienta adecuada para el diseño de estructuras de datos. En efecto, partiendo de la base de que una clase de estructuras de datos (por ej. las pilas), junto con sus operaciones de creación, consulta y modificación son un t.a.d. se consigue, en primer lugar, un método de diseño descendente de estructuras de datos. En efecto, si tenemos que diseñar una estructura de datos con unas ciertas operaciones, esto es un t.a.d. T1, buscaremos tipos T2, T3,... Tn, cada uno de un nivel de abstracción más bajo (i.e. cada uno más cercano a las estructuras proporcionadas por el lenguaje utilizado) e implementaremos T1 en T2, T2 en T3, etc., hasta llegar al nivel de abstracción del lenguaje.
2. Los tipos abstractos de datos «equilibran» el diseño descendente. En efecto, en un diseño con t.a.d.'s las posibilidades abstracciones funcionales utilizadas serán operaciones sobre un t.a.d., en consecuencia el nivel de abstracción de cada operación irá ligado al nivel de abstracción de los datos que trate.

Recapitulando, un tipo abstracto de datos es un conjunto de objetos junto con una familia de operaciones con las que se generan y manipulan los objetos del tipo. Entre las operaciones se encontrarán las constantes como operaciones de cero argumentos (en algunos casos, por ejemplo en los tipos de base, todos los objetos son considerados constantes, en otros, las constantes serán sólo los elementos significativos del tipo, por ejemplo en el tipo pila, la pila vacía). Esta definición corresponde al concepto de álgebra en álgebra universal, lo cual ha permitido establecer toda una teoría algebraica, de t.a.d.'s, de cara a su especificación, diseño, implementación y verificación [1, 14, 19].

Para terminar, veamos que quiere decir, en programación, implementar un tipo de datos.

La implementación de un tipo de datos tiene dos partes. En primer lugar, es necesario definir la representación de los objetos del tipo que se desea implementar en término de los objetos del tipo que sirve de implementación. En segundo lugar es necesario definir las operaciones del tipo como procedimiento o funciones que trabajan sobre

la representación. Por ejemplo, supongamos que deseamos implementar, en la forma habitual, el tipo pila de enteros, con operaciones pvacia (constante que nos define la pila vacía), push (que apila un elemento en la pila), pop (que nos suprime el último elemento apilado) y top (que nos dice cual es el último elemento apilado). En un pseudo-Pascal, la implementación sería:

```
Representación

piladeent = record
              A: array [1.. max] of integer; h:0.. max
              end

Operaciones

procedure pvacia (var p:piladeent)
 begin p. h: = , end
procedure push (var p:piladeent; n:integer)
 begin
   with p do
     if h = max then error
       else begin h: = h+1; A [h]: = n
            end
end
   procedure pop (var p:piladeent)
    begin
      with p do
        if h = 0 then error
          else h: = h–1
   end

   function top (p:piladeen): integer
    begin
      with p do
        if h = 0 then error
          else top: = A [h]
   end
```

A partir del momento en que la implementación está hecha; si el lenguaje está diseñado para trabajar con tipos abstractos de datos se podrían declarar variables de dicho tipo que podrían ser manipuladas con las operaciones implementadas *y sólo con ellas*, es decir si declaramos la variable:

var: pilal: piladeent

podríamos efectuar operaciones como pvacia (pilal), push (pilal, 5) o pop (pilal), o hacer x:=top (pilal) (si x es una variable entera), pero en ningún caso podríamos acceder directamente a la representación de pilal y hacer operaciones del tipo: pilal.h := pilal.h-5. Las ventajas que nos aporta esta restricción son las siguientes:

–Modularidad e independencia de la representación: si se comprueba que otra implementación del tipo de datos es más eficiente, basta cambiar la implementación sin alterar el resto del programa.
– Fiabilidad: la modularidad de un programa disminuye su complejidad y por tanto aumenta su fiabilidad. Además, la programación conjunta de todas las operaciones que afectan a un tipo de datos garantiza mejor su coherencia interna.

3. Procesos concurrentes como tipos abstractos de datos

Un proceso concurrente puede ser considerado como un tipo abstracto de datos. Esta afirmación, que a continuación probaremos, servirá de base a todo el estudio comparativo que haremos en este artículo de los distintos mecanismos de sincronización.

Hasta el momento, los tipos abstractos de datos que hemos visto eran tipos de «valores» (simples o estructurados), esto es, las operaciones de un tipo de datos podían, básicamente, dividirse en dos clases: las que producían «valores» del tipo en cuestión y las operaciones de consulta que devolvían «valores» de otros tipos. Por ejemplo, en el caso de las pilas las operaciones pvacia, push y pop producen «valores» de tipo pila, mientras que top devuelve enteros. Para considerar los procesos como tipos de datos, solo tenemos que ampliar la idea intuitiva de «valor», y pensar que un valor puede ser un «estado» de un proceso o una instrucción.

En concreto, consideramos que un proceso es un tipo de datos, cuyos valores son los estados en que se puede encontrar el proceso, cuyas operaciones son: una operación de inialización, que se ejecutará al comenzar a trabajar el proceso, una serie de operaciones sobre el proceso que serán ejecutadas a petición de otros procesos y, finalmente, una operación que, en forma indeterminista indica qué operaciones (que llamaremos órdenes) está deseando ejecutar sobre otros procesos. Por ejemplo, en un sistema concurrente formado por tres procesos: productor-buffer-consumidor. El buffer podría venir definido por las operaciones:

bvacio: buffer
(inicialización del buffer)
enviar (buffer, mensaje): buffer
(operación que mete un mensaje en el buffer)
recibir (buffer): buffer
(operación que saca un mensaje del buffer y, por medio de una interacción como ya veremos, envía al consumidor)
sigorden (buffer): orden
(operación indeterminista que indica qué operación u operaciones está el buffer deseando ejecutar sobre los otros procesos). Las órdenes que el buffer puede querer ejecutar serán:

producir
consumir (consumidor, mensaje)

Cada una de las órdenes que un proceso está dispuesto a emitir a otros procesos tiene asociada una operación de estos procesos y es considerada complementaria de otro orden dentro del juego de órdenes de los mismos.

Siguiendo con el ejemplo, el productor tendría como operaciones:

Prodprimermens: productor
(inicialización del productor, produce el primer mensaje)
producir (productor): productor
(operacion por la que el productor produce un nuevo mensaje)
sigorden (productor): orden
(operación que indica la siguiente orden que quiere ejecutar el productor). Las órdenes del productor son:
enviar (buffer, mensaje)
El consumidor tendría como operaciones:
inic: consumidor
(inicialización del consumidor)
consumir (consumidor, mensaje): consumidor
(operación por la que el consumidor consume un nuevo mensaje)
sigorden (consumidor): orden
Las órdenes del consumidor son:
recibir

En este caso, la orden del buffer *consumir* (consumidor, mensaje) tiene asociada la operación consumir del consumidor, y es complementaria de la orden *recibir* del consumidor, análogamente para el resto de las órdenes.

El esquema de funcionamiento e interacción en un sistema de procesos concurrentes será considerado de la si-

guiente forma: Los procesos comienzan con su inicialización, si en un momento dado dos procesos p1 y p2 están deseando ejecutar órdenes complementarias, se puede producir la interacción que consiste en que sobre cada uno de los procesos se ejecute la operación asociada a la orden emitida por el otro proceso. Las interacciones se considera que se producen en forma indeterminista.

Siguiendo con el ejemplo anterior el sistema funcionaría de la siguiente forma:

En principio se inicializarían los tres procesos con sus respectivas operaciones de inicialización. En este momento tendremos sigorden (productor) = *enviar* (buffer, ml) donde ml es el mensaje recién producido, sigorden (buffer) = *producir* y sigorden (consumidor) = *recibir*. En este momento, las órdenes que el productor y el buffer desean ejecutar son complementarias, con lo que podría producirse la interacción, ejecutándose sobre el buffer la operación

enviar (buffer, ml)

y sobre el productor

producir (productor)

Ahora, tendremos:

sigorden (productor) = *enviar* (buffer, m2)

(m2 será el segundo mensaje producido)

sigorden (buffer) = *producir* o *consumir* (consumidor, ml)

(en este momento, el buffer desea, o bien interaccionar sobre el productor o bien sobre el consumidor, pues contiene un mensaje que puede ser consumido).

sigorden (consumidor) = *recibir*

En este momento el buffer podría interaccionar con el consumidor y con el productor, indeterminísticamente se produciría una interacción, supongamos que vuelve a interaccionar con el productor, i.e. sobre el productor se ejecutaría la operación producir y sobre el buffer la operación enviar, ahora la situación sería:

sigorden (productor) = *enviar* (buffer, m3)

(m3 es el tercer mensaje producido)

sigorden (buffer) = *producir* o *consumir* (consumidor, ml)

sigorden (consumidor) = *recibir*

De nuevo se podría producir las dos interacciones, si ahora el buffer interaccionara con el consumidor, sobre el buffer se ejecutaría la operación recibir, que eliminaría ml del buffer y sobre el consumidor la operación consumir (consumidor, ml). En este momento, la situación sería:

sigorden (productor) = *enviar* (buffer, m3)

sigorden (buffer) = *producir* o *consumir* (consumidor, m2)

sigorden (consumidor) = *recibir*

etc.

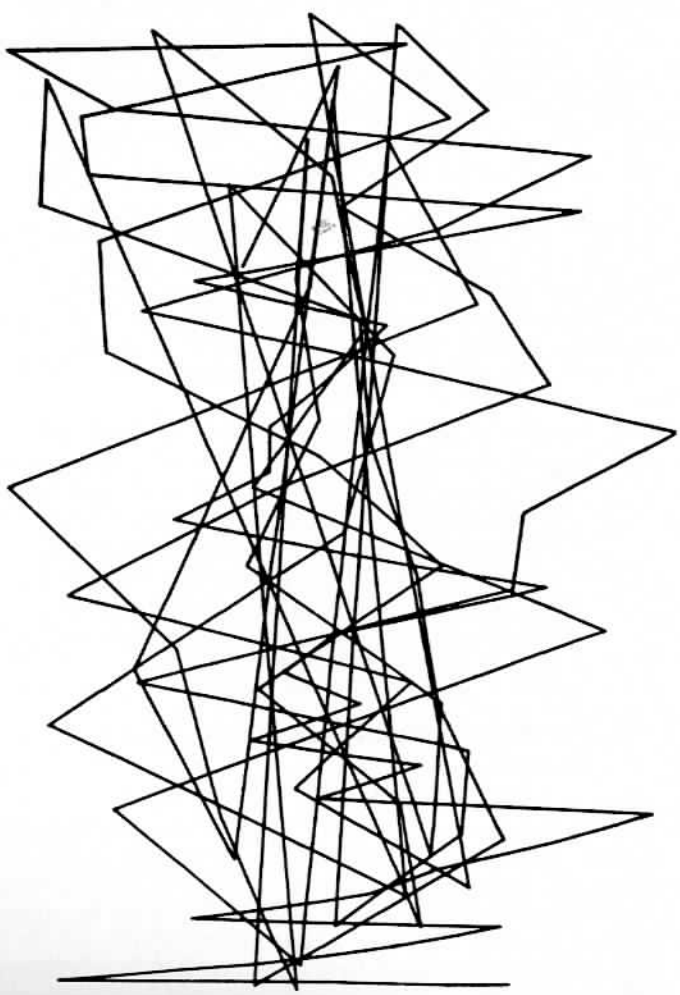
Resumiendo, un proceso es un tipo abstracto de datos, cuyas operaciones son ejecutadas «a petición» de otros procesos, pudiendo él a su vez «pedir» la ejecución de operaciones de los otros procesos. Eso es, un proceso es un tipo de datos que «interacciona» con otros procesos (tipos de datos) con la ejecución cruzada de órdenes.

4. Monitores

Los tipos abstractos de datos entraron, desde el principio, por la puerta grande de la programación concurrente. La primera construcción de un nivel de abstracción realmente alto, que apareció en los lenguajes de programación concurrente fue la de monitor [22], extensión del concepto de clase de simula [9] (protoaparición de los tipos abstractos de datos en un lenguaje de programación). Previamente, habían aparecido construcciones como los semáforos y los buffers, que constituían particularizaciones del concepto de monitor.

La idea inicial de monitor consistía, básicamente, en extender el concepto de clase en el sentido de garantizar la ejecución de sus operaciones bajo régimen de exclusión mutua, o lo que es lo mismo, mientras un proceso está ejecutando una operación del monitor, nadie más puede manipular el monitor. En principio, un proceso puede pedir la ejecución de una operación del monitor en cualquier momento, incluso aunque el monitor no esté preparado para responder: por ejemplo, el productor puede ejecutar la operación recibir sobre el buffer (considerado como monitor) aunque éste esté vacío; en tal caso, la respuesta del monitor es una orden de espera que causa una «salida del proceso del monitor» o lo que es lo mismo, otro proceso puede ejecutar cualquier operación del monitor. Cuando la causa de la espera ha desaparecido (por ejemplo, porque se haya enviado algún mensaje al buffer) la operación que se había abandonado prosigue su ejecución.

Visto desde el punto de vista del modelo descrito en la sección anterior, el esquema de funcionamiento con monitores consiste en lo siguiente:



Hay dos clases de procesos: los «procesos» propiamente dichos y los monitores. Dos «procesos» no pueden interaccionar si no es a través de un monitor, esto es, la interacción es siempre «proceso» –monitor: En principio, puede estar admitida la interacción monitor-monitor, pero en tal caso la sincronización se complica ya que las órdenes de espera deben propagarse a través de las llamadas pendientes de operaciones, i.e. si el «proceso» P ejecuta una operación del monitor M1, éste una del monitor M2,... éste una del monitor Mn, si en este momento se produce una situación de espera, la orden debe propagarse hasta el proceso P, evitando que ningún monitor quede bloqueado. Para que se produzca la interacción basta que uno de los dos la desee; esto es, si uno está dispuesto a interaccionar el otro también lo está siempre. La interacción se produce por la ejecución de una orden de uno de ellos sobre el otro, esto es la interacción no es mutua. Un «proceso» puede interaccionar sobre un monitor con cualquiera de las operaciones del monitor. Sin embargo, el monitor sólo puede interaccionar con el «proceso» con una orden de espera o con una orden de continuación (con los resultados obtenidos por el monitor).

Como puede verse, el modelo de concurrencia está basado en la asimetría. Para que dos procesos interaccionen basta que uno esté preparado, la interacción se produce sobre un proceso únicamente, y finalmente, lo que puede ser lo más grave, unos procesos tienen restringida la interacción a determinadas operaciones, mientras que otros no. La razón de esta asimetría viene justificada en el sentido de que los monitores se asocian, en un sistema de procesos concurrentes, a los recursos (físicos o lógicos) y en consecuencia rigen la competencia entre el resto de los procesos por su adquisición. Sin embargo, existen varias razones para huir de esta asimetría:

- La primera razón es de orden formal: La asimetría dificulta la definición de modelos teóricos, lo que implica una dificultad de comprensión de la concurrencia (comprendida a través de los monitores).
- La segunda razón es de orden metodológico (y por tanto práctico): La división arbitraria de procesos en dos clases: los que utilizan los recursos y los que son considerados o gobiernan los recursos dificulta la programación de casos como los siguientes: Dos procesos que interaccionan libremente, o un proceso que es simultáneamente recurso y usuario de otros recursos como puede ser el caso de módulos de un sistema operativo. En concreto, esta dificultad artificial de programar creada por el uso de monitores puede verse por ejemplo en el sistema SOLO, que siendo en un sentido admirable por lo que representó en su día, puede verse que para lo que, conceptualmente, bastaría un proceso se utilizan (en terminología de Pascal concurrente, lenguaje en el que está escrito el sistema SOLO) un monitor, una clase y un proceso (por ejemplo, en el control de pantalla).

5. Dp, Csp, Tasking, Ccs

Las razones expuestas anteriormente, junto con otras de orden más práctico llevaron a la definición de otros modelos en los que se eliminaba, total o parcialmente, la asimetría introducida por los monitores. Estos modelos han sido fundamentalmente, DP (distributed processes) de Brinch Hansen [5] , CSP (communicating sequential processes) de Hoare [23], tasking de ADA [10] y CCS (Calculus of communicating systems) de Milner [27].

DP constituye la extensión más evidente del concepto de monitor.

En DP un proceso viene definido por una serie de recursos (datos), una operación de inicialización, y una serie de operaciones (procedimientos) que son ejecutadas a petición de otros procesos y que actúan sobre los recursos del proceso. La forma de funcionamiento y sincronización se establece como sigue: un proceso P comienza ejecutando su inicialización, en un momento dado pueden ocurrir tres cosas:

1. Que termine el cálculo que estaba realizando.
2. Que deba de esperar a que se cumpla una cierta condición.
3. Que quiera ejecutar una operación sobre otro proceso Q.

En cualquiera de los dos primeros casos, si uno o más procesos están intentando ejecutar una de sus operaciones, en forma no determinista se elegirá una de las peticiones y la intentará satisfacer, al terminar, si se hubiera partido de una situació de tipo dos, y ahora se cumpliera la condición que esperaba, continuará con el trabajo pendiente; en caso de que nadie esté intentando ejecutar una de sus operaciones, esperará hasta que esto ocurra. En el caso 3) el proceso P esperará hastas que **Q** ejecute la operación solicitada, una vez hecho ésto continuará con su trabajo.

Es decir, en términos de nuestro modelo, DP. puede ser explicado así:

- Un proceso es un tipo de datos que interacciona con otros. Un proceso P interacciona con otro **Q** por la ejecución de una orden de P sobre **Q**. Para que una interacción se pueda producir basta que un proceso la desee.

Como se puede ver, se han eliminado parte de las asimetrías de los monitores:

- Sólo hay una clase de procesos.
- Cada proceso puede interaccionar con otro sin restricción inicial en el juego de órdenes..

Sin embargo, se mantienen las asimetrías en la forma de interacción:

- Sólo un proceso acciona sobre el otro.
- Sólo un proceso tiene que desear la interacción.

Estas dos asimetrías, más o menos conceptuales, se concretan en otra asimetría más observable:

- Existen dos formas de espera distintas: la ocasionada por las situaciones 1) y 2), y la ocasionada por la situación 3); en la primera, el proceso es capaz de atender requerimientos de otros procesos; en la segunda, el proceso no puede hacer nada. Esta asimetría provoca, una vez más, consecuencias desagradables que pueden dificultar, en algunos casos, la programación en DP y en otros casos provocar ineficiencias inútiles. En concreto, si estudiamos el ejemplo productor-buffer-consumidor, podemos observar lo siguiente:

- Supongamos que el productor ha producido un mensaje m. A partir de este momento, intentará ejecutar la orden sobre el buffer: *enviar* (buffer, m), con lo que pasará a la espera; supongamos que su requerimiento es atendido, el productor todavía esperará hasta que se complete la ejecución de la orden, esto es, hasta que el buffer haya terminado de almacenar

el mensaje. Sin embargo, esto puede causar una cierta ineficiencia: si el buffer no está lleno, no existe razón para que una vez ha sido emitida la orden, el productor no comience a producir un nuevo mensaje, es decir, sin esperar a que el buffer acabe de almacenar el mensaje. Este sería un caso de espera inútil, para minimizarla sería necesario que las tareas, a realizar en los procedimientos accesibles por otros procesos, fueran mínimas: sin embargo, esto dificultaría artificialmente la programación.

- Supongamos, que en la misma situación que en el caso anterior (el productor ha producido un mensaje), el buffer se encuentra lleno, en principio, podríamos efectuar la misma argumentación que antes (i.e. que el productor, una vez emitida la orden, puede seguir trabajando) suponiendo que es al proceso buffer a quien afecta el problema y no al productor. Sin embargo, parece razonable pensar que no tiene sentido que el productor produzca más mensajes si el buffer no va a ser capaz de almacenarlos. El problema entonces aparece en el hecho de que un proceso debe esperar como consecuencia de una situación ocurrida en el interior de otro proceso y que, en cierta manera les es ajena (sería distinto si el buffer, se negara a atender el requerimiento del productor de meter un mensaje, si está lleno), lo que conlleva una falta de transparencia y privacidad de los procesos y, por tanto, una falta de fiabilidad.

CSP es un modelo en que los procesos interaccionan por el envío de mensajes. Es decir, las únicas operaciones que un proceso puede ejecutar sobre otro son enviar y recibir. Además, para que dos procesos P y **Q** puedan interaccionar deben ocurrir dos cosas: 1) P y **Q** deben conocerse mutuamente; y 2) debe querer enviar mensajes a **Q** y **Q** debe querer recibir un mensaje de P (o viceversa). En este caso, como puede verse, la simetría es total. En términos de nuestro modelo CSP vendría descrito, simplemente, poniendo la limitación de que las órdenes pueden llevar: o bien dos parámetros (el nombre del proceso sobre el que se quiere ejecutar la orden y el mensaje que se pasa) para la orden de enviar, o bien un parámetro (el nombre del proceso) para la orden de recibir.

Como puede comprenderse la mayor limitación viene precisamente en la limitación en el bajo nivel de abstracción de la forma de interacción, que en determinados casos (cuando la interacción no se ajusta a lo previsto y sea compleja) dificultará la programación en CSP.

Tasking es la forma de concurrencia utilizada en el lenguaje ADA. Básicamente, es similar a DP con la eliminación de una de las asimetrías de que adolece DP:

- Para que se realice una interacción, es necesario que los dos procesos estén de acuerdo.

Sin embargo, se mantiene la asimetría en la forma de interacción, i.e. la interacción consiste en la ejecución de *una* orden sobre *uno* de los procesos. No obstante, esta asimetría no parece causar graves problemas. De hecho, los mayores problemas de ADA (en cuanto a concurrencia) parecen venir, no del modelo de concurrencia utilizado, sino de restricciones impuestas por el lenguaje. En concreto, de la imposibilidad de definir procesos (tasks) parametrizados.

Finalmente, CCS, es un modelo (o mejor, un cálculo) para el estudio de la sincronización de sistemas de procesos concurrentes, CCS es, probablemente, de todos los modelos estudiados el que tiene menos vocación de lenguaje de programación y más de herramienta teórica: en este momento es, probablemente, junto con otros enfoques en cierta forma emparentados (como las redes de Petri [] o las relaciones infinitarias de Nivat [30] una de las herramientas más utilizadas para el estudio de la concurrencia, por ejemplo, constituye la base utilizada por Hoare para la descripción formal de CSP.

En CCS un proceso es, básicamente, un autómata de estados finitos no determinista. El alfabeto de cada uno de los procesos (autómatas) es el conjunto de acciones que ejecuta el proceso. Un subconjunto de este alfabeto lo constituyen las acciones externas (en nuestro contexto órdenes) que ejecuta el proceso. Cada acción externa tiene un complementario que puede pertenecer al alfabeto de uno o más procesos del sistema. La interacción de dos procesos ocurre cuando éstos ejecutan simultáneamente una acción externa y su complementaria.

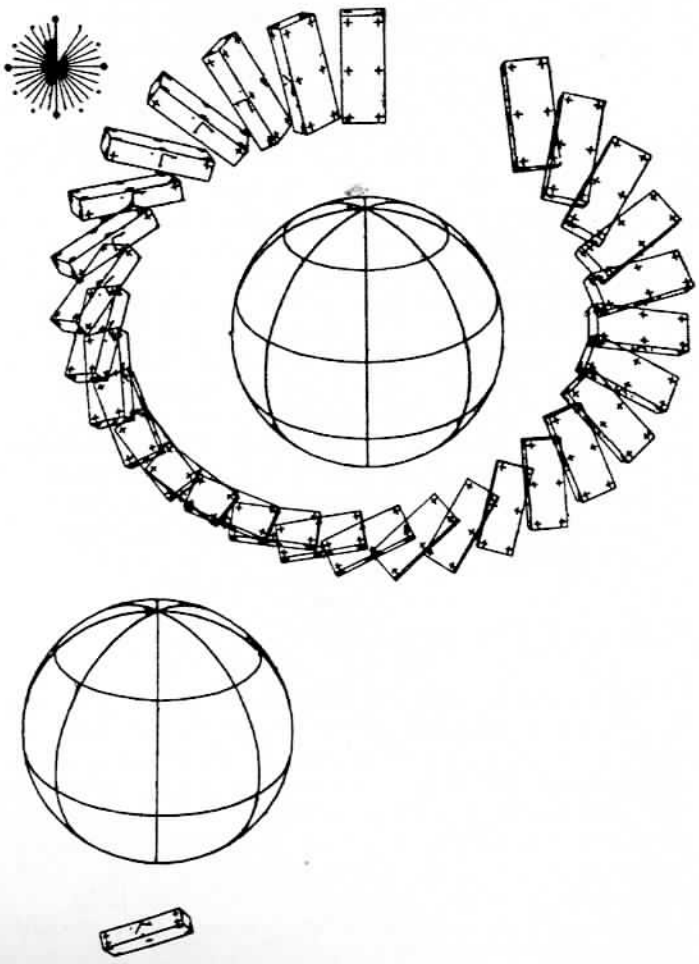
Como puede verse, nuestro modelo de tipos abstractos de datos está basado (al menos en un cierto nivel de intuición) en CCS, la diferencia, evidentemente, estriba en las teorías subyacentes (t.a.d.'s y autómatas), aunque, incluso a este nivel existe parentesco: el álgebra es el fundamento de ambas.

«Ha sonado la hora del Algebra»

Gunther Grass: «El rodaballo»

6. Referencias

[1] ADJ (Goguen, J.A.; Thatcher, J.W.; Wagner, E.G.) *«An initial algebra approach to the specification, correctness and implementation of abstract data types»*, en R. Yeh (ed.) *«Current Trends in program-*



ming methodology; IV: Data structuring», pp. 80-149, Prentice-Hall, 1978.
[2] ADJ (Thatcher, J.W.; Wagner, E.G.; Wright, J.B.) *«More on advice on structuring compilers and proving them correct»*, IBM Research Report RC-7588, 1979.
[3] Bergstra, J.A.; Tucker, J.V. *«Algebraically specified programming systems and Hoare's logic»*, en Proceedings ICALP-81, S. Even, O. Kariv (eds.), LNCS 115, Springer-Verlag, 1981.
[4] Brinch Hansen, P. *«The architecture of concurrent programs»*, Prentice-Hall, 1977.
[5] Brinch Hansen, P. *«Distributed processes: a concurrent programming concept»*; CACM 21, 11 (Nov. 1978), pp. 934-941.
[6] Brodie, M.L.; Zilles, S.N. (eds.) *«Proceedings of the workshop on data abstraction, data bases and conceptual modelling»*, Colorado 1980.
[7] Broy, M.; Wirsing, M. *«Programming languages as abstract data types»*, proc. 5th.; CAAP, M. Dauchet (ed.), Lille 1980.
[8] Dahl, O.J.; Dijkstra, E.W.; Hoare, C.A.R. *«Structured Programming»*, Academic Press, 1972.
[9] Dahl, O.J.; Myhrhaug, B.; Nygaard, K. *«The SIMULA 67 Common base language»*, Publ. S-22, Norwegian Computing Center, Oslo 1970.
[10] Department of Defense *«The programming language ADA. Reference Manual»*, LNCS 106, Springer-Verlarg, 1981.
[11] Dijstra, E.W. *«Hierarchical ordering of sequential processes»*, Acta Informática 1, 2, pp. 115-138 (1972).
[12] Dijkstra, E.W. *«American programming's plight»*, Software Engineering Notes 6, 1 (Jan. 1981), p. 5
[13] Dosch, W.; Mascari, G.; Wirsing, M. *«On the algebraic specification of data bases»*, aparecerá en Int. Conf. on Very Large Data Bases, Mexico City, 1982.
[14] Ehrich, M.D. *«On the theory of specification, implementation and parameterization of abstract data types»*, J.A.C.M. 29, 1 (Jan. 1982), pp. 206-227.
[15] Gaudel, M.C. *«Generation et preuve de compilateurs basées sur une sémantique formelle des langages de programmation»*. These d'Etat, París 1980.
[16] Goguen, J.A. *«Some design principles and theory for OBJ-O, a language to express and execute algebraic specifications of programs»*, en *«Proc. Inf. Conf. on Math. Studies of Inform. Processing»*, LNCS 75, pp. 429-475, Springer-Verlag, 1978.
[17] Goguen, J.A.; Burstall, R.M. *«CAT, a system for the structured elaboration of correct programs from structured specifications»*, Research Report, Computer Sc. Dept. SRI Int.
[18] Goguen, J.A.; Parsaye-Ghomi, K. *«Algebraic denotational semantics using parametrized abstract modules»*, en *«Formalization of programming concepts»*, J. Díaz e I. Ramos (eds.), LNCS 107, Springer-Verlag, 1981.
[19] Guttag, J.V.; Horning, J.J. *«The algebraic specification of abstract data types»* Acta Informática, 10, (Jan. 1978), pp. 27-52.
[20] Guttag, J.V.; Horowitz, E.; Musser, D.R. *«Abstract data types and software validation»*, Research Report ISI/RR-76-48, 1976.
[21] Guttag, J.V. *«Abstract data types and the development of data structures»*, C.A.C.M. 20, 6 (June 1977), pp. 396-404.
[22] Hoare, C.A.R. *«Monitors: an operating system structuring concept»*, C.A.C.M. 17, 10 (Oct. 1974), pp. 549-557.
[23] Hoare, C.A.R. *«Communicating sequential processes»*, C.A.C.M. 21, 8 (Aug. 1978), pp. 666-677.
[24] Levesque H.; Mylopoulus, J. *«A procedural semantics for semantic networks»*, en *«Associative networks»*, N. Findler (ed.), pp. 93-120, Academic Press 1979.
[25] Lucena, C.J.P.; Pequeno, T.H.C. *«Program derivation using data types: A case study»*, IEEE Trans. on Soft. Eng., Vol. SE 5,6 (Nov. 1979), pp. 586-592.
[26] Maibaum, T.S.E. *«Data base instances, abstract data types and data base specifications»*, próxima publicación.
[27] Milner, R. *«A calculus of communicating systems»*, LNCS 92, Springer-Verlag, 1980.
[28] Mosses, P. *«A constructive approach to compiler correctness»* en ICALP 80, J. de Bakker, J. Van Leuwen (eds.), LNCS 85, pp. 449-469, Springer-Verlag, 1980.
[29] Musser, D.R. *«Abstract data types specification in the AFFIRM system»*, IEEE Trans. on Soft. Eng., Vol SE-6, 1 (Jan. 1980), pp. 24-32.
[30] Nivat, M. *«Infinitary relations»* Research Report 54, INRIA, 1981.
[31] Orejas, F. *«An equational approach to concurrency»* proc. 2as. Jornadas Hispano-Francesas de Informática Teórica y Programación, F.J. Garijo (ed.), San Sebastián, 1982.
[32] Orejas, F. *«On the equational specification of concurrent processes»*. Sin escribir todavía.
[33] Pair, C. *«Abstract data types and algebraic semantics of programming languages»*, Theoretical Computer Science 18, 1 (April 1982), pp. 1-32.
[34] Peterson, J.L. *«Petri Nets»*, Computing Surveys 9, 3 (Sept. 1977).
[35] Wirth, N. *«Program development by stepwise refinment»*, C.A.C.M. 14, 4 (April 1971), pp. 221-227.