1 Tipos de datos

> The un-named is heaven and earth's origin;
> Naming is the mother of the thousand things.
> Whenever there is no desire, one beholds the mistery;
> Whenever there is desire, one beholds the manifestations.
> These two have the same point of departure but differ the naming.
> Their identity is *hsüan*(kaos)-*hsüan* beyond *hsüan*, all mystery's gate
>
> Lao Tzu (traducción Alan Wats)
> *Tao Te Ching*

Los tipos de datos son uno de los temas fundamentales de ciencias de la computación. Una prueba de ello es la amplia literatura en temas tales como estructuras de datos, tipos abstractos de datos, algebras many-sorted, etc.

Tanto en las máquinas de Turing originales como en la posterior elaboración del estilo von Newman de computación, la separación de estructuras de datos y de control es esencial en la noción de computación. La programación imperativa se basa en estos modelos. Si bien en la programación funcional las funciones - las cuales cumplen el rol de los programas - son datos, también aquí la idea de abstracción de datos es una de las herramientas fundamentales para el desarrollo de programas.

1.1 Álgebras

Los tipos de datos son (familias de) conjuntos con operaciones. Por consiguiente, el concepto matemático que va a usarse para modelarlos es el de álgebra. En general las álgebras usadas para modelar tipos de datos son más complejas que las que aparecen en matemática. En particular, es necesario usar álgebras *many sorted*, es decir donde se dispone de una familia de conjuntos base y operaciones que solo están defindas para algunos de los conjuntos dados. Existen básicamente dos estilos de definir álgebras para tipos de datos. La primera consiste en enumerar las operaciones con su correspondiente aridad y establecer las propiedades que estas deben satisfacer. Para ciertas formas de definir estas operaciones se pueden construir álgebras que las satisfacen y que además tengan buenas propiedades (por ejemplo ser la más concreta, la más abstracta o la única salvo isomorfismo). Otra forma de definir un tipo de datos es elegir un modelo y definir las operaciones sobre este. Si bien este esquema es menos abstracto que el anterior, dado que es necesario definir quienes son los elementos del álgebra, para el objetivo de este curso no hay grandes diferencias con el anterior, ya que todos los tipos de datos a ser tratados dispondrán de un modelo canónico.

En cualquiera de los casos es necesario tener una forma de nombrar los conjuntos de valores involucrados así también como las operaciones. Para modelar esta idea, se recurre al concepto de signatura. Una *signatura* esta dada por una secuencia S de nombres llamados *sorts* (o géneros) y una secuencia Ω de nombres de operaciones. A cada operación de Ω se le asocia una aridad, la cual consiste en una secuencia no vacía de elementos de S. Por ejemplo, una signatura para los naturales puede ser

$$(nat \mid 0, succ, +)$$

donde la aridad de cada operación es la obvia:

$$\begin{aligned} 0 &: \to nat \\ succ &: nat \to nat \\ + &: nat \times nat \to nat \end{aligned}$$

La notación usada para la aridad es $f : s_1 \times \cdots \times s_n \to s_{n+1}$ para el caso en el cual la aridad de f es $(s_1, \ldots, s_n, s_{n+1})$

Un *álgebra* A para una signatura (S, Ω) consiste de una secuencia de conjuntos A_s y de una secuencia de funciones f^A para todo $s \in S, f \in \Omega$. Si la aridad de f es $(s_1, \ldots, s_n, s_{n+1})$, entonces la función f^A tiene como dominio $A_{s_1} \times \cdots, A_{s_n}$ y como codominio $A_{s_{n+1}}$.

Volviendo al ejemplo de los natuarles, un álgebra canónica para la signatura dada es tomar como A_{nat} al conjunto de los números naturales Nat y como operaciones a las usuales. Debe tenerse en cuenta que si se implementa un programa que trabaje sobre esta álgebra, las únicas operaciones disponibles serán las enunciadas en la signatura. Si se necesitan otras operaciones es necesario cambiar de álgebra o definir a estas en término de las anteriores.

En lo que sigue no siempre vamos a definir las signaturas de manera explícita. Por ejemplo, diremos que

$$(Nat \mid 0, succ, +)$$

es el álgebra de los naturales, donde se entiende ahora que, como es usual en matemática, $0, succ$ y $+$ son operaciones sobre los naturales y no nombres.

1.2 Tipo de interés

Si bien las álgebras que estamos considerando están formadas por una familia de conjuntos, siempre vamos a considerar a uno de ellos en especial como el principal, llamado a veces en la literatura *tipo de interés*. En lo que sigue, al definir un álgebra vamos a mencionar sólo este conjunto, dejando implícitos a los otros conjuntos involucrados. De esta manera vamos a hablar por ejemplo del álgebra de los naturales definida como

$$(Nat \mid 0, succ, +, (= 0))$$

Si bien también está involucrado el conjunto de los booleanos, no lo mencionamos de manera explícita.

Dada un álgebra, es posible clasificar las operaciones de acuerdo a su dominio y codominio. Esto nos va a permitir definir de manera más simple el concepto de implementación.

Sea $(A \mid f_0, \ldots, f_n)$ un álgebra. Las operaiciones f_i se clasifican en:

generadoras: Son aquellas operaciones con las cuales se pueden crear elementos del tipo tomando como argumentos elementos de otros conjuntos. El tipo de estas operaciones va a ser de la forma

$$f_i : T_0 \times \cdots \times T_m \to A$$

Donde los T_i son todos distintos de A.

modificadoras: Son aquellas operaciones que tienen su codominio en A y al menos uno de sus argumentos proviene también de A, es decir

$$f_i : T_0 \times \cdots \times A \times \cdots \times T_m \to A$$

en este caso los T_i pueden ser cualesquiera.

observadoras: Son aquellas operaciones cuyo codominio es distinto de A y al menos uno de sus argumentos proviene de A

$$f_i : T_0 \times \cdots \times A \times \cdots \times T_m \to T$$

donde T es distinto de A y los T_i son cualesquiera.

1.3 Ejemplos

Ene esta sección consideramos algunas álgebras útiles en programación

Ejemplo *(Booleanos).* El primer ejemplo es el de los booleanos. El álgebra tiene como conjunto asociado $Bool = \{true, false\}$ y como operaciones pueden elegirse las necesarias para la aplicación en consideración. Por ejemplo

$$(Bool \mid true, false, \equiv, \vee, \neg)$$

Las operaciones $true, false$ son generadoras y el resto modificadoras.

Ejemplo *(Naturales).* Consideramos ahora una posible álgebra para los naturales un poco más extensa que la ya presentada. Si bien las operaciones que se agregan podrían haber sido definidas en términos de las anteriores, el hecho de ponerlas como parte del álgebra permite implementaciones diferentes las cuales pueden ser más eficientes

$$(Nat \mid 0, succ, +, *, (=))$$

En este caso la operación 0 es generadora, $succ, +, *$ son modificadoras y $=$ es observadora.

Ejemplo *(Enteros).* De manera análoga, podemos considerar ahora un álgebra para los enteros. La operación $-$ de este álgebra va a ser interpretada como una operación unaria y no como la resta entre dos números.

$$(Int \mid 0, succ, -, +, *, (=))$$

Ejemplo *(Pilas).* El álgebra de las llamadas pilas (en inglés *stacks*) no es otra cosa que listas donde se inserta sólo por delante y se extraen elementos también por delante.

$$([A] \mid [\ ], \triangleright, hd, tl, (= [\ ]))$$

La operación $[\ ]$ es generadora, la operaciones $\triangleright$ y tl son modificadoras y el resto observadoras.

Ejemplo *(Colas).* El álgebra de las colas (en inglés *queues*, palabra francesa en realidad) es un álgebra de listas donde se inserta sólo por detrás y se extraen elementos por delante.

$$([A] \mid [\ ], \triangleleft, hd, tl, (= [\ ]))$$

Ejemplo *(Conjuntos).* Un álgebra posible para los conjuntos es la siguiente

$$(\{A\} \mid \emptyset, ins, rm, \in, (= \emptyset), \cup, \cap, \setminus)$$

Si se restringe el álgebra a las primeras cinco operaciones se suele llamar al tipo *diccionario*.

Ejemplo*(Multiconjuntos).* Un álgebra para multiconjuntos es la siguiente

$$(\lfloor A \rfloor \mid \emptyset, ins, min, rMin, \in, (= \emptyset), \uplus, \setminus)$$

Si se restringe el álgebra a las primeras cuatro operaciones se suele llamar al tipo *cola de prioridad*.

1.4 Implementaciones

Una de herramientas básicas para la resolución de problemas es la posibilidad de separar diferentes tareas para poder resolverlas de manera independiente y luego combinar los resultados. La modularización es un ejemplo de esto, donde una subtarea que tiene una especificación dada puede resolverse de manera independiente de la tarea principal pudiendo inferirse la corrección de esta última usando sólo la especificación de la subtarea, es decir independientemente de la implementación. La corrección de la subtarea respecto de su propia especificación es un problema aparte. Además, diferentes programas que satisfagan esa especificación pueden usarse, haciendo posible mejorar la eficiencia de un programa modificando sólo una parte sin comprometer la corrección del resto.

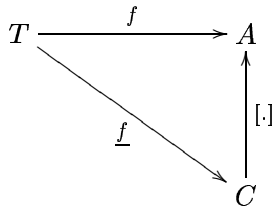
La idea básica de la abstracción de datos es permitir construir algoritmos abstractos definidos para un tipo de datos dado y luego, como tarea separada, implementar este tipo de datos en otro, sea porque el tipo original no esta disponible en el lenguaje de programación, sea por razones de eficiencia (por ejemplo pueden implementarse colas de manera tal que el costo de todas las operaciones sea casi constante). Para que este método sea efectivo es necesario contar con las herramientas teóricas que permitan determinar cuando una implementación es válida. Por otro lado, es importante conocer las propiedades del tipo abstracto, dado que van a ser necesarias para derivar el algoritmo abstracto.

Definición*(Implementación).* Dada un álgebra A llamada álgebra abstracta, y otra álgebra C llamada concreta, una implemetación de A en términos de C esta dada por

- Una función de abstracción $[\![.]\!] : C \to A$ que sea suryectiva. La suryectividad es necesaria para que cualquier valor abstracto posible tenga (al menos) un valor concreto que lo implemente.
- Para cada operación generadora $f : T \to A$ una operación $\underline{f} : T \to C$ tal que para todo $x \in T$

  $$[\![\underline{f}.x]\!] = f.x$$

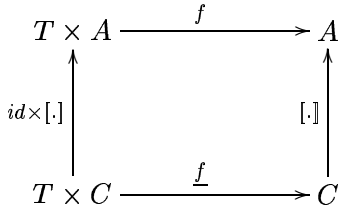
  tal como lo ilustra el siguiente diagrama



- Para cada operación modificadora $f : T \times A \to A$ una operación $\underline{f} : T \times C \to C$ tal que para todo $x \in T, c \in C$

$$[\![\underline{f}.x.c]\!] = f.x.[\![c]\!]$$

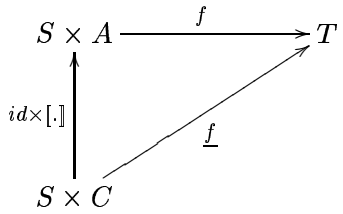
como es ilustrado en el siguiente diagrama



- Para cada operación observadora $f : S \times A \to T$ una operación $\underline{f} : S \times C \to T$ tal que para todo $x \in S, c \in C$

$$\underline{f}.x.c = f.x.[\![c]\!]$$

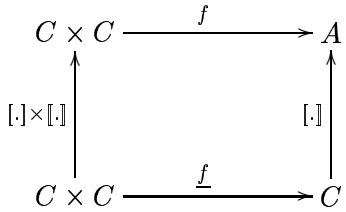
lo cual ilustra el siguiente triángulo



Esta definición no está escrita en su máxima generalidad, dado que el tipo de las operaciones consideradas puede ser más general. La generalización obvia a otros tipos se deja al lector (por ejemplo, una operación $f : A \times A \to A$ será implementada por $\underline{f} : C \times C \to C$ si para todo $b \in C, c \in C$

$$[\![\underline{f}.b.c]\!] = f.[\![b]\!].[\![c]\!]$$

o, escrito de manera diagramática



Las ecuaciones aseguran que cualquier operación concreta refleja via la abstracción lo que debe ocurrir en el álgebra abstracta.

La definición de implementación sólo esta dada para operaciones totales. Sin embargo es común que las operaciones de las álgebras no estén definidas para todos los valores posibles, por ejemplo las operaciones *hd* y *tl* del álgebra de listas sólo están definidas para listas no vacías. En estos casos, la implementación será correcta si las ecuaciones pertinentes son válidas para cualquier valor del dominio de la operación abstracta en consideración. Para valores fuera del dominio la operación concreta puede tomar cualquier valor (o no estar definida).

1.5 Ejemplos

en esta sección se presentan algunos ejemplos de implementación de álegbras.

Ejemplo *(Enteros con pares de naturales).* Una forma usual de definir los números enteros a partir de los naturales es usando pares y la siguiente función de abstracción

$$[\![(m,n)]\!] = m - n$$

Dejamos como ejercicio demostrar que la función de abstracción es suryectiva.

Vamos a considerar aqui solo las operaciones + y - (unaria) dejando las otras para el lector. Aprovechando las propiedades que sabemos deben satisfacer las operaciones concretas, podemos derivar estas obteniendo la demostración como producto esta derivación. Por ejemplo, la operación de suma (usamos $\oplus$ para la operación concreta)

$[\![(m,n) \oplus (p,q)]\!]$

$=$ $\{$*propiedad que debe satisfacer* $\oplus\}$

$[\![(m,n)]\!] + [\![(p,q)]\!]$

$=$ $\{$*definición de* $[\![.]\!]\}$

$(m-n) + (p-q)$

$=$ $\{$*álgebra*$\}$

$m + p - (n+q)$

$=$ $\{$*definición de* $[\![.]\!]$ *dado que tanto* $m+p$ *como* $n+q$ *son naturales*$\}$

$[\![(m+p, n+q)]\!]$

Si ahora definimos

$$(m,n) \oplus (p,q) = (m+p, n+q)$$

tenemos una demostración de que se satisface la propiedad requerida para operaiones de modificación, reescribiendo la derivación de la siguiente manera:

$[\![(m,n) \oplus (p,q)]\!]$

$=$ $\{$*definición de* $\oplus$, *Leibniz*$\}$

$[\![(m+p, n+q)]\!]$

$=$ $\{$*definición de* $[\![.]\!]\}$

$m + p - (n+q)$

$=$ $\{$*álgebra*$\}$

$(m-n) + (p-q)$

$=$ $\{$*definición de* $[\![.]\!]\}$

$[\![(m,n)]\!] + [\![(p,q)]\!]$

Para el operador unario $-$ damos sólo la derivación del operador concreto $\ominus$.

$[\![\ominus(m,n)]\!]$

$=$ $\{$*propiedad que debe satisfacer* $\ominus\}$

$-[\![(m,n)]\!]$

$=$ $\{definición\ de\ [\![.]\!]\}$

$-(m-n)$

$=$ $\{álgebra\}$

$n-m$

$=$ $\{definición\ de\ [\![.]\!]\}$

$[\![(n,m)]\!]$

Luego, definimos $\ominus(m,n) = (n,m)$

Ejemplo *(Booleanos con naturales).*

Una posible implementación de los booleanos es con números naturales, usando la siguiente función de abstracción

$$[\![n]\!] = n \neq 0$$

esto es, el valor 0 representará al booleano $false$ y cualquier número distinto de 0 al booleano $true$. Esta implementación es usada por algunos lenguajes de programación. Daremos una posible implementación de las operaciones concretas y dejaremos la demostración de corrección de la implementación como ejercicio.

$$\begin{aligned} \underline{true} &= 8 \\ \underline{false} &= 0 \\ p \underline{\wedge} q &= p * q \\ p \underline{\vee} q &= p + q \\ \underline{\neg} p &= 18 * n(p = 0) \end{aligned}$$

donde recordemos que n es la función que devuelve 1 si el predicado es verdadero y 0 si es falso.

Puede hacerse una implementación similar usando enteros (con la misma función de abstracción y usar valor absoluto para definir las operaciones (ejercicio).

Ejemplo *(Multiconjuntos con listas ordenadas).*

La implementación de multicnojuntos con listas ordenadas tiene una complejidad aceptable para multiconjuntos no demasiado grande, dado que todas las operaciones pueden implementarse en tiempo lineal. Más adelante vamos a ver implementaciones que pemiten una complejidad logarítmica para todas las operaciones de una colas de prioridad (con complejidad constante para el mínimo) , inclusive con la operación de unión de dos multiconjuntos. La función de abstracción es la usual, la cual puede definirse recursivamente como

$$\begin{aligned} [\![.]\!] &: [A] \to \lfloor A \rfloor \\ [\![[\ ]]\!] &= \emptyset \\ [\![x \triangleright xs]\!] &= \lfloor x \rfloor \uplus [\![xs]\!] \end{aligned}$$

Damos ahora la implementación de algunas de las operaciones, dejanco el resto como ejercicio.

$$\underline{\emptyset} = [\ ]$$

$$
\begin{array}{rcl}
xs \underline{\uplus} ys & = & mrg.xs.ys \\
xs \underline{\setminus} ys & = & diff.xs.ys \\
xs \underline{= \emptyset} & = & xs = [\ ]
\end{array}
$$

donde la función mrg es el merge de dos listas ordenadas que se usó para definir el *mergesort* y la función $diff$ se define como sigue

$$
\begin{array}{rcl}
diff.[\ ].ys & = & [\ ] \\
diff.xs.[\ ] & = & xs \\
diff.(x \triangleright xs).(y \triangleright ys) & = & (\ \ x < y \Rightarrow x \triangleright diff.xs.(y \triangleright ys) \\
 & & \Box\ x = y \Rightarrow diff.xs.ys \\
 & & \Box\ x > y \Rightarrow diff.(x \triangleright xs).ys) \\
 & &)
\end{array}
$$

Queda como ejercicio demostrar la corrección de la implementación de las operaciones.