View metadata, citation and similar papers at core.ac.uk
brought to you by CORE
provided by UPCommons. Portal del coneixement obert de la UPC

Reconciliando modularidad y eficiencia mediante atajos

Jordi Marco, Xavier Franch
jmarco@lsi.upc.es, franch@lsi.upc.es
Dept. Llenguatges i Sistemes Informàtics (LSI)
Universitat Politècnica de Catalunya (UPC)
c/ Jordi Girona, 1-3 (Campus Nord)
08034 Barcelona

Resumen

Se presenta en este artículo una propuesta para el desarrollo de programas eficientes en el marco de la programación con tipos abstractos de datos (TAD), con el objetivo de respetar la estructura modular de los programas propia de este ámbito. La propuesta se centra en el concepto de atajo como camino eficiente de acceso a los datos, alternativo al acceso mediante las operaciones propias del TAD, y se desarrolla sobre un TAD concreto, el almacén de elementos. La definición de los atajos es altamente formal, mediante especificaciones algebraicas interpretadas con semántica inicial, de manera que el resultado tiene un significado bien definido y encaja sin problemas en el marco de la programación con TAD. La eficiencia se asegura mediante una implementación adecuada del tipo, que proporciona acceso constante a los datos siguiendo los atajos sin penalizar las otras operaciones del TAD. Se ilustra la utilidad de la propuesta mediante un ejemplo concreto.

1. Introducción

La programación modular con tipos abstractos de datos (abreviadamente, TAD) [LG86] es una metodología consolidada para el desarrollo de programas a gran escala. En ella, es fundamental la distinción entre la especificación y la(s) implementación(ones) del TAD, que se traduce en la existencia de módulos diferentes para las mismas y que se resume en el denominado principio de la transparencia de la información: los usuarios de un TAD deben utilizarlo en base a las propiedades que establece la especificación, independientemente de las características que presenta su implementación, que permanece oculta. Este principio simplifica las relaciones entre los módulos que contienen las especificaciones y las implementaciones y, por ello, facilita el desarrollo de programas, pues es más sencillo escribir el código, probarlo, reusarlo y mantenerlo.

No obstante, el principio de la transparencia de la información choca, a veces frontalmente, con un requisito bastante usual sobre los programas: su eficiencia, fundamentalmente caracterizada por su tiempo de ejecución. Ello se debe a que el acceso a los datos debe respetar las propiedades que definen el TAD, que se establecieron de una manera abstracta sin considerar los problemes inherentes a su posterior implementación (como debe ser). Si el contexto de uso del TAD exige un alto grado de eficiencia de la implementación, la reutilización plena del TAD resulta imposible y debe procederse a algún tipo de modificación para adaptarlo a los requisitos; incluso, es posible que estas modificaciones sean de tal envergadura que se opte por descartar el TAD y crear un *software* enteramente nuevo.

Esta confrontación entre la eficiencia y la modularidad de los programas es un problema bien conocido en el ámbito de la programación con TADs, recogido en textos clásicos en este campo [AHU83, HS94, etc.] y resuelto casi siempre sacrificando la modularidad en aras de la eficiencia. Nos proponemos en este artículo presentar un marco general que permita reconciliar ambos criterios, obteniendo programas eficientes que reutilicen sin modificación alguna los TADs que sea necesario, y respetando el principio de transparencia de la información. La propuesta se basa en la definición de un método alternativo de acceso a los datos, los llamados atajos, a los que daremos un significado formal claramente definido

mediante su especificación algebraica y cuya implementación será altamente eficiente. Desarrollamos nuestro trabajo estudiando un TAD concreto, el TAD *ALMACÉN* de los almacenes de elementos, aunque la propuesta puede extrapolarse a cualquier otro TAD que se caracterice por albergar colecciones de datos.

El resto del artículo se organiza como sigue. En la sección 2, se introduce el concepto intuitivo de atajo. La sección 3 presenta el TAD *ALMACÉN* tal y como se definiría en la programación con TADs tradicional, mientras que en la sección 4 se modifica el TAD añadiéndole atajos. La sección 5 propone una implementación eficiente del TAD con atajos. En la sección 6 se estudia un ejemplo concreto de uso de atajos que muestra su utilidad. Finalmente, la sección 7 expone las conclusiones y el trabajo futuro.

2. Concepto de atajo

Nos proponemos pues definir un método alternativo de acceso a las estructuras de datos usadas para implementar TADs. Es decir, los usuarios de un TAD podrán acceder a los datos que contiene no sólo mediante las operaciones definidas en la especificación, que son las que determinan el modelo matemático subyacente del TAD, sino también siguiendo unos caminos alternativos cuando el uso de dichas operaciones sea demasiado costoso. A estos caminos alternativos los denominamos *atajos*.

Los atajos se crean y se destruyen dinámicamente a medida que se insertan y se sacan datos en una estructura; cada dato tiene un, y sólo un, atajo asociado, y cada atajo sirve para acceder a un único dato. Los atajos son fijos durante todo el tiempo que el dato reside en la estructura. El tiempo de acceder a un dato siguiendo un atajo se considerará O(1), lo más rápido posible; este coste quedará garantizado con la estrategia de implementación que presentaremos.

Existe un inconveniente evidente, que solucionaremos a lo largo del artículo: el añadido de atajos a un TAD hace que éste pierda el significado matemático que le dio el usuario. Es más, puede dudarse que exista siquiera un modelo bien definido. Veremos que, efectivamente, el significado del TAD cambiará, pero lo hará de una manera controlada, obteniéndose un nuevo TAD con un significado bien definido. El estudio lo haremos sobre un caso concreto, el TAD *ALMACÉN*, y comentaremos en las conclusiones cómo se podría generalizar el método que proponemos a TADs arbitrarios.

3. El TAD *ALMACÉN*

Presentamos en esta sección el TAD *ALMACÉN*, que sirve para almacenar una colección de elementos, con operaciones de creación, añadido, supresión y consulta. Supondremos que los elementos son pares <clave, información>, de manera que la supresión y la consulta serán por clave; el tipo de las claves debe proveer una operación de comparación. Consideramos un error borrar o consultar elementos usando claves indefinidas; por ello, añadiremos una operación para saber si una clave está indefinida.

Utilizamos como lenguaje de especificación la notación Merlí, que nos permite construir especificaciones algebraicas con ecuaciones condicionales, interpretadas con semántica inicial [EM85]; puede consultarse [FBB93, Fra94] para una exposición detallada, aunque basta decir que los constructores disponibles son parecidos a los que ofrecen los lenguajes usuales de esta familia (OBJ3, ACT-ONE, etc.). Los errores los tratamos de manera similar a [ADJ78]: agrupamos todos los términos en una cláusula aparte y suponemos su propagación implícita.

Para simplificar los detalles de la exposición, definiremos el TAD no parametrizado, de manera que su semántica inicial consistirá en una clase de álgebras (totales) y no en un funtor. Por ello, el TAD *ALMACÉN* importa dos universos que definen las claves y la información.

La especificación es ciertamente sencilla, y tan solo debe controlarse que la supresión de elementos no provoque ningún error (para ello necesitamos un par de ecuaciones condicionales).

universo ALMACÉN **define**
  usa CLAVE, INFORMACIÓN, BOOL
  tipo almacén
  operaciones crear: -> almacén
    añadir: almacén clave info -> almacén
    borrar: almacén clave -> almacén
    consultar: almacén clave -> info
    definida?: almacén clave -> bool
  errores borrar(crear, k); consultar(crear, k)
  ecuaciones
    añadir(añadir(A, k, v1), k, v2) = añadir(A, k, v2)
    [¬ig(k1, k2)] □ añadir(añadir(A, k1, v1), k2, v2) = añadir(añadir(A, k2, v2), k1, v1)
    [definida?(A, k)] □ borrar(añadir(A, k, v), k) = borrar(A, k)
    [¬definida?(A, k)] □ borrar(añadir(A, k, v), k) = A
    [¬ig(k1, k2)] □ borrar(añadir(A, k1, v1), k2) = añadir(borrar(A, k2), k1, v1)
    consultar(añadir(A, k, v), k) = v
    [¬ig(k1, k2)] □ consultar(añadir(A, k1, v1), k2) = consultar(A, k2)
    definida?(crea, k) = falso
    definida?(añadir(A, k1, v), k2) = ig(k1, k2) ∆ definida?(A, k2)
funiverso

Fig. 1. Un TAD para los almacenes de elementos.

Determinamos el modelo del TAD *ALMACÉN* interpretando las ecuaciones con semántica inicial. Dadas las propiedades establecidas sobre *añadir*, se puede concluir que el modelo, por lo que al conjunto de soporte del tipo *almacén* se refiere, son las funciones parciales g: K ∅ V, siendo K el conjunto de soporte de las claves y V el conjunto de soporte de la información. Las operaciones del modelo son la interpretación intuitiva de las operaciones del TAD sobre dichas funciones; por ejemplo, *crear* se interpreta como la función g que cumple dom(g) = Ø. La demostración de que realmente éste es el modelo cae fuera de los objetivos del artículo.

Una vez hemos construido la especificación del TAD, podemos construir una o más implementaciones[1]. Cada implementación tendrá unas características diferentes por lo que a la complejidad asintótica de las operaciones se refiere. En la fig. 2 se ofrece una tabla comparativa de algunas implementaciones del TAD: mediante dispersión (*hashing*), árboles binarios de búsqueda AVL, y vectores desordenados; el parámetro de eficiencia n representa el número de claves definidas. En el caso de la dispersión, se da por supuesto que la función distribuirá correctamente las claves en la tabla. Puede verse, pues, que hay implementaciones que tienen un tiempo de acceso a los elementos no constante, incluso lineal.

	Inserción	Supresión	Consulta
Dispersión	O(1)	O(1)	O(1)
Árbol AVL	O(log n)	O(log n)	O(log n)

[1] En este artículo, nos centramos en implementaciones en memoria principal (por lo que medimos su eficiencia con una notación asintótica, en concreto la O grande [Knu76, Bra85]), aunque este hecho es irrelevante para el método que se propone. En [Mar96] se proporcionan implementaciones en memoria secundaria.

Vector desordenado	O(n)	O(n)	O(n)

Fig. 2. Complejidad asintótica de las operaciones del TAD ALMACÉN.

4. El TAD *ALMACÉN* con atajos

El objetivo de esta sección es modificar el TAD *ALMACÉN* añadiendo atajos para acceder de manera alternativa a los pares <clave, información> contenidos en los objetos del tipo. Empezamos adaptando la signatura, escribiremos las ecuaciones y determinaremos el nuevo modelo del TAD. Dejamos el estudio de la implementación del TAD para la próxima sección.

Por lo que se refiere a los atajos, definimos un nuevo tipo que llamamos *atajo*. Los valores de este tipo se generan mediante las operaciones *prim_atajo* y *sig_atajo*, que se declaran privadas para que el cliente del TAD no pueda crear objetos de tipo *atajo*; esta tarea recae íntegramente en *ALMACÉN*. Proveemos además de una operación pública de comparación de atajos.

La generación de atajos tiene lugar al añadir nuevos pares <clave, información> al almacén. Por ello, declaramos que *añadir* devuelve dos resultados: el almacén con el nuevo par y, además, un atajo a dicho nuevo par. Este atajo puede ser guardado por el cliente que haya añadido el par; además, la signatura del TAD provee de una nueva operación, *atajo_de_clave*, para que en cualquier otro momento pueda obtenerse el atajo a un par concreto.

Por último, se añaden operaciones de acceso a la estructura mediante atajos: supresión, consulta (tanto de información como de clave) y modificación. Además, se introduce una operación para saber si un atajo está definido; se considerará error acceder a la estructura mediante atajos indefinidos.

En cuanto a la especificación del tipo, nos centramos en sus aspectos más relevantes (v. fig. 3). Para facilitar la escritura, introducimos una operación privada que añade los pares <clave, información> ya con un atajo. Ello es posible desde el momento que cada clave tiene un y sólo un atajo asociado. Esta operación presenta diversas propiedades que se establecen en la especificación: se presenta como error romper la relación (única y permanente) entre una clave y su atajo, y se declaran las mismas relaciones que exhibía el TAD *ALMACÉN* sin atajos (ecuaciones 1 y 2).

Un punto clave de la especificación es la estrategia de generación de atajos. Introducimos una operación privada *nuevo_atajo* que se ocupa de asociar un atajo a una clave que entra en el almacén. La política más sencilla consiste en añadir el atajo siguiente al último generado, usando *sig_atajo*, y siendo *prim_atajo* el atajo asociado a la primera clave que entró en almacén. No obstante, deben considerarse las supresiones de elementos, que dejan libres atajos generados anteriormente. Aunque de cara a la especificación podríamos desdeñar la posibilidad de reaprovechar estos atajos, si queremos posteriormente construir implementaciones viables es imprescindible reciclar esos atajos libres (que dejarán huecos en el espacio que ocupa el almacén) en las nuevas inserciones. Eso sí, destacamos que este reaprovechamiento conlleva un peligro: es imposible asegurar que todos los clientes que tenían copias del valor de un atajo reaprovechado son conscientes que la clave asociada ha cambiado; podría darse el caso de accesos al almacén mediante una copia del atajo creada con anterioridad.

Hemos añadido una nueva operación privada, *anotar_atajo*, que toma nota de que un atajo que se ha empleado en el almacén ha quedado libre (ecuación 7); cuando se borra un elemento, se realiza dicha anotación; las ecuaciones 5 a 9 aseguran que el último atajo liberado es el primero que se encuentra en un *anotar_atajo* en una exploración del término de fuera hacia dentro. La especificación de *nuevo_atajo* queda: si no hay atajos para reaprovechar, el nuevo atajo es el siguiente al último generado (ecuación 10); si hay algún atajo libre, se reaprovecha el último liberado (ecuación 11). Se introducen diversas

operaciones auxiliares para concretar esta política, cuya especificación no presenta mayores complicaciones.

Notemos que, con atajos, el modelo inicial del TAD es diferente del anterior. Por un lado, el conjunto de soporte del tipo *almacén* debe incorporar los atajos y, así, una primera propuesta de modelo serían los pares de funciones g: K ∅ V ≈ h: K ∅ A que cumplen dom(g) = dom(h) y que h es inyectiva, siendo A, K y V los conjuntos de soporte de los atajos, las claves y la información, respectivamente. El hecho de que h sea inyectiva asegura que una clave tiene un y sólo un atajo asociado, mientras que la relación entre los dominios asegura que los atajos sólo existen asociados a claves definidas.

```
universo ALMACÉN define
    usa CLAVE, INFORMACIÓN, BOOL
    tipos almacén, atajo
    operaciones
        crear: ∅ almacén
        añadir: almacén clave info ∅ <almacén, atajo>
        borrar, consultar y definida? como antes
        borrar_por_atajo: almacén atajo ∅ almacén
        consultar_por_atajo: almacén atajo ∅ info
        atajo_definido?: almacén atajo ∅ bool
        modificar_por_atajo: almacén atajo info ∅ almacén
        clave_de_atajo: almacén atajo ∅ clave
        atajo_de_clave: almacén clave ∅ atajo
        privada añadir_con_atajo: almacén clave info atajo ∅ almacén
        privada prim_atajo: ∅ atajo
        privada sig_atajo: atajo ∅ atajo
        privada nuevo_atajo, último_atajo_liberado, último_atajo_generado: almacén ∅ atajo
        privada anotar_atajo: almacén atajo ∅ almacén
        privada hay_huecos?: almacén ∅ bool
        ig_atajo: atajo atajo ∅ bool
    errores [atajo_definido?(A, q)  ¬ig(k, clave_de_atajo(A, q))]  añadir_con_atajo(A, k, v, q)
            [clave_definida?(A, k)  ¬ig_atajo(q, atajo_de_clave(A, k))]  añadir_con_atajo(A, k, v, q)
            ...
            [¬atajo_definido?(A, q)]  borrar_por_atajo(A, q)
    ecuaciones
       1) [¬ig(k1, k2)]  añadir_con_atajo(añadir_con_atajo(A, k1, v1, q1), k2, v2, q2) =
                                  = añadir_con_atajo(añadir_con_atajo(A, k2, v2, q2), k1, v1, q1)
       2) añadir_con_atajo(añadir_con_atajo(A, k, v1, q), k, v2, q) = añadir_con_atajo(A, k, v2, q)
       3) [definida?(A, k)]  añadir(A, k, v) = <añadir_con_atajo(A, k, v, q), q>,
                siendo q = atajo_de_clave(A, k)
       4) [¬definida?(A, k)]  añadir(A, k, v) = <añadir_con_atajo(A, k, v, q, q>, siendo q = nuevo_atajo(A)
       ...
       5) [¬ig_atajo(p, q)]  borrar_por_atajo(añadir_con_atajo(A, k, v, p), q) =
                             = añadir_con_atajo(borrar_por_atajo(A, q), k, v, p)
       6) [definida?(A, k)]  borrar_por_atajo(añadir_con_atajo(A, k, v, q), q) = borrar_por_atajo(A, q)
       7) [¬definida?(A, k)]  borrar_por_atajo(añadir_con_atajo(A, k, v, q), q) = anotar_atajo(A, q)
       8) [¬ig_atajo(p, q)]  borrar_por_atajo(anotar_atajo(A, p), q) = anotar_atajo(borrar_por_atajo(A, q), p)
       9) [¬ig_atajo(p, q)]  anotar_atajo(añadir_con_atajo(A, k, v, p), q) =
                             = añadir_con_atajo(anotar_atajo(A, q), k, v, p)
       ...
```

10) [¬hay_huecos?(A)] ⇒ nuevo_atajo(A) = sig_atajo(último_atajo_generado(A))

11) [hay_huecos?(A)] ⇒ nuevo_atajo(A) = último_atajo_liberado(A)

...

funiverso

Fig. 3. Un TAD para los almacenes de elementos con atajos.

Además, debe añadirse como parte de dicho modelo una secuencia *s* que contenga los atajos liberados, $s \in A^*$; es indispensable que este componente forme parte del modelo para poder generar atajos. Decimos una secuencia y no un conjunto porque los atajos se asignarán por orden inverso de aparición (es decir, la secuencia será en realidad una pila) tal como establece la especificación[2]. Notemos que ha de cumplirse que ningún atajo aparezca a la vez en dom(h^{-1}) y en *s*. Igualmente, la unión de los atajos de dom(h^{-1}) y *s* no puede dejar "agujeros". Las operaciones del TAD que involucran el tipo *almacén* pueden definirse en base a este conjunto de soporte; por ejemplo, *modificar_por_atajo*(*A*, *q*, *v*) modifica la función *g* del modelo de *A* en el punto $h^{-1}(q)$, de manera que $g(h^{-1}(q)) = v$.

Por otro lado, el conjunto de soporte del tipo *atajo* es cualquier dominio biyectivo al conjunto denotado por el álgebra cociente de términos de dicho tipo, y que no colapse los tipos usados (por ejemplo, que no permita deducir *cierto* = *falso*). Dadas las ecuaciones que relacionan las operaciones generadoras de atajos (o, mejor dicho, dada la ausencia de tales ecuaciones) el álgebra cociente de términos de tipo *atajo* se caracteriza por tener como conjunto de soporte las clases [$sig_atajo^n(prim_atajo)$], $n \geq 0$, (el superíndice debe interpretarse como aplicación reiterada de la operación). Entre estos dominios, destacamos el de los números naturales, y entonces podríamos interpretar las operaciones de la manera siguiente: 0, la interpretación de *prim_atajo*; sumar uno, la interpretación de *sig_atajo*; y la igualdad de los naturales, la interpretación de *ig_atajo*. Dicho en otras palabras, podemos considerar los números naturales como una interpretación válida de los atajos.

5. Implementación del TAD *ALMACÉN* con atajos

Nos interesamos aquí en la formulación de una estrategia de implementación válida para los almacenes con atajos. En realidad, lo que realmente nos interesa es la representación del tipo, pues la codificación de las operaciones será una consecuencia directa de la misma; al final de la sección damos una idea sobre cómo construirla de manera formal.

La representación consta de tres partes. Primero, y aprovechando que los naturales son un modelo válido de los atajos, consideramos la existencia de un vector *AT* de 0 a *N* - 1 indexado por atajo. Si el número máximo de claves que contendrá el almacén es conocido *a priori*, la *N* ya está determinada; si no, podemos contemplar la memoria dinámica del sistema como un vector de *N* posiciones[3] y suponer pues que *AT* modeliza la memoria dinámica del programa: el esquema es entonces el mismo. En cuanto al contenido de las posiciones de *AT*, serán pares <clave, información>.

Por otro lado, queda claro que las posiciones libres de este vector (es decir, aquéllas que correspondan a atajos indefinidos) se gestionarán en forma de pila. La pila la podemos dividir en dos trozos: el trozo superior, que contendrá los atajos libres que ya han sido utilizados, en

---

[2] Podría pensarse en la conveniencia de dejar indeterminado cual es el atajo concreto a reaprovechar. No obstante, en el marco de la semántica inicial, nos vemos obligados a fijar la política de reaprovechamiento para evitar que aparezcan inconsistencias que colapsen alguno de los conjuntos de soporte involucrados (por ejemplo, podría llegar a deducirse la igualdad de dos informaciones diferentes). En la sección 7, como parte del trabajo futuro, proponemos estudiar la posibilidad de cambiar a otra semántica que nos dé una mayor libertad.

[3] Siempre podríamos establecer la existencia de una biyección entre los naturales de 0 a *N* - 1 y las *N* direcciones de memoria dinámica capaces de almacenar los datos del almacén.

orden inverso de liberación (tal como establece la especificación); y el trozo inferior, que contendrá los atajos todavía no generados, en orden. En realidad, esta gestión del sitio libre es la habitual de una estructura encadenada implementada en un vector [Fra94], donde puede verse que el coste de insertar y borrar en la pila es O(1). Aquí también supondremos que, en el caso de utilizar memoria dinámica, el sistema la gestiona de la forma adecuada a nuestros propósitos.

Por último, es necesario asegurar el acceso eficiente por clave, que no queda garantizado con lo que tenemos hasta ahora. Para ello, añadimos un almacén M sin atajos (tal como se ha presentado en la sección 2), siendo la información asociada a la clave el atajo que sirve de acceso al vector anterior. Así, dada una clave, obtenemos rápidamente el atajo y, si es necesario, utilizamos este atajo para acceder a la información.

En la fig. 4 se presenta un ejemplo de estructura para $N = 8$. En ella residen tres claves, con atajos 0, 3 y 4. Los atajos 1 y 2 se utilizaron en el pasado y están listos para ser reaprovechados, empezando por el 2 (que se liberó el último). Los atajos del 5 al 7 no se han generado nunca y no se generarán hasta que se hayan reaprovechado los anteriores. Estos cinco atajos forman la pila de atajos libres. M se ha pintado como un vector aunque, en realidad, la implementación puede ser cualquiera de las presentadas en la fig. 2.

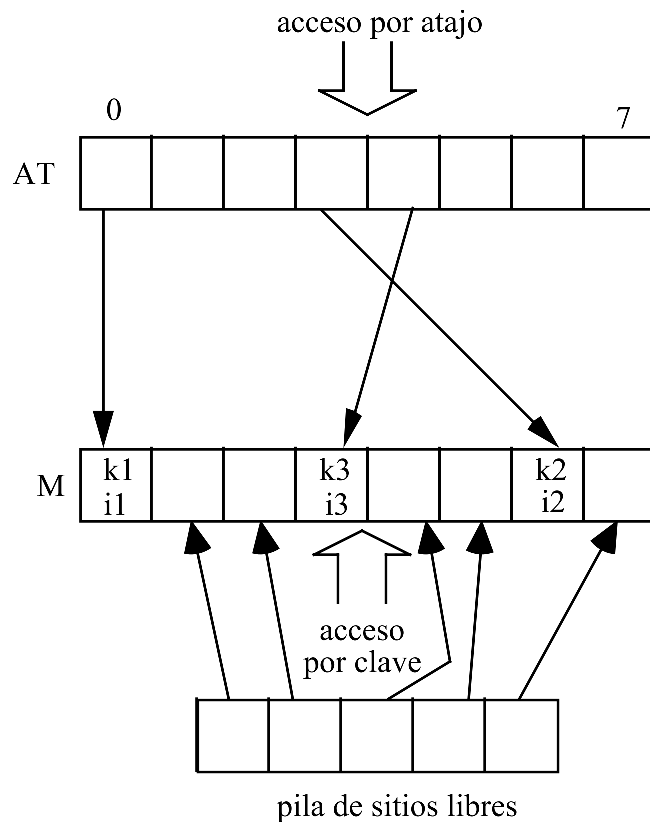

Fig. 4. Representación del almacén con atajos.

Estudiamos ya la eficiencia de las operaciones del TAD sobre esta representación. Las de acceso al almacén por atajo que no impliquen modificar la estructura de la tabla M son O(1), lo que era nuestro objetivo primordial. Por otra parte, las operaciones que acceden por clave o que, accediendo por atajo, modifican la tabla M, tienen una complejidad que depende de la implementación concreta de la tabla; en todo caso, la propiedad importante es que dicha complejidad no crece por el hecho de que existan atajos. En la fig. 5 se muestra la eficiencia para una implementación de la tabla mediante árboles AVL. En resumen, puede concluirse que hemos definido un método de acceso a los datos de complejidad O(1), tal y como nos

habíamos propuesto, sin penalizar el acceso por clave propio de la estrategia de implementación escogida.

T(crear)□O(1)
T(añadir, borrar, consultar, definida?)□O(log *n*)
T(borrar_por_atajo)□O(log *n*)
T(consultar_por_atajo, atajo_definido?, modificar_por_atajo)□O(1)
T(clave_de_atajo)□O(1), T(atajo_de_clave)□O(log *n*)

Fig. 5. Tiempo de ejecución de las operaciones del TAD ALMACÉN con atajos usando AVLs.

Notemos, por último, que la estructura es resistente a reubicaciones físicas de las claves de la tabla *M*[4]. Es decir, en caso que los algoritmos de gestión de la tabla (por ejemplo, la supresión en una tabla de dispersión gestionada mediante *open addressing*) cambien la posición física que ocupa una clave en *M*, el atajo correspondiente a su información seguirá siendo el mismo. De esta manera, nuestra estrategia de implementación consigue que los movimientos de datos sean transparentes al usuario del TAD, lo cual es un beneficio adicional que favorece la reusabilidad directa del tipo.

Quedaría por asegurar que la implementación es correcta respecto la especificación. Para ello, utilizamos la noción de corrección propuesta en [Hoa72], que se basa en la formulación de la función de abstracción de la representación del tipo, que nos asocia a cada valor de la implementación del TAD un término que representa su valor abstracto. Como la función es parcial, añadimos el invariante de la representación, que determinará su dominio. Por último, estableceremos la igualdad en términos de la representación. A partir de aquí, establecemos la especificación pre-post de cada operación y derivamos su código, que será correcto por construcción. Dada su extensión, no se incluye la verificación en el presente artículo; ver [Mar96] para consultar la derivación completa.

6. Ejemplo

Para ilustrar la utilidad de los atajos en el diseño de nuevas estructuras de datos, estudiamos un caso concreto que hemos extraído de [AHU83]: el *ranking* de tenistas. Precisamente, este texto usa el ejemplo para ilustrar la necesidad de acceder directamente a la representación de las estructuras cuando las restricciones de eficiencia así lo requieran; por ello, creemos especialmente indicada su resolución con nuestra propuesta.

El enunciado es el siguiente. Se pretende mantener un *ranking* tenístico mediante las operaciones:

- *alta*(*R*, *nombre*): añade el tenista *nombre* al *ranking R* en la última posición (i.e., en una posición una unidad mayor que la que anteriormente era la última). Las posiciones se numeran a partir del uno.
- *rival*(*R*, *nombre*): devuelve la posición en el *ranking R* del tenista que ocupa la posición inmediatamente anterior que *nombre*.
- *cambiar*(*R*, *k*): intercambia la posición en el *ranking R* de los tenistas que ocupan las posiciones k y k - 1, $k > 1$.

La fig. 6 muestra la implementación eficiente del TAD que propone [AHU83]. Se combinan un vector *pos* indexado por la posición en el *ranking* con una tabla de dispersión que utiliza

[4] Recordemos que el atajo es fijo y, por ello, no hay reubicaciones en *AT*.

los nombres de los tenistas como claves. Las estructuras están relacionadas, de manera que las posiciones de *pos* apuntan a los nodos de la tabla, y cada nodo de la tabla tiene asociada a la clave el índice a *pos*; las flechas dobles del esquema indican esta relación. Esta propuesta ilustra algunos de los problemas típicos de integración de un TAD clásico (el almacén tal y como se ha presentado en la fig. 1) en uno nuevo: es necesario conocer la implementación concreta del TAD para declarar el tipo de los apuntadores que residen en el vector; la función de inserción debe modificarse para devolver el apuntador al elemento insertado; y debemos asegurar que la dirección física de los datos no cambiará.

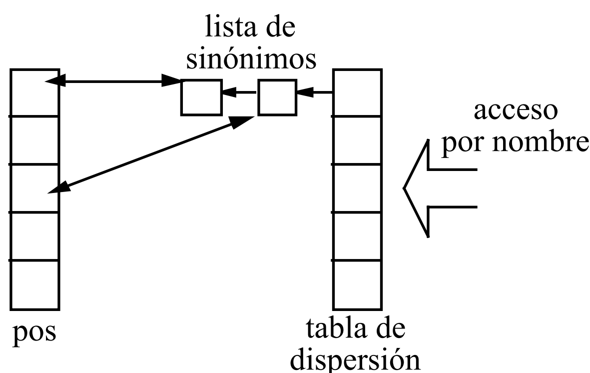

Fig. 6. Esquema de implementación del ranking de tenistas.

Si nos decidiéramos por primar la modularidad de la estructura, para solucionar los problemas anteriores, mediante la reutilización del TAD correspondiente al almacén (sin atajos) implementado por dispersión, no podríamos acceder a las posiciones de la tabla con el apuntador (que queda escondido), sino tan sólo con la clave. Además, aunque la eficiencia temporal es teóricamente óptima, siempre existe el riesgo de que la función de dispersión elegida no distribuya correctamente las claves en la tabla y que la tabla degenere. Por último, notemos que si hubiéramos elegido una implementación mediante AVL (para sacar listados ordenados, por ejemplo), la eficiencia de *rival* y *cambiar* sería de complejidad logarítmica; si bien en el primer caso esto es inevitable (pues se accede al árbol por clave), en el segundo parece claramente mejorable.

Nuestra propuesta se concreta en reutilizar el TAD *ALMACÉN* con atajos tal como se ha presentado en las secciones anteriores. De esta manera, *pos* vuelve a almacenar apuntadores (en realidad, atajos) a los datos en vez de claves, con el consiguiente ahorro de espacio y evitando pues los posibles problemas con la función de dispersión una vez los datos entran en la tabla. La eficiencia de las operaciones queda igual que en la propuesta de [AHU83] pero, en cambio, solucionamos todos los problemas mencionados: los atajos no dependen del tipo concreto de los apuntadores; la función de inserción del TAD con atajos ya devuelve el camino de acceso al dato insertado y, por ello, no es necesaria adaptarla a la nueva situación; y no importa que la dirección física de los datos cambie, pues el atajo se mantiene. Además, en el caso que utilizaramos una implementación por AVL (o, en general, cualquiera que no tenga tiempo O(1) de acceso a los datos por clave), *cambiar* quedaría O(1), pues podría accederse al almacén mediante el atajo.

7. Conclusiones y trabajo futuro

Hemos presentado una propuesta que trata de reconciliar dos criterios normalmente opuestos en la programación con tipos abstractos de datos: eficiencia y modularidad. Esta propuesta se basa en la definición un tipo *atajo* que modeliza el concepto de apuntador a los datos. Hemos desarrollado nuestro trabajo estudiando un TAD particular, el almacén de datos, formulando una especificación algebraica para el mismo, identificando su modelo matemático, y proponiendo una implementación modular que respeta los invariantes de la representación de

los módulos que la componen. Como resultado, y ésta es la aportación de nuestro trabajo, los TAD pueden combinarse de manera eficiente sin ninguna modificación (permitiendo reusabilidad plena), conservando la eficiencia de la implementación por lo que respecta a las operaciones sin atajos, permitiendo acceso directo (i.e., O(1)) a los datos siguiendo el atajo, y conservando en todo momento un significado matemático bien definido. Queremos destacar que la mayor parte de todo lo que hemos dicho aquí puede aplicarse a cualquier otro TAD que almacene una colección de elementos: listas, grafos, árboles, etc.

En cuanto al trabajo futuro, hay tres líneas principales de investigación. Por un lado, queremos expresar la propuesta de una manera genérica, manteniendo el nivel de formalismo aquí expuesto. Por "genérica" queremos decir: aplicable a cualquier TAD implementado de cualquier manera. Para este objetivo, hemos pensado definir una jerarquía de clases donde el añadido de atajos a los TAD se plasme en la herencia de una clase adecuada.

Por otro lado, queremos estudiar si hay otros marcos formales más adecuados que la semántica inicial (principalmente, estamos pensando en las semánticas de comportamiento y laxa). Como ya se ha comentado, la adopción de dicha semántica es la que nos ha obligado a determinar con exactitud cuales son los atajos que se van asignando a los elementos que entran en una estructura, lo que ha generado una especificación extensa y, en cierto modo, ya orientada a la implementación posterior. En cambio, con otro tipo de semántica, la operación *nuevo_atajo* se podría especificar con una única ecuación:

atajo_definido?(A, nuevo_atajo(A)) = falso

que simplemente constata que el atajo que se asigna a una nueva clave no debe estar ya asignado en el almacén actual, dejando la puerta abierta a la política concreta de asignación de atajos.

Por último, está claro que el modelo de memoria dinámica que se ha presentado aquí no es el más adecuado a la realidad y por ello nos proponemos mejorarlo. En concreto, queremos derogar dos suposiciones que hemos asumido: que el espacio de memoria dinámica se puede asimilar a un vector (con una dimensión fija), y que la gestión del espacio liberado se realiza ineludiblemente con política LIFO. El segundo punto es tal vez el menos importante, pues siempre podríamos gestionar las posiciones liberadas nosotros mismos, usando un *pool* de celdas libres, con la política que nos parezca conveniente. La primera cuestión, en cambio, es más peliaguda: el uso de memoria dinámica no permite normalmente determinar el número máximo de posiciones a utilizar y, además, la memoria dinámica no puede dedicarse a un único TAD sino que es compartida por todos los objetos del programa (y, eventualmente, de otros programas).

Referencias

[ADJ78] J. Goguen, J. Thatcher, E. Wagner. *An Initial Algebra Approach to the Specification, Correctness and Implementation of Abstract Data Types*. En *Current Trends in Programming Methodology*, R. Yeh (ed.), Prentice-Hall, 1978.

[AHU83] A. Aho, J. Hopcroft, J. Ullman. *Data Structures and Algorithms*. Addison-Wesley, 1983.

[Bra85] G. Brassard. "Crusade for a better Notation". SIGACT News, 16(4), 1985.

[EM85] H. Ehrig, B. Mahr. *Fundamentals of Algebraic Specification* (1). Springer-Verlag, 1985.

[FBB93] X. Franch, X. Burgués, P. Botella. "Report del Llenguatge de Programació Merlí" (escrito en catalán). Report LSI-93-T, UPC, 1993.

[Fra94] X. Franch. *Estructuras de Datos: Especificación, Diseño e Implementación*. Edicions UPC, colección Politext nº 30, 1994.

[Hoa72] C.A.R. Hoare. "Proof of correctness of Data Representation". Acta Informatica 1(1), 1972.

[HS94] E. Horowitz, S. Sahni. *Fundamentals of Data Structures in Pascal*. Computer Science Press, 4ª edición, 1994.

[Knu76] D. Knuth. "Big Omicron and Big Omega and Big Theta". SIGACT News, 8(2), 1976.

[LG86] B. Liskov, J. Guttag. *Abstraction and Specification in Program Development*. The MIT Press, 1986.

[Mar96] J. Marco. *Dreceres: "Pointers" Abstractes* (escrito en catalán). Proyecto Fin de Carrera de los Estudios de Ingeniería Informática, Facultat d'Informàtica de Barcelona (U.P.C.), dirigido por J.L. Balcázar, 1996. Resumido en el report de invstigación LSI-97-25-R.