Adding Alternative Access Paths to Abstract Data Types[1]

Xavier Franch, Jordi Marco

Universitat Politècnica de Catalunya
Department Llenguatges i Sistemes Informàtics
Jordi Girona, 1-3 (Campus Nord). E-08034 Barcelona (Catalonia, Spain)
e-mail: {franch | jmarco}@lsi.upc.es

Abstract. We present in this paper a proposal for developing efficient programs in the abstract data type (ADT) programming framework, keeping the modular structure of programs and without violating the information hiding principle. The proposal focuses in the concept of "shortcut" as an efficient way of accessing to data, alternative to the access by means of the primitive operations of the ADT. We develop our approach in a particular ADT, a store of items. We define shortcuts in a formal manner, using algebraic specifications interpreted with initial semantics, and so the result has a well-defined meaning and fits in the ADT framework. Efficiency is assured with an adequate representation of the type, which provides O(1) access to items in the store without penalising the primitive operations of the ADT.

1 Introduction

Modular programming with abstract data types (ADT) [LG86] is a widespread methodology for programming in the large. In this field, it is crucial the distinction between the specification and the implementation of ADTs, which results in the existence of different modules for them and which can be summarised with the information hiding principle: an ADT must be used just regarding the properties stated in the specification, without any knowledge of the characteristics of its implementation, which remains hidden. This principle simplifies the relationships between modules and supports the development of programs, because it is easier to code them, to test them, to reuse them and to maintain them.

However, the information hiding principle collides, often dramatically, with a very usual requirement on programs: their efficiency, mainly characterised by their execution time. The reason is that the access to a data structure implementing an ADT must follow the properties that define it, which were stated in an abstract manner without taking into account the problems related to its subsequent implementation (as it must be). In case of a context using ADTs with strong efficiency requirements (for instance, program analysis tools construction, system programming, geometric computing and combinatorial computing), their full reusability can become impossible and it may be necessary to carry out many modifications to fit it to this context; even more, such modifications can be so important to decide throwing away the implementation and developing a new one.

This conflict between efficiency and modularity is a well known problem in the ADT framework, recognised as such in the most important textbooks on data types and data structures [AHU83, HS94, CLR90, etc.], and solved in many cases sacrificing modularity to achieve efficiency. Fortunately, there are many widespread ADT-libraries that have coped the problem by incorporating the notion of *location* (i.e., a cursor —an integer referring to an array position— or a pointer) in ADT interfaces. This is the case for instance of STL [MS96] and LEDA [MN99], both of them providing a similar solution to the problem: when a new element is stored in the data structure, its location is returned as part of the result, being later usable as parameter in other operations (removal, lookup and modification). Unfortunately, these libraries present some drawbacks due to the fact that they are designed with the concept of location incorporated in the component from the very beginning. Therefore, the implementations that can be used for the ADT are restricted to a fixed set (which makes these libraries not flexible enough), the behaviour is less clear (locations and elements appear at the same level) and some classical low-level problems appear (for instance, meaningless uses of cursors and pointers).

Our goal in this paper is to define a general framework to reconcile both criteria, efficiency and modularity, obtaining thus efficient programs reusing existing implementations of ADTs without any modification, and following the information hiding principle. The proposal is based on the definition of an alternative way to access data, that we call *shortcuts*. Shortcuts are added to existing ADTs in a systematic manner, obtaining new ADTs (compatible with the previous ones) with this alternative access paths incorporated. Then, the users of the new ADT will be able to access the data therein not only by means of the operations introduced in the original specification (that are the ones defining the underlying mathematical model), but also using other new ones which

[1] This work is partially supported by the spanish research programme CICYT under contract TIC97-1158.

follow these alternative paths, when the use of the former operations is considered unacceptably expensive. We are going to develop the proposal on a particular ADT, a *STORE* of items, although the conclusions of our work can be applied to any other container-like ADT, i.e. those ones arranging collections of items with an arbitrary (but completely defined) policy.

The rest of the paper is organised as follows. In section 2 we introduce the notion of shortcut and we make clear the differences with respect to the classical pointer and cursor facilities. The ADT is presented in sections 3 and 4, without and with shortcuts, respectively. Section 5 proposes the model of stores with shortcuts, while section 6 shows the implementation. Finally, section 7 gives the conclusions and some future work. Also, we include two appendixes with the complete specification (in OBJ-3) and implementation (in Ada-95) of the ADT.

2 The Notion of Shortcut

We define a shortcut as an access path to elements stored in a container with the following properties:

- **Abstraction**. The elements in the container can be accessed without knowing how are they stored in the container and, therefore, without knowing the underlying representation of the container (with arrays, pointers, linked, in tree-form, ...). This property allows integrating containers in programs in a modular way.
- **Efficiency**. The access to the elements in a container by means of a shortcut is achieved in constant time O(1), making then possible to use them even with high efficiency constraints.
- **Evolution**. Shortcuts are created and destroyed as the elements are inserted in and deleted from in the container, respectively.
- **Persistency**. The shortcut bound to an element remains the same while it is inside the container, even if the underlying representation requires rearrangements. This property is critical for the approach being general enough, and it is a point that distinguishes our proposal from others.
- **Preservation**. The addition of shortcuts to a container does not modify its functional behaviour. This is assured by incorporating the concept of shortcut into the formal specification of the container (see section 4 for more details). Preservation of behaviour makes possible the substitution of old ADTs by new ones.
- **Security**. The access to the elements by means of shortcuts is safe because meaningless access to them is avoided. In particular, our approach avoids dangling shortcuts or accessing a data structure using a shortcut created to access another data structure.
- **Uniqueness**. An element has one and only one shortcut bound, and a shortcut is bound to one and only one element.

One could think that shortcuts behave like pointers (or cursors), but this is not true. It should be clear that pointers lack many of the fundamental properties listed above: they are neither abstract, nor persistent nor secure and they do not preserve behaviour. First, they are not abstract enough because accessing the container by means of pointers implies knowing some details about the ADT implementation; even more, a pointer declaration may become obsolete due to implementation changes, forcing then programs to be modified and damaging thus software maintenance. Second, a pointer to a data element may become obsolete if the physical location of an element changes due to the implementation strategy (for instance, many hashing implementations or various self-organising data structures). Next, it is not possible to take control on dangling references, because pointers can be used to access data that does not currently exist, causing an error in execution time; this is due to the very nature of dynamic memory, which is not bound to any particular data structure, acting as a kind of global variable to the whole program. Last, since pointers are a kind of backdoor to ADTs, they can be used to change their functional behaviour interfering with the existing specification and making then the component unreliable.

3 The Abstract Data Type *STORE*

From now on, we focus on the study of a particular ADT[2], the ADT *STORE*, defined as a collection of items, with operations of insertion, removal and retrieval of items. Just to fix a particular definition of stores, we use a short version of the one defined by Booch in [Boo87], although this selection is arbitrary. Items are pairs <key, value>, and so the removal and the retrieval are key-based; keys must provide a comparison operation, *eq*. As stated in [Boo87], it is an error trying to remove or to retrieve items using undefined keys, and also trying to

[2] However, it should remain clear that the results are valid any other container-like ADT.

insert a pair with a key that is already therein.

We use in the paper an algebraic specification language with conditional equations, interpreted with initial semantics [EM85], close to OBJ-3 [GW88] but with many simplifications to make it more readable[3]. To simplify matters, we manage errors as in [ADJ78], grouping all the error expressions in a separated area and assuming implicit error propagation. For simplicity purposes too, we define the ADT as a non-parameterised one, obtaining thus classes of (total and heterogeneous) algebras as models, instead of functors. The specification of the type is straightforward (see fig. 1).

```
specification STORE imports KEY+VALUE+BOOL
  sort store
  operations create: -> store
             insert: store key value -> store
             remove: store key -> store
             retrieve: store key -> value
             defined?: store key -> bool
  errors insert(insert(A, k, v), k, v'); remove(create, k); retrieve(create, k)
  equations
     [eq(k, k') ≡ false] => insert(insert(A, k, v), k', v') ≡ insert(insert(A, k', v'), k, v)
     remove(insert(A, k, v), k) ≡ A
     [eq(k, k') ≡ false] => remove(insert(A, k, v), k') ≡ insert(remove(A, k'), k, v)
     retrieve(insert(A, k, v), k) ≡ v
     [eq(k, k') ≡ false] => retrieve(insert(A, k, v), k') ≡ retrieve(A, k')
     defined?(create, k) ≡ false
     defined?(insert(A, k, v), k') ≡ eq(k, k') ∨ defined?(A, k')
end STORE
```

Fig. 1. An ADT for a store of items

We fix the model of the ADT *STORE* interpreting the equations with initial semantics. Given the properties stated on *insert* (see first equation) we can say that the model (with respect to the carrier set of the *store* sort) is the set of partial functions $K \rightarrow V$, being K and V the carrier sets of the sorts *key* and *value*, respectively. The operations of the model are the intuitive interpretation of the ADT operations over these functions; for instance, *create* is interpreted as the function g satisfying $\mathrm{dom}(g) = \emptyset$.

Implementations for the ADT will make use of hashing, AVL trees and so *on*. Every implementation has a different behaviour with respect to execution time, and it can be the case of implementations with a non-constant access time to items, even linear time (e.g., unordered arrays)[4].

4 The Abstract Data Type *STORE* with Shortcuts

The goal of this section is to extend the ADT *STORE* by adding shortcuts to access directly the items contained in stores. As a design requirement, we want a specification not only correct but also useful. This means mainly two things. First, the new ADT must be compatible with the former one, in the sense that the old operations must be preserved with the same signature and with the same behaviour as before, when shortcuts are not taken into account. On the other hand, the specification must allow feasible implementations; the main consequence of this is recycling of free shortcuts, although this is not necessary from the specification point of view.

We begin by introducing a new sort *shortcut*. The values of this sort are generated using two operations *first_sc:* $\rightarrow$ *shortcut* and *next_sc: shortcut* $\rightarrow$ *shortcut*, which are declared as private to avoid out-of-control creation of shortcuts by *STORE* users; shortcut creation is restricted to *STORE*. We provide also with a (public) operation of shortcut comparison, *eq_sc*.

Shortcut creation takes place when adding new pairs to the store. So, we add a new operation *last_sc: store* $\rightarrow$ *shortcut*, which return the *shortcut* to be used to access the last pair <key, value> inserted into the *store*. Typically, this operation should be called once a new pair enters the store. The obtained shortcut can be stored in

---

[3] Real OBJ-3 is used in the appendix 1 to provide a complete specification for the *STORE* with shortcuts.

[4] In this paper, we focus on implementations in main memory (and so we measure efficiency with the asymptotic big-Oh notation [Knu76, Bra85]).

other data structures, and then coupling of ADTs (for building new data structures) can be carried out both in an efficient and modular way. In addition to this, the ADT provides a new operation, *sc_for_key*, to obtain the shortcut bound to a pair at any moment.

Last, we add operations to access the store by means of shortcuts: removal (*remove_sc*), retrieval (*retrieve_sc*) and modification (*modify_sc*). Furthermore, we introduce an operation to find out if a given shortcut is defined (*defined_sc?*), because we consider an error to access the structure using an undefined shortcut; in fact, as we have mentioned earlier, this kind of control is one of advantages with respect the usual notion of pointer.

sorts store, shortcut
operations create, insert, remove, retrieve and defined? as in fig. 1
last_sc: store → shortcut
remove_sc: store shortcut → store
retrieve_sc: store shortcut → value
defined_sc?: store shortcut → bool
modify_sc: store shortcut value → store
key_for_sc: store shortcut → key
sc_for_key: store key → shortcut
eq_sc: shortcut shortcut → bool

Fig. 2. Public signature of an ADT for stores with shortcuts

We address now to the specification of the type, focusing just on its most interesting parts (see the appendix 1 for a full version in OBJ-3). To simplify the final product, we introduce a new private operation to add pairs <key, value> with its shortcut, *insert_sc: store key value shortcut → store*. We need two error expressions (see fig. 3, expressions 1 and 2), to avoid key or shortcut repetition (the first error coming from Booch's definition, the second one coming from our approach). Note the absence of the commutative equation over *insert* that appeared in the store without shortcuts (section 3). This is due to the fact that we need now to maintain the ordering of insertions to distinguish different shortcuts; in other case the following property would hold:

last_sc(insert_sc(insert_sc(A, k, v, q), k', v', q') ≡ last_sc(insert_sc(insert_sc(A, k', v', q'), k, v, q)

which is obviously wrong.

In order to obtain shortcuts for the store, we introduce another private operation *new_sc: store → shortcut* which is the one responsible to associate a shortcut to a new pair entering in the store. Then, we can bound the public *insert* operation with the private *insert_sc* one:

(E1) insert(A, k, v) ≡ insert_sc(A', k, v, new_sc(A))

where *A'* will be defined later.

A point is worth to be mentioned: as far as *insert_sc* is not commutative, *insert* is not commutative also. This is really a difference in the underlying model, but in fact it does not impact on the practical use of the type. Changing to another type of semantics, as we mention in the future work (section 7) would solve this problem.

The simplest policy to generate new shortcuts would consist in obtaining the shortcut successor of the last generated one. However, this criteria works not well when removals are taken into account, because removals set free previously generated shortcuts. Although from the specification point of view we could reject the possibility of reusing these shortcuts, we decide not to do that, because feasible implementations of the ADT will need to reassign released shortcuts in further insertions (to avoid holes in the underlying data structure). The specification of *new_sc* results in (see fig. 3, equations from 3 to 5): if there are no shortcuts to reassign, the new shortcut is the next of the last generated one; if there is at least one shortcut to reassign, it is the last released one.

We should mention that reassignment conveys a danger: it is impossible to be sure that all the users of a store having copies of the reassigned shortcut are aware that the pair bound to the shortcut has changed; it could be the case of accessing the store by means of a copy of the shortcut created before its last assignment. In fact, it would be not difficult to take care of this, adding new operations on the ADT to create and destroy copies of shortcuts in a controlled way; however, we have decided not to do that because the same problem arises when considering keys instead of shortcuts, and usually this situation is not explicitly handled in usual container-like ADTs.

To specify the auxiliary operations appearing in these two equations, we need to keep track of released shortcuts, with the help of a new operation *mark_sc: store shortcut → store*. Marks appear when items are removed (either by key or by shortcut) and disappear only when the shortcut is assigned again, by means of an *unmark_sc: store*

shortcut → *store* operation. These operations are specified in fig. 3, equations from 6 to 15.

1) **error** [defined?(A, k) ≡ true] ⇒ insert_sc(A, k, v, q)
2) **error** [defined_sc?(A, q) ≡ true] ⇒ insert_sc(A, k, v, q)

3) new_sc(create) ≡ first_sc
4) [holes?(insert_sc(A, k, v, q)) ≡ false] ⇒
new_sc(insert_sc(A, k, v, q)) ≡ next_sc(last_sc_generated(insert_sc(A, k, v, q)))
5) [holes?(insert_sc(A, k, v, q)) ≡ true] ⇒
new_sc(insert_sc(A, k, v, q)) ≡ last_sc_released(insert_sc(A, k, v, q))

6) remove_sc(insert_sc(A, k, v, q), q) ≡ mark_sc(A, q)
7) remove(insert_sc(A, k, v, q), k) ≡ mark_sc(A, q)
8) [eq(k, k') ≡ false] ⇒ remove(insert_sc(A, k, v, q), k') ≡ insert_sc(remove(A, k'), k, v, q)
9) [eq_sc(q, q') ≡ false] ⇒ remove_sc(insert_sc(A, k, v, q), q') ≡ insert_sc(remove_sc(A, q'), k, v, q)
10) [eq_sc(q', q) ≡ false] => insert_sc(mark_sc(A, q'), k, v, q) ≡ mark_sc(insert_sc(A, k, v, q), q')
11) **error** insert_sc(mark_sc(A, q), k, v, q)

12) unmark_sc(create, q) ≡ create
13) unmark_sc(insert_sc(A, k, v, q'), q) ≡ insert_sc(unmark(A, q), k, v, q')
14) unmark_sc(mark_sc(A, q), q) ≡ unmark_sc(A, q)
15) [eq_sc(q', q) ≡ false] => unmark_sc(mark_sc(A, q'), q) ≡ mark_sc(unmark_sc(A, q), q')

Fig. 3. An excerpt of the specification of an ADT for stores with shortcuts

The *unmark* operation plays an important role also when considering insertions. In equation (E1), the store *A'* should eliminate any remaining mark of the new shortcut, just in case it were a released shortcut. This is necessary to maintain the consistence of the store with respect to the state of shortcuts. So, equation (E1) takes as final form:

(E1) insert(A, k, v) ≡ insert_sc(unmark_sc(A, new_sc(A)), k, v, new_sc(A))

The rest of the specification is straightforward.

5 Semantics of the Abstract Data Type *STORE* with Shortcuts

As we already expected, the initial model of the ADT with shortcuts is different from the one without them. We are going to fix this model concerning the algebras bound to the sorts of interest, *store* and *shortcut*. As far as we are working in the initial semantics framework, we are going to identify which are the terms representative of the classes in the quotient-term algebra (of the appropriate sorts), then we will formulate the model and we will establish the correspondence between the representative terms and the values of the model.

Model of *STORE* for the sort *store*

Given the equations of the type, the equivalence classes of the quotient term algebra will include combinations of the operations *insert_sc* and *mark_sc* over the empty store. Arbitrarily, we choose as representative of a class any of the terms with the marks appearing after insertions:

t_{repr} ::= mark_sc(...(mark_sc(
insert_sc(...(insert_sc(create, k_1, v_1, q_1), ..., k_n, v_n, q_n),
q_{n+1}), ..., q_r)

such that: ∀i, j: 1 <= i, j <= n: i ≠ j ⇒ ¬eq(k_i, k_j) ∧
∀i, j: 1 <= i, j <= r: i ≠ j ⇒ ¬eq_sc(q_i, q_j)

The parameters k_i and v_i, 1 <= *i* <= *n*, are the pairs <key, value> in the store, while q_{n+1}, ..., q_r are all the released (and not reassigned) shortcuts.

It is clear that the model must include information about the correspondence between keys and values, and keys and shortcuts, and also the knowledge about which are the released shortcuts. So, we formulate as the carrier set corresponding to the sort *store*:

$STORE_{store}$ ::= $(K \rightarrow V) \times (K \leftrightarrow A) \times A^*$ satisfying that
∀(g, h, s)∈ $STORE_{store}$:

$$dom(g) = dom(h) \wedge$$
$$s \cap ran(h) = \emptyset \wedge s \cup ran(h) = [first\_sc, next\_sc^n(first\_sc)]$$

being $n = \| s \cup ran(h) \| - 1$, being $next\_sc^n(first\_sc)$ the application n times of *next_sc* on *first_sc*, and being A, K and V the carrier sets of shortcuts, keys and values, respectively. We mix sequences and sets when using $\cup$ and $\cap$ with an intuitive meaning. The first function maps keys to values, the second one binds keys and shortcuts with a bijection (as required by the uniqueness property), while the sequence keeps track of released shortcuts. As the specification obliges shortcuts to be reassigned in reverse order of release, the sequence must be seen as a stack[5].

The correspondence between the carrier set of $STORE_{store}$ and the representative term t_{repr} of the classes in the quotient-term algebra of *STORE* is established as:

$$t_{repr} \leftrightarrow (g, h, s) \text{ such that: } s = q_r \,.\, q_{r-1} \,.\, \ldots \,.\, q_{n+1} \,.\, \lambda \wedge$$
$$dom(g) = dom(h) = \{k_1, \ldots, k_n\} \wedge$$
$$\forall i: 1 <= i <= n: (g(k_i) = v_i \wedge h(k_i) = q_i)$$

The operations of the ADT can be defined in terms of this carrier set; for instance, the interpretation of *modify_sc(A, q, v)*, being (g, h, s) the model of A, requires $q \in ran(h)$ and redefines the function g in the point $h^{-1}(q)$ such that $g(h^{-1}(q)) = v$.

Model of *STORE* for the sort *shortcut*

On the other hand, the carrier set for the sort *shortcut* is any domain isomorphic to the quotient-term algebra for this sort. Given the absence of equations establishing relationships between the constructor operations for *shortcut*, the quotient-term algebra for the sort *shortcut* is characterised by having as carrier set the equivalence classes $[next\_sc^n(first\_sc)]$, $n >= 0$. Among them, we remark the domain of natural numbers; in this case, the operations can be interpreted in the following way: 0, the interpretation of *first_sc*; +1, the interpretation of *next_sc*; and =, the interpretation of *eq_sc*. So, we can consider natural numbers as a valid model of shortcuts.

6 Implementing the Abstract Data Type *STORE* with Shortcuts

We focus in this section in the efficient implementation of stores with shortcuts. In fact, we are interested in determining the representation of the sorts, because the code for the operations can be derived automatically from it (we have done it [MF97]).

The essential point consists on adding a mapping from shortcuts to pairs <key, value> in the new ADT, while reusing the old ADT substituting values by the shortcut that identifies them. This is precisely a point worth mentioning that makes our approach different from other existing ones, as LEDA and STL: shortcuts can be added to any given implementation of the ADT, without any kind of restriction. In appendix 2 we can see that the Ada-95 implementation takes profit of the generics mechanism to implement this idea.

Then, the representation of the store has three parts (see fig. 4). First, we consider the existence of an array *SC* of N positions to implement the mapping between shortcuts and pairs <key, value>. So, we implement shortcuts with natural numbers that indexing the array. The cells of *SC* will contain for the moment the pairs <key, value>.

On the other hand, the free positions of the array (those ones representing undefined shortcuts) will be managed as a stack. This stack has two parts: the upper one, containing the free shortcuts used before, in reverse order of release; and the lower one, containing shortcuts not used before, in increasing order of natural numbers. In fact, this kind of free space management is the usual one in chained data structures implemented with arrays [AHU83, HS94], with an O(1) complexity.

Last, we reuse the given implementation of stores (hashing tables, AVL trees, etc.), passing the shortcuts as values bound to the keys. As a result, given a key, we obtain the shortcut with the efficiency of the former implementation and, if necessary, we can use it to recover the corresponding value in constant time. Therefore, the cost of all the previous operations is maintained. It is worth noting also that the data structure is robust with respect to movements of the keys in M (for instance, when deleting in an open addressing hashing table).

---

[5] We could think of not fixing which is the shortcut to reassign. However, in the initial semantics framework, we are obliged to determine the concrete reassignment policy to avoid inconsistences collapsing some of the carrier sets involved in the model (for instance, we could demonstrate the equality of two different values). In section 7, as future work, we mention the possibility of moving to other semantics providing a higher degree of flexibility.

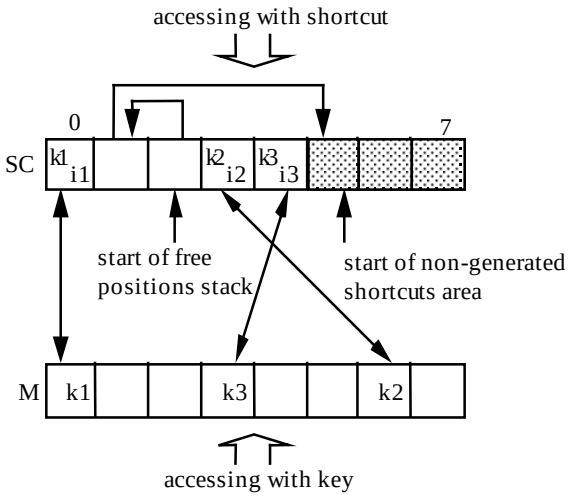

Fig. 4. A valid state of the implementation for stores with shortcuts

The operations accessing and not modifying the store by means of the shortcut are O(1), which was one of our goals. On the other hand, the operations accessing by key, or accessing by shortcut but modifying the store, have a complexity that depends on the underlying implementation of *M*; the important fact is that this complexity does not get worst with the addition of shortcuts. On the other hand, the representation needs *N**(space(shortcut)+space(key)) extra space. However, even this waste will generate a later saving of space, when shortcuts substitute keys (generally strings, which require most space than a shortcut).

Instead of the array *SC*, dynamic memory could have been used to store the pairs <key, value>. In this case, released shortcuts (in this case, pointers) would be managed directly by the memory allocator. The main consequence is that we can not assure that they are recycled with the chosen policy stated in the specification. A way to handle this problem would be to incorporate in the specification the memory allocator policy itself. In any case, this difference has not practical consequences.

7 Conclusions and Future Work

We have presented a proposal aimed at reconciling two usually contradictory criteria in the ADTs framework: modularity and efficiency. To do this, we add a new type to implement the concept of shortcut as alternative path access to elements in the ADT, and we add many new operations to make proper use of shortcuts. Shortcuts are interesting because, besides of assuring O(1) access time to elements in the ADT, they present some nice properties: they are abstract (independent of the implementation of the ADT), persistent (movements inside the data structure do not affect them), secure (meaningless accesses are not possible) and they preserve behaviour (the new ADT behaves as the old one, and the efficiency of the former operations keep the same). These properties are the ones that distinguish clearly shortcuts from low-level concepts as pointers or cursors.

We have developed our work studying a concrete ADT, the store of items, writing down an algebraic specification for the type, identifying its mathematical model (which behaves in a predictable manner), and proposing an adequate (efficient) implementation for it. We would like to remark that most of our work can be applied to every other container-like ADT.

Concerning future work, there are two main lines of research. On the one hand, we are working on expressing our proposal in a generic manner (that is, suitable for a wide variety of ADTs with arbitrary implementations), with the same level of formalism as the one outlined here. To do this, we are defining a parameterised ADT which retains the most fundamental common properties of a wide variety of containers (in fact, the container itself acts as parameter), so that we can reformulate the methodology on it.

On the other hand, we want to study if other formal frameworks are more adequate than initial semantics. As it has been already pointed out, initial semantics forces us to determine in a precise manner which shortcuts are the ones assigned to new elements, and this is the reason why we have obtained a large specification which also suffers from implementation bias. For instance, the operation *new_sc* could be specified instead with the single equation $defined\_sc?(A, new\_sc(A)) \equiv false$, which states that the shortcut assigned to a new pair must not be already assigned in the current store, but without fixing the assignment policy.

References

[ADJ78] J. Goguen, J. Thatcher, E. Wagner. *An Initial Algebra Approach to the Specification, Correctness and Implementation of Abstract Data Types*. *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.

[Boo87] G. Booch. *Software Components with Ada* (second edition). The Benjamin/Cummings Publishing Company Inc., 1987.

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

[CLR90] T.H. Cormen, C.E. Leiserson, R.L. Rivest. *Introduction to Algorithms*. The MIT Press, 1990.

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

[GW88] J.A. Goguen, T. Winkler. *Introducing OBJ-3*. Tecnical Report SRI-CFL-88-9, August 88.

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

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

[Knu98] D.E. Knuth. *Sorting and Searching* (second edition). Addison-Wesley, 1998.

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

[MF97] J. Marco, X. Franch. *Shortcuts: Abstract Pointers*. Technical Report LSI-R-97-25, Universitat Politècnica de Catalunya, 1997.

[MN99] K. Mehlhorn and S. Näher. *The LEDA Platform of Combinatorial and Geometric Computing*. Cambridge University Press, 1999.

[MS96] D.R. Musser, A. Saini. *STL Tutorial and Reference Guide*. Addison-Wesley, 1996.

Appendix 1. OBJ3 specification

In this appendix we present the OBJ3 specification of the store with shortcuts.

```
obj STORE[key :: TRIV, value :: TRIV] is
   sorts store store? value? shortcut shortcut? key? .

   subsorts store < store? .
   subsorts Elt.value < value? .

   subsorts shortcut < shortcut? .
   subsorts Elt.key < key? .

 *** original operations
   op create : -> store .
   op insert : store Elt.key Elt.value -> store .
   op insert : store Elt.key Elt.value -> store? .
   op remove : store Elt.key -> store .
   op remove : store Elt.key -> store? .
   op defined? : store Elt.key -> Bool .
   op retrieve : store Elt.key -> Elt.value .
   op retrieve : store Elt.key -> value? .
 *** new public operations
   op lastsc : store -> shortcut .
   op removesc : store shortcut -> store .
   op removesc : store shortcut -> store? .
   op retrievesc : store shortcut -> Elt.value .
   op retrievesc : store shortcut -> value? .
   op definedsc? : store shortcut -> Bool .
   op modifysc : store shortcut Elt.value -> store .
   op modifysc : store shortcut Elt.value -> store? .
   op keyforsc : store shortcut -> Elt.key .
   op keyforsc : store shortcut -> key? .
   op scforkey : store Elt.key -> shortcut .
   op scforkey : store Elt.key -> shortcut? .
 *** begin private operations
   op insertsc : store Elt.key Elt.value shortcut -> store .
   op insertsc : store Elt.key Elt.value shortcut -> store? .
   op marksc : store shortcut -> store .
   op unmark : store shortcut -> store .
   op firstsc : -> shortcut .
   op nextsc : shortcut -> shortcut .
   op newsc : store -> shortcut .
   op holes? : store -> Bool .
   op lastscgenerated : store -> shortcut .
   op lastscreleased : store -> shortcut .
   op _>_ : shortcut shortcut -> Bool .
 *** end private operations
   op UndefinedShortcut : -> key? .
   op UndefinedShortcut : -> value? .
   op UndefinedShortcut : -> store? .
   op keyIsNotInserted : -> shortcut? .
   op MultipleInserting : -> store? .
   op keyIsNotInserted : -> store? .
   op keyIsNotInserted : -> value? .

   var k k1 : Elt.key . var v v1 : Elt.value . var A A1 : store .
   var q q1 : shortcut .

   eq insert(A,k,v) = insertsc(unmark(A,newsc(A)),k,v,newsc(A)) .

   cq insertsc(marksc(A,q1),k,v,q) = marksc(insertsc(A,k,v,q),q1)
                                     if q =/= q1 .

   eq unmark(create,q) = create .
   eq unmark(insertsc(A,k,v,q),q1) = insertsc(unmark(A,q1),k,v,q) .
   cq unmark(marksc(A,q1),q) = unmark(A,q) if q == q1 .
   cq unmark(marksc(A,q),q1) = marksc(unmark(A,q1),q) if q =/= q1 .

   eq lastsc(insertsc(A,k,v,q)) = q .
   eq lastsc(marksc(A,q)) = lastsc(A) .
         *** because OBJ3 equations are really rewriting rules ...

   eq newsc(create) = firstsc .
   cq newsc(insertsc(A,k,v,q)) = nextsc(lastscgenerated(insertsc(A,k,v,q)))
                 if holes?(insertsc(A,k,v,q)) == false .
   cq newsc(insertsc(A,k,v,q)) = lastscreleased(insertsc(A,k,v,q))
                 if holes?(insertsc(A,k,v,q)) == true .
   eq newsc(marksc(A,q)) = lastscreleased(marksc(A,q)) .
         *** because OBJ3 equations are really rewriting rules ...
```

```
   eq holes?(create) = false .
   eq holes?(insertsc(A,k,v,q)) = holes?(A) .
          *** it's not necessary because ...
   eq holes?(marksc(A,q)) = true .

   eq lastscreleased(marksc(A,q)) = q .

   eq lastscgenerated(insertsc(create,k,v,q)) = q .
   cq lastscgenerated(insertsc(insertsc(A,k,v,q),k1,v1,q1)) = q1
                      if q1 > lastscgenerated(insertsc(A,k,v,q)) .
   cq lastscgenerated(insertsc(insertsc(A,k,v,q),k1,v1,q1)) =
                      lastscgenerated(insertsc(A,k,v,q))
                      if lastscgenerated(insertsc(A,k,v,q)) > q1 .

   eq nextsc(q) > firstsc = true .
   eq firstsc > nextsc(q) = false .
   eq nextsc(q) > nextsc(q1) = q > q1 .

   eq remove (insertsc(A,k,v,q),k) = marksc(A,q) .
   cq remove (insertsc(A,k,v,q),k1) = insertsc(remove(A,k1),k,v,q)
                                      if k =/= k1 .
   eq remove (marksc(A,q),k) = marksc(remove(A,k),q) .
          *** because OBJ3 equations are really rewriting rules ...

   eq removesc (insertsc(A,k,v,q),q) = marksc(A,q) .
   cq removesc (insertsc(A,k,v,q),q1) = insertsc(removesc(A,q1),k,v,q)
                                        if q =/= q1 .
   eq removesc (marksc(A,q),q1) = marksc(removesc(A,q1),q) .
          *** because OBJ3 equations are really rewriting rules ...

   eq retrieve (insertsc(A,k,v,q),k) = v .
   cq retrieve (insertsc(A,k,v,q),k1) = retrieve(A,k1) if k =/= k1 .
   eq retrieve (marksc(A,q),k) = retrieve(A,k) .
          *** because OBJ3 equations are really rewriting rules ...

   eq retrievesc (insertsc(A,k,v,q),q) = v .
   cq retrievesc (insertsc(A,k,v,q),q1) = retrievesc(A,q1)
                                          if q =/= q1 .
   eq retrievesc (marksc(A,q),q1) = retrievesc(A,q1) .
          *** because OBJ3 equations are really rewriting rules ...

   eq defined? (create,k) = false .
   eq defined? (insertsc(A,k,v,q),k1) = k == k1 or defined?(A,k1) .
   eq defined? (marksc(A,q),k) = defined?(A,k) .
          *** because OBJ3 equations are really rewriting rules ...

   eq definedsc? (create,q) = false .
   eq definedsc? (insertsc(A,k,v,q),q1) = q == q1 or definedsc?(A,q1) .
   eq definedsc? (marksc(A,q),q1) = definedsc?(A,q1) .
          *** because OBJ3 equations are really rewriting rules ...

   eq modifysc(insertsc(A,k,v,q),q,v1) = insertsc(A,k,v1,q) .
   cq modifysc(insertsc(A,k,v,q),q1,v1) =
                  insertsc(modifysc(A,q1,v1),k,v,q)
                  if q =/= q1 .
   eq modifysc(marksc(A,q1),q,v) = marksc(modifysc(A,q,v),q1) .

   eq keyforsc (insertsc(A,k,v,q),q) = k .
   cq keyforsc (insertsc(A,k,v,q),q1) = keyforsc(A,q1)
                                        if q =/= q1 .
   eq keyforsc (marksc(A,q),q1) = keyforsc(A,q1) .
          *** because OBJ3 equations are really rewriting rules ...

   eq scforkey (insertsc(A,k,v,q),k) = q .
   cq scforkey (insertsc(A,k,v,q),k1) = scforkey(A,k1)
                                        if k =/= k1 .
   eq scforkey (marksc(A,q),k) = scforkey(A,k) .
          *** because OBJ3 equations are really rewriting rules ...

   cq insertsc(A,k,v,q) = MultipleInserting if defined?(A,k) .
   cq insertsc(A,k,v,q) = MultipleInserting if definedsc?(A,q) .
   eq remove (create,k) = keyIsNotInserted .
   eq retrieve(create,k) = keyIsNotInserted .
   eq removesc (create,q) = UndefinedShortcut .
   eq retrievesc(create,q) = UndefinedShortcut .
   eq modifysc(create,q,v) = UndefinedShortcut .
   eq keyforsc(create,q) = UndefinedShortcut .
   eq scforkey(create,k) = keyIsNotInserted .
endo
```

Appendix 2. Implementation of the map with shortcuts

In this appendix we present the specification (generic package) and the implementation (package body) of the store with shortcuts using ADA95.

Generic Package

```
--/* package corresponding to the original ADT */

WITH Store;

GENERIC

--/* parameters of the original package */

  TYPE key IS PRIVATE;

  TYPE value IS PRIVATE;

  Number_Of_Buckets: IN Positive;

  WITH FUNCTION Hash_Of (The_key: IN key) RETURN Positive;

PACKAGE StoreSC IS

--/* new type that provide efficient acces to elements */
  TYPE shortcut IS PRIVATE;

--/* redefinition of the type of the original ADT */
  TYPE storesc IS PRIVATE;


--/* new operations to acces by means of shortcuts */
  FUNCTION last_sc (In_The_store  : IN storesc) RETURN shortcut;
           -- Returns the shortcut bound to the last added element
           -- (pair consisting of key and value).
           -- If there is no one element bound returns
           -- an undefined shortcut.

  PROCEDURE remove_sc (In_The_store  : IN OUT storesc;
                       The_Shortcut: IN shortcut);

  FUNCTION retrieve_sc (In_The_store  : IN storesc;
                        The_Shortcut: IN shortcut) RETURN value;
           -- Returns the value bound to the shortcut.
           -- If there is no one value bound to the shortcut
           -- the Undefined_Shortcut is raised.

  FUNCTION defined_sc (In_The_store  : IN storesc;
                       The_Shortcut: IN shortcut) RETURN boolean;

  FUNCTION key_for_sc (In_The_store  : IN storesc;
                       The_Shortcut: IN shortcut) RETURN key;
           -- Returns the key bound to the shortcut.
           -- If there is no one key bound to the shortcut
           -- the Undefined_Shortcut is raised.

  FUNCTION sc_for_key (In_The_store  : IN storesc;
                       The_key: IN key) RETURN shortcut;
           -- Returns the shortcut bound to the key.
           -- If there is no one shortcut bound to the key
           -- the Key_Is_Not_Inserted is raised.

--/* operations from the original package */
           -- The behaviour of these operations is the same that
           -- the operations of the original package.

  PROCEDURE insert  (In_The_store   : IN OUT storesc;
                     The_key   : IN key;
                     And_The_value: IN value);

  PROCEDURE remove(In_The_store   : IN OUT storesc;
                   The_key  : IN key);

  FUNCTION defined (In_The_store: IN storesc;
                    The_key: IN key) RETURN boolean;

  FUNCTION retrieve(In_The_store: IN storesc;
                    The_key: IN key) RETURN value;
```

```
--/* exceptions from the original package */
  Overflow: exception;
  Key_Is_Not_Inserted: exception;
  Multiple_Inserting: exception;

--/* new exceptions */
  Undefined_Shortcut: exception;

 PRIVATE
--/* implementation of the ADT with shortcuts */
    TYPE node;
    TYPE shortcut IS ACCESS node;

    -- Instanciation of the original package
    PACKAGE old_store IS NEW store(key => key,
                                   value => shortcut,
                                   Number_Of_Buckets => Number_Of_Buckets,
                                   Hash_Of => Hash_Of);

    TYPE storesc IS RECORD
      m: old_store.store;             -- The old store.
      last_sh: shortcut := NULL;      -- To obtain the shortcut bound
                                      -- to the last element inserted
    END RECORD;

    TYPE node IS RECORD
      k: key;
      v: value;
      deleted: boolean := FALSE;     -- To avoid meaningless access
      previous: shortcut := NULL;    -- To bring up-to-date the insert order
      next: shortcut := NULL;
    END RECORD;

END Storesc;
```

Package Body

```
PACKAGE BODY StoreSC IS


--/* new operations */


  FUNCTION last_sc (In_The_store  : IN storesc) RETURN shortcut IS
    BEGIN
       RETURN In_The_store.last_sh;
    END last_sc;

  PROCEDURE remove_sc (In_The_store  : IN OUT storesc;
                       The_Shortcut: IN shortcut) IS
    BEGIN
       The_Shortcut.deleted:= TRUE;
       In_The_store.last_sh:= The_Shortcut.previous;
       IF The_Shortcut.previous /= NULL THEN
          The_Shortcut.previous.next:= The_Shortcut.next;
       END IF;
       IF The_Shortcut.next /= NULL THEN
          The_Shortcut.next.previous:= The_Shortcut.previous;
       ELSE
          In_The_store.last_sh:= The_Shortcut.previous;
       END IF;
       The_Shortcut.previous:= NULL;
       The_Shortcut.next := NULL;
       old_store.remove(In_The_store.m,The_Shortcut.k);
    END remove_sc;


  FUNCTION retrieve_sc (In_The_store  : IN storesc;
                        The_Shortcut: IN shortcut) RETURN value IS
    BEGIN
       IF The_Shortcut.deleted THEN
          RAISE Undefined_Shortcut;
       ELSE
          RETURN The_Shortcut.v;
       END IF;
     exception
       WHEN Constraint_Error => raise Undefined_Shortcut;
    END retrieve_sc;

  FUNCTION defined_sc (In_The_store  : IN storesc;
                       The_Shortcut: IN shortcut) RETURN boolean IS
    BEGIN
       RETURN The_Shortcut /= NULL AND NOT The_Shortcut.deleted;
    END defined_sc;

  FUNCTION key_for_sc (In_The_store  : IN storesc;
                       The_Shortcut: IN shortcut) RETURN key IS
    BEGIN
       IF The_Shortcut.deleted THEN
          RAISE Undefined_Shortcut;
       ELSE
          RETURN The_Shortcut.k;
       END IF;
     exception
       WHEN Constraint_Error => raise Undefined_Shortcut;
    END key_for_sc;

  FUNCTION sc_for_key (In_The_store  : IN storesc;
                       The_key: IN key) RETURN shortcut IS
    BEGIN
       RETURN old_store.retrieve(In_The_store.m,The_key);
    END sc_for_key;
```

```
--/* operations from the original package */


  PROCEDURE insert(In_The_store   : IN OUT storesc;
                   The_key   : IN key;
                   And_The_value: IN value) IS
    aux: shortcut;
    BEGIN
       aux:= NEW node'
                     (k => The_key,
                      v => And_The_value,
                      deleted => FALSE,
                      previous => NULL,
                      next => NULL);
       old_store.insert(In_The_store.m,The_key,aux);
       aux.previous := In_The_store.last_sh;
       In_The_store.last_sh:= aux;
     exception
       WHEN Storage_Error => raise Overflow;
    END insert;

  PROCEDURE remove(In_The_store   : IN OUT storesc;
                   The_key  : IN key) IS
    aux: shortcut;
    BEGIN
       aux:= old_store.retrieve(In_The_store.m,The_key);
       aux.deleted:= TRUE;
       IF aux.previous /= NULL THEN
          aux.previous.next:= aux.next;
       END IF;
       IF aux.next /= NULL THEN
          aux.next.previous:= aux.previous;
       ELSE
          In_The_store.last_sh:= aux.previous;
       END IF;
       aux.previous := NULL;
       aux.next := NULL;
       old_store.remove(In_The_store.m,The_key);
    END remove;

  FUNCTION defined (In_The_store: IN storesc;
                    The_key: IN key) RETURN boolean IS
    BEGIN
       RETURN old_store.defined(In_The_store.m,The_key);
    END defined;

  FUNCTION retrieve(In_The_store: IN storesc;
                    The_key: IN key) RETURN value IS
    aux: shortcut;
    BEGIN
       aux:= old_store.retrieve(In_The_store.m,The_key);
       RETURN aux.v;
    END retrieve;

END storesc;
```