STUDIA UNIV. BABEŞ–BOLYAI, INFORMATICA, Volume LI, Number 1, 2006

A UNIFORM ANALYSIS OF LISTS BASED ON A GENERAL NON-RECURSIVE DEFINITION

VIRGINIA NICULESCU

Abstract. The paper presents a general, non-recursive definition of lists, in order to be used as a starting point for a uniform analysis of them. A general parameterized abstract data type *List* is defined, based on the type parameter *Position*, and type parameter *TE* (the type of elements contained in the list). By instantiating the parameter *Position* to the concrete types *Index*, *SNode* and *DNode* we obtained the abstract data types: *IndexedList*, *SinglyLinkedList*, and *DoublyLinkedList*. For them different representations could be considered.

This definition that starts from a general parameterized ADT has the advantage of uniform formal introduction of any type of lists. The presentation is open to other possible instantiations of *Position* parameter. In order to illustrate this, the case of unrolled linked lists is presented.

Also, this approach emphasizes the differences between the abstract data types of linked lists and the linked representation of the structures.

1. Introduction

Lists are very important data structures, which are widely used in computer science.

A recursive definition of an ADT for untyped, and mutable lists, specifies the domain as:

$$List = \{l | l \text{ is empty} \vee l = (e, l1),\ \text{where } l1 : List \wedge e \text{ is an entity}\}$$

and the operations:

(1) a constructor for creating an empty list;
(2) an operation for testing whether or not a list is empty;
(3) an operation for prepending an entity to a list (cons in Lisp);
(4) an operation for determining the first component (or the "head") of a list (car in Lisp);

Received by the editors: March 10, 2006.

2000 *Mathematics Subject Classification.* 68P05, 68P01.

1998 *CR Categories and Descriptors.* E.1 [**Data**]: Data Structures – *Lists, stacks, and queues*.

(5) an operation for referring to the list consisting of all the components of a list except for its first (or its "tail") (cdr in Lisp);

This definition is preferred in functional languages, but in imperative programming, in which iteration is preferred to recursion, this definition is not so appropriate.

Under the imperative paradigm, a list is usually defined as an instance of an abstract data type (ADT) formalizing the concept of an ordered collection of entities. However, there are a lot of types of list data structures, and also in the literature could be found different definitions for ADT List. Also, the ADTs List are introduced very often in an informal way. Different implementations of list data structures are discussed more than a general ATD definition for them, and because of this there is not a uniform approach of the subject in the literature [2, 4, 5, 6, 7].

In practice, lists are usually implemented using arrays or linked lists of some sort; due to lists sharing certain properties with arrays and linked lists. Many times, the term list is used synonymously with linked list. *Sequence* is another used name, emphasizing the ordering and suggesting that it may not be a linked list. However, it is generally assumed that elements can be inserted into a list in constant time, while access of a random element in a list requires linear time; this is to be contrasted with an array (or vector), for which the time complexities are reversed.

Lists have the following properties:

- The contents or data type of lists may or may not vary at runtime.
- Lists may be typed. This implies that the entries in a list must have types that are compatible to the list base type.
- They may be sorted or unsorted.
- Random access over lists may or may not be possible.
- Equality of lists:
  - : - In mathematics, sometimes equality of lists is defined simply in terms of object identity: two lists are equal if and only if they are the same object.
  - : - In modern programming languages, equality of lists is normally defined in terms of structural equality of the corresponding entries, except that if the lists are typed, then the list types may also be relevant.

We propose, in this paper, a formal, general, non-recursive definition for a typed, unsorted ADT *List*. This could be used, then, as the starting point for a formal introduction of all the aspects referring to lists.

2. ADT List

We consider a list being a collection of elements of the same type (TE), where each element has a certain position. Each element in a list has a *successor* element and a *predecessor* element in the list, with two exceptions: the first element that has only a successor, and the last element that has only a predecessor.

The constructive approach is used for defining abstract data types.

We define a parameterized Abstract Data Type *List*, with two type parameters:

(i) The type parameter TE, which represents the type of the constitutive elements, characterized by at least two operations: *assign*, and *equals*.

(ii) *Position* is a type parameter that emphasizes the type of elements' positions. The instances of the type parameter *Position* are characterized by the existence of two operations: *next*, and *prev*. The properties of this type parameter are strictly connected to the type *List*. Knowing a position of an element in a list, we have, based on the list definition, to be able to extract the element stored at that position, and to compute successor and predecessor elements in the list.

Domain

$List(Position, TE) = \{l | l$ is a list of elements of type TE,
in which each element has a position of type $Position\}$

Operations:

(1) $createEmpty(l)$
pre: true
post: $l : List$ and l is empty

(2) $length(l)$
pre: $l : List$
post: $result =$ the number of the elements of the list l

(3) $getFirstPosition(l)$
pre: $l : List$
post: $result = \begin{cases} \perp, l \text{ is empty list} \\ p, p : Position \text{ is the first position in the non-empty list } l \end{cases}$

(4) $getLastPosition(l)$
pre: $l : List$
post: $result = \begin{cases} \perp, l \text{ is empty list} \\ p, p : Position \text{ is the last position in the non-empty list } l \end{cases}$

(5) $valid(l, p)$
pre: $l : List$ and $p : Position$
post: $result = \begin{cases} \text{true, if } p \text{ is a valid position in } l \\ \text{false, otherwise} \end{cases}$

(6) $addFirst(l, e)$
pre: $l : List$ and $e : TE$
post: $l^{'} = (e, l)$

(7) $insert(l, p, e)$
pre: $l : List$ and $p : Position$ and $e : TE$ and $valid(p, l)$
post: $l^{'}$ is the list l after inserting e on the next position of p

(8) $delete(l, p)$
pre: $l : List$ and $p : Position$ and $valid(p, l)$
post: $l^{'}$ is the list l after removing the element on the position p

(9) $next(l, p)$
pre: $l : List$ and $p : Position$ and $valid(p, l)$
$result = \begin{cases} \text{the next position of } p\text{, if } p \text{ is not the last position} \\ \perp\text{, if } p \text{ is the last position} \end{cases}$

(10) $prev(l, p)$
pre: $l : List$ and $p : Position$ and $valid(p, l)$
post: $result = \begin{cases} \text{the previous position of } p\text{, if } p \text{ is not the first position} \\ \perp\text{, if } p \text{ is the first position} \end{cases}$

(11) $getElement(l, p)$
pre: $l : List$ and $p : Position$ and $valid(p, l)$
post: $result =$ the element e at the position p and $e : TE$

(12) $setElement(l, p, e)$
pre: $l : List$ and $p : Position$ and $valid(p, l)$ and $e : TE$
post: the element at the position p is equal to e

(13) $iterator(l)$
pre: $l : List$
post: $result =$ iterator of type $ListIterator$ on the list l

We have used the notation $\perp$ for the undefined position, which is a special value of *Position* type. (The undefined position is always not valid, but a value which is not valid for a list does not have to be equal to $\perp$.) In the formal specification of an operation with parameter l, $l^{'}$ denotes l after the execution of that operation. We consider *ListIterator*, the type of a bidirectional "Read-Write" iterator on lists, characterized by the following operations (beside the creational operations):

(1) $valid(it, l)$
pre: $l : List$ and $it : ListIterator$
post: $result = \begin{cases} \text{true, if } it \text{ indicates no element in the list } l \\ \text{false, otherwise} \end{cases}$

(2) $next(it, l)$
pre: $l : List$ and $it : ListIterator$ and $valid(it, l)$
post: $it^{'}$ indicates the next position of that indicated by it

(3) $prev(it, l)$
pre: $l : List$ and $it : ListIterator$ and $valid(it, l)$
post: $it^{'}$ indicates the previous position of that indicated by it

(4) $element(it, l)$
pre: $l : List$ and $it : ListIterator$ and $valid(it, l)$
post: $result = e$, where e is the element indicated by it

(5) $insert(it, l, e)$
pre: $l : List$ and $it : ListIterator$ and $e : TE$ and $valid(it, l)$
post: $l^{'}$ is l after inserting e on the next position of that indicated by it

(6) $delete(it, l)$
pre: $l : List$ and $it : ListIterator$ and $valid(it, l)$
post: $l^{'}$ is l after removing the element on the position indicated by it

Since *insert* and *delete* operations are defined for *ListIterator*, a list could also be modified using an iterator. More restrictive iterator types could be considered, too.

The position of one element in the list allows us to obtain the element, and it may be given either by giving the index of the element in the list, or by given a reference to the location where that element is stored.
So, possible choices for *Position* are:

(1) $Index = \{i | i \in \mathbb{N}\}$
The resulted lists are characterized by the
ADT $List(Index, TE) = IndexedList(TE)$.

(2) $SNode = \{(e, n) | e : TE \wedge n : \text{ref}(SNode)\}$
The resulted lists are characterized by the
ADT $List(\text{ref}(SNode), TE) = SinglyLinkedList(TE)$.

(3) $DNode = \{(p, e, n) | e : TE \wedge n, p : \text{ref}(DNode)\}$
The resulted lists are characterized by the
ADT $List(\text{ref}(DNode), TE) = DoublyLinkedList(TE)$.

The notation ref($Type$) specifies a type of references to values of type $Type$. The types $ref(SNode)$ and $ref(DNode)$ use the value *null* (it could be a null pointer or a null reference to an entry into an array) for specifying $\perp$, and the type $Index$ could use the value 0 for the same purpose.

By choosing concrete instances of type *Position* we obtain abstract data types, which are parameterized only by the elements' type TE. For an ADT like this, we may choose different representations of the values of the domain *List* and corresponding implementations of the operations. So, different list data structures are obtained.

Other instantiations for *Position* may be considered. For example, for a *unrolled linked* or *chunk* list that is a linked list in which each node contains an array of data values, the *Position* could be defined as a reference to the type

$$\begin{aligned}ArrayNode = \{&(A, next, max, ind)|\\ &A : Array(TE) \wedge next : \text{ref}(ArrayNode) \wedge max, ind \in \mathbb{N}^*\}\end{aligned}$$

where max represents the number of the elements stored in the node, and ind represents the current position into the node. The unrolled linked lists increase the cache performance while decreasing the memory overhead for references.

3. *IndexedList*

*IndexedList*s are characterized by the fact that elements are accessed directly by their indices. The internal representation of the lists is independent of their interface, so we may considere an array representation, but also a linked representation for *IndexedList*.

If we consider that the lists are represented using a dynamic array (its size is not constant) of elements of type TE, we obtain a list data structure in which direct access to the elements is possible in an $O(1)$ time, but *delete* and *insert* operations need each $O(n)$ time. Usually, this data structure is called *ArrayList* or *Sequence*.

Another possibility to implement *IndexedList* is based on a linked representation. In this case, each element is placed in a node that contains not only the value of the element, but also a pointer to the node that contains the next element in the list; in a doubly linked representation, a pointer to the node that contains the previous element in the list is stored, too. This representation of the *IndexedList* does not improve the time complexity of the operations *insert* and *delete*, since the positions are still referred to by indices. The improvement could be obtained if the insertions and deletions are done using an iterator, and not using the operations of the interface.

The majority languages defines in their collection libraries implementations of the ADT *IndexedList*. This is the case of Java [3], where the interface *List* corresponds to the interface of *IndexedList*, and the classes *ArrayList* and *LinkedList* are implementations of this. Therefore, the class *LinkedList* has not the type that corresponds to a linked list since its interface is an *IndexedList* interface. Only, its representation is a linked one. The same situation there is in C++, in STL library [8].

The *IndexedList* ADT is preferred to be implemented, since its interface does not contain parameters of type pointer or reference, which may bring some problems.

4. *LinkedList*

We will treat together the ADT *SinglyLinkedList* and ADT *DoublyLinkedList* since their problems are similar.

The positions, in this case, are expressed by the addresses where the elements are stored, and the elements are accessible through them. For representation, we may consider both static and dynamic linked allocation.

Dynamic linked allocation is based on nodes that contains the element's value and pointers to the node of the next element (and to the previous element, for the *DoublyLinkedList*). A representation for *SinglyLinkedList*s may use a pointer to the first node, and a representation for *DoublyLinkedList*s may use two pointers to the first and last nodes.

For *DoublyLinkedList*s, the *insert* and *delete* operations take $O(1)$ time, fact that corresponds to the general meaning of lists.
Another advantage of *DoublyLinkedList*s is that an easy reverse traversal is possible.

In the *SinglyLinkedList* case, *insert* operation also takes $O(1)$ time. But *delete* operation takes $O(n)$ time, because it has to compute the previous position of the element to be deleted, and the *prev* operation takes $O(n)$ time, in this case.

Static linked storage means that not dynamically allocation is used, but an associative array. For *SinglyLinkedList*s each entry of the associative array contains a value of type TE and a link to the sucessor element. Inside an array the reference(address) of an element is given by its index, so in this case a link is expressed as an index. The free space is also managed as a linked list. In the *DoublyLinkedList* case, an entry of the associative array also contains a link to the predecessor element.

5. Lists with Cursor

For each ADT List, a representation with a cursor could be chosen. In this case, a current position is stored (in the cursor) for any list, so the *Position* parameter p of the operations: *insert*, *delete*, *next*, *prev*, *getElement*, and *valid*, is included in the *List* parameter.

This representation is not common for indexed lists, but it brings some advantages for linked lists. The complexity of the operations is not influenced by the storing of this current position, but the advantage is given by the fact that the user has not direct access to the addresses where the elements are stored, and so the information is better protected. For linked lists, this cursor acts as an iterator on that list.

6. Conclusions

There are a lot of definitions of lists in literature, and they are very often informally defined. We have presented in this paper a formal definition for a general ADT List, from which specialized ADTs List are obtained by instantiation of the type parameter used for specifying positions in lists. The discussed ADTs, which are obtained after *Position* instantiation are *IndexedList*, *SinglyLinkedList*, and *DoublyLinkedList*. This analysis that starts from a general parameterized ADT has the advantage of a uniform formal introduction of any type of lists. The presentation is open to other possible instantiations of *Position* parameter.

This formal and general definition of lists emphasizes very clearly the differences between an indexed list (in which elements are referred to by using their number into the list) implemented in a linked way (as LinkedList in Java), and an implementation of ADT LinkedList.

Acknowledgment: This general and formal definition of lists was developed after long and interesting discussions that I had with Gabriela Serban referring to Data Structures Course. I am grateful for her suggestions and observations.

References

[1] Aho, A. V., Data Structures and Algorithms. Addison-Wesley, 1983.
[2] Amsbury,W., Data Structures - From Arrays to Priority Queues. 1985.
[3] Arnold, K., Holmes, D., Gosling, J., *The Java Programming Language*. Addison-Wesley, 2000.
[4] Horowitz, E, Sartaj Sahni, Fundamentals of Data Structures in C++. Computer Science Press, New York, 1995.
[5] Cormen, T. H., Leiserson, C. E., Rivest, R. L., Stein, C., Introduction to algorithms, sec. ed. MIT Press 2001.
[6] Mount, D. M., Data Structures, University of Maryland, 1993.
[7] Standish, T.A.,Data Structures, Algorithms and Software Principles. Addison-Wesley, 1994.
[8] Musser, D.R., Scine A., *STL Tutorial and Reference Guide: C++ Programming with Standard Template Library*, Addison-Wesley, 1995.

Department of Computer Science, Babeş-Bolyai University, Cluj-Napoca
E-mail address: vniculescu@cs.ubbcluj.ro