FIT2004: Supplementary notes for lecture 1

Abstract Data Types

DANIEL ANDERSON [1]

When dealing with and analysing data structures, it often helps to think about them in a more formal, abstract way. Doing so allows us to not only understand how an algorithm or data structure works, but to also prove rigorously that it works correctly.

Summary: Abstract Data Types

In this lecture, we cover:

- Thinking about abstract data types (ADTs) algebraically and formally
- The `List` ADT
- The `(Binary) Tree` ADT
- Formally reasoning about algorithms on ADTs

Recommended Resources: Algebraically defined ADTs

- `http://www.csse.monash.edu.au/~lloyd/tildeAlgDS/List/`
- `http://www.csse.monash.edu.au/~lloyd/tildeAlgDS/Tree/`

The List ADT

The list is a flexible ADT, typically described by a sequential linked structure consisting of elements chained together one after the other. Because of this, it is highly suited to the sequential processing of elements, but not so useful for situations requiring random access to elements. We can describe a list algebraically with the following set of definitions.

Types:

```
list e = nil | cons(e × (list e))
```

Constant:

```
nil
```

Operations:

```
null :  list e → boolean
head :  list e → e
tail :  list e → list e
cons :  e × (list e) → list e
```

Rules:

```
null(nil) = true
null(cons(x,L)) = false

head(cons(x,L)) = x
head(nil) = error

tail(cons(x,L)) = L
tail(nil) = error
```

Shorthand:

$[x_1, x_2, \ldots, x_n]$ = cons(x_1,cons(x_2, ...cons(x_n,nil)))

[1] Faculty of Information Technology, Monash University: daniel.anderson@monash.edu. These notes are based on the lecture slides developed by Arun Konagurthu for FIT2004.

What does this all mean?

Definition:

We defined the list type algebraically as

```
list e = nil | cons(e  (list e)).
```

This can be read as:

"A **list** of elements of type **e** is (=) either empty (**nil**) or (|) is **cons**tructed (**cons**) using an element of type **e** and ($\times$) another list of type e".

Simply, a list is either empty, or it is an element (called the head of the list)followed by a list. This is a *recursive* definition of a data type.

Operations:

The operations define what we can do with a list. Here, we have defined that we can check whether a list is empty (called `null`), ask for the head element of the list (`head`). ask for the tail of the list (`tail`), which is the remaining list when we exclude the head and construct a list from a given head element and a given tail list (`cons`).

Rules:

While the operations define what we can do with a list, the rules tell us how these operations behave. The `null` operation returns `true` for the empty list `nil` and `false` otherwise. The `head` operation returns the head element of the list, or is an error if invoked on the empty list. The `tail` operation returns the tail of the list, or is an error if invoked on an empty list.

Functions on lists

In addition to the fundamental operations, we can also define *functions* on lists using their formal algebraic structure. For example, we have the `length` function which computes the length of a list. It can be defined algebraically like so.

```
length(nil) = 0
length(const(h, T)) = 1 + length(T)
```

Just like our previous definitions, the `length` function has been defined recursively. The length of the empty list (`nil`) is zero, and the length of any other list is 1 plus the length of the tail of the list. As you will have noticed, recursive definitions must always have a base case, otherwise they simply do not make sense.

We can also define the `append` function, which joins two lists together. It can be written algebraically like this.

```
append(nil, L2) = L2
append(L1, L2) = append(cons(h1, T1), L2) = cons(h1, append(T1, L2))
```

Here, we have used to definition of the list `L1` to expand it into `cons(h1, T1)` in order to make the definition of `append` easier to understand, since it involves splitting the list `L1` into its head and tail components. We could have instead written

```
append(nil, L2) = L2
append(L1, L2) = cons(head(L1), append(tail(L1), L2),   L1 ≠ nil
```

which would have the same meaning as above. The meaning of append should not be too hard to understand. Appending a list (`L2`) to the empty list `nil` leaves us with the same list, while appending a non-empty list `L1` to `L2` is equivalent to **cons**tructing a new list with the head of `L1` and a tail which is the concatenation of `tail(L1)` and `L2`. Take some time to absorb and understand the recursive behaviour of the `append` function.

Formal proofs using algebraic functions

Why do we really care about defining the `append` function formally rather than just saying "join two lists together?" The reason is because we can use these formal definitions to formally prove certain properties about lists.

Theorem: The append function is associative

The append function is an *associative operation.* Formally, this means that

```
append(L1,(append(L2,L3))) = append(append(L1,L2),L3).
```

In other words, when joining three lists together, it does not make a difference whether we join the first and second or the second and third lists first.

Proof

We will prove that the `append` function is associative using induction on the list. First we prove our base case, that is that the `append` function is associative when `L1 = nil`.

Base Case:

```
append(nil ,(append(L2,L3)))                    // using the definition of append
  = append(L2,L3)

append(L2,L3)                                    // using the definition of append
  = append(append(nil,L2),L3)
```

therefore,

```
append(nil ,(append(L2,L3))) = append(append(nil,L2),L3).
```

which is our claim in the case where `L1 = nil`. We now prove the general case using induction on the list.

Inductive Step:

Suppose as our inductive hypothesis that some list `T1` satisfies

```
append(T1,(append(L2,L3))) = append(append(T1,L2),L3).                    *
```

Define a new list `L1 = cons(h1, T1)` for some element `h1`. We aim to prove that

```
append(L1,(append(L2,L3))) = append(append(L1,L2),L3).
```

We begin by using the definition of `L1` that we have constructed, then applying the definitions of the append function and the list operations.

```
append(L1,(append(L2,L3)))                       // Apply our definition of L1.
  = append(cons(h1, T1), append(L2, L3))

append(cons(h1, T1), append(L2, L3))             // Apply the definition of append
  = cons(h1, append(T1, append(L2, L3)))
```

We can now use the inductive hypothesis (*) to proceed. This is the most important part of the proof!

```
cons(h1, append(T1, append(L2, L3)))          // Apply the inductive hypothesis (*)
  = cons(h1, append(append(T1, L2), L3))
```

We can now clean up what remains and deduce the desired result.

```
cons(h1, append(append(T1, L2), L3))          // Apply the definition of append
  = append(cons(h1, append(T1, L2)), L3)

append(cons(h1, append(T1, L2)), L3)          // Apply the definition of append
  = append(append(cons(h1, T1), L2)), L3)

append(append(cons(h1, T1), L2)), L3)          // Apply our definition of L1
  = append(append(L1,L2),L3)
```

We have hence shown for `L1 = cons(h1, T1)` that

```
append(L1,(append(L2,L3))) = append(append(L1,L2),L3).
```

Therefore, by induction on the list `L1`, we can conclude that for all lists `L1, L2, L3`

```
append(L1,(append(L2,L3))) = append(append(L1,L2),L3).
```

Since the list structure was defined recursively, it makes sense that induction should be used to prove properties about it, since an inductive proof is a recursive kind of proof.

Some more functions on lists

We can define some more useful functions on sorted lists. We have the `merge` function which merges two sorted lists into one sorted list.

```
merge(nil, nil) = nil
merge(L1, nil) = L1
merge(nil, L2) = L2

merge(L1, L2) = cons(head(L1), merge(tail(L1), L2))   if head(L1) < head(L2)
merge(L1, L2) = cons(head(L2), merge(L1, tail(L2)))   if head(L1) ≥ head(L2)
```

We can also `reverse` a list.

```
reverse(nil) = nil
reverse(cons(h, T)) = append(T, cons(h, nil))
```

Recursive vs. iterative structures

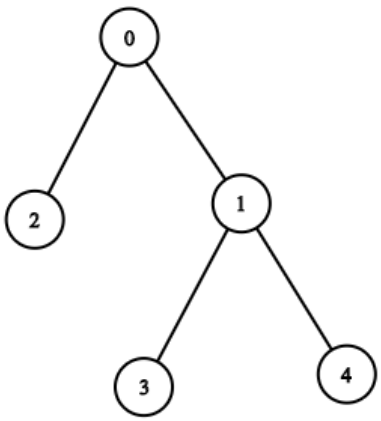
Many of the operations on the list data type and indeed on many other recursively defined data types are defined recursively. Expressing operations and functions recursively is often conceptually simpler than expressing them iteratively and can frequently simplify proofs by providing a natural framework on which to perform proof by induction.

In practice however, iterative routines are often simpler to implement and more efficient. Depending on the structure though, it is often more difficult to prove the correctness of iterative routines.

The Tree ADT

A (rooted) tree is a hierarchical data structure consisting of a set of vertices (or nodes) and edges (or arcs), where each edge connects a child vertex to its parent. The special *root* node is the only node without a parent, where all other nodes have a unique parent. A tree with n vertices has exactly $n - 1$ edges, which leads them to having several very nice properties:

- For any two vertices, there is a unique path of edges between them
- A tree contains no cycle of vertices connected by edges



A binary rooted tree containing five vertices and four edges

A tree is called a binary tree if each node has no more than two children. We can define a binary tree algebraically like this.

Types:

```
type tree e = nilTree | fork e × (tree e) × (tree e)
```

Operations:

```
empty:  tree e → boolean
leaf :  tree e → boolean
fork :  e × tree e × tree e → tree e
left :  tree e → tree e
right:  tree e → tree e
contents:  tree e → e
```

Rules:

```
empty(nilTree) = true
empty(fork(x, T1, T2)) = false

leaf(fork(x, nilTree , nilTree)) = true
leaf(fork(x, T1, T2)) = false , if not empty(T1) or not empty(T2)
leaf(nilTree) = error

left(fork(x, T1, T2)) = T1
left(nilTree) = error

right(fork(x, T1, T2)) = T2
right(nilTree) = error

contents(fork(x, T1, T2)) = x
contents(nilTree) = error
```

Here, the `nilTree` represents an empty tree (one with no vertices). A tree is defined recursively as being either empty (the `nilTree`) or a single element corresponding to the root, and two subtrees (trees corresponding to the children of the root vertex) which are "forked" together.

The `empty` operation tells us whether or not a tree is empty and the `leaf` operations tells us whether a tree consists of a single vertex. The `left` operations returns the left subtree, the `right` operation returns the right subtree, and the `contents` operation returns the element contained in the root vertex.

Common functions on trees

Much like we could compute the length of a list, we can compute the *height* of a tree, which corresponds to how many layers of vertices the tree has. `height` can be defined recursively like this.

```
height(nilTree) = 0
height(fork(e, L, R)) = 1 + max(height(L), height(R))
```

We can also define the weight function, which corresponds to the number of vertices in the tree.

```
weight(nilTree) = 0
weight(fork(e, L, R)) = 1 + weight(L) + weight(R)
```

Tree traversal

Tree traversal is the process of sequencing the elements of a tree. There are three classic ways of recursively traversing a tree in which each element is visited exactly once in some order. The difference that distinguishes the tree is the order in which the root element is visited or processed. We sometimes call the process of converting a tree into a list via a traversal *flattening*.

```
flatten:  Tree t → list e
```

Inorder traversal: In an inorder traversal, the root node is visited in between visiting the left and right subtrees.

```
// Inorder flattening
flatten(nilTree) = nil
flatten(fork(e, L, R)) = append(flatten(L), cons(e, flatten(R)))
```

Preorder traversal: In a preorder traversal, the root node is visited before visiting the left and right subtrees.

```
// Preorder flattening
flatten(nilTree) = nil
flatten(fork(e, L, R)) = append(cons(e, flatten(L)), flatten(R))
```

Postorder traversal: In a postorder traversal, the root node is visited after visiting the left and right subtrees.

```
// Postorder flattening
flatten(nilTree) = nil
flatten(fork(e, L, R)) = append(flatten(L), append(flatten(R), cons(e, nil)))
```

These three traversals are examples of *depth-first* traversals. Other non-recursive traversals such as *breadth-first* traversals are also possible. A breadth-first traversal visits nodes in height order, so the root is visited first, then the children at height 1, then the children at height 2, and so on.