CSCI 3155, Recitation 2
Grammars, Languages, Abstract Syntax

Quick Review

- Some relevant terminology: syntax, semantics, concrete syntax, abstract syntax, higher-order abstract syntax, context-free grammar, language, sentence/string, Bauchus-Naur Form, production, derivation, parse tree, ambiguity, precedence, terminal, non-terminal

- This is the library we will use for this recitation:

  ```scala
  sealed abstract class Tree[T]
  case class Empty[T] extends Tree[T]
  case class Node[T](l: Tree[T], d: T, r: Tree[T]) extends Tree[T]
  ```

- Given the library above, write a recursive function that performs an *inorder* walk of the tree and generates a list of the values in tree.

  ```scala
  def inorder[T](t: Tree[T]): List[T] = t match{
    case Empty() => Nil
    case Node(l, d, r) => inorder(l) ::: (d :: inorder(r))
  }
  ```

  What is the complexity of this method?

  **Solution:** `:::` takes linear time in the number of elements in the list to walk the list, find last element and update pointers. We call append for every `Node()` in the tree. This gives us *quadratic* complexity in the nodes of the tree.

- What does a method that solves the same problem look like if we want it to have complexity linear in the nodes of the tree?

  **Hint:** Use a helper method with the following signature:

  ```scala
  inorderHelper[T](t: Tree[T], a: List[T]): List[T] =
  ```
Solution:

```scala
def inorderFast[T](t: Tree[T]): List[T] = {
  def inorderHelper[T](t: Tree[T], a: List[T]): List[T] = t match {
    case Empty() => a
    case Node(l, d, r) => inorderHelper(l, d::inorderHelper(r, a))
  }
  inorderHelper(t, Nil)
}

How do we know this solution is better?

Solution: Notice that the we only make use of the constant time operation prepend (`::`) and never use append (`:::`). We make linear number, in terms of nodes in the tree, of recursive calls to `inorderHelper()` and prepend. This gives an overall linear complexity in the number of nodes in the tree.

For clarity, here are the execution traces for the two methods on the following tree:

```scala
val tree = Node(Node(Empty(), 2, Empty()), 1, Node(Empty(), 3, Empty()))

inorder(tree)

inorder(Node(Empty(), 2, Empty())) ::: (1 :: inorder(Node(Empty(), 3, Empty())))
(inorder(Empty()):::(2::inorder(Empty()))):::(1::(inorder(Empty()):::(3::in(Empty()))))
(Nil ::: (2 :: Nil)) ::: (1 :: (Nil ::: (3 :: Nil)))
(Nil ::: List(2)) ::: (1 :: (Nil ::: List(3)))
//append takes linear time in the length of the lists,
//so eventually we end up with:
(List(2) ::: (1 :: List(3)))
(List(2) ::: List(1,3))
List(2,1,3)
```

inorderFast(tree)

```scala
inorderFast(Node(Node(Empty(), 2, Empty()), 1, Node(Empty(), 3, Empty())))
inorderH(Node(Empty(), 2, Empty()), 1 :: inorderH(Node(Empty(), 3, Empty()), Nil))
inorderH(Node(Empty(), 2, Empty()), 1 :: (inH(Empty(), 3 :: inorderH(Empty(), Nil))))
```
The point is that half of the operations in the execution trace of `inorder()` are the costly `:::`, whereas the execution trace of `inorderFast()` uses only prepends (::) and no `:::` operations.

**Structural Induction Proofs**

- We would like to prove that the two methods we just wrote produce the same result. Let’s formalize this.

- **Theorem 1.** For any tree value `t`, list expressions `acc`, `l_1` and `l_2`, if `inorder(t) →∗ l_1` and `inorderHelper(t, acc) →∗ l_2`, then `l_1 :: acc →∗ l'` and `l_2 →∗ l'` for some list expression `l'`.

- **Hint:** You are allowed to assume (without proof) that operations `:::` and `::` are associative.

**Solution:** We show the claim holds by structural induction on the tree value `tree`.

- **Base Case:** Let `tree = Empty()`. Then,

  \[
  \text{inorder}(\text{tree}) = \text{inorder}(\text{Empty}()), \text{rewrite } \text{tree} \\
  \quad →∗ \text{Nil}
  \]

  Next,

  \[
  \text{inorderHelper}(\text{tree}, \text{acc}) = \text{inorderHelper}(\text{Empty}(), \text{Nil}), \text{rewrite } \text{tree, acc} \\
  \quad →∗ \text{Nil}
  \]

  Finally, `Nil::Nil →∗ Nil` and `Nil →∗ Nil`.

- **How do we grow trees?**

  Here are a few simple ways of growing trees:
In general, one way to extend a tree is depicted below:

This suggests that we have to be careful in our Induction Hypothesis. It should captures the two subtrees!

– *Induction Hypothesis:*
Suppose for some tree values \( \text{left}, \text{right} \), and all list expressions \( \text{acc}, \ l_1, \ l_2, \ l', \ \text{acc}', \ r_1, \ r_2, \ r' \), if \( \text{inorder}(\text{left}) \rightarrow^* l_1, \ \text{inorderHelper}(\text{left}, \text{acc}) \rightarrow^* l_2 \), and \( \text{inorder}(\text{right}) \rightarrow^* r_1, \ \text{inorderHelper}(\text{right}, \text{acc'}) \rightarrow^* r_2 \), then \( l_1 :: \text{acc} \rightarrow^* l' \) and \( l_2 \rightarrow^* l' \), and \( r_1 :: \text{acc'} \rightarrow^* r' \) and \( r_2 \rightarrow^* r' \).

- Need to show: Assume IH, show Inductive Step.

- Inductive Step: Let \( \text{tree} = \text{Node}(\text{left}, \ d_{\text{tree}}, \ \text{right}) \).

Then,

\[
\text{inorderHelper}(\text{tree}, \text{acc'}) = \text{inorderHelper}(\text{Node}(\text{left}, \ d_{\text{tree}}, \ \text{right}), \text{acc'}) \rightarrow^* \text{inorderHelper}(\text{left}, \ d_{\text{tree}} :: \text{inorderHelper}(\text{right}, \text{acc'})) \\
\rightarrow^* \text{inorderHelper}(\text{left}, \ d_{\text{tree}} :: \ \text{r}_2), \text{by IH on right.}
\]

Let us call \( \text{acc} = (\ d_{\text{tree}} :: \ r') \), then we have:

\[
\rightarrow^* l_2, \text{ by IH on left,} \\
\rightarrow^* l', \text{ by IH part } l_2 \rightarrow^* l'.
\]

Next,

\[
\text{inorder}(\text{tree}) = \text{inorder}(\text{Node}(\text{left}, \ d_{\text{tree}}, \ \text{right})), \text{ rewrite tree,} \\
\rightarrow^* \text{inorder}(\text{left}) ::: (\ d_{\text{tree}} :: \text{inorder}(\text{right})), \text{ evaluation} \\
\rightarrow^* l_1 :: (\ d_{\text{tree}} :: \ r_1), \text{ by Induction Hypothesis}
\]

At this point we stop and notice the following:

\[
(d_{\text{tree}} :: r_1) :: \text{acc'} \rightarrow^* d_{\text{tree}} :: (r_1 :: \text{acc'}), \\
\text{by associativity of append and prepend operations} \\
\rightarrow^* d_{\text{tree}} :: r', \text{by IH on } r_1 \text{ and acc'}
\]

So then:

\[
\rightarrow^* (l_1 :: (d_{\text{tree}} :: r_1)) :: \text{acc'} \\
\rightarrow^* l_1 :: (d_{\text{tree}} :: r'), \text{ again, by associativity of :: and :::,} \\
= l_1 :: \text{acc}, \text{ by our naming of acc,} \\
\rightarrow^* l', \text{ by Induction Hypothesis on } l_1 \text{ and acc,}
\]

We showed that both functions ultimately evaluate to a common list expression.

This completes the proof. \( \square \)