Climbing Trees and Visiting Neighbors

Christopher R. Genovese

Department of Statistics & Data Science

22 Oct 2024
Session #15

Plan

Discussion of Mini-Exercise

Plan

Discussion of Mini-Exercise

Comments on Style

Plan

Discussion of Mini-Exercise

Comments on Style

Finish Activity: Making Change

Plan

Discussion of Mini-Exercise

Comments on Style

Finish Activity: Making Change

Climbing Trees

Plan

Discussion of Mini-Exercise

Comments on Style

Finish Activity: Making Change

Climbing Trees

Graphs

Announcements

- Hope you had a nice break!
- `fpc` code, python 3.12
- **Reading**:
  - `DFS.pdf` under `Readings` in the `documents` repo.
- **Homework**: `migit-2` due Tuesday 05 Nov, additional assignments on the way.

Plan

Discussion of Mini-Exercise

Comments on Style

Finish Activity: Making Change

Climbing Trees

Graphs

Mini-Exercise

Consider function composition: `(.)` (or if you prefer `compose`)

```
(.) : (b -> c) -> (a -> b) -> (a -> c)
(.) . (.) :
(.) . (.) . (.) :
(.) . (.) . (.) . (.) :
```

Remember that all the type variables are general and can match any type.

Mini-Exercise

Consider function composition: `(.)` (or if you prefer `compose`)

```
(.) : (b -> c) -> (a -> b) -> (a -> c)
(.) . (.) :
(.) . (.) . (.) :
(.) . (.) . (.) . (.) :
```

Remember that all the type variables are general and can match any type.

Let `f : u -> v`, then `(.) f` (or equivalently `(f . __)`) has type `(a2 -> u) -> (a2 -> v)`.

Then `(.) . (.) f` matches `b` to `a2 -> u` and `c` to `a2 -> v` giving type

`(a1 -> a2 -> u) -> (a1 -> a2 -> v)`

Mini-Exercise

Continuing in this vein, we have

```
(.) : (b -> c) -> (a -> b) -> (a -> c)
(.) . (.) : (b -> c) -> (a1 -> a2 -> b) -> a1 -> a2 -> c
(.) . (.) . (.) : (b -> c) -> (a1 -> a2 -> a3 -> b) -> a1 -> a2 -> a3 -> c
(.) . (.) . (.) . (.) : (b -> c) -> (a1 -> a2 -> a3 -> a4 -> b)
                                 -> a1 -> a2 -> a3 -> a4 -> c
```

Put in some extraneous parentheses to make it clearer

```
(.) : (b -> c) -> (a -> b) -> (a -> c)
(.) . (.) : (b -> c) -> (a1 -> a2 -> b) -> (a1 -> a2 -> c)
(.) . (.) . (.) : (b -> c) -> (a1 -> a2 -> a3 -> b)
                           -> (a1 -> a2 -> a3 -> c)
(.) . (.) . (.) . (.) : (b -> c) -> (a1 -> a2 -> a3 -> a4 -> b)
                                 -> (a1 -> a2 -> a3 -> a4 -> c)
```

Mini-Exercise: Polymorphic Combinators

This is just a special case of `map`. (With which Functor?)

```
map :  Functor f => (a -> b) -> f a -> f b
map . map : Functor f, g =>
map . map . map : Functor f, g, h =>
map . map . map . map : Functor f, g, h, k =>
```

Again, all the type variables are general and can match any type, and each use of map can be with a *different functor*.

Mini-Exercise: Polymorphic Combinators

This is just a special case of `map`. (With which Functor?)

```
map :  Functor f => (a -> b) -> f a -> f b
map . map : Functor f, g =>
map . map . map : Functor f, g, h =>
map . map . map . map : Functor f, g, h, k =>
```

Again, all the type variables are general and can match any type, and each use of map can be with a *different functor*.

Let `z : u -> v`, then `map z` has type `f u -> f v`, and `map . map z` has type `g (f u) -> g (f v)`.

Mini-Exercise: Polymorphic Combinators

Continuing in this vein, we have

```
map :  Functor f =>
       (a -> b) -> f a -> f b
map . map : Functor f, g =>
       (a -> b) -> g (f a) -> g (f b)
map . map . map : Functor f, g, h =>
       (a -> b) -> h (g (f a)) -> h (g (f b))
map . map . map . map : Functor f, g, h, k =>
       (a -> b) -> k (h (g (f a))) -> k (h (g (f b)))
```

and so on.

Mini-Exercise: Polymorphic Combinators

This same pattern will prove useful shortly. We will consider a function `traverse` with type

```
traverse :  (a -> f b) -> t a -> f (t b)
traverse . traverse :  (a -> f b) -> t2 (t1 a) -> f (t2 (t1 b))
```

and so on.

Plan

Discussion of Mini-Exercise

Comments on Style

Finish Activity: Making Change

Climbing Trees

Graphs

Comments on Style

- Formatting
- Naming
- Comments

- Organization and Layering

- Separate Calculations and Actions (and Data)

- Parse, don't Validate
  - Example: Boolean/None blindness and email addresses
  - `head` and partial functions
  - Strengthening inputs rather than weakening outputs (ex: NonEmptyList)
  - Smart constructors
  - Controlling the boundaries

- Make invalid states unrepresentable

Plan

Discussion of Mini-Exercise

Comments on Style

Finish Activity: Making Change

Climbing Trees

Graphs

Making Change

Problem: Given a monetary value `amount` in cents and a set of coin denominations `coins`, also in cents, generate *all* the distinct ways to make change for the former in terms of the latter.

Write this as a function `change_for` (or `change-for` or `changeFor` depending on your language conventions). Also write a function `best_change_for`, with the same arguments, that gives the way to make change with the smallest number of coins required.

The function `change_for` should return a list of vectors, where each vector contains the coins used to make change. The returned list should contain no repeats (accounting for order). The function `best_change_for` should return any single way of making change (a vector of denominations) of minimal size.

For example:

```
change_for(17, [1, 5, 10, 25])

# should return these results:
[[10, 5, 1, 1],
 [5, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1],
 [1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1],
 [10, 1, 1, 1, 1, 1, 1, 1],
 [5, 5, 1, 1, 1, 1, 1, 1, 1],
 [5, 5, 5, 1, 1]]
```

Making Change: Discussion Questions

Don't write any code for now. Discuss these questions **in order** with your group:

1. What are the "base cases", i.e. trivially small problems that we can solve directly?
2. How can I reduce this problem into simpler/smaller problems?
3. How are the subproblem results combined?
4. How can you ensure that there are no repeats in the generated list?

Generate some test ideas with a partner!

With a partner, sketch out the pseudocode for a recursive function that solves the problem.

Plan

Discussion of Mini-Exercise

Comments on Style

Finish Activity: Making Change

Climbing Trees

Graphs

Trees and Recursion

Trees represent hierarchical relationships among entities.

They are a special type of **graphs** and so are typically depicted with nodes and edges.

We will talk about graphs, but starting with trees lets us get the main ideas down clearly.

A tree with n nodes is a connected graph with exactly $n - 1$ edges. It is acyclic and there is a unique path between any two nodes.

The *Divide and Conquer* strategy that we saw last time (i.e., recursion) is intimately connected to trees.

Trees arise in countless other problems, including backtracking algorithms (which we will see), various graph algorithms, geometric methods, and high-dimensional data analysis.

Let's jump in with a simple exercise.

Mini-Activity

In the Zippers assignment, we looked at "vector zippers" that navigated trees described by nested lists.

When testing the trees produced in R, it is helpful to have a way to check that a tree is as expected. Comparing R lists *in toto* is not automatic, but comparing strings is.

Goal: Take a nested list (as in the vector zipper case) and convert it to a string. Call this function `to_string`

The key requirement is that the conversion function is a bijection: two trees are equal if and only if their strings are equal.

While this is especially relevant in R, you can do it in Python or any language.

Hint: Traverse the list printing nodes, but when you see a subtree, apply `to_string` to handle it. Choose a "down" character (e.g., `(`) and an "up" character (e.g., `)`) and a separate character (e.g., `,`) that are emitted before and after subtrees and between children, respectively.

Seeing the Forest for the Trees

There are many, many flavors of tree, but all share a similar structure. Consider:

```
type BinaryTree a = Tip | Branch (BinaryTree a) a (BinaryTree a)

type RoseTree a = Node a (List (RoseTree a))

-- Heterogenous version of Rose Tree, different types on Leaf and Branch
type HTree b l = Leaf l | Branch b (NonEmptyList HTree b l)

type Trie k m a = Trie { data : Maybe a
                       , children : Map k (Trie k m a)
                       , annotation : m      -- m will be a Monoid
                       }

-- One way to think about unrooted trees; there are others.
-- Fin n is the type of integers 1..n. OrderedPairs a is like Pair a a
-- but represents pairs (x, y) where x < y.

type UnrootedTree a = forall n.
         Pair(Injection Fin(n - 1) (OrderedPairs (Fin n)), Fin n -> a)
```

Group work: Building Trees

Using Binary or Rose trees as you prefer, write a quick working implementation of

```
-- Binary case
unfold : (b -> (a, Maybe b, Maybe b)) -> b -> BinaryTree a

-- Rose case
unfold : (b -> (a, List b)) -> b -> RoseTree a
```

The `b` type represents a **seed**. We start with a seed and generate a partial subtree that may include additional seeds from which we generate new subtrees at that position.

Example in the Binary case: `complete_btree(depth)` creates a complete `BinaryTree Int` of the given depth.

Walking and Mapping Trees

A **traversal** of a tree is an organized visit to every node in the tree exactly once.

- Preorder: The root of a subtree is visited before its children.
- Postorder: The children of a subtree are visited before the root.
- Inorder (Binary case): Root-Left-Right order

We often do an ***effectful*** computation at each node.

How much flexibility do we have in the ordering of the nodes? How can you pass information with you as you move along the tree? (Consider how you might number the leaves of a tree by depth.)

Mapping a tree converts a tree to another tree of the same shape. In other words, trees are ***functors***.

Choose one of these and do a quick implementation, using the trees you built.

Variant: Traverse a nested data structure composed of e.g., vectors, dictionaries, tuples, Iterable objects, et cetera.

Printing Trees

Write a function `to_string` that converts a tree (of whichever form you like) to a printable string that represents the structure of the tree.

Demo

Consider a divide and conquer approach.

- What information do you need to maintain at any point?
- How does recursion figure in to this? How does the recursion reflect the structure of the tree?
- Do you see a Monoid structure here? (You don't have to use it today.)

Class Work: Searching with Tries

Consider a simple form of the "Trie," a special type of tree that can be used for efficiently associating values with a set of strings.

We consider a simplified form here

```
type Trie a = Trie { data : Maybe a
                   , children : Map Char (Trie a)
                   }
```

where `Map k v` is an associative map (e.g., dictionary) from key type `k` to value type `v`.

A string like "foobar" starts at the root node (associated with the empty string) and moves to the child of a node based on successive characters.

We have

```
empty  : Trie a
insert : String -> Trie a -> Trie a
lookup : String -> Trie a -> Maybe a
```

Let's sketch out a trie implementation together.

Plan

Discussion of Mini-Exercise

Comments on Style

Finish Activity: Making Change

Climbing Trees

Graphs

Graphs

Next Time

The End