Programming Languages and Techniques (CIS120)

Lecture 5
September 10, 2018

Datatypes and Trees
Announcements

• Homework 1 due tomorrow at 11:59pm!
• Homework 2 available soon
  – due Tuesday, September 18th
• Read Chapters 5 – 7
Recap: Lists, Recursion, & Tuples
Recursive functions over lists follow a general pattern:

```ocaml
let rec length (l : string list) : int =
begin match l with
| [] -> 0
| (x :: rest) -> 1 + length rest
end
```

```ocaml
let rec contains (l:string list) (s:string) : bool =
begin match l with
| [] -> false
| (x :: rest) -> s = x || contains rest s
end
```
Structural Recursion Over Lists

Structural recursion builds an answer from smaller components:

```
let rec f (l : ... list) ... : ... =
begin match l with
| []  -> ...
| ( hd :: rest ) -> ... f rest ...
end
```

The branch for `[]` calculates the value `(f [ ] )` directly.
  – this is the `base case` of the recursion

The branch for `hd::rest` calculates
  `(f(hd::rest))` given `hd` and `(f rest ).`
  – this is the `inductive case` of the recursion
Given the definition below, which of the following is correct?

\[
\begin{align*}
(f \ [1;2] \ [3;4]) \\
= \ [3;4;1;2] \\
(f \ [1;2] \ [3;4]) \\
= \ [1;2;3;4] \\
(f \ [1;2] \ [3;4]) \\
= \ [4;3;2;1] \\
(f \ [1;2] \ [3;4]) \\
= \ [1;3;2;4] \\
\text{none of the above}
\end{align*}
\]
Given the definition below, which of the following is correct?

```plaintext
let rec f (l1:int list) (l2:int list) : int list =
begin
match l1 with
| [] -> l2
| x::xs -> x :: (f xs l2)
end
```

1. \((f \[1;2\] \[3;4\]) = \[3;4;1;2\]\)
2. \((f \[1;2\] \[3;4\]) = \[1;2;3;4\]\)
3. \((f \[1;2\] \[3;4\]) = \[4;3;2;1\]\)
4. \((f \[1;2\] \[3;4\]) = \[1;3;2;4\]\)
5. none of the above

Answer: 2
let rec f (l1:int list) (l2:int list) : int list =
    begin match l1 with
    | []  -> l2
    | x::xs -> x :: f xs l2
    end

f [1; 2] [3;4]
⇒ 1 :: (f [2] [3;4])
⇒ 1 :: 2 :: (f [] [3;4])
⇒ 1 :: 2 :: [3;4]
=  [1;2;3;4]
Design Pattern for Recursion

1. **Understand the problem**
   What are the relevant concepts and how do they relate?

2. **Formalize the interface**
   How should the program interact with its environment?

3. **Write test cases**
   - If the main input to the program is an immutable list, make sure the tests cover both empty and non-empty cases.

4. **Implement the required behavior**
   - If the main input to the program is an immutable list, look for a recursive solution...
     - Is there a direct solution for the empty list?
     - Suppose someone has given us a partial solution that works for lists up to a certain size. Can we use it to build a better solution that works for lists that are one element larger?
OCaml provides two ways of packaging multiple values together into a single compound value:

• **Lists:**
  – *arbitrary-length* sequence of values of a single, *fixed type*
  – example: a list of email addresses

• **Tuples:**
  – *fixed-length* sequence of values of *arbitrary types*
  – example: tuple of name, phone #, and email
What is the type of this expression?

- `int`
- `int list`
- `int list list`
- `(int * int list) list`
- `int * (int list) * (int list list)`
- `(int * int * int) list`
- `none (expression is ill typed)`
What is the type of this expression?

(1, [1], [[1]])

1. int
2. int list
3. int list list
4. (int * int list) list
5. int * (int list) * (int list list)
6. (int * int * int) list
7. none  (expression is ill typed)

Answer: 5
What is the type of this expression?

```
[ (1,true); (0, false) ]
```

- `int * bool`
- `int list * bool list`
- `(int * bool) list`
- None (expression is ill typed)
What is the type of this expression?

\[ (1, \text{true}); (0, \text{false}) \]

1. \text{int} * \text{bool}
2. \text{int list} * \text{bool list}
3. (\text{int} * \text{bool}) \text{ list}
4. (\text{int} * \text{bool}) \text{ list list}
5. \text{none}  \quad (\text{expression is ill typed})

Answer: 3
Datatypes and Trees
Building Datatypes

• Programming languages provide a variety of ways of creating and manipulating structured data

• We have already seen:
  – *primitive datatypes* (int, string, bool, ... )
  – *lists* (int list, string list, string list list, ... )
  – *tuples* (int * int, int * string, ...)

• Rest of Today:
  – *user-defined* datatypes
  – type abbreviations
HW 2 Case Study: Evolutionary Trees

• Problem: reconstruct evolutionary trees* from DNA data.
  – What are the relevant abstractions?
  – How can we use the language features to define them?
  – How do the abstractions help shape the program?

*Interested? Check out this suggested reading:
Dawkins: The Ancestor's Tale: A Pilgrimage to the Dawn of Evolution
DNA Computing Abstractions

• **Nucleotide**
  – Adenine (A), Guanine (G), Thymine (T), or Cytosine (C)

• **Helix**
  – A sequence of nucleotides: e.g. AGTCCGATTACAGAGA...
  – Genetic code for a particular species (human, gorilla, etc)

• **Phylogenetic tree**
  – Binary tree with helices (species) at the nodes and leaves

![Phylogenetic tree diagram]
OCaml lets programmers define new datatypes.

**Example:**

```ocaml
type day =
| Sunday
| Monday
| Tuesday
| Wednesday
| Thursday
| Friday
| Saturday
```

- The constructors *are* the values of the datatype
  - e.g. `A` is a nucleotide and `[A; G; C]` is a nucleotide list.
Pattern Matching Simple Datatypes

• Datatype values can be analyzed by pattern matching:

```plaintext
let string_of_n (n:nucleotide) : string =

begin
  match n with
  | A -> "adenine"
  | C -> "cytosine"
  | G -> "guanine"
  | T -> "thymine"
end
```

• There is one case per constructor
  – you will get a warning if you leave out a case or list one twice

• As with lists, the pattern syntax follows that of the datatype values (i.e. the constructors)
A Point About Abstraction

• *We could* represent data like this by using integers:
  – Sunday = 0, Monday = 1, Tuesday = 2, etc.

• But:
  – Integers support different operations than days do: *Wednesday - Monday = Tuesday* (?!)
  – There are *more* integers than days (What day is 17?)

• Confusing integers with days can lead to bugs
  – Many *scripting* languages (PHP, Javascript, Perl, Python,...) violate such abstractions (true == 1 == “1”), leading to pain and misery...

Most modern languages (Java, C#, C++, OCaml,...) provide user-defined types for this reason.
OCaml also lets us *name* types without making new abstractions:

- **type** helix = nucleotide list
- **type** codon = nucleotide * nucleotide * nucleotide

i.e. a **codon** is the same thing a triple of *nucleotides*

```ocaml
let x : codon = (A, C, C)
```

Makes code easier to read & write.
Datatype constructors can also carry values

```
type measurement =
  | Missing
  | NucCount  of nucleotide * int
  | CodonCount of codon * int
```

Values of type `measurement` include:
- Missing
- NucCount(A, 3)
- CodonCount((A,G,T), 17)
Pattern Matching Datatypes

- Pattern matching notation combines syntax of tuples and simple datatype constructors:

```ocaml
let get_count (m:measurement) : int = 
  begin match m with 
  | Missing         -> 0
  | NucCount(_, n)  -> n
  | CodonCount(_, n) -> n
  end
```

- Datatype patterns bind variables (e.g. ‘n’) just like lists and tuples
What is the type of this expression?

(type nucleotide = | A | C | G | T
type helix = nucleotide list

What is the type of this expression?

(A, "A")

type nucleotide list

nucleotide *

helix

string *

string

none (expression is ill typed)
What is the type of this expression?

\[(A, \ "A")\]

1. nucleotide
2. nucleotide list
3. helix
4. nucleotide * string
5. string * string
6. none  \(\text{(expression is ill typed)}\)

Answer: 4
Recursive User-defined Datatypes

- Datatypes can mention themselves!

```plaintext
type tree =
  | Leaf of helix
  | Node of tree * helix * tree
```

- Recursive datatypes can be taken apart by pattern matching (and recursive functions).
Example values of type `tree`

```haskell
let t1 = Leaf [A;G]
let t2 = Node (Leaf [G], [A;T], Leaf [A])
let t3 = Node (Leaf [T], [T;T],
              Node (Leaf [G;C], [G], Leaf []))
```
Trees are everywhere
Family trees
Organizational charts
Filesystem Directory Structure
Domain Name Hierarchy

cornell ... upenn

cisco...yahoo

nasa ... nsf

arpa ... navy

cis  seas  wharton ...
Game trees
Natural-Language Parse Trees

```
S
  /\  \\
 NP /  \ VP
  /\    /\
 D the  V cooks N
  /\   /\    /\  \\
 N chef V NP D N
      /\  /\  /\  \\
     the the the soup
```
Binary Trees

A particular form of tree-structured data
A binary tree is either empty, or a node with at most two children, both of which are also binary trees.

A leaf is a node whose children are both empty.
Trees need not be *balanced* they may be missing some branches.
Drawn Upside Down

root node

leaf node
Binary Trees in OCaml

```ocaml
type tree =
  | Empty
  | Node of tree * int * tree

let t : tree =
  Node (Node (Empty, 1, Empty),
        3,
        Node (Empty, 2,
               Node (Empty, 4, Empty)))
```

```
3

1

2

4
```
Representing trees

type tree =
| Empty
| Node of tree * int * tree

Node (Node (Empty, 0, Empty), 1, Node (Empty, 3, Empty))

Node (Empty, 0, Empty)

Empty
More on trees

see tree.ml
treeExamples.ml
Trees as Containers

- Like lists, trees aggregate ordered data
- As we did for lists, we can write a function to determine whether the data structure contains a particular element
Searching for Data in a Tree

let rec contains (t:tree) (n:int) : bool =
  begin match t with
  | Empty -> false
  | Node(lt,x,rt) -> x = n ||
     (contains lt n) || (contains rt n)
  end

• This function searches through the tree, looking for n
• In the worst case, it might have to traverse the entire tree
Search during (contains t 8)
Recursive Tree Traversals

Pre-Order
Root – Left – Right

In Order
Left – Root – Right

Post-Order
Left – Right – Root

(* Code for Pre-Order Traversal *)
let rec f (t:tree) : ... =
    begin match t with
        | Empty -> ...
        | Node(l, x, r) ->
            let root = ... x ... in (* process root *)
            let left = f l in (* recursively process left *)
            let right = f r in (* recursively process right *)
            combine root left right
    end

The traversals vary the order in which these are computed...