Trees and Structural Induction

15-150 Spring 2018
Lecture 5
Dilsun Kaynar
So far in the course

• Basic ML programming
  – Write well-typed functions with recursion
  – Aggregate data structures such as tuples and lists

• Specifications

• Proofs
  – Reasoning with evaluation and equivalence
  – Simple and strong induction
  – Structural induction
Today

• How to define your own types (recursive/non-recursive) using datatype declarations
• Represent and compute with trees in ML
• More specifications and proofs
Declaring your own types

DATATYPES
Datatype declarations

- Introduces a **new** type that is distinct from all other types

```plaintext
datatype day = Sun | Mon | Tue | Wed | Thurs | Fri | Sat
```
Datatype declarations

• Introduces a new type that is distinct from all other types

```
datatype day = Sun | Mon | Tue | Wed | Thurs | Fri | Sat
```

- type constructor
- value constructors
Exercise

datatype day = Sun | Mon | Tue | Wed | Thurs | Fri | Sat

(* previous : day -> day
  REQUIRES: true
  ENSURES: previous(d) returns the value representing the day before d
*)
Clausal function declaration

datatype day = Sun | Mon | Tue | Wed | Thurs | Fri | Sat

(* previous : day -> day
   REQUIRES: true
   ENSURES: previous (d) returns the value representing the day before d
*)

fun previous(d) = case d of
               | Sun => Sat
               | Mon => Sun
               | Tue => Mon
               | Wed => Tue
               | Thurs => Wed
               | Fri => Thurs
               | Sat => Fri

constructors in patterns
Clausal function declaration

datatype day = Sun | Mon | Tue | Wed | Thurs | Fri | Sat

(* previous : day -> day
   REQUIRES: true
   ENSURES: previous(d) returns the value representing the day before d
*)

fun previous Sun = Sat
  | previous Mon = Sun
  | previous Tue = Mon
  | previous Wed = Tue
  | previous Thurs = Wed
  | previous Fri = Thurs
  | previous Sat = Fri

constructors in patterns
Proof by cases

- Datatype declaration for day is *not* recursive.
- Suggests a simple form of structural induction with only bases cases.
Representing trees with datatypes

BINARY TREES
Recursive datatype declaration

datatype tree = Empty | Node of tree * int * tree
Recursive datatype declaration

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

- constant constructor
- constructor that takes an argument
Recursive datatype declaration

```
datatype tree = Empty | Node of tree * int * tree
```

A tree is
- either Empty
- or Node(l, x, r) where l is a tree, x is an int and r is a tree
- and that’s it.
(* height : tree -> int
   REQUIRES: true
   ENSURES: height returns the height of t
            with height (Empty) being 0
   *)
fun height (Empty : tree) : int = 0 | height (Node(t1, x, t2)) = 1 + Int.max (height(t1), height(t2))
height is total
Theorem: For all values $t: \text{tree}$, $\text{height}(t)$ returns a value.
**Theorem:** For all tree $t:\text{tree}$, $\text{height}(t)$ returns a value.

**Proof:** By structural induction on $t$. 
Recursive datatype declaration

```plaintext
datatype tree = Empty | Node of tree * int * tree
```

- base case
- recursive cases
Principle of Induction for trees

**Theorem:** For all $T$: tree, $P(T)$.

**Proof** by structural induction on $T$.

**Base case:** $T = \text{Empty}$

Show $P(\text{Empty})$

**Inductive step:** $T = \text{Node} \ (t_1,x,t_2)$

I.H. $P(t_1)$ and $P(t_2)$

Show $P(\text{Node} \ (t_1,x,t_2))$
ANOTHER KIND OF TREE
A new datatype for trees

datatype tree = Leaf of int | Node of tree * tree
datatype tree = Leaf of int | Node of tree * * tree

(* flatten : tree -> int list
   REQUIRES: true
   ENSURES: flatten(t) returns a list of the leaf values as they are encountered in the inorder traversal of t
*)
**flatten**

```plaintext
datatype tree = Leaf of int | Node of tree * tree

(* flatten : tree -> int list
   REQUIRES: true
   ENSURES: flatten(t) returns a list of the leaf
            values as they are encountered in the
            inorder traversal of t
*)

fun flatten (Leaf(x) : tree) : int list = [x]
  | flatten (Node(t1, t2)) = flatten (t1) @ flatten (t2)
```
flatten with accumulator

(* flatten2 : tree * int list-> int list
REQUIRES: true
ENSURES: ...
*)
flatten with accumulator

(* flatten2 : tree * int list-> int list
  REQUIRES: true
  ENSURES: flatten2(t, acc) == flatten(t) @ acc *)
flatten with accumulator

(* flatten2 : tree * int list -> int list
  REQUIRES: true
  ENSURES: flatten2(t, acc) == flatten(t) @ acc *)

fun flatten2 (Leaf(x), acc) = x :: acc
| flatten2 (Node(t1,t2), acc) =
    flatten2(t1,(flatten2(t2,acc)))
flatten with accumulator

(* flatten2 : tree * int list -> int list  
  REQUIRES: true  
  ENSURES: flatten2(t, acc) == flatten(t) @ acc *)

fun flatten2 (Leaf(x), acc) = x ::: acc  
| flatten2 (Node(t1,t2), acc) = 
  flatten2(t1,(flatten2(t2,acc)))

fun flatten' (t: tree) : int list = 
  flatten2(t,[])
Correctness of flatten2

**Theorem:** For all values $T : \text{tree}$ and $acc : \text{int list}$, $\text{flatten2}(t,acc) \equiv \text{flatten}(t)@acc$.

**Code:**

```latex
\text{fun flatten2 (Leaf(x), acc) = x :: acc}
\text{ | flatten2 (Node(t1,t2), acc) =}
\text{ \quad flatten2(t1,(flatten2(t2,acc)))}
```

```latex
\text{fun flatten (Leaf(x) : tree) : int list = [x]}
\text{ | flatten (Node(t1, t2)) = flatten (t1) \@ flatten (t2)}
```
Auxiliary results

Lemma 1: For all values $x : \text{int}$ and $\text{acc}: \text{int list}$,
\[ x :: \text{acc} \cong [x] @ \text{acc}. \]

Lemma 2: The function flatten is total.

Lemma 3: The function @ is total and associative.
Proof

By structural induction on \( T \).

**Base case:** \( T = \text{Leaf}(x) \)

NTS. For all \( acc : \text{int list} \),
\[
\text{flatten2}(\text{Leaf}(x), acc) \cong \text{flatten}((\text{Leaf}(x)) @ acc).
\]

Showing.
\[
\text{flatten2}(\text{Leaf}(x), acc) \cong x :: acc \quad [1^{\text{st}} \text{clause of flatten2}]
\]
\[
\cong [x] @ acc \quad [\text{Lemma 1}]
\]
\[
\cong \text{flatten}(\text{Leaf}(x)) @ acc
\]
\[
[1^{\text{st}} \text{clause of flatten}]
\]
Inductive step

By structural induction on $T$.

**Inductive step**: $T = \text{Node}(t_1, t_2)$

I.H. For all $acc_1: \text{int list}$,

\[
\text{flatten2}(t_1, acc_1) \equiv \text{flatten}(t_1) \ @ acc_1
\]

For all $acc_2: \text{int list}$

\[
\text{flatten2}(t_2, acc_2) \equiv \text{flatten}(t_2) \ @ acc_2
\]

NTS. For all $acc: \text{int list}$,

\[
\text{flatten2}(\text{Node}(t_1, t_2), acc) \equiv \text{flatten}(\text{Node}(t_1, t_2)) \ @ acc
\]
Inductive step

Showing.

\[
\text{flatten2}(\text{Node}(t_1,t_2), \text{acc})
\]

\[= \text{flatten2}(t_1, \text{flatten2}(t_2, \text{acc})) \quad [\text{2nd clause of flatten2}] \]

\[= \text{flatten2}(t_1, \text{flatten}(t_2) \mathbin{@} \text{acc}) \quad [\text{I.H. for } t_2 \text{ and ref. trans.}] \]

\[= \text{flatten}(t_1) \mathbin{@} ((\text{flatten}(t_2) \mathbin{@} \text{acc})) \quad [\text{I.H. for } t_1, \text{ Lemmas 2 and 3}] \]

\[= (\text{flatten}(t_1) \mathbin{@} \text{flatten}(t_2)) \mathbin{@} \text{acc} \quad [\text{Lemma 3}] \]

\[= (\text{flatten}(\text{Node}(t_1,t_2)) \mathbin{@} \text{acc}) \quad [\text{2nd clause of flatten}] \]
Operator/operand tree

```haskell
datatype optree = Op of optree * (int * int -> int) * optree
    | Val of int

(* eval : optree -> int
  REQUIRES: all functions in T are total
  ENSURES: eval(T) reduces to the integer value that is the result of the computation
          described by T (assuming post-order traversal)
*)

fun eval(Val x : optree ) : int = x
    |eval(Op(l,f,r)) = f(eval l, eval r)
```