Inductive Datatypes

COS 326

Speaker: Andrew Appel
Princeton University

Inductive data types

- We can use data types to define inductive data
- A binary tree is:
  - a Leaf containing no data
  - a Node containing a key, a value, a left subtree and a right subtree

```
type key = string
type value = int

type tree =
  Leaf
| Node of key * value * tree * tree
```

Inductive data types

```
type key = int
type value = string

type tree =
  Leaf
| Node of key * value * tree * tree
```

```
let rec insert (t:tree) (k:key) (v:value) : tree =
```

Inductive data types

```
type key = int
type value = string

type tree =
  Leaf
| Node of key * value * tree * tree
```

```
let rec insert (t:tree) (k:key) (v:value) : tree =
  match t with
  | Leaf ->
  | Node (k', v', left, right) ->
```

Again, the type definition specifies the cases you must consider

Inductive data types

```
type key = int
type value = string

type tree =
  Leaf
| Node of key * value * tree * tree
```

```
let rec insert (t:tree) (k:key) (v:value) : tree =
  match t with
  | Leaf -> Node (k, v, Leaf, Leaf)
  | Node (k', v', left, right) ->
```

Inductive data types

```
type key = int
type value = string

type tree =
  Leaf
| Node of key * value * tree * tree
```

```
let rec insert (t:tree) (k:key) (v:value) : tree =
  match t with
   | Leaf -> Node (k, v, Leaf, Leaf)
   | Node (k', v', left, right) ->
       if k < k' then
         Node (k', v', insert left k v, right)
       else if k > k' then
         Node (k', v', left, insert right k v)
       else
         Node (k, v, left, right)
```

Inductive data types

```
type key = int
type value = string

type tree =
  Leaf
| Node of key * value * tree * tree
```

```
let rec insert (t:tree) (k:key) (v:value) : tree =
  match t with
  | Leaf -> Node (k, v, Leaf, Leaf)
  | Node (k', v', left, right) ->
      if k < k' then
        Node (k', v', insert left k v, right)
      else if k > k' then
        Node (k', v', left, insert right k v)
      else
        Node (k, v, left, right)
```

Inductive data types

```
type key = int
type value = string

type tree =
  Leaf
| Node of key * value * tree * tree
```

```
let rec insert (t:tree) (k:key) (v:value) : tree =
  match t with
  | Leaf -> Node (k, v, Leaf, Leaf)
  | Node (k', v', left, right) ->
      if k < k' then
        Node (k', v', insert left k v, right)
      else if k > k' then
        Node (k', v', left, insert right k v)
      else
        Node (k, v, left, right)
```

Inductive data types: Another Example

- Recall, we used the type "int" to represent natural numbers
  - but that was kind of broken: it also contained negative numbers
  - we had to use a dynamic test to guard entry to a function:

```
let double (n : int) : int =
  if n < 0 then
    raise (Failure "negative input!")
  else
    double_nat n
```

  - it would be nice if there was a way to define the natural numbers exactly, and use OCaml's type system to guarantee no client ever attempts to double a negative number

Inductive data types

- Recall, a natural number n is either:
  - zero, or
  - m + 1
- We use a data type to represent this definition exactly:

Inductive data types

- Recall, a natural number n is either:
  - zero, or
  - m + 1
- We use a data type to represent this definition exactly:

```
type nat = Zero | Succ of nat
```

Inductive data types

- Recall, a natural number n is either:
  - zero, or
  - m + 1
- We use a data type to represent this definition exactly:

```
type nat = Zero | Succ of nat

let rec nat_to_int (n : nat) : int =
 match n with
   Zero -> 0
 | Succ n -> 1 + nat_to_int n
```

Inductive data types

- Recall, a natural number n is either:
  - zero, or
  - m + 1
- We use a data type to represent this definition exactly:

```
type nat = Zero | Succ of nat

let rec nat_to_int (n : nat) : int =
 match n with
   Zero -> 0
 | Succ n -> 1 + nat_to_int n

let rec double_nat (n : nat) : nat =
  match n with
   | Zero -> Zero
   | Succ m -> Succ (Succ(double_nat m))
```

Lists!

- Recall, a list is either:
  - nil, or
  - the cons of a *head* value with a *tail* list

- We use a data type to represent this definition exactly:

```
type 'a list =  []  |  :: of 'a * 'a list
```

Summary

- OCaml data types: a powerful mechanism for defining complex data structures:
  - They are precise
    - contain exactly the elements you want, not more elements
  - They are general
    - recursive, non-recursive (mutually recursive and polymorphic)
  - The type checker helps you detect errors
    - missing cases in your functions