Thinking Inductively

Speaker: David Walker

COS 326

Princeton University

Inductive Programming

An *inductive data type* T is a data type defined by:

- base cases
  - don’t refer to T
- inductive cases
  - build new data of type T from pre-existing data of type T
  - the pre-existing data is guaranteed to be *smaller* than the new values

Inductive Programming

An *inductive data type* T is a data type defined by:

- base cases
    - don’t refer to T
- inductive cases
    - build new data of type T from pre-existing data of type T
    - the pre-existing data is guaranteed to be *smaller* than the new values

Example: a tree

- base case:
    - the leaf of the tree
- inductive case:
    - the internal nodes of the tree
    - the left- and right- subtrees are the “smaller” data

Inductive Programming

To *program* a function over inductive data:

- think: what does my function need to do to be correct?
- solve the programming problem for the base cases
    - solve them one-by-one
- solve the programming problem for inductive cases:
    - solve them one-by-one
    - *assume your function already works correctly on smaller data values*
    - *call your function, when necessary, on smaller data values*

Inductive Proving

To *prove* a function over inductive data is correct:

- think: what is the correctness theorem for this function?
- prove the function correct for the base cases
  - prove them one-by-one
- prove the function correct for the inductive cases:
  - prove them one-by-one
  - *assume your function already works correctly on smaller data values*
  - *use this assumption to reason about calls over smaller data values*
  - this assumption is called the *induction hypothesis* of your proof

Inductive Proving

To *prove* a function over inductive data is correct:

- think: what is the correctness theorem for this function?
- prove the function correct for the base cases
  - prove them one-by-one
- prove the function correct for the inductive cases:
  - prove them one-by-one
  - *assume your function already works correctly on smaller data values*
  - *use this assumption to reason about calls over smaller data values*
  - this assumption is called the *induction hypothesis* of your proof

To be a good programmer, you also need to be a good prover.

LISTS: AN INDUCTIVE DATA TYPE

Lists are Inductive Data

In OCaml, a list value is:

- [ ] (the empty list)
- v :: vs (a value v followed by a shorter list of values vs)

Base Case

Inductive Case

Lists are Inductive Data

In OCaml, a list value is:

[ ]   (the empty list)

v :: vs   (a value v followed by a shorter list of values vs)

An example:

- 2 :: 3 :: 5 :: [ ] has type int list
- is the same as: 2 :: (3 :: (5 :: [ ]))
- "::" is called "cons"

An alternative syntax (“syntactic sugar” for lists):

- [2; 3; 5]
- But this is just a shorthand for 2 :: 3 :: 5 :: []. If you ever get confused fall back on the 2 basic *constructors*, :: and []

Typing Lists

Typing rules for lists:

(1) [ ] may have any list type, t list

(2) if e1 : t and e2 : t list
then (e1 :: e2) : t list

Typing Lists

Typing rules for lists:

(1) [ ] may have any list type t list

(2) if e1 : t and e2 : t list
then (e1 :: e2) : t list

More examples:

(1 + 2) :: (3 + 4) :: [ ] : ??

(2 :: [ ]) :: (5 :: 6 :: [ ]) :: [ ] : ??

[[2]; [5; 6]] : ??

Typing Lists

Typing rules for lists:

(1) [ ] may have any list type t list

(2) if e1 : t and e2 : t list
then (e1 :: e2) : t list

More examples:

(1 + 2) :: (3 + 4) :: [ ] : int list

(2 :: [ ]) :: (5 :: 6 :: [ ]) :: [ ] : int list list

[[2]; [5; 6]] : int list list

(Remember that the 3rd example is an abbreviation for the 2nd)

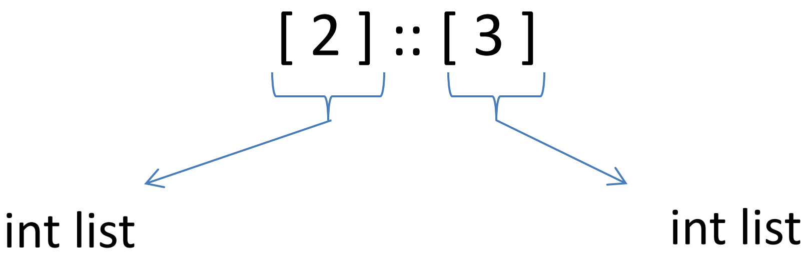
Another Example

What type does this have?

[2] :: [3]

Another Example

What type does this have?



[2] :: [3]

int list

int list

```
# [2] :: [3];;
Error: This expression has type int but an
         expression was expected of type
         int list
#
```

Another Example

What type does this have?

$$\underbrace{[\ 2\ ]}_{\text{int list}} :: \underbrace{[\ 3\ ]}_{\text{int list}}$$

Give me a simple fix that makes the expression type check?

Another Example

What type does this have?

$$\underbrace{[\ 2\ ]}_{\text{int list}} :: \underbrace{[\ 3\ ]}_{\text{int list}}$$

Give me a simple fix that makes the expression type check?

Either: 2 :: [3] : int list

Or: [2] :: [[3]] : int list list

Analyzing Lists

Just like options, there are two possibilities when deconstructing lists. Hence we use a match with two branches

```
(* return Some v, if v is the first list element;
   return None, if the list is empty *)

let head (xs : int list) : int option =
```

Analyzing Lists

Just like options, there are two possibilities when deconstructing lists. Hence we use a match with two branches

```
(* return Some v, if v is the first list element;
   return None, if the list is empty *)

let head (xs : int list) : int option =
  match xs with
  | [] ->
  | hd :: _ ->
```

we don't care about the contents of the tail of the list so we use the underscore

Analyzing Lists

Just like options, there are two possibilities when deconstructing lists. Hence we use a match with two branches

```
(* return Some v, if v is the first list element;
   return None, if the list is empty *)

let head (xs : int list) : int option =
  match xs with
  | [] -> None
  | hd :: _ -> Some hd
```

This function isn't recursive -- we only extracted a small , fixed amount of information from the list -- the first element

A more interesting example

```
(* Given a list of pairs of integers,
   produce the list of products of the pairs

   prods [(2,3); (4,7); (5,2)] == [6; 28; 10]
*)
```

A more interesting example

```
(* Given a list of pairs of integers,
   produce the list of products of the pairs

   prods [(2,3); (4,7); (5,2)] == [6; 28; 10]
*)

let rec prods (xs : (int * int) list) : int list =
```

A more interesting example

```
(* Given a list of pairs of integers,
   produce the list of products of the pairs

   prods [(2,3); (4,7); (5,2)] == [6; 28; 10]
*)

let rec prods (xs : (int * int) list) : int list =
  match xs with
  | [] ->
  | (x,y) :: tl ->
```

A more interesting example

```
(* Given a list of pairs of integers,
   produce the list of products of the pairs

   prods [(2,3); (4,7); (5,2)] == [6; 28; 10]
*)

let rec prods (xs : (int * int) list) : int list =
  match xs with
  | [] -> []
  | (x,y) :: tl ->
```

A more interesting example

```
(* Given a list of pairs of integers,
   produce the list of products of the pairs

   prods [(2,3); (4,7); (5,2)] == [6; 28; 10]
*)

let rec prods (xs : (int * int) list) : int list =
  match xs with
  | [] -> []
  | (x,y) :: tl -> ?? :: ??
```

the result type is int list, so we can speculate that we should create a list

A more interesting example

```
(* Given a list of pairs of integers,
   produce the list of products of the pairs

   prods [(2,3); (4,7); (5,2)] == [6; 28; 10]
*)

let rec prods (xs : (int * int) list) : int list =
  match xs with
  | [] -> []
  | (x,y) :: tl -> (x * y) :: ??
```

the first element is the product

A more interesting example

```
(* Given a list of pairs of integers,
   produce the list of products of the pairs

   prods [(2,3); (4,7); (5,2)] == [6; 28; 10]
*)

let rec prods (xs : (int * int) list) : int list =
  match xs with
  | [] -> []
  | (x,y) :: tl -> (x * y) :: ??
```

to complete the job, we must compute the products for the rest of the list

A more interesting example

```
(* Given a list of pairs of integers,
   produce the list of products of the pairs

   prods [(2,3); (4,7); (5,2)] == [6; 28; 10]
*)

let rec prods (xs : (int * int) list) : int list =
  match xs with
  | [] -> []
  | (x,y) :: tl -> (x * y) :: prods tl
```

Three Parts to Constructing a Function

(1) Think about how to *break down* the input into cases:

```
let rec prods (xs : (int*int) list) : int list =
  match xs with

  | [] ->  ...

  | (x,y) :: tl ->   ...
```

(2) *Assume* the recursive call on smaller data is correct.

(3) Use the result of the recursive call to *build* correct answer.

```
let rec prods (xs : (int*int) list) : int list =
   ...
  | (x,y) :: tl ->   ...  prods tl ...
```

Another example: zip

```
(* Given two lists of integers,
   return None if the lists are different lengths
   otherwise stitch the lists together to create
     Some of a list of pairs

   zip [2; 3] [4; 5] == Some [(2,4); (3,5)]
   zip [5; 3] [4] == None
   zip [4; 5; 6] [8; 9; 10; 11; 12] == None
*)
```

(Give it a try.)

Another example: zip

```
let rec zip (xs : int list) (ys : int list)
  : (int * int) list option =
```

Another example: zip

```
let rec zip (xs : int list) (ys : int list)
  : (int * int) list option =

  match (xs, ys) with
```

Another example: zip

```
let rec zip (xs : int list) (ys : int list)
  : (int * int) list option =

  match (xs, ys) with
  | ([], []) ->
  | ([], y::ys') ->
  | (x::xs', []) ->
  | (x::xs', y::ys') ->
```

Another example: zip

```
let rec zip (xs : int list) (ys : int list)
 : (int * int) list option =

 match (xs, ys) with
 | ([], []) -> Some []
 | ([], y::ys') ->
 | (x::xs', []) ->
 | (x::xs', y::ys') ->
```

Another example: zip

```
let rec zip (xs : int list) (ys : int list)
  : (int * int) list option =

  match (xs, ys) with
  | ([], []) -> Some []
  | ([], y::ys') -> None
  | (x::xs', []) -> None
  | (x::xs', y::ys') ->
```

Another example: zip

```
let rec zip (xs : int list) (ys : int list)
  : (int * int) list option =

  match (xs, ys) with
  | ([], []) -> Some []
  | ([], y::ys') -> None
  | (x::xs', []) -> None
  | (x::xs', y::ys') -> (x, y) :: zip xs' ys'
```

is this ok?

Another example: zip

```
let rec zip (xs : int list) (ys : int list)
  : (int * int) list option =

  match (xs, ys) with
  | ([], []) -> Some []
  | ([], y::ys') -> None
  | (x::xs', []) -> None
  | (x::xs', y::ys') -> (x, y) :: zip xs' ys'
```

No! zip returns a list option, not a list!
We need to match it and decide if it is Some or None.

Another example: zip

```
let rec zip (xs : int list) (ys : int list)
 : (int * int) list option =

 match (xs, ys) with
 | ([], []) -> Some []
 | ([], y::ys') -> None
 | (x::xs', []) -> None
 | (x::xs', y::ys') ->
   (match zip xs' ys' with
     None -> None
   | Some zs -> (x,y) :: zs)
```

Is this ok?

Another example: zip

```
let rec zip (xs : int list) (ys : int list)
  : (int * int) list option =

  match (xs, ys) with
  | ([], []) -> Some []
  | ([], y::ys') -> None
  | (x::xs', []) -> None
  | (x::xs', y::ys') ->
       (match zip xs' ys' with
          None -> None
        | Some zs -> Some ((x,y) :: zs))
```

Another example: zip

```
let rec zip (xs : int list) (ys : int list)
  : (int * int) list option =

  match (xs, ys) with
  | ([], []) -> Some []
  | (x::xs', y::ys') ->
      (match zip xs' ys' with
         None -> None
       | Some zs -> Some ((x,y) :: zs))
  | (_, _) -> None
```

Clean up.
Reorganize the cases.
Pattern matching proceeds in order.

A bad list example

```
let rec sum (xs : int list) : int =
  match xs with
  | hd::tl -> hd + sum tl
```

A bad list example

```
let rec sum (xs : int list) : int =
  match xs with
  | hd::tl -> hd + sum tl
```

```
#       Characters 39-78:
  ..match xs with
      hd :: tl -> hd + sum tl..
Warning 8: this pattern-matching is not exhaustive.
Here is an example of a value that is not matched: []
val sum : int list -> int = <fun>
```