Programming Languages and Techniques (CIS120)

Lecture 3

Value-Oriented Programming (continued)

Lists and Recursion

Review: Value-Oriented Programming

- OCaml promotes a **value-oriented** style:

  Most of what we write are *expressions* denoting *values*

- We can visualize running an OCaml program as a sequence of *calculation* or *simplification* steps that eventually lead to values

```
(300 + 12) * 60 + 17
↦ 312 * 60 + 17
↦ 18720 + 17
↦ 18737
```

(Top-level) Function Declarations

function name

parameter names

parameter types

```
let total_seconds (hours:int)
                  (minutes:int)
                  (seconds:int)
                : int =
  (hours * 60 + minutes) * 60 + seconds
```

result type

function body (an expression)

Function Calls

Once a function has been declared, it can be invoked by writing the function name followed by a sequence of arguments. The whole expression is a *function application*.

```
total_seconds 5 30 22
```

(Note: the sequence of arguments is *not* parenthesized.)

Calculating With Functions

- To calculate the value of a function application, first calculate values for its arguments and then *substitute* them for the parameters in the body of the function.

```
   total_seconds (2 + 3) 12 17
⟼ total_seconds 5 12 17
⟼ (5 * 60 + 12) * 60 + 17     substitute args in body
⟼ (300 + 12) * 60 + 17
⟼ 312 * 60 + 17
⟼ 18720 + 17
⟼ 18737
```

```
let total_seconds (hours:int)
                  (minutes:int)
                  (seconds:int)
                : int =
(hours * 60 + minutes) * 60 + seconds
```

What is the value computed for 'answer' in the following program? (0 .. 9)

```
let answer : int =
  let x = 3 in
  let f (y : int) = y + x in
  let x = 1 in
  f x
```

```
let answer : int =
  let f (y : int) = y + 3 in
  let x = 1 in
  f x
```

```
let answer : int =
  let f (y : int) = y + 3 in
  f 1
```

```
let answer : int =
  1 + 3
```

```
let answer : int =
  4
```

Lists

A Value-Oriented Approach
to Sequential Data

What is a list?

A list value is either:

`[]`	the *empty* list, sometimes called *nil*
or	
`v :: tail`	a head value v, followed by a list of the remaining elements, the *tail*

- Here, the '`::`' infix operator *constructs* a new list from a head element and a shorter list.
  - This operator is pronounced "cons" (short for "construct")
- Importantly, *there are no other kinds of lists*.
- Lists are an example of an *inductive datatype*.

Example Lists

To build a list, cons together elements, ending with the empty list:

`1::2::3::4::[]`	a list of (four) ints
`“abc”::”xyz”::[]`	a list of (two) strings
`(false::[])::(true::[])::[]`	a list of lists that each contain booleans
`[]`	the empty list

Explicitly parenthesized

'`::`' is an binary operator like + or ^; it takes an element and a *list* of elements as inputs:

`1::(2::(3::(4::[])))`	a list of four numbers
`“abc”::(”xyz”::[])`	a list of two strings
`true::[]`	a list of one boolean
`[]`	the empty list

*Unlike + and ^, cons is right associative. a :: b :: c means a :: (b :: c) and not (a :: b) :: c

Convenient Syntax

Much simpler notation: enclose a list of elements in `[` and `]` separated by `;`

`[1;2;3;4]`	a list of (four) ints
`[“abc”;”xyz”]`	a list of (two) strings
`[[false];[true]]`	a list of lists that each contain booleans
`[]`	the empty list

Convenient Syntax

The two ways of writing lists can be freely mixed.

```
1 :: [2;3;4]
```

a list of (four) ints

NOT Lists

These are *not* lists:

`[1;true;3;4]`	different element types*
`1::2`	2 is not a list
`3::[]::[]`	different element types

*Lists in OCaml are *homogeneous* – all of the list elements must be of the same type.

List Types

The type of lists of integers is written

```
int list
```

The type of lists of strings is written

```
string list
```

The type of lists of booleans is written

```
bool list
```

The type of lists of lists of strings is written

```
(string list) list
```

or

```
string list list
```

etc.

Which of the following expressions has the type

`int list` ?

1) `[3; true]`

2) `[1;2;3]::[1;2]`

3) `[]::[1;2]::[]`

4) `(1::2)::(3::4)::[]`

5) `[1;2;3;4]`

Answer: 5

Which of the following expressions has the type

`(int list) list ?`

```
1) [3; true]

2) [1;2;3]::[1;2]

3) []::[1;2]::[]

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

5) [1;2;3;4]
```

Answer: 3

Calculating With Lists

- Calculating with lists is like calculating with arithmetic expressions. Just simplify each subexpression in the list expression.

  (2+3)::(12 / 5)::[]

$\longmapsto$ 5::(12 / 5)::[]   because 2+3 $\Rightarrow$ 5

$\longmapsto$ 5::2::[]   because 12/5 $\Rightarrow$ 2

A list is a *value* whenever all of its elements are values.

Inspecting lists

- So far, we've seen how to *build* lists in OCaml
- To write list-processing programs, we also need to be able to *inspect* existing lists (so that we can process their parts)…

Pattern Matching

OCaml provides a *pattern matching* construct for inspecting a list and naming its subcomponents.

match expression syntax is:

begin match … with
| … -> …
| … -> …
end

```
let foo (l : int list) : int =
  begin match l with
  | [] -> 42
  | first::rest -> first+10
  end
```

case branches

Case analysis is justified because there are only *two* shapes a list can have.

Note that `first` and `rest` are identifiers that are bound in the body of the branch

- `first` names the head of the list; its type is the element type.
- `rest` names the tail of the list; its type is the list type

The type of the match expression is the (one) type shared by its branches.

Calculating with match

- Consider how to evaluate a match expression:

```
foo [1;2;3]
```

$\longmapsto$

```
begin match [1;2;3] with
  | [] -> 42
  | first::rest -> first + 10
end
```

Calculating with match

- Consider how to evaluate a match expression:

```
foo [1;2;3]
```

$\longmapsto$

```
begin match 1::(2::(3::[])) with
  | [] -> 42
  | first::rest -> first + 10
end
```

Note: [1;2;3] means 1::(2::(3::[]))

Calculating with match

- Consider how to evaluate a match expression:

```
foo [1;2;3]
```

$\longmapsto$

```
begin match 1::(2::(3::[])) with
  | [] -> 42
  | first::rest -> first + 10
end
```

match checks each branch in sequence:

(1). pattern [] *does not* match 1::(2::(3::[]))

Calculating with match

- Consider how to evaluate a match expression:

```
foo [1;2;3]
```

$\longmapsto$

```
begin match 1::(2::(3::[])) with
  | [] -> 42
  | first::rest -> first + 10
end
```

match checks each branch in sequence:

~~(1). pattern [] *does not* match 1::(2::(3::[]))~~

(2). pattern `first::rest` *does* match 1::(2::(3::[]))

`first` = 1

`rest` = (2::(3::[]))

Calculating with match

- Consider how to evaluate a match expression:

```
foo [1;2;3]
```

$\longmapsto$

```
begin match [1;2;3] with
  | [] -> 42
  | first::rest -> first + 10
end
```

$\longmapsto$

1 + 10

$\longmapsto$

11

match checks each branch in sequence:

~~(1). pattern [] *does not* match 1::(2::(3::[]))~~

(2). pattern `first::rest` *does* match 1::(2::(3::[]))

`first` = 1

`rest` = (2::(3::[]))

...so: substitute in that branch.

The Inductive Nature of Lists

A list value is either:

`[]` the *empty* list, sometimes called *nil*

or

`v :: tail` a *head* value v, followed by a list value containing the remaining elements, the *tail*

- Why is this well-defined? The definition of list mentions 'list'!
- Solution: 'list' is *inductive*:
  - The empty list [] is the (only) list of 0 elements
  - To construct a list of n+1 elements, add a head element to an *existing* list of n elements
  - The set of list values contains all and only values constructed this way
- Corresponding computation principle: *recursion*

Recursion

Recursion principle:

Compute a function value for a given input by combining the results for strictly smaller subcomponents of the input.

- The structure of the computation follows the inductive structure of the input.

- Example:

```
length 1::2::3::[]   =  1 + (length 2::3::[])
length 2::3::[]      =  1 + (length 3::[])
length 3::[]         =  1 + (length [])
length []            =  0
```

Recursion Over Lists in Code

The function calls itself *recursively* so the function declaration must be marked with rec.

Lists are either empty or nonempty. *Pattern matching* determines which.

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

If the list is non-empty, then “x” is the first string in the list and “rest” is the remainder of the list.

Patterns specify the **structure** of the value and (optionally) give **names** to parts of it.

Calculating with Recursion

```
length [“a”; “b”]
```

⟼ *(substitute the list for l in the function body)*

```
begin match “a”::“b”::[] with
| [] -> 0
| ( x :: rest ) -> 1 + length rest
end
```

⟼ *(second case matches with rest = “b”::[])*

```
1 + (length “b”::[])
```

⟼ *(substitute the list for l in the function body)*

```
1 + (begin match “b”::[] with
     | [] -> 0
     | ( x :: rest ) -> 1 + length rest
     end )
```

⟼ *(second case matches again, with rest = [])*

```
1 + (1 + length [])
```

⟼ *(substitute [] for l in the function body)*

…

⟼ 1 + 1 + 0 ⇒ 2

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

Recursive function patterns

Recursive functions over lists follow a general pattern:

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

```
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

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?