OCaml IV: Lists and IO

CAS CS 320: Principles of Programming Languages

Thursday, February 1, 2024

Administrivia

- Homework 1 is due today by 11:59 pm.
- Homework 2 is posted today and due on Thursday, Feb 8, by 11:59 pm.

Administrivia

- Homework 1 is due today by 11:59 pm.
- Homework 2 is posted today and due on Thursday, Feb 8, by 11:59 pm.

Reading Assignment

- OCP Section 3.1: **Lists**

building lists (OCP 3.1.1): three syntactic forms

```
(* empty list, also called “nil” *)
[]
(* prepending elt to lst with “cons” :: *)
elt :: lst
(* list with n expressions, all of the same type *)
[e_1; e_2; . . . ; e_n]
```

building lists (OCP 3.1.1): three syntactic forms

```
(* empty list, also called “nil” *)
[]
(* prepending elt to lst with “cons” :: *)
elt :: lst
(* list with n expressions, all of the same type *)
[e_1; e_2; . . . ; e_n]
```

Specific examples:

```
# let lst1 = [] ;;
# let lst2 = 'a' :: lst1 ;;
# let lst3 = 'b' :: lst2 ;;
# let lst4 = 'c' :: lst3 ;;
val lst4 : char list = ['c'; 'b'; 'a']
```

some notational conventions (OCP 3.1.1)

- metavariable **e** usually denotes an **expression** that can be evaluated further.

metavariable **v** usually denotes a **value**, i.e. an expression that can***not*** be evaluated further.

some notational conventions (OCP 3.1.1)

- metavariable **e** usually denotes an **expression** that can be evaluated further.

metavariable **v** usually denotes a **value**, i.e. an expression that can***not*** be evaluated further.

- **[e_1;e_2;e_3]** is sugar for **(e_1 :: (e_2 :: (e_3 ::[])))**.

for example: **['c' ; 'b' ; 'a']** is syntactic sugar for **('c' :: ('b' :: ('a' :: [])))**.

some notational conventions (OCP 3.1.1)

- metavariable **e** usually denotes an **expression** that can be evaluated further.

  metavariable **v** usually denotes a **value**, i.e. an expression that can***not*** be evaluated further.

- **[e_1;e_2;e_3]** is sugar for **(e_1 :: (e_2 :: (e_3 ::[])))**.

  for example: **['c' ; 'b' ; 'a']** is syntactic sugar for **('c' :: ('b' :: ('a' :: [])))**.

- **e ==> v** means **e** evaluates to **v** in finitely many, possibly 0, steps.

  for example, **3+1 ==> 4**, **2*(3+1)==> 8**, and **7 ==> 7**.

examples of expressions which are not values

```
2 + 1

2 * 2

"a" ^ "b"

((2+1) :: ((2*2) :: (5 :: [])))
```

examples of expressions which are not values

```
2 + 1

2 * 2

"a" ^ "b"

((2+1) :: ((2*2) :: (5 :: [])))
```

examples of expressions which are values

```
3

4

"ab"

(3 :: (4 :: (5 :: [])))
```

dynamic semantics (OCP 3.1.1)

evaluation rules:

$$\frac{e_1 \Longrightarrow v_1 \qquad e_2 \Longrightarrow v_2}{(e_1 :: e_2) \Longrightarrow v_1 :: v_2}$$

$$\frac{e_i \Longrightarrow v_i \text{ for all } i \text{ in } \{1,\ldots,n\}}{[e_1; \ldots ; e_n] \Longrightarrow [v_1; \ldots ; v_n]}$$

dynamic semantics (OCP 3.1.1)

evaluation rules:

```
e_1 ==> v_1      e_2 ==> v_2
------------------------------
(e_1 :: e_2) ==>  v_1 :: v_2


e_i ==> v_i for all i in {1,…,n}
------------------------------------
[e_1; … ; e_n] ==> [v_1; … ; v_n]
```

somewhat less transparent:

```
      e_i ==> v_i for all i in {1, … , n}
---------------------------------------------------
(e_1::( … ::(e_n::[]))) ==> (v_1::( … ::(v_n::[])))
```

<u>**example**</u> (OCP 3.1.1)

simple enough that we all agree that:

```
[2+1 ; 2+2 ; (2+3)*2] ==> [3 ; 4 ; 10]
```

example (OCP 3.1.1)

simple enough that we all agree that:

`[2+1 ; 2+2 ; (2+3)*2] ==> [3 ; 4 ; 10]`

we use evaluation rules to produce a formal evaluation:

$$
\dfrac{\texttt{2+1 ==> 3} \qquad \dfrac{\texttt{2+2 ==> 4} \qquad \texttt{(2+3)*2 ==> 10}}{\texttt{[2+2 ; (2+3)*2] ==> [4 ; 10]}}}{\texttt{[2+1 ; 2+2 ; (2+3)*2] ==> [3 ; 4 ; 10]}}
$$

remark: this formal evaluation is **not** unique, because (like in the book OCP) we have allowed "==>" to mean "zero or more finitely many steps".

static semantics (OCP 3.1.1)

typing rules:

$$\frac{\texttt{[] : 'a list}}{\texttt{[] : t list}} \qquad \frac{\texttt{e : t} \qquad \texttt{[] : t list}}{\texttt{[e] : t list}}$$

$$\frac{\texttt{e\_1 : t} \qquad \texttt{e\_2 : t list}}{\texttt{(e\_1 :: e\_2) : t list}}$$

$$\frac{\texttt{e\_i : t for all i in \{1, … ,n\}}}{\texttt{[e\_1; … ; e\_n] : t list}}$$

example (OCP 3.1.1)

simple enough that we all agree that:

```
[3 ; 4 ; 10] : int list
```

example (OCP 3.1.1)

simple enough that we all agree that:

`[3 ; 4 ; 10] : int list`

we use typing rules to formally derive a typing:

$$\dfrac{3 : \texttt{int} \qquad \dfrac{4 : \texttt{int} \qquad \dfrac{10 : \texttt{int} \qquad \dfrac{\texttt{[]} : \texttt{'a list}}{\texttt{[]} : \texttt{int list}}}{\texttt{[10]} : \texttt{int list}}}{\texttt{[4 ; 10]} : \texttt{int list}}}{\texttt{[3 ; 4 ; 10]} : \texttt{int list}}$$

example (OCP 3.1.1)

simple enough that we all agree that:

`[3 ; 4 ; 10] : int list`

we use typing rules to formally derive a typing:

$$
\dfrac{3 : \texttt{int} \qquad \dfrac{4 : \texttt{int} \qquad \dfrac{10 : \texttt{int} \qquad \dfrac{\texttt{[]} : \texttt{'a list}}{\texttt{[]} : \texttt{int list}}}{\texttt{[10]} : \texttt{int list}}}{\texttt{[4 ; 10]} : \texttt{int list}}}{\texttt{[3 ; 4 ; 10]} : \texttt{int list}}
$$

question: Is the typing derivation uniquely defined?

accessing lists (OCP 3.1.2)

- two ways of building lists: with **nil "[]"** and **cons "::"**

- to take apart a list into its component pieces, we have to say what to do with the list if it is *empty* **[]**, and what to do if it is *non-empty* of the form **(elt :: lst)**.

accessing lists (OCP 3.1.2)

- two ways of building lists: with **nil** **"[]"** and **cons** **"::"**
- to take apart a list into its component pieces, we have to say what to do with the list if it is *empty* **[]**, and what to do if it is *non-empty* of the form **(elt :: lst)**.
- best way to do this is with **pattern matching.**
- example: function **length** computes the length of a list:

```
let rec length lst =
  match lst with
  | [] -> 0
  | (h :: t) -> 1 + length t
```

another example (OCP 3.1.2)

function **append** *splices* (i.e. *concatenates*) two lists:

```
let rec append lst1 lst2 =
  match lst1 with
  | [] -> lst2
  | h :: t -> h :: append t lst2
```

yet another example (OCP 3.1.2)

function **sum** adds all the entries in a list of integers:

```
let rec sum lst =
  match lst with
  | [] -> 0
  | h :: t -> h + sum t
```

(non) mutating lists (OCP 3.1.3)

- lists in OCaml are *immutable*, i.e., there is no way to change an element of a list from one value to another.

- instead, in OCaml, we create new lists out of old lists.

(non) mutating lists (OCP 3.1.3)

- lists in OCaml are *immutable*, i.e., there is no way to change an element of a list from one value to another.
- instead, in OCaml, we create new lists out of old lists.
- example: **inc_fst** increments first entry in integer list:

```
let inc_fst lst =
  match lst with
  | [] -> []
  | h :: t -> (h + 1) :: t
```

(non) mutating lists (OCP 3.1.3)

- lists in OCaml are *immutable*, i.e., there is no way to change an element of a list from one value to another.

- instead, in OCaml, we create new lists out of old lists.

- another example: **inc_snd** increments second entry in list:

```
let inc_snd lst =
  match lst with
  | [] -> []
  | [h] -> [h]
  | h1 :: (h2 :: t) -> h1 :: ( (1 + h2) :: t)
```

pattern mathching with lists (OCP 3.1.4)

syntax: **match e with p_1 -> e_1 | . . . | p_n -> e_n**

dynamic semantics:

(1) evaluate **e** to a value **v**

(2) if **p_i** is the first pattern to match **v**, then evaluate **e_i** to value **v_i** and return **v_i**

remark: it is a little more complicated to set up the dynamic semantics of **math-with** with formal evaluation rules (see OCP 3.1.4 for details)

pattern mathching with lists (OCP 3.1.4)

syntax: **match e with p_1 -> e_1 | . . . | p_n -> e_n**

static semantics:

```
e : t_a     p_i : t_a and e_i : t_b  for all i in {1,…,n}
------------------------------------------------------------
   match e with p_1 -> e_1 | . . . | p_n -> e_n : t_b
```

deep pattern matching (OCP 3.1.5)

patterns can be *nested*, which allows us to look deeply into the structure of a list.

Examples:

_ :: []	matches all lists with	**??**	element
_ :: _	matches all lists with	**??**	elements
_ :: _ :: []	matches all lists with	**??**	elements
_ :: _ :: _ :: _	matches all lists with	**??**	elements

deep pattern matching (OCP 3.1.5)

patterns can be *nested*, which allows us to look deeply into the structure of a list.

Examples:

_ :: []	matches all lists with **1** element
_ :: _	matches all lists with **1 or more** elements
_ :: _ :: []	matches all lists with **2** elements
_ :: _ :: _ :: _	matches all lists with **3 or more** elements

tail recursion (OCP 3.1.9)

- a function is *tail recursive* if it calls itself recursively but does not perform any computation after the recursive call returns, and immediately returns to its caller the value of its recursive call.

- example of a **non-tail recursive** function:

```
(* factorial function with pattern matching *)
let rec fact n =
    match n with
    | 0 -> 1
    | n -> n * fact (n - 1) ;;
```

- several advantages of tail-recursion … **what are they?**

tail recursion (OCP 3.1.9)

- a function is *tail recursive* if it calls itself recursively but does not perform any computation after the recursive call returns, and immediately returns to its caller the value of its recursive call.

- one way *(not the only one)* of implementing factorial function with **tail recursion** is to use a **help** function:

```
let fact n =
  let rec helper n current =
    match n with
    | 0 -> current
    | n -> helper (n-1) (n*current) (* why not (current*n)?? *)
  in helper n 1 ;;
```

tail recursion (OCP 3.1.9)

- a function is *tail recursive* if it calls itself recursively but does not perform any computation after the recursive call returns, and immediately returns to its caller the value of its recursive call.

- example of a function already in **tail recursive** form – no need to transform it:

```
(* GCD function with pattern matching *)
let rec gcd n m =
    match m with
    | 0 -> n
    | m -> gcd m (n mod m) ;;
```

tail recursion (OCP 3.1.9)

- a function is *tail recursive* if it calls itself recursively but does not perform any computation after the recursive call returns, and immediately returns to its caller the value of its recursive call.

- example of a function not in **tail recursive** form:

```
(* power function with pattern matching *)
let rec pow x y =
    match y with
    | 0 -> 1
    | y -> x * pow x (y - 1) ;;
```

[tail recursion]() (OCP 3.1.9)

- a function is *tail recursive* if it calls itself recursively but does not perform any computation after the recursive call returns, and immediately returns to its caller the value of its recursive call.

- transforming the power function into **tail recursive** form:

```
(* power function in tail-recursive form *)
let pow x y =
 let rec helper x y acc =
   match y with
   | 0 -> acc
   | y -> helper x (y-1) (x*acc) (* why not (acc*n)?? *)
 in helper x y 1 ;;
```

tail recursion (OCP 3.1.9)

- a function is *tail recursive* if it calls itself recursively but does not perform any computation after the recursive call returns, and immediately returns to its caller the value of its recursive call.

- example of a function not in **tail recursive** form:

```
(* function foo is not tail-recursive *)
let rec foo n =
  match n with
  | 0 -> 1
  | n -> n/(foo (n-1));; (* substitute "/" for "*" in fact *)
```

problem: define function foo_tr in tail-recursive form which is equivalent to function foo.

(THIS PAGE INTENTIONALLY LEFT BLANK)