CS103 Daily Doggie Bag 5

The most important topics and details of the day.

January 31, 2006

**Topic 1. Recursive functions.** Assume the following recursive definitions of the exponentiation and factorial functions:

```ocaml
let rec expt =
  (fun x -> if x = 0 then 1 else 2 * expt(x - 1))

let rec fact =
  (fun x -> if x = 0 then 1 else x * fact(x - 1))
```

To illustrate the meaning of these definitions, think of them as analogues of the following C functions:

```c
int expt(int x) {
    int n1, n2 = 1;
    if (x == 0) { return 1; }
    n1 = x - 1;
    n2 = expt(n1);
    return (2 * n2);
}

int fact(int x) {
    int n1, n2 = 1;
    if (x == 0) { return 1; }
    n1 = x - 1;
    n2 = fact(n1);
    return (x * n2);
}
```

**Topic 2. Recursion and induction.** Recursion and induction are deeply interrelated concepts. Induction is a method of proof analysis on natural numbers (mathematical induction), and recursively defined structures in general. Recursive programming is simplified by thinking inductively.

**Topic 3. Mathematical induction.** Mathematical induction is perhaps the most basic forms of induction. It is defined on the natural numbers, \( \mathbb{N} = \{0, 1, 2, 3, \ldots\} \). When we want to prove that property \( P \) holds for \( \mathbb{N} \), the general form is as follows:

- **proof:** By mathematical induction.
- **base case:** Prove that \( P(0) \) holds.
- **induction hypothesis:** Assume that \( P(j) \) holds for all \( 0 \leq j < n \).
- **NB:** The IH holds for all values between 1 and \( n - 1 \); this will come into play in the homework.
- **induction step:** Given the induction hypothesis, prove that \( P(n) \) holds.
Mathematical induction is like a machine that lets us crank through all possible cases of the natural numbers; the base case gets us started at 0, and then the induction step gets us to 1, 2, 3,...

example:

proposition: For all \( n \in \mathbb{N} \), \( \text{expt } n \downarrow 2^n \).

proof: By mathematical induction on \( n \).

basis \( n = 0 \): If \( n = 0 \), then \( 2^0 = 1 \), and \( \text{expt } 0 \downarrow 1 \) by definition of \( \text{expt} \).

induction hypothesis: For all \( 0 \leq j < n \), \( \text{expt } j \downarrow 2^j \).

induction step: By definition of \( \text{expt} \), \( \text{expt } n \downarrow 2 \ast \text{expt } (n-1) \). But \( 2^n = 2 \ast 2^{n-1} \) by properties of arithmetic, and by the induction hypothesis \( \text{expt } (n-1) \downarrow 2^{n-1} \), hence \( \text{expt } n \downarrow 2 \ast 2^{n-1} \), i.e. \( \text{expt } n \downarrow 2^n \).

\( \square \)

example:

proposition: For all \( n \in \mathbb{N} \), \( \text{fact } n \downarrow n! \).

proof: By mathematical induction on \( n \).

basis \( n = 0 \): If \( n = 0 \), then \( n! = 1 \) by definition, and \( \text{expt } 0 \downarrow 1 \) by definition of \( \text{expt} \).

induction hypothesis: For all \( 0 \leq j < n \), \( \text{fact } j \downarrow j! \).

induction step: By definition of \( \text{fact} \), \( \text{fact } n \downarrow n \ast \text{fact } (n-1) \). But \( n! = n \ast (n-1)! \) by definition, and by the induction hypothesis \( \text{fact } (n-1) \downarrow (n-1)! \), hence \( \text{fact } n \downarrow n \ast (n-1)! \), i.e. \( \text{fact } n \downarrow n! \).

\( \square \)

**Topic 4. Syntactic Sugar.** So far, we have been naming functions using the usual “let” form for declarations. However, OCaml provides an appealing abbreviation for declaring functions:

\[
\begin{align*}
\text{let } f \; = \; & (\text{fun } x \rightarrow e) \triangleq \text{let } f \; x \; = \; e \\
\text{let rec } f \; = \; & (\text{fun } x \rightarrow e) \triangleq \text{let rec } f \; x \; = \; e \\
\text{let } f \; = \; & (\text{fun } x \rightarrow e) \text{ in } e' \triangleq \text{let } f \; x \; = \; e \text{ in } e' \\
\text{let rec } f \; = \; & (\text{fun } x \rightarrow e) \text{ in } e' \triangleq \text{let rec } f \; x \; = \; e \text{ in } e'
\end{align*}
\]

example:

\[
\begin{align*}
\text{let double } x \; = \; & 2 \ast x \triangleq \text{let double } \; = \; (\text{fun } x \rightarrow 2 \ast x) \\
\text{let rec } \text{fact } n \; = \; & \text{if } n \; = \; 0 \text{ then } 1 \text{ else } n \ast \text{fact } (n-1)
\end{align*}
\]

(In general, \( \triangleq \) means “equal by definition”). This is an example of *syntactic sugar*, a trick whereby we define more succinct and appealing language forms as macros for expressions in the basic syntax, rather than *ab initio*. This simplifies the theoretical development and implementation of the language, while still offering convenience to the programmer.

**Topic 5. Commenting conventions.** An absolutely essential aspect of programming is understandable documentation, to explain your code to yourself and others. Type annotations are an excellent way to document functions—in fact, the *declarative* nature of types is one of the most appealing characteristics of type analyses. Of course, with type inference, direct annotations are cumbersome and unnecessary; thus, type specifications should be moved to the comments, along with semantic descriptions. Comments should have the general form:
(*)
  <name> : \tau
  in : <formal parameters, expected invariants>
  out : <precise description of semantics>
*)

example:

(*)
  fact : int -> int
  in : x >= 0
  out : x!
*)

let rec fact x = if x = 0 then 1 else x * expt (x-1)

This convention will be expected in the homework assignments.