SOFTWARE VERIFICATION
AND COMPUTER PROOF
(lesson 1, part 2)

Enrico Tassi – Inria Sophia-Antipolis
Functional programming

1. Programs are data
2. Types provide guidance for building and destructing data
Programs are data

- There is no “return” keyword: expression = statement
- The program that always returns 4 is:
  4
- This program also returns (or better computes to) 4:
  if (false || true) then 2 + 2 else 7
  if true then 2 + 2 else 7
  2 + 2
  4
- This program also computes to 4:
  2 + (if leb 7 2 then 4 else 2)
  2 + (if false then 4 else 2)
  2 + 2
  4
## Types: build and destruct

<table>
<thead>
<tr>
<th>type</th>
<th>build</th>
<th>destruct</th>
</tr>
</thead>
<tbody>
<tr>
<td>bool</td>
<td>true : bool</td>
<td>match b with</td>
</tr>
<tr>
<td></td>
<td>false : bool</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>end</td>
</tr>
<tr>
<td>nat</td>
<td>O : nat</td>
<td>match n with</td>
</tr>
<tr>
<td></td>
<td>S : nat -&gt; nat</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>end</td>
</tr>
<tr>
<td>list nat</td>
<td>nil : list nat</td>
<td>match l with</td>
</tr>
<tr>
<td></td>
<td>cons : nat -&gt; list nat -&gt; list nat</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>end</td>
</tr>
</tbody>
</table>
Types: build (example)

Definition a_natural_number : nat :=
   O     (* ok *)
   S     (* no, S : nat → nat *)
   S O   (* ok *)
   S (S O) (* ok *)

Definition another_natural_number : nat := S a_natural_number.
Types: destruct (example 1)

Execution of a “match ... with ... end”:
match (S (S O)) with
| O => ...
| S n => ...
end.

1. the term under exam (S (S O)) is compare with every branch
1.1 first branch:
   O  => ... this branch is not taken ...
   S (S O)
1.2 second branch. Remark: n is a name that will be bound.
   S n  => .... the branch is taken, and n takes (S O)
   S (S O)
Types: destruct (example 2)

match (S (S O)) with
| O => ...
| S n => ... what is the value of n here? .... end.

Let's run this program:
  match (2 + 1) with O => O | S n1 => n1 end
  match 3 with O => O | S n1 => n1 end
  match S (S (S O)) with O => O | S n1 => n1 end
  S (S O)
## Types: build and destruct

<table>
<thead>
<tr>
<th>type</th>
<th>build</th>
<th>destruct</th>
</tr>
</thead>
<tbody>
<tr>
<td>option nat</td>
<td>None : option nat</td>
<td>match o with</td>
</tr>
<tr>
<td></td>
<td>Some : nat → option nat</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>prod A B</td>
<td>pair : A → B → (A * B)</td>
<td>match p with</td>
</tr>
<tr>
<td>(A * B)</td>
<td>(a, b)</td>
<td></td>
</tr>
<tr>
<td></td>
<td>if (a : A) and (b : B)</td>
<td></td>
</tr>
</tbody>
</table>
Types: build and destruct (example)

match Some 3 with
| None => ...
| Some n => ... what is the value of n here? ....
end.

Definition option_add1 (on : option nat) : option nat :=
match on with
| None => None
| Some n => Some (n + 1)
end.
Last example

From the exercises of yesterday:

\[
\begin{align*}
3 + 5 + (\text{slnat} \ (3 :: 4 :: \text{nil})) \\
3 + 5 + (3 + (\text{slnat} \ (4 :: \text{nil}))) \\
8 + (3 + 4 + (\text{slnat} \ \text{nil})) \\
8 + (3 + 4 + 0) \\
8 + 7 \\
15
\end{align*}
\]

Fixpoint slnat (l : list nat) :=
match l with
| nil => 0
| x :: xs => x + (slnat xs)
end.
A few more exercises.

- define a function “map” that takes function \( f : \text{nat} \to \text{nat} \) and a list \( l : \text{list nat} \). It returns a list that is obtained by applying \( f \) to each element of \( l \) (preserving the order).

E.g.
\[
\text{(map (fun x : nat => x * 2) (1 :: 2 :: 3 :: nil))} \\
(2 :: 4 :: 6 :: nil)
\]

- define a function “append” that takes two lists and returns a list containing all the elements of the first list followed by the elements of the second list.

E.g.
\[
\text{(append (1 :: 2 :: nil) (3 :: 4 :: nil))} \\
(1 :: 2 :: 3 :: 4 :: nil)
\]