Introducing Lists

- Given $A : \text{Set}$ we write $\text{list } A : \text{Set}$ for the set of finite sequences over $A$.
- Lists are widely used in functional programming languages like Lisp, Scheme, CAML, Haskell and F#.
- In Coq we define lists as an inductive type (like $\mathbb{N}$):

  ```coq
  Inductive list (A : Set) : Set :=
  | nil : list A
  | cons : A -> list A -> list A.
  ```
Introducing Lists

- E.g. the sequence of natural numbers 1, 2, 3 becomes:
  \[
  \text{cons } 1 \ (\text{cons } 2 \ (\text{cons } 3 \ \text{nil})) : \text{list}\ \mathbb{N}
  \]

- We abbreviate \texttt{cons a} as \texttt{a :: l}. Hence the previous example becomes:
  \[
  1 :: 2 :: 3 :: \text{nil} : \text{list}\ \mathbb{N}
  \]

- Note that the roles of : and :: are reverse in Haskell.

- Functional programming languages also use an even more compact notation:
  \[
  [1, 2, 3] : \text{list}\ \mathbb{N}
  \]
As for $\mathbb{N}$ we can define functions by structural recursion over lists.

An example is *append* (written $\+\+$):

\[
\begin{align*}
\+\+ & : \text{list } A \to \text{list } A \to \text{list } A \\
\text{n}i\text{l} + + m & = m \\
(a :: l) + + m & = a :: (l + + m)
\end{align*}
\]

Which function on the natural numbers resembles $\+\+$?

In Coq we use **Fixpoint**, see *l12.v*.
List induction

- Like induction for natural numbers, there is induction for lists.
- Given a predicate over lists $P : \text{list} A \rightarrow \text{Prop}$, we can show that it holds for all lists $(\forall l : \text{list} A, P l)$ by showing:
  - base It holds for $\text{nil}$
    \[ P \text{ nil} \]
  - step It is preserved by cons:
    \[ \forall a : A \forall m : \text{list} A, P m \rightarrow P (a :: m) \]

- To summarize
  \[
  (P \text{ nil})
  \rightarrow (\forall a : A \forall m : \text{list} A, P m \rightarrow P (a :: m))
  \rightarrow \forall l : \text{list} A, P l
  \]

- In Coq we use the induction tactic (as for $\mathbb{N}$).
Lists are a monoid

Using list induction we can show that lists form a monoid:

\[
\begin{align*}
\text{nil} ++ m & = m \\
\text{l} ++ \text{nil} & = \text{l} \\
l ++ (m ++ n) & = (l ++ m) ++ n
\end{align*}
\]

However, this is not a commutative monoid:

\[
[1, 2] ++ [3] = [1, 2, 3] \neq [3] ++ [1, 2] = [3, 1, 2]
\]

Actually \((\text{list } A, \text{nil}, ++)\) is the free monoid over \(A\), because:

- \(\text{list } A\) contains all the elements of \(A\) (as singleton lists \([a]\)).
- It doesn’t satisfy any additional equations (hence it is unconstrained, i.e. free).
We introduce an operation \( \text{rev} : \text{list} \ A \) on lists which reverses a list. E.g.

\[
\text{rev} \ [1, 2, 3] = [3, 2, 1]
\]

\( \text{rev} \) uses an auxiliary operation

\[
\text{snoc} : \text{list} \ A \rightarrow A \rightarrow \text{list} \ A
\]

which appends an element at the end of a list.

\( \text{snoc} = \text{cons} \) backwards.

Both operations can be defined by structural recursion over lists:

\[
\text{snoc} \ \text{nil} \ a = a\]

\[
\text{snoc} \ (b :: l) \ a = b :: (\text{snoc} \ l \ a)
\]

\[
\text{rev} \ \text{nil} = \text{nil}
\]

\[
\text{rev} \ (a :: l) = \text{snoc} \ (\text{rev} \ l) \ a
\]
Clearly reversing twice gets us back to the initial list, e.g.

\[
\text{rev} (\text{rev} [1, 2, 3]) \\
= \text{rev} [3, 2, 1] \\
= [1, 2, 3]
\]

In predicate logic:

\[
\forall l : \text{list } A, \text{rev} (\text{rev } l) = l
\]

We need to show a lemma about snoc:

\[
\forall l : \text{list } A, \forall a : A, \text{rev} (\text{snoc } l a) = a :: \text{rev } l
\]

Both can be established using list induction, see l12.v.