Mathematics for Computer Scientists 2 (G52MC2)

L12 : Lists

Thorsten Altenkirch

School of Computer Science
University of Nottingham

November 19, 2009

Introducing Lists

- Given $A : \mathrm{Set}$ we write $\mathrm{list}\, A : \mathrm{Set}$ 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 $\mathbb{N}$):

```
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:

$$\mathrm{cons}\,1\,(\mathrm{cons}\,2\,(\mathrm{cons}\,3\,\mathrm{nil})) : \mathrm{list}\,\mathbb{N}$$

- We abbreviate $\mathrm{cons}\,a\,l$ as $a :: l$. Hence the previous example becomes:

$$1 :: 2 :: 3 :: \mathrm{nil} : \mathrm{list}\,\mathbb{N}$$

- Note that the roles of $:$ and $::$ are reverse in Haskell.
- Functional programming languages also use an even more compact notation:

$$[1, 2, 3] : \mathrm{list}\,\mathbb{N}$$

Structural recursion

- As for $\mathbb{N}$ we can define functions by structural recursion over lists.
- An example is *append* (written $+\!\!+$) :

$$\begin{aligned} +\!\!+ \quad &: \quad \text{list}\, A \to \text{list}\, A \to \text{list}\, A \\ nil +\!\!+ m \quad &= \quad m \\ (a :: l) +\!\!+ m \quad &= \quad a :: (l +\!\!+ m) \end{aligned}$$

- 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 $P : \mathrm{list}\, A \to \mathrm{Prop}$, we can show that it holds for all lists $(\forall l : \mathrm{list}\, A, P\, l)$ by showing:

  base It holds for nil

  $$P\, \mathrm{nil}$$

  step It is preserved by cons:

  $$\forall a : A \forall m : \mathrm{list}\, A, P\, m \to P\, (a :: m)$$

- To summarize

$$\begin{aligned}&(P\, \mathrm{nil})\\ &\to (\forall a : A \forall m : \mathrm{list}\, A, P\, m \to P\, (a :: m))\\ &\to \forall l : \mathrm{list}\, A, P\, l\end{aligned}$$

- 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:

$$\begin{aligned} \mathrm{nil} +\!\!+ m &= m \\ l +\!\!+ \mathrm{nil} &= l \\ l +\!\!+ (m +\!\!+ n) &= (l +\!\!+ m) +\!\!+ n \end{aligned}$$

- However, this is not a commutative monoid:

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

- Actually $(\mathrm{list}\,A, \mathrm{nil}, +\!\!+)$ is the *free monoid* over A, because:
    - $\mathrm{list}\,A$ contains all the elements of A (as singleton lists $[a]$).
    - It doesn't satsify any additional equations (hence it is unconstrained, i.e. free).

Reverse

- We introduce an operation $\mathrm{rev} : \mathrm{list}\, A$ on lists which reverses a list. E.g.

$$\mathrm{rev}\,[1,2,3] = [3,2,1]$$

- rev uses an auxilliary operation

$$\mathrm{snoc} : \mathrm{list}\, A \to A \to \mathrm{list}\, A$$

  which appends an element at the end of a list.
- snoc = cons backwards.
- Both operations can be defined by structural recursion over lists:

$$\begin{aligned} \mathrm{snoc}\,\mathrm{nil}\, a &= a \\ \mathrm{snoc}\,(b :: l)\, a &= b :: (\mathrm{snoc}\, l\, a) \end{aligned}$$

$$\begin{aligned} \mathrm{rev}\,\mathrm{nil} &= \mathrm{nil} \\ \mathrm{rev}\,(a :: l) &= \mathrm{snoc}\,(\mathrm{rev}\, l)\, a \end{aligned}$$

Reversing twice

- Clearly reversing twice gets us back to the initial list, e.g.

$$\begin{aligned}
&\mathrm{rev}\,(\mathrm{rev}\,[1,2,3]) \\
&= \mathrm{rev}\,[3,2,1] \\
&= [1,2,3]
\end{aligned}$$

- In predicate logic:

$$\forall l : \mathrm{list}\,A, \mathrm{rev}\,(\mathrm{rev}\,l) = l$$

- We need to show a lemma about snoc:

$$\forall l : \mathrm{list}\,A, \forall a : A, \mathrm{rev}\,(\mathrm{snoc}\,l\,a) = a :: \mathrm{rev}\,l$$

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