Lists : a general sequence of elements

Marc Coiffier

Contents

```
'utils require import
```

- Required module: utils

Sometimes in life, we want to do a series of tasks in a certain order, or sort objects in a sequence. In mathematical disciplines, lists play a similar semantic role, allowing us to express sequences of related elements.

The elements of a sequence

The simplest kind of list is either an empty list, or a list containing one element, followed by another list. Given a type A of elements, we can define lists of type A in the following context :

```
'List_context {
  Type '.List ->
  A 'a -> .List 'l -> .List ? ? '.cons ->
  .List '.nil -> } def
```

Armed with this context, defining the usual constructors for the List type and its members becomes easy :

```
'List List_context .List ? ? ? "List A" defconstr
A 'a -> List 'l -> 'cons List_context
    .cons ( a l ( .List .cons .nil ) ) ! ! ! "cons a l" defconstr ! !
'nil List_context .nil ! ! ! "nil" defconstr
[ 'List 'nil 'cons ] { export } each
```

The list recursor, $\lambda(l : List\,A).\,\mu(l)$, has type $\forall(l : List\,A)\,(\mathrm{List}^{P} : List\,A \to Set_1),\,(\forall(a : A)\,(l_0 : List\,A),\,\mathrm{List}^{P}\,l_0 \to \mathrm{List}^{P}\,(cons\,a\,l_0)) \to \mathrm{List}^{P}\,nil \to \mathrm{List}^{P}\,l$. We can now start to define non-trivial combinators that work on lists, such as "map" and "append" :

List combinators

```
!
'list_map Type 'A -> Type 'B -> A 'x -> B ? 'f -> List ( A ) 'l ->
   l (
     List ( B )
     A 'x -> List ( A ) 'l -> cons ( B f ( x ) l ) ! !
     nil ( B )
   ) 4 lambdas def
```

list_map has type $(null)$.

```
'list_append Type 'A -> List ( A ) dup 'x -> 'y -> x (
   List ( A )
   cons ( A )
   y ) ! ! ! def
```

list_append has the type $\forall(A : Set_0)$, $List\,A \rightarrow List\,A \rightarrow List\,A$.