

15–150: Principles of Functional Programming

Brief Introduction to Lists

Michael Erdmann

Spring 2026

Type: $\mathbf{t\ list}$, for any type $\mathbf{t}$.

Values: $[v_1, \dots, v_n]$, with each v_i a value of type $\mathbf{t}$ and $n \geq 0$.

When $n = 0$, one really means the empty list, which appears in ML as $[]$ and is pronounced “nil”.

Expressions: The collection of expressions for this type includes all values of type $\mathbf{t\ list}$.

It further includes all expressions of the form $\mathbf{e::es}$, with $\mathbf{e : t}$ and $\mathbf{es : t\ list}$.

For example, a valid expression is $\mathbf{1::[2,3]}$. This gives the value $[1,2,3]$ of type $\mathbf{int\ list}$.

The operator $\mathbf{::}$ is right associative. It is pronounced “cons”.

Typing Rules:

$$\begin{array}{l} [] : \mathbf{t\ list} \\ \mathbf{e::es} : \mathbf{t\ list} \quad \text{if } \mathbf{e : t} \text{ and } \mathbf{es : t\ list}. \end{array}$$

Evaluation: Evaluation of lists proceeds from left to right, until one obtains values. Formally:

$$\begin{array}{l} [] \text{ is a value} \\ \mathbf{e::es} \xRightarrow{1} \mathbf{e'::es} \quad \text{if } \mathbf{e} \xRightarrow{1} \mathbf{e'} \\ \mathbf{v::es} \xRightarrow{1} \mathbf{v::es'} \quad \text{if } \mathbf{es} \xRightarrow{1} \mathbf{es'} \text{ and } \mathbf{v} \text{ is a value.} \end{array}$$

Pattern Matching: One can pattern match on the structure of a list. For instance, in a **case** expression one might have a clause of the form

$$| \mathbf{x::xs} \Rightarrow \mathbf{e}$$

When type-checking $\mathbf{e}$, the compiler will use $\mathbf{x : t}$ and $\mathbf{xs : t\ list}$ (assuming that is consistent with other patterns and the rest of the **case**).

At runtime, if the pattern $\mathbf{x::xs}$ is matched by a value of the form $\mathbf{v::vs}$, the evaluator will create local bindings $[v/x, vs/xs]$. The evaluator will evaluate $\mathbf{e}$ with these bindings in scope.

Let us write a short function that computes the length of a list. Then we will use structural induction to prove that the function is total.

```
(* length : int list -> int
   REQUIRES: true
   ENSURES: length(L) returns the number of integers in L.
*)

fun length ([] : int list) : int = 0
  | length (x::xs) = 1 + length(xs)
```

Theorem: `length` is total.

Proof: Establishing totality means proving that `length(L)` reduces to a value for all list values `L` of type `int list`.

We will prove that assertion by structural induction on `L`.

BASE CASE: `L = []`.

NEED TO SHOW: `length []` reduces to a value.

SHOWING:

$$\begin{array}{ll} \text{length []} & \\ \implies 0 & \text{[first clause of length]} \end{array}$$

0 is a value, so we have established the base case.

INDUCTIVE CASE: `L = x::xs`, for some values `x : int` and `xs : int list`.

INDUCTION HYPOTHESIS: `length(xs)  $\hookrightarrow$  v` for some value `v`.

NEED TO SHOW: `length(x::xs)  $\hookrightarrow$  v'` for some value `v'`.

SHOWING:

$$\begin{array}{ll} \text{length}(x::xs) & \\ \implies 1 + \text{length}(xs) & \text{[step, second clause of length]} \\ \implies 1 + v & \text{[Induction Hypothesis]} \\ \implies v' & \text{[for some value } v', \text{ assuming + total]} \end{array}$$

That establishes the inductive case.

The base case and the inductive case together establish the Theorem, by a principle of structural induction for `int list`.