15–150: Principles of Functional Programming
Brief Introduction to Lists

Michael Erdmann & Frank Pfenning

Spring 2020

Type: `t list`, for any type `t`.

Values: `[`$\mathtt{v}_1$`, ..., `$\mathtt{v}_n$`]`, with each $\mathtt{v}_i$ a value of type `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 `t list`.
It further includes all expressions of the form `e::es`, with `e : t` and `es : t list`.

For example, a valid expression is `1::[2,3]`. This gives the value `[1,2,3]` of type `int list`.
The operator `::` is right associative. It is pronounced "cons".

Typing Rules:

$$\begin{array}{rcll} \texttt{[]} & \texttt{:} & \texttt{t list} & \\ \texttt{e::es} & \texttt{:} & \texttt{t list} & \text{if } \texttt{e : t} \text{ and } \texttt{es : t list}. \end{array}$$

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

$$\begin{array}{rcll} \texttt{[]} & \text{is a} & \texttt{value} & \\ \texttt{e::es} & \stackrel{1}{\Longrightarrow} & \texttt{e'::es} & \text{if } \texttt{e} \stackrel{1}{\Longrightarrow} \texttt{e'} \\ \texttt{v::es} & \stackrel{1}{\Longrightarrow} & \texttt{v::es'} & \text{if } \texttt{es} \stackrel{1}{\Longrightarrow} \texttt{es'} \text{ and } \texttt{v} \text{ is a value.} \end{array}$$

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

```
| x::xs => e
```

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

At runtime, if the pattern `x::xs` is matched by a value of the form `v::vs`, the evaluator will create local bindings `[v/x, vs/xs]`. The evaluator will evaluate `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:

```
        length []
⟹  0                                  [first clause of length]
```

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

INDUCTION STEP: `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:

```
    length(x::xs)
⟹  1 + length(xs)                          [step, second clause of length]
⟹  1 + v                                              [Inductive Hypothesis]
⟹  v'                  [for some value v', assuming SML addition is correct]
```

That establishes the induction step.

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