Lists

We can have pairs of pairs of pairs... but we still “commit” to the amount of data when we write down a type.

Lists can have any number of elements:

- [] is the empty list (a value)
- More generally, [v1, v2, ..., vn] is a length n list
- If e1 evaluates to v and e2 evaluates to a list [v1, v2, ..., vn], then e1::e2 evaluates to [v, v1, v2, ..., vn] (a value).
- null e evaluates to true if and only if e evaluates to []
- If e evaluates to [v1, v2, ..., vn], then hd e evaluates to v1 and tl e evaluates to [v2, ..., vn].
  - If e evaluates to [], both hd e and tl e raise run-time exceptions. (Different from type errors; more on this later.)
List types

A given list’s elements must all have the same type.

If the elements have type \( t \), then the list has type \( t \ list \). Examples: \( \text{int list} \), \( (\text{int*int}) \ list \), \( (\text{int list}) \ list \).

What are the type rules for ::, null, hd, and tl?

- Possible exceptions do not affect the type.

Hmmm, that does not explain the type of \([\ ]\) ?

- It can have any list type, which is indicated via ’a list.
- That is, we can build a list of any type from \([\ ]\).
- *Polymorphic* types are 3 weeks ahead of us.
  - Teaser: null, hd, and tl are not keywords!
Recursion again

Functions over lists that depend on all list elements will be recursive:

- What should the answer be for the empty list?
- What should they do for a non-empty list? (In terms of answer for the tail of the list.)

Functions that produce lists of (potentially) any size will be recursive:

- When do we create a small (e.g., empty) list?
- How should we build a bigger list out of a smaller one?
Let bindings

Motivation: Functions without local variables can be poor style and/or really inefficient.

Syntax: \texttt{let } b_1 \ b_2 \ldots \ b_n \ \texttt{in } e \ \texttt{end} where each \( b_i \) is a \textit{binding}.

Typing rules: Type-check each \( b_i \) and \( e \) in context including previous bindings. Type of whole expression is type of \( e \).

Evaluation rules: Evaluate each \( b_i \) and \( e \) in environment including previous bindings. Value of whole expression is result of evaluating \( e \).

Elegant design worth repeating:

- Let-expressions can appear anywhere an expression can.
- Let-expressions can have any kind of binding.
  - Local functions can refer to any bindings \textit{in scope}. 
More than style

Exercise: hand-evaluate bad_max and good_max for lists [1,2] [1,2,3], and [3,2,1].

Extra Credit Exercise: As a function of $n$, how long will it take to calculate

- $\text{bad\_max}([1, 2, \ldots, n])$?
- $\text{bad\_max}([n, n-1, \ldots, 1])$?
Summary and general pattern

Major progress: recursive functions, pairs, lists, let-expressions

Each has a syntax, typing rules, evaluation rules.

Functions, pairs, and lists are very different, but we can describe them in the same way:

• How do you create values? (function definition, pair expressions, empty-list and ::)

• How do you use values? (function application, #1 and #2, null, hd, and tl)

This (and conditionals) is enough for your homework though:

•andalso and orelse help

• You need options (next slide)

• Soon: much better ways to use pairs and lists (pattern-matching)
Options

“Options are like lists that can have at most one element.”

- Create a \( t \) option with NONE or SOME \( e \) where \( e \) has type \( t \).
- Use a \( t \) option with isSome and valOf

Why not just use (more general) lists? An interesting style trade-off:

- Options better express purpose, enforce invariants on callers, maybe faster.
- But cannot use functions for lists already written.