Lists

With nested tuples the *type* of a variable still *commits* to a particular *amount* of data

On the other hand lists can have any number of elements But all lists elements have the same type

Building lists

- The empty list is a value: `[]`
- *A list of values is a value*

If `e1` evaluates to `v` and `e2` evaluates to a list `[v1,...vn]`, then e1::e2 evaluates to `[v,... ,vn]`

`e1::e2 (pronouced cons)`

Accessing Lists

Using standard lib functions

- `null e`, takes a list and evaluates to `true`, only if e evaluates to `[]`
- if `e` evaluates to [v1,v2,vn] then `hd e` evaluates to `v1`
    - raises exception if `e` evaluates to `[]`
- if `e` evaluates to [v1,v2,vn] then tl e evaluates to [v2,vn]
    - raises exeception if `e` evaluates to `[]`
    - notice result is a `list`

Type Checking lists

For any *type t* the type `t list` describes lists where all elements have type t The empty list has type alpha list `a list` For the cons operation `e1::e2`, to type check, `e1` needs to have a `t`(ype) and `e2` has type `t list`, then the result is type `t list`

- null: 'a list -> bool
- hd: 'a list -> 'a
- tl: 'a list -> 'a list

Recursion

key points

Functions over lists are usually recursive - only way to get to all the elements

- What should the answer be for the empty list?
- What should the answer be for a non-empty list
    - Typically in terms of the answer for the tail of the list