Lists

List constructors

- Syntax: `Exp ::= '[' ']'| Exp '::'Exp`
- `[]` is the empty list
- $hd::ts$ is the list with *head hd* and *tail tl*
- $[] \neq t_1::t_2$ and $t_1 \neq t_1::t_2$ and $t_2 \neq t_1::t_2$
- $t_1::t_2 = t_1'::t_2'$ if and only if $t_1 = t_1'$ and $t_2 = t_2'$

Syntax rules for the constructor `::`

- right-associative
- lower precedence than unary and binary infix operators
- higher precedence than
  - the tuple constructor
  - anonymous function expression (**`fun`** `... -> ...`)
  - conditional expression (**`if`** `...` **`then`** `...` **`else`** `...`)
- `[`e_1`;`e_2`;...;`e_n`]` is a useful shorthand
  - `[1] = 1::[]`
  - `[1;2;3] = 1::2::3::[]`
  - `[1,`**`true`**`] = (1,`**`true`**`)::[]`
  - `1,[`**`true`**`] = 1,`**`true`**`::[]`

Warning

- the operator `;` inside square brackets has its own precedence rules!
- `;` has lower precedence than the tuple constructor
  `[1,true;2,false]=[(1,true);(2,false)]=(1,true)::(2,false)::[]`
- advice: use parentheses if you are not sure of precedence rules!

Lists

Type constructor for lists

Lists must be homogeneous: all elements have the same type

- unary postfix constructor `list`
- higher precedence than the `->` and $*$ constructors
- $t \neq t$ `list`, t_1`->`$t_2 \neq t$ `list`, $t_1 * t_2 \neq t$ `list`
- t_1 `list` $= t_2$ `list` if and only if $t_1 = t_2$

Examples

```
# [1;2] (* a list of integers *)
- : int list = [1; 2]
# [true;false;true] (* a list of booleans *)
- : bool list = [true; false; true]
# [1,true] (* a list of pairs int*bool *)
- : (int * bool) list = [(1, true)]
# [1,true;2,false] (* a list of pairs int*bool *)
- : (int * bool) list = [(1, true); (2, false)]
# [[1;2];[0;3;4];[]] (* a list of lists of integers *)
- : int list list = [[1; 2]; [0; 3; 4]; []]
```

Lists

Static semantics

- `[]` has type α `list` or `'a list` in the OCaml concrete syntax
- e_1`::`e_2 has type t `list` if and only if
  e_1 has type t and e_2 has type t `list`
- e_1`::`e_2 is **not** type correct if
  - there is no type t s.t. e_1 has type t and e_2 has type t `list`
  - or e_1 or e_2 is **not** type correct

Polymorphic types

- α `list` is a polymorphic type or type scheme
- α is a type variable
- meaning: the set of values intersection of `int list`, `bool list`,
  `(int*bool)list`, `(int -> int)list`, `int list list`, ...
  that is, t `list` for all types t
- mostly used with function types: $\alpha{*}\beta$`->`α, α`->`β`->`α, α `list->int`, ...

Lists

Concatenation

Binary infix operator

`Exp ::= Exp '@' Exp`

- left-associative
- lower precedence than the `::` constructor

Concatenation is not a constructor!

Counter-examples:

- *e*`@[]` = `[]@`*e* = *e*
- `[]@[1;2;3] = [1]@[2;3] = [1;2]@[3] = [1;2;3]@[] = [1;2;3]`

Lists

Static semantics of concatenation

- e_1`@`e_2 has type t `list` if and only if
  e_1 and e_2 have type t `list`
- e_1`@`e_2 is **not** type correct if
  - there is no type t s.t. e_1 and e_2 have type t `list`
  - e_1 or e_2 is **not** type correct

Other details on concatenation

- notation `(@)` to represent the corresponding curried function of polymorphic type `'a list -> 'a list -> 'a list`
- time complexity is linear ($O(n)$) in the length (n) of the left operand

Lists

Examples

```
# [1;2]@[3]@[4;5;6]
- : int list = [1; 2; 3; 4; 5; 6]
# [[1]]@[2]::[[3]]
- : int list list = [[1]; [2]; [3]]
# ([1]@[2])::[[3]]
- : int list list = [[1; 2]; [3]]
# (@)
- : 'a list -> 'a list -> 'a list = <fun>
# (@) [1;2]
- : int list -> int list = <fun>
# (@) [1;2] [3;4;5]
- : int list = [1; 2; 3; 4; 5]
```