Some functions on Lists: Append

<table>
<thead>
<tr>
<th>Base pattern</th>
<th>Ind pattern</th>
</tr>
</thead>
<tbody>
<tr>
<td>( \text{let rec } \text{append } (l1,l2) = )</td>
<td>( l, ) ( \text{append} ) ( \text{of} ) ( )</td>
</tr>
</tbody>
</table>

- Find the right **induction** strategy
  - **Base** case: pattern + expression
  - **Induction** case: pattern + expression

Well designed datatype gives strategy

**hd**, **tl** are all functions ...

**Bad ML style:** More than aesthetics!

- Pattern-matching better than test-extract:
  - ML checks all cases covered
  - ML checks no redundant cases
  - ...at compile-time:
    - fewer errors (crashes) during execution
    - get the bugs out ASAP!

**Another Example: Calculator**

We want an arithmetic calculator to evaluate expressions like:

- \( 4.0 + 2.9 = 6.9 \)
- \( 3.78 - 5.92 = -2.14 \)
- \( (4.0 + 2.9) \times (3.78 - 5.92) = -14.766 \)

**Q:** What's a ML datatype for such expressions?

**Another Example: Calculator**

We want an arithmetic calculator to evaluate expressions like:

- \( 4.0 + 2.9 = 6.9 \)
- \( 3.78 - 5.92 = -2.14 \)
- \( (4.0 + 2.9) \times (3.78 - 5.92) = -14.766 \)

What's a ML function for evaluating such expressions?

\[
\text{let } \text{add } l = \begin{cases} 
\text{match } l \text{ with} \\
\text{ hd } \text{ nil } & \text{hd } \text{ of} \\
\text{ Cons } h t \rightarrow (\text{hd } h) + (\text{hd } t) 
\end{cases}
\]
Begin at the beginning ...

1. Programmer enters expression
2. ML checks if expression is “well-typed”
   - Using a precise set of rules, ML tries to find a unique type for the expression meaningful type for the expr
3. ML evaluates expression to compute value
   - Of the same “type” found in step 2

Expressions (Syntax) --> Values (Semantics) -> Types

Expressions built from sub-expressions
Types computed from types of sub-expressions
Values computed from values of sub-expressions

Base Types

Base Type: int

Base Type: float

Base Type: string

Expressions built from sub-expressions
Types computed from types of sub-expressions
Values computed from values of sub-expressions
Expressions built from sub-expressions
Types computed from types of sub-expressions
Values computed from values of sub-expressions

Base Type: string

Expressions built from sub-expressions
Types computed from types of sub-expressions
Values computed from values of sub-expressions

Base Type: bool

Equality testing is built-in for all expr, values, types
- but compared expressions must have same type
...except for?
- function values ... why?

Type Errors

Expressions built from sub-expressions
Types computed from types of sub-expression
If a sub-expression is not well-typed then whole expression is not well-typed

Complex types: Tuples

\[(e_1, e_2, \ldots) : T_1 \times T_2 \times \ldots\]
Complex types: Tuples

- Can be of any fixed size
- Elements can have different types
- Tuples can be nested in other tuples

(2+2, 7>8); (4, false)

- Can be of any fixed size
- Elements can have different types
- Tuples can be nested in other tuples

(9+3, "abcd", false)

Complex types: Tuples

- Can be of any fixed size
- Elements can have different types
- Tuples can be nested in other tuples

(2+2, 7>8);

(4, false)

(9+3, "abcd", false)

Complex types: Records

Records are tuples with named elements...

{name="sarah"; age=31; pass=false}

But wait...

All evaluation rules look like:

Complex types: Lists

- Unbounded size
- Can have lists of anything (e.g. lists of lists)

[1;2;3;4;5;6]
Complex types: Lists

All elements have the same type

Complex types: list ..construct

Cons “operator”

Can only “cons” element to a list of same type

Complex types: list ..construct

Append “operator”

Can only append lists of the same type

Complex types: list ... deconstruct

Reading the elements of a list:
- Two “operators”: hd (head) and tl (tail)

List: Heads and Tails
**List: Heads and Tails**

**Head**

```
e : T list
  hd e : T
  tl e : T list
```

```
e = (hd [[]]; [1;2;3])
```

- `e : T list` is the type of the list `e`
- `hd e` returns the head of the list `e`
- `tl e` returns the tail of the list `e`

**Tail**

```
e : T list
  e = (hd [[]]; ["a"])
```

- `e : T list` is the type of the list `e`
- `hd e` returns the head of the list `e`
- `tl e` returns the tail of the list `e`

- Examples:
  - `(hd [[]];[1;2;3]) = (hd [[]];["a"])`
  - `int list e1` and `string list e2` type examples

---

**Recap**

1. **Programmer enters expression**
2. **ML checks if expression is “well-typed”**
   - Using a precise set of rules, ML tries to find a unique type for the expression meaningful type for the expression
3. **ML evaluates expression to compute value**
   - Of the same “type” found in step 2

---

**If-then-else expressions**

```
if (1 < 2) then 5 else 10
```

- `if (1 < 2) then ["ab","cd"] else ["x"]` returns `["ab","cd"]`

---

**If-then-else expressions**

```
if false then [1;2] else 5
```

- `if false then ["x"] else [1;2;3]` returns `["x"]`

---

**If-then-else expressions**

```
if (1 < 2) then [1;2] else 5
```

- `if (false then [1;2] else 5)` returns `[1;2]`

---

**If-then-else expressions**

```
e1 = true e2 = v2
if e1 then e2 else e3
```

- `if e1 then e2 else e3` has type `T` if `e1` has type `bool` and `e2` and `e3` have the same type `T`

---

**If-then-else expressions**

```
e1 = false e2 = v2
if e1 then e2 else e3
```

- `if e1 then e2 else e3` has type `V` if `e1` has type `bool` and `e2` and `e3` have the same type `V`
If-then-else expressions

\[ \begin{align*}
e1 : \text{bool} & \quad \text{e2} : T & \quad \text{e3} : T \\
\text{if } e1 \text{ then } e2 \text{ else } e3 & : T
\end{align*} \]

- Then-subexp, Else-subexp must have same type!
  - Equals type of resulting expression

\[
\begin{align*}
\text{if } 1 > 2 \text{ then } [1, 2] \text{ else } [] & \quad \text{if } 1 < 2 \text{ then } [] \text{ else } ["a"] \\
\text{int list} & \quad \text{string list}
\end{align*}
\]

\[(\text{if } 1 > 2 \text{ then } [1, 2] \text{ else } [])(\text{if } 1 < 2 \text{ then } [] \text{ else } ["a"])]\]