An SQL Frontend on top of OCaml for Data Analysis

Yan Dong, Yahui Song, Chin Wei Ngan
National University of Singapore
2nd Sep 2022 @ IFL’22 in Copenhagen
From loops to functional stream processing

```java
List<Album> albums = ...;
List<String> result = new ArrayList<>();

for (Album album : albums) {
    if (album.getYear() != 1999) {
        continue;
    }

    if (album.getGenre() != Genre.ALTERTATIVE) {
        continue;
    }

    result.add(album.getName());

    if (result.size() == 5) {
        break
    }
}

Collections.sort(result);
```

This code is equivalent to:

```java
List<String> result =
    albums.stream()
        .filter(album -> album.getYear == 1999)
        .filter(album -> album.getGenre() == Genre.ALTERTATIVE)
        .limit(5)
        .map(Album::getName)
        .sorted()
        .collect(Collectors.toList());
```
Motivation Example - Data Processing

- Incomes sorted by month in decreasing order in 2021

```ocaml
let income_by_month : order list -> (string * float) list =
  fun orders ->
    let group f l =
      let eq a b = compare (f a) (f b) = 0 in
      List.to_seq l |> Seq.group eq |> Seq.map List.of_seq |> List.of_seq
    in
    List.filter (fun o -> o.month >= "2021-01" && o.month <= "2021-12") orders
    |> List.sort (fun a b -> compare a.month b.month)
    |> group (fun o -> o.month)
    |> List.map (fun l ->
      List.fold_left
      (fun (m, income) o -> (m, o.price +. income))
      ((List.hd l).month, 0.0) l)
    |> List.sort (fun (_, s1) (_, s2) -> compare s2 s1)
```

OCaml code written in a functional manner

With SQL Select Query, the logic can be rewritten in a more concise way

```sql
let income_by_month : order list -> (string * float) list =
  fun orders ->
    SELECT o.month, {sum o.price} FROM o <- orders
    WHERE o.month >= "2021-01" && o.month <= "2021-12"
    GROUP BY o.month ORDER BY {sum o.price} DESC
```
Motivation Example - Data Processing

- Incomes sorted by month in decreasing order in 2021

```ocaml
type order = { id: int; price: float; month: string }

let group f l =
  let eq a b = compare (f a) (f b) = 0 in
  List.to_seq l |> Seq.group eq |> Seq.map List.of_seq |> List.of_seq

let income_by_month : order list -> (string * float) list =
  fun orders ->
    | List.filter (fun o -> o.month >= "2021-01" && o.month <= "2021-12")
    | List.sort (fun a b -> compare a.month b.month)
    | group (fun o -> o.month)
    | List.map (fun l ->
      List.fold_left
        (fun (m, income) o -> (m, o.price +. income))
        ((List.hd l).month, 0.0) l)
    | List.sort (fun (_, s1) (_, s2) -> compare s2 s1)

let income_by_month : order list -> (string * float) list =
  fun orders ->
    SELECT o.month, {sum o.price} FROM o <- orders
    WHERE o.month >= "2021-01" && o.month <= "2021-12"
    GROUP BY o.month ORDER BY {sum o.price} DESC
```

Implement databases by modifying general-purposed programming languages

OCaml code written in a functional manner

With SQL Select Query, the logic can be rewritten in a more concise way
We propose: *SelectML = OCaml + Select Query*

- SelectML is built on top of OCaml
  - Static typed & Functional

- SelectML introduces SQL Select Query onto OCaml
  - Supporting a declarative way for data processing

- Suitable for operating list data
  - Also supporting arrays, sequences, and user-defined data types

- Orthogonal with other OCaml language features
  - Core language, module language, object language, ...
System Overview

- OCaml frontend

\[
\text{Source Program} \xrightarrow{\text{Parsing}} \text{Parsetree} \xrightarrow{\text{Typing}} \text{Typedtree} \xrightarrow{\text{Translating}} \text{Lambda}
\]

- SelectML is implemented by modifying the parsing and typing phases
System Overview

- OCaml frontend

SelectML is implemented by modifying the parsing and typing phases

- Adding the syntax support for SelectML
- Conduct type checking before code refractoring
- Code refractoring via a formally defined translation schema with a set of query plans
SelectML Language Design

• OCaml + Select Expression
  • SELECT, DISTINCT, FROM, WHERE, GROUP BY, HAVING, ORDER BY
  • Aggregate functions & applications

• Common features for data analysis
  • mapping, filtering, grouping, ordering

```
1  (* as arguments *)
2  f 123 (SELECT x FROM x <- xs)
3
4  (* as return values *)
5  let g xs = SELECT x FROM x <- xs
6
7  (* as operands *)
8  x :: SELECT y FROM y <- ys
```

Variables \( x \)
Expressions \( e ::= \ldots \mid q \mid \{ e e \} \)
Select Expressions \( q ::= \text{SELECT} [\text{DISTINCT}] e [\text{FROM} s] [\text{WHERE} e]
\mid [\text{GROUP BY} e] [\text{HAVING} e] [\text{ORDER BY} o] \)
Source Expressions \( s ::= x \gets e \mid (x_1, \ldots, x_n) \gets e \mid s, s \)
Order Expressions \( o ::= e [\text{ASC} \mid \text{DESC} \mid \text{USING} e] \mid o, o \)
• FROM clause
  • `xs AS x` changed to `'x←xs'`
  • `x::'a, xs::'a list`

```sql
/* SQL */
SELECT x, y FROM xs AS x, ys AS y;

/* SelectML */
SELECT x, y FROM x <- xs, y <- ys;
```

• GROUP BY clause
  • `COUNT(y)` changed to `{count y}`

```sql
/* SQL */
SELECT x, COUNT(y) FROM t GROUP BY x;

/* SelectML */
SELECT x, {count y} FROM (x, y) <- t GROUP BY x;
```
• **FROM clause**
  - `xs AS x` changed to `x ← xs`
  - `x :: 'a`, `xs :: 'a` list

  ```
  /* SQL */
  SELECT x, y FROM xs AS x, ys AS y;
  ```

  ```
  (* SelectML *)
  SELECT x, y FROM x ← xs, y ← ys;;
  ```

• **GROUP BY clause**
  - `COUNT(y)` changed to `{count y}`

  ```
  /* SQL */
  SELECT x, COUNT(y) FROM t GROUP BY x;
  ```

  ```
  (* SelectML *)
  SELECT x, (count y) FROM (x, y) ← t GROUP BY x;;
  ```

• **WHERE & HAVING clause**
  - Boolean expressions
  - WHERE: before GROUP BY
  - HAVING: after GROUP BY

  ```
  SELECT x FROM x ← xs WHERE f x;;
  SELECT {h x} FROM x ← xs HAVING g {h x};;
  ```

• **Aggregate functions**
  - Line 2:
    (`'a`, `'b`) `agg` is the built-in type for aggregate functions
  - Line 4~9:
    Common aggregate functions: AVG, COUNT, SUM, MIN, MAX

  ```
  type (a, b, c) aggfunc = 'c * (c -> 'a -> 'c) * (c -> 'b)
  type (_, _) agg = Agg : (a, 'b, c) aggfunc -> ('a, 'b) agg
  ```

  ```
  val mkagg : c -> (c -> 'a -> 'c) -> (c -> 'b) -> ('a, 'b) agg
  let count = mkagg () (fun n _ -> n + 1) (fun n -> n);;
  ```

  ```
  (* usage inside Select Expression *)
  SELECT {count x} FROM x ← [1;2;3];
  ```
• ORDER BY clause
  • Ordering key must be comparable expressions
  • Ordering directions: ASC, DESC, USING cmp_func

```haskell
let odd_first a b =
    let x = a mod 2 = 0 in
    let y = b mod 2 = 0 in
    compare x y;

SELECT x, y
FROM (x, y) <- [("a", 2); ("a", 3); ("b", 4); ("b", 5)]
ORDER BY x ASC, y USING odd_first;

-- (string × int) SelectML.src=[("a",3); ("a",2);("b",5);("b",4)]
```
• ORDER BY clause
  • Ordering key must be comparable expressions
  • Ordering directions: ASC, DESC, USING cmp_func

```c
let odd_first a b =
  let x = a mod 2 = 0 in
  let y = b mod 2 = 0 in
  compare x y;

SELECT x, y
FROM (x, y) <- ["a", 2); ("a", 3); ("b", 4); ("b", 5)]
ORDER BY x ASC, y USING odd_first;
```

• SELECT clause
  • DISTINCT for deduplication

• Type of Select Expressions
  • Assume the type of SELECT clause is 'a
  • Line 1~8: general cases
    • type 'a list
  • Line 10~16: returns exactly one row
    • type 'a

```c
SELECT x FROM x <- [1;1;2;2;3;3];
- : int list = [1; 1; 2; 2; 3; 3]

SELECT DISTINCT x FROM x <- [1;1;2;2;3;3];
- : int list = [1; 2]

SELECT y, x FROM (x, y) <- [("a", 1); ("b", 2)];
- : (int * string) list = [(1, "a"); (2, "b")]

(* without FROM, WHERE, and HAVING *)
SELECT x, y;

SELECT x, y GROUP BY z;

SELECT x, y ORDER BY z;

(* aggregation without GROUP BY, WHERE, and HAVING *)
SELECT {count x} FROM x <- t;
```
Query Plans and Translation Schema

- Source code → Query plans (8 kinds)

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Description</th>
<th>Operation</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\mathcal{E}$</td>
<td>Empty Source</td>
<td>$\sigma_e(\mathcal{P})$</td>
</tr>
<tr>
<td>$\mathcal{D}_\chi(e)$</td>
<td>Data Source</td>
<td>$\Pi_{e/\chi}(\mathcal{P})$</td>
</tr>
<tr>
<td>$\mathcal{P}_1 \times \mathcal{P}_2$</td>
<td>Cartesian Product</td>
<td>$e\mathcal{G}_{e/\chi}(\mathcal{P})$</td>
</tr>
<tr>
<td>$\mathcal{U}(\mathcal{P})$</td>
<td>Deduplication</td>
<td>$e\mathcal{S}_{e/\chi}(\mathcal{P})$</td>
</tr>
</tbody>
</table>

```sql
1 type order = { id: int; price: float; month: string }
2
3 SELECT o.month, {sum o.price} FROM o <- orders
4 WHERE o.month >= "2021-01" && o.month <= "2021-12"
5 GROUP BY o.month ORDER BY {sum o.price} DESC
```
Query Plans and Translation Schema

- Translation Schema: Query plans → Typed OCaml code
  - Semantics of the query plans are captured by primitives listed in module SelectML

```
orders
|> SelectML.input
|> SelectML.filter (fun o ->
  o.month >= "2021-01" && o.month <= "2021-12")
|> SelectML.map (fun o -> o.month, o.price)
|> SelectML.group (fun (__col_1, __col_2) -> __col_1)
  (let Agg (init1, update1, final1) = Stdlib.firstrow in
  let Agg (init2, update2, final2) = Stdlib.sum in
  Agg ((init1, init2),
    (fun (acc1, acc2) (x1, x2) ->
      update1 acc1 x1, update2 acc2 x2),
    (fun (acc1, acc2) -> final1 acc1, final2 acc2)))
|> let key (__col_1, __col_3) -> __col_3) in
|> SelectML.sort (fun a b -> compare (key b) (key a))
|> SelectML.output
```
Translation Schema in Detail

• Query plans are translated to primitives defined in Stdlib module SelectML

• Primitives defined on line 1~10 are used by intermediate computations

• Primitives defined on line 12~14 are used for casting type

```plaintext
1 type 'a t = 'a list
2 val one : 'a t -> 'a
3 val singleton : 'a -> 'a t
4 val product : ('a -> 'b -> 'c) -> 'a t -> 'b t -> 'c t
5 val map : ('a -> 'b) -> 'a t -> 'b t
6 val filter : ('a -> bool) -> 'a t -> 'a t
7 val sort : ('a -> 'a -> int) -> 'a t -> 'a t
8 val unique : 'a t -> 'a t
9 val group_all : ('a, 'b) agg -> 'a t -> 'b
10 val group : ('a -> 'c) -> ('a, 'b) agg -> 'a t -> 'b t
11
12 type 'a src = 'a list
13 val input : 'a src -> 'a t
14 val output : 'a t -> 'a src
```
Translation Schema in Detail

• Query plans are translated to primitives defined in Stdlib module SelectML

• Primitives defined on line 1~10 are used by intermediate computations

• Primitives defined on line 12~14 are used for casting

```ocaml
1 type 'a t = 'a list
2 val one : 'a t -> 'a
3 val singleton : 'a -> 'a t
4 val product : ('a -> 'b -> 'c) -> 'a t -> 'b t -> 'a t
5 val map : ('a -> 'b) -> 'a t -> 'b t
6 val filter : ('a -> bool) -> 'a t -> 'a t
7 val sort : ('a -> 'a -> int) -> 'a t -> 'a t
8 val unique : 'a t -> 'a t
9 val group_all : ('a, 'b) agg -> 'a t -> 'b
10 val group : ('a -> 'c) -> ('a, 'b) agg -> 'a t ->
11
type 'a src = 'a list
12 val input : 'a src -> 'a t
13 val output : 'a t -> 'a src
```
Implementation

• Core implementation: ~ 1000 LOC in OCaml

• Test for validation: ~ 60 testcases, manually marked with the expected outputs

• Flexible Customisations
  ✓ Customise the semantics of Select Expression: rewrite the WHERE implementation to keep the falsas
  ✓ Change the input and output type of Select Expression: from 'a list to 'a array
  ✓ Change the intermediate type of Select Expression: from list to array.
  ✓ Generalization with Functors: to deal with both list type and array type.

https://github.com/dyzsr/ocaml-selectml
Implementation

• Core implementation: ~ 1000 LOC in OCaml

• Test for validation: ~ 60 testcases, manually marked with the expected outputs

• Flexible Customisations

  ✓ Customise the semantics of Select Expression
  ✓ Change the input and output type
  ✓ Change the intermediate type of Select Expression
  ✓ Generalization with Functors: to deal with both list type and array type.

https://github.com/dyzsr/ocaml-selectml
Possible More Features

1. Joins $\Leftrightarrow$ Cartesian product + filter: may reduce memory consumption

```scala
val join : ('a -> 'b -> 'c option) -> 'a t -> 'b t -> 'c t
val join_eq : ('a -> 'd) -> ('b -> 'd) -> 'a t -> 'b t -> 'c t
```

```sql
(* when hash key can be determined *)
SELECT ... FROM x <- xs JOIN y <- ys ON x = y;
(* translation *)
SelectML.join_eq (fun x -> x) (fun y -> y) xs ys;

(* when hash key cannot be determined *)
SELECT ... FROM x <- xs JOIN y <- ys ON f x y;
(* translation *)
SelectML.join (fun x y -> if f x y then Some (x, y) else None) xs ys;
```

2. Window Functions
   - They are like aggregate functions, but without causing rows to become grouped into a single output
   - Additional window functions: LEAD, LAG, RANK, …

```
| x=1, count=3 |
| x=1, count=3 |
| x=1, count=3 |
| x=2, count=2 |
| x=2, count=2 |
| x=3, count=1 |
```

```
| x=1, count=3 |
| x=1, count=3 |
| x=1, count=3 |
| x=2, count=5 |
| x=2, count=5 |
| x=3, count=6 |
```

(a) Window Frames of **PARTITION BY**  (b) Window Frames of **ORDER BY**

```
1 INSERT INTO t VALUES (1,1),(1,2),(1,3),(2,2),(2,3),(3,3);
2 SELECT x, COUNT(y) OVER (PARTITION BY x) FROM t;
3 SELECT x, COUNT(y) OVER (ORDER BY x) FROM t;
```

```
<table>
<thead>
<tr>
<th>x</th>
<th>count</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>3</td>
</tr>
<tr>
<td>2</td>
<td>5</td>
</tr>
<tr>
<td>3</td>
<td>6</td>
</tr>
</tbody>
</table>
```

5 /* PARTITION BY */  6 /* ORDER BY */
Possible Optimizations

1. Rule-based optimisation
   • Eliminate unnecessary plans \( \Pi_{(c_1,c_2)}(\Pi_{(x,y)/(c_1,c_2)}(P)) \implies \Pi_{(x,y)}(P) \)
   • Push down filters \( \sigma_{x<1 \land y>1}(D_x(xs) \times D_y(ys)) \implies \sigma_{x<1}(D_x(xs)) \times \sigma_{y>1}(D_y(ys)) \)

2. Cost-based optimisation
   • Select the optimal algorithm for a query plan, based on the runtime information of data

3. Indexes
   • Optimising table scan
   • May require modifications on the OCaml type system

```
/* SQL */
SELECT x FROM xs AS x WHERE x BETWEEN lb AND ub
SELECT x, ys AS y WHERE x < 1
SELECT x FROM xs AS x, ys AS y WHERE x < y
```

```
let f (module XS : (int * int) list ORDER BY 0) = 
  SELECT x, y FROM (x, y) <- (module XS) WHERE lb <= x && x <= ub;

type order = { id: int; price: float; month: string }

let f' (module XS : order list ORDER BY price) = 
  SELECT x FROM x <- (module XS) WHERE lb <=. x && x <=. ub;
```
Create Database

• From REPL (Read–eval–print loop) to Database

  • Behaviors

  • CREATE introduces a top-level variable xs of type \( \text{int} \times \text{int} \) table
  
  • INSERT inserts two new rows \((1,2), (3,4)\) to xs
  
  • UPDATE updates xs by setting increasing the first column by 1

```haskell
module SelectMLType = sig
  ...;
  val insert : 'a list \rightarrow 'a src \rightarrow unit
  val update : ('a \rightarrow 'a) \rightarrow 'a src \rightarrow unit
end
```

```haskell
#CREATE TABLE xs : int * int;;
#INSERT INTO xs VALUES (1,2), (3,4);;
#UPDATE (a, b) <- xs SET (a+1, b);;
SELECT a, b FROM (a, b) <- xs WHERE a < 4;;
```

```haskell
1 SelectML.insert [(1,2); (3,4)] xs;;
2 SelectML.update (fun a, b -> (a+1, b)) xs;;
```
Conclusions

• SelectML = OCaml + Select Query

• Language Design: Select Expression & Aggregate functions

• Semantics: Typing & Planning rules & Translation schema

• Implementation: Different Flexible Customisations

• More possible features and future work
  • Joins, window functions, optimisations, indexes
  • Connection to database via libraries such as 'caqti' with 'postgresql' database
Conclusions

• SelectML = OCaml + Select Query

• Language Design: Select Expression & Aggregate functions

• Semantics: Typing & Planning rules & Translation schema

• Implementation: Different Flexible Customisations

• More possible features and future work
  • Joins, window functions, optimisations, indexes
  • Connection to database via libraries such as 'caqti' with 'postgresql' database

Thanks!