Introduction to Functional Programming in *OCaml*

Roberto Di Cosmo, Yann Régis-Gianas, Ralf Treinen

Week 2 - Sequence 4: Case study: A small (typed) database

Putting everything together

- A database for a contact list with 3 kinds of queries: `insert`, `delete`, `search`.
- The database engine is a function of type:

```
database -> query -> status * database * contact
```

- The status is `true` if the query went well.

A small typed database I

```
(* A phone number is a sequence of four integers. *)
type phone_number = int * int * int * int;;
# type phone_number = int * int * int * int
```

A small typed database II

```
(* A contact has a name and a phone number. *)
type contact = {
  name         : string;
  phone_number : phone_number
};;
# type contact = {
  name : string;
  phone_number : phone_number;
}
(* Here is a dumb contact. *)
let nobody = { name = ""; phone_number = (0, 0, 0, 0) };;
# val nobody : contact =
  {name = ""; phone_number = (0, 0, 0, 0)}
```

A small typed database III

```
(* A database is a collection of contacts. *)
type database = {
  number_of_contacts : int;
  contacts : contact array;
};;
# type database = {
  number_of_contacts : int;
  contacts : contact array;
}
```

A small typed database IV

```
(* [make n] is the database with no contact and at most [n] contacts
    stored inside. *)
let make max_number_of_contacts =
  {
    number_of_contacts = 0;
    contacts = Array.make max_number_of_contacts nobody
  };;
# val make : int -> database = <fun>
```

A small typed database V

```
(* Queries are represented by a code and a contact.
   - If the code is 0 then the contact must be inserted.
   - If the code is 1 then the contact must be deleted.
   - If the code is 2 then we are looking for a contact with the same name
     in the database.
 *)
type query = {
  code    : int;
  contact : contact;
}
let search db contact =
  let rec aux idx =
    if idx >= db.number_of_contacts then
      (false, db, nobody)
    else if db.contacts.(idx).name = contact.name then
      (true, db, db.contacts.(idx))
```

A small typed database VI

```
    else
      aux (idx + 1)
  in
  aux 0;;
# type query = { code : int; contact : contact; }
val search :
  database -> contact -> bool * database * contact = <fun>
```

A small typed database VII

```
let insert db contact =
  if db.number_of_contacts >= Array.length db.contacts then
    (false, db, nobody)
  else
    let (status, db, _) = search db contact in
    if status then (false, db, contact) else
      let cells i =
        if i = db.number_of_contacts then contact else
    db.contacts.(i)
      in
      let db' = {
          number_of_contacts = db.number_of_contacts + 1;
          contacts = Array.init (Array.length db.contacts) cells
        }
      in
      (true, db', contact);;
```

A small typed database VIII

```
# val insert :
  database -> contact -> bool * database * contact = <fun>
```

A small typed database IX

```
let delete db contact =
  let (status, db, contact) = search db contact in
  if not status then (false, db, contact)
  else
    let cells i = if db.contacts.(i).name = contact.name then nobody
    else db.contacts.(i) in
    let db' = {
        number_of_contacts = db.number_of_contacts - 1;
        contacts = Array.init (Array.length db.contacts) cells
      }
    in
    (true, db', contact);;
# val delete :
  database -> contact -> bool * database * contact = <fun>
```

A small typed database X

```
(* Engine parses and interprets the query. *)
let engine db (code, contact) =
  if code = 0 then insert db contact
  else if code = 1 then delete db contact
  else if code = 2 then search db contact
  else (false, db, nobody);;
# val engine :
  database -> int * contact -> bool * database * contact =
  <fun>
```

A small typed database XI

```
let db = make 5;;
# val db : database =
  {number_of_contacts = 0;
   contacts =
    [|{name = ""; phone_number = (0, 0, 0, 0)};
      {name = ""; phone_number = (0, 0, 0, 0)};
      {name = ""; phone_number = (0, 0, 0, 0)};
      {name = ""; phone_number = (0, 0, 0, 0)};
      {name = ""; phone_number = (0, 0, 0, 0)}|]}
```

A small typed database XII

```
let (status, db, contact) = engine db (0, { name = "luke";
    phone_number = (1, 2, 3, 4) });;
# val status : bool = true
val db : database =
  {number_of_contacts = 1;
   contacts =
    [|{name = "luke"; phone_number = (1, 2, 3, 4)};
      {name = ""; phone_number = (0, 0, 0, 0)};
      {name = ""; phone_number = (0, 0, 0, 0)};
      {name = ""; phone_number = (0, 0, 0, 0)};
      {name = ""; phone_number = (0, 0, 0, 0)}|]}
val contact : contact =
  {name = "luke"; phone_number = (1, 2, 3, 4)}
```

A small typed database XIII

```
let (status, db, contact) = engine db (0, { name = "darth";
    phone_number = (4, 3, 2, 1) });;
# val status : bool = true
val db : database =
  {number_of_contacts = 2;
   contacts =
    [|{name = "luke"; phone_number = (1, 2, 3, 4)};
      {name = "darth"; phone_number = (4, 3, 2, 1)};
      {name = ""; phone_number = (0, 0, 0, 0)};
      {name = ""; phone_number = (0, 0, 0, 0)};
      {name = ""; phone_number = (0, 0, 0, 0)}|]}
val contact : contact =
  {name = "darth"; phone_number = (4, 3, 2, 1)}
```

A small typed database XIV

```
let (status, db, contact) = engine db (2, { name = "luke";
    phone_number = (1, 2, 3, 4) });;
# val status : bool = true
val db : database =
  {number_of_contacts = 2;
   contacts =
    [|{name = "luke"; phone_number = (1, 2, 3, 4)};
      {name = "darth"; phone_number = (4, 3, 2, 1)};
      {name = ""; phone_number = (0, 0, 0, 0)};
      {name = ""; phone_number = (0, 0, 0, 0)};
      {name = ""; phone_number = (0, 0, 0, 0)}|]}
val contact : contact =
  {name = "luke"; phone_number = (1, 2, 3, 4)}
```

A small typed database XV

```
let (status, db, contact) = engine db (1, { name = "luke";
    phone_number = (4, 3, 2, 1) });;
# val status : bool = true
val db : database =
  {number_of_contacts = 1;
   contacts =
    [|{name = ""; phone_number = (0, 0, 0, 0)};
      {name = "darth"; phone_number = (4, 3, 2, 1)};
      {name = ""; phone_number = (0, 0, 0, 0)};
      {name = ""; phone_number = (0, 0, 0, 0)};
      {name = ""; phone_number = (0, 0, 0, 0)}|]}
val contact : contact =
  {name = "luke"; phone_number = (1, 2, 3, 4)}
```

A small typed database XVI

```
let (status, db, contact) = engine db (2, { name = "luke";
    phone_number = (1, 2, 3, 4) });;
# val status : bool = false
val db : database =
  {number_of_contacts = 1;
   contacts =
    [|{name = ""; phone_number = (0, 0, 0, 0)};
      {name = "darth"; phone_number = (4, 3, 2, 1)};
      {name = ""; phone_number = (0, 0, 0, 0)};
      {name = ""; phone_number = (0, 0, 0, 0)};
      {name = ""; phone_number = (0, 0, 0, 0)}|]}
val contact : contact =
  {name = ""; phone_number = (0, 0, 0, 0)}
```

A purely functional database engine

A "non destructive" program

- This database engine has type:

```
database -> query -> status * database * contact
```

- As shown in this type, a **new** database is created each time a query is processed.
- Hence, previous versions of the database are still valid.
- In imperative programming, applying a query would modify the database instead.

This is **a purely functional program**.

Purely functional programs

Side-effects considered harmful

- Functional programming encourages a style in which functions **produce values instead of modifying the memory** as in imperative programming.
- The evaluation of a function does not depend on the state of the program but only on its arguments. Exactly like in Mathematics!
- Mathematical specification can therefore be used on functional programs.
- For instance, for all database `d` and for all contact `c`,

  if `insert db c = (true, db', _)`
  then `search db' c = (true, db', c)`

- As it does not depend on the state of the machine,
  a functional program can be used anytime.
  It is more **composable** than an imperative one.

Weaknesses of our implementation

Imprecise typing of query results

- Search queries return a contact while insertion queries return a new database.
- The type of `engine` forces us to use a single type of query results.
- The type of `engine` should be the **union** of query results types.

Inefficient duplications of databases

- Each time a contact is inserted, the database is duplicated!
- We should use a datastructure that enables more sharing.

Forthcoming **algebraic datatypes**
will be an elegant answer to all these problems!