Macaque

Interrogation sûre et flexible de base de données depuis OCaml

Gabriel Scherer, sous la direction de Jérôme Vouillon

Laboratoire PPS (Paris 7)

Plan

Problématique

Base de données : SQL

SELECT truc, machin **FROM table WHERE** chose

Problématique

Base de données : SQL

SELECT truc, machin **FROM table WHERE** chose

Méthode flexible :

```
# let interroge table predicat =
    "SELECT * FROM " ^ table ^ " WHERE " ^ predicat ;;
val interroge : string -> string -> string
```

Problématique

Base de données : SQL

SELECT truc, machin **FROM table WHERE** chose

Méthode flexible :

```
# let interroge table predicat =
    "SELECT * FROM " ^ table ^ " WHERE " ^ predicat ;;
val interroge : string -> string -> string
```

Méthode sûre (PG'OCaml) :

```
# let interroge predicat =
    PGSQL(dbh) "select * from ingredients where !predicat" ;;
val interroge : bool -> (int32 * string option) list
```

Compromis ?

```
PGSQL(dbh)
  "select event.id, minor_version, major_version,
          last_updated, category, status, start, finish,
          event.room, event.location, title, event.description
   from announcement.event, announcement.category
   where start < $finish :: timestamp and finish > $start :: timestamp
     and event.category = category.id
     and path like $pat"
   order by start

PGSQL(dbh)
  "select event.id, minor_version, major_version,
          last_updated, category, status, start, finish,
          event.room, event.location, title, event.description
   from announcement.event, announcement.category
   where start >= $date :: timestamp
     and event.category = category.id
     and path like $pat
   order by start"
```

Objectifs

- Sûreté : vérification syntaxique et typage statique
- Flexibilité : paramétrer une requête par des expressions, des sous-requêtes...
- Travailler avec le langage OCaml

Notre approche

Typage :

- utiliser le typage le plus expressif possible pour des requêtes SQL
- n'utiliser que le typage pour la vérification statique (pas de vérification de la base de données)

Notre approche

Typage :

- utiliser le typage le plus expressif possible pour des requêtes SQL
- n'utiliser que le typage pour la vérification statique (pas de vérification de la base de données)

Syntaxe :

- une extension Camlp4
- choix d'une syntaxe par "compréhensions" :
  $\{ (x, k) \mid x \in \mathbb{C},\ k \in \mathbb{Z},\ |x|^2 < k \}$
  mais d'autres syntaxes restent possibles

Notre approche

Typage :

- utiliser le typage le plus expressif possible pour des requêtes SQL
- n'utiliser que le typage pour la vérification statique (pas de vérification de la base de données)

Syntaxe :

- une extension Camlp4
- choix d'une syntaxe par "compréhensions" :
  $\{ (x, k) \mid x \in \mathbb{C},\ k \in \mathbb{Z},\ |x|^2 < k \}$
  mais d'autres syntaxes restent possibles

La flexibilité découle naturellement de cette approche.

Exemples

```
# let coordonnees_majeurs utilisateurs =
    << {u.nom; u.adresse} | u in $utilisateurs$; u.age >= 18 >>
;;
val coordonnees_majeurs :
    <adresse: <t: 'a> t; age: <t : int32_t> t; nom: <t: 'b> t; ..> view ->
    <adresse: <t: 'a> t; nom: <t : 'b> t> view
```

Exemples

```
# let coordonnees_majeurs utilisateurs =
    << {u.nom; u.adresse} | u in $utilisateurs$; u.age >= 18 >>
;;
val coordonnees_majeurs :
    <adresse: <t: 'a> t; age: <t : int32_t> t; nom: <t: 'b> t; ..> view ->
    <adresse: <t: 'a> t; nom: <t : 'b> t> view


<:insert< $utilisateurs$ :=
                {id = nextval $uid_seq$; nom = nouveau.nom} |
                  nouveau in $nouveaux_utilisateurs$ >>

<:update< p in $personnel$ := {salaire = 1.05 * p.salaire} |
                                    p.salaire < 2 * $smic$ >>
```

Exemple complet

```
let utilisateurs = <:table< mon_schema.utilisateurs (
  id integer NOT NULL,
  nom text NOT NULL,
  age integer,
  adresse text
) >>
let ciotadens = << u | u in $utilisateurs$; u.adresse = "La Ciotat" >>
```

Exemple complet

```
let utilisateurs = <:table< mon_schema.utilisateurs (
  id integer NOT NULL,
  nom text NOT NULL,
  age integer,
  adresse text
) >>
let ciotadens = << u | u in $utilisateurs$; u.adresse = "La Ciotat" >>

let print_user u =
  let age_str = match Sql.getn u#age with
  | Some age -> Printf.sprintf "%ld ans" age
  | None -> "age inconnu" in
  Printf.printf "%s : %s\n" (Sql.get u#nom) age_str
```

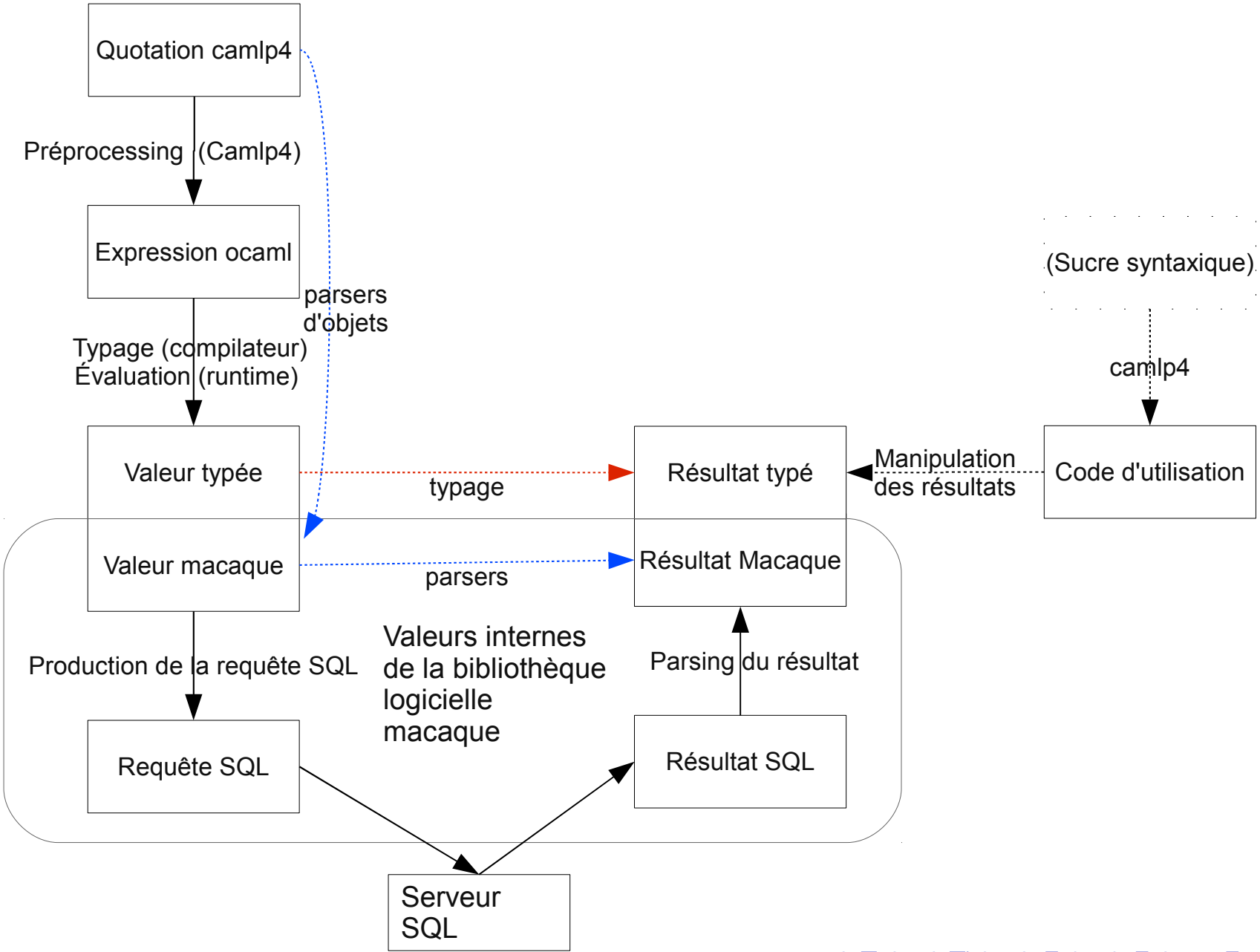
Exemple complet

```
let utilisateurs = <:table< mon_schema.utilisateurs (
  id integer NOT NULL,
  nom text NOT NULL,
  age integer,
  adresse text
) >>
let ciotadens = << u | u in $utilisateurs$; u.adresse = "La Ciotat" >>

let print_user u =
  let age_str = match Sql.getn u#age with
  | Some age -> Printf.sprintf "%ld ans" age
  | None -> "age inconnu" in
  Printf.printf "%s : %s\n" (Sql.get u#nom) age_str

let () =
  let dbh = PGOCaml.connect () in (* connexion au serveur SQL *)
  let result = Query.view dbh ciotadens in (* envoi de la requête *)
  print_endline "Liste des ciotadens :";
  List.iter print_user result;
  PGOCaml.close dbh
```

Schéma global



Quelques points intéressants, difficiles ou délicats

- la “nullabilité” des valeurs est calculée statiquement

Quelques points intéressants, difficiles ou délicats

- la “nullabilité” des valeurs est calculée statiquement
- forme de surcharge des opérateurs (polymorphisme d’inclusion)

Quelques points intéressants, difficiles ou délicats

- la “nullabilité” des valeurs est calculée statiquement
- forme de surcharge des opérateurs (polymorphisme d'inclusion)
- valeurs composées de “première classe”

```
# let produit_cartesien vue_a vue_b =
    << {a = elem_a; b = elem_b} |
        elem_a in $vue_a$; elem_b in $vue_b$ >>
;;
val produit_cartesien : 'a Sql.view -> 'b Sql.view ->
< a : <t : 'a row_t> Sql.t; b : <t : 'b row_t> Sql.t > Sql.view
```

Quelques points intéressants, difficiles ou délicats

- la "nullabilité" des valeurs est calculée statiquement
- forme de surcharge des opérateurs (polymorphisme d'inclusion)
- valeurs composées de "première classe"

```
# let produit_cartesien vue_a vue_b =
    << {a = elem_a; b = elem_b} |
        elem_a in $vue_a$; elem_b in $vue_b$ >>
;;
val produit_cartesien : 'a Sql.view -> 'b Sql.view ->
< a : <t : 'a row_t> Sql.t; b : <t : 'b row_t> Sql.t > Sql.view
```

- la sûreté dépend en partie de la génération de code par Camlp4 (cas GROUP BY détaillé dans l'article)

```
val field :
  <t : 'a #row_t; nul : non_nullable; ..> t ->
  string list unsafe ->
  ('a -> <t : 't; nul : 'n; ..> t) unsafe ->
  <t : 't; nul : 'n> t
```

Des sorties qui font peur

```
<< t | t in $recette$; t.nom = "omelette" >>
```

Des sorties qui font peur

```
<< t | t in $recette$; t.nom = "omelette" >>
```

```
let t = Sql.row (Sql.unsafe "t") (Sql.View.table Base.recette) in
Sql.view
(Sql.simple_select t)
[ ("t", (Sql.untyped_view recette) ]
[ (Sql.Op.( = )
    (Sql.field t (Sql.unsafe [ "nom" ]) (Sql.unsafe (fun t -> t#nom)))
    (Sql.Value.string "omelette") :>
  < t : Sql.bool_t > Sql.t) ]
```

Des sorties qui font peur

```
<< t | t in $recette$; t.nom = "omelette" >>
```

```
let t = Sql.row (Sql.unsafe "t") (Sql.View.table Base.recette) in
Sql.view
(Sql.simple_select t)
[ ("t", (Sql.untyped_view recette) ]
[ (Sql.Op.( = )
    (Sql.field t (Sql.unsafe [ "nom" ]) (Sql.unsafe (fun t -> t#nom)))
    (Sql.Value.string "omelette") :>
  < t : Sql.bool_t > Sql.t) ]
```

```
{ .. ; from = [("t", {descr = [("id", Non_nullable TInt32);
                               ("nom", Nullable (Some TString))];
                      producer = <fun>; record_parser = <fun>;
                      data = Table (None, "recette")})]; .. }
```

Des sorties qui font peur

```
<< t | t in $recette$; t.nom = "omelette" >>
```

```
let t = Sql.row (Sql.unsafe "t") (Sql.View.table Base.recette) in
Sql.view
(Sql.simple_select t)
[ ("t", (Sql.untyped_view recette) ]
[ (Sql.Op.( = )
    (Sql.field t (Sql.unsafe [ "nom" ]) (Sql.unsafe (fun t -> t#nom)))
    (Sql.Value.string "omelette") :>
  < t : Sql.bool_t > Sql.t) ]
```

```
{ .. ; from = [("t", {descr = [("id", Non_nullable TInt32);
                               ("nom", Nullable (Some TString))];
                      producer = <fun>; record_parser = <fun>;
                      data = Table (None, "recette")})]; .. }
```

```
object
  method id = (Atom (Int32 1l), Non_nullable TInt32)
  method nom = (Atom (String "omelette"), Nullable (Some TString))
end
```

Et du code qui donne envie ?

```
<< group {nb = count[u.nom]} by {age = u.age} | u in $utilisateurs$ >>
```

```
<< group {betise = u.nom} by {age = u.age} | u in $utilisateurs$ >>
```

Error : This expression has type Sql.grouped_row but an expression was expected of type

< nul : Sql.non_nullable; t : 'a #Sql.row_t; .. > Sql.t

Et du code qui donne envie ?

```
<< group {nb = count[u.nom]} by {age = u.age} | u in $utilisateurs$ >>
```

```
<< group {betise = u.nom} by {age = u.age} | u in $utilisateurs$ >>
```

Error : This expression has type Sql.grouped_row but an expression was expected of type

```
  < nul : Sql.non_nullable; t : 'a #Sql.row_t; .. > Sql.t
```

```
<:value< match $date$ with
         | null -> current_timestamp
         | t -> if $conf_delay$ then $insert_delay$ t else t >>
```

Limitations et perspectives

Les descriptions de table sont à modifier quand la base de donnée évolue.

Macaque repose sur PG'OCaml, donc n'est pas portable en l'état (PostgreSQL).

Macaque ne supporte pas (encore) l'ensemble du langage SQL.
Il faut ajouter soigneusement chaque fonctionnalité.
La méthode PG'OCaml n'a pas ce soucis.

Idées de concepts à prendre en compte :

- clés étrangères
- sérialisation des types utilisateurs
- procédures paramétrées

Macaque est un logiciel libre ($\simeq$ 3000 lignes de caml)

`http://macaque.forge.ocamlcore.org/`

Utilisateurs et commentaires appréciés.

Des questions ?