Macaque : Interrogation sûre et flexible de base de données depuis OCaml

Gabriel Scherer * — **Jérôme Vouillon** **

* *École Normale Supérieure Paris*
45 rue d'Ulm 75005 Paris
gabriel.scherer@ens.fr

** *CNRS et Université Paris 7*
175 Rue du Chevaleret 75013 Paris
Jerome.Vouillon@pps.jussieu.fr

RÉSUMÉ. *Macaque est une bibliothèque OCaml permettant d'interagir avec un serveur SQL. Elle permet de construire des requêtes vérifiées statiquement, de façon modulaire. Une extension Camlp4 apporte une syntaxe concrète inspirée des compréhensions. Des types fantômes sont utilisés pour encoder, en utilisant les types objets d'OCaml, certaines propriétés fines des valeurs SQL, comme la nullabilité.*

ABSTRACT. *Macaque is an OCaml library providing an interface to SQL databases. Modular, statically typed queries can be expressed. A Camlp4 syntax extension brings a concrete syntax using comprehensions. Subtle properties of SQL values, such as nullability, are encoded into the OCaml type system using phantom types.*

MOTS-CLÉS : *OCaml, base de donnée, typage, langages embarqués, macros, compréhensions*
KEYWORDS: *OCaml, databases, types, embedded languages, macros, comprehensions*

Introduction

Les bases de données sont des méthodes solides et reconnues de stockage d'information pour les besoins d'une application. Un programmeur qui voudrait s'en servir est cependant confronté à une situation

désagréable : il doit communiquer avec un programme externe (le serveur de base de données) en lui envoyant des requêtes en format texte, c'est-à-dire oublier, dans cette partie de son programme, tout le confort des données structurées et des moyens d'expression de son langage.

La méthode la plus flexible pour construire des requêtes SQL est la production d'une chaîne de caractères correspondant à la requête :

```
let interroge table predicat =
  "SELECT * FROM " ^ table ^ " WHERE " ^ predicat
```

Son inconvénient majeur et qu'elle transporte toutes les données sous forme de chaînes de caractères, qui n'apportent aucune information de typage, donc aucune vérification de correction (même syntaxique) à la compilation. En particulier, il y a facilement des problèmes de sécurité si des portions de la requête peuvent provenir d'un utilisateur malicieux du programme.

La méthode la plus sûre pour écrire des requêtes est de vérifier leur validité, au moment de la compilation, en interrogeant directement le serveur SQL, comme le fait le projet PG'OCaml [Jon] :

```
let interroge predicat =
  PGSQL(dbh) "select * from ma_table where $predicat"
```

Pour que cette vérification statique soit possible, la requête doit contenir assez d'information : le serveur PostgreSQL sait vérifier le type des données mais ne dispose ni d'un moteur d'inférence sophistiqué, ni d'une forme de polymorphisme. En particulier, on a dû préciser ici la table étudiée. On ne peut pas écrire de fonction générique `interroge`, qui fonctionne sur n'importe quelle table. On a donc perdu en flexibilité.

Quels sont les compromis acceptables ? Macaque[1] est un langage de requête, embarqué dans OCaml, qui se veut à la fois sûr et flexible. C'est une extension syntaxique couplée à une bibliothèque logicielle à l'interface fortement typée, qui permettent d'écrire des requêtes modulaires en conservant la sûreté à laquelle sont habitués les programmeurs OCaml. Macaque est un logiciel libre, disponible [SV09] sur le site OCamlForge.

1. MAcros for CAml QUEries

Nous présentons une vue d'ensemble de Macaque, ses principes généraux et un aperçu de ses fonctionnalités par des exemples (section 1). Nous donnons ensuite une vue d'ensemble de son architecture (section 2). Des transformations sont effectuées à plusieurs niveaux : pré-processeur, compilation (typage), transformation des requêtes pendant l'exécution, communication avec un serveur SQL externe... Elles illustrent bien la diversité des aspects à considérer lors de la conception d'un langage embarqué. Nous présentons enfin deux points particuliers qui illustrent bien les problématiques spécifiquement liées au langage SQL. Tout d'abord, l'encodage des types des valeurs SQL, utilisant des types fantômes (section 3). Ensuite, la construction `GROUP BY`, dont la traduction contient des subtilités, nécessaires à la vérification statique (section 4).

1. Principes et exemples

Macaque vise deux objectifs principaux :

flexibilité par la composabilité : pouvoir construire une requête à partir de composants existants

sécurité statique par le typage : minimiser les erreurs produites à l'exécution, en transposant les contraintes de correction d'une requête SQL dans le langage des types de OCaml, afin qu'elles soient validées statiquement.

Macaque est constitué d'une extension syntaxique Camlp4 (section 1.2), qui propose une syntaxe concrète pour décrire des requêtes ou fragments de requêtes, et d'une bibliothèque logicielle (un module OCaml nommé `Sql`), qui contient tout le traitement des requêtes. L'extension syntaxique traduit la syntaxe concrète en des appels à cette bibliothèque, dont l'interface typée garantit la sûreté de la plupart des manipulations.
Pour cela, on définit de nombreux types Caml correspondant aux différents constituants d'une requête SQL, mais le programmeur est amené à utiliser essentiellement les trois types suivants :

Valeurs

Macaque utilise un type différent du type OCaml `int` pour représenter les entiers SQL, en particulier parce qu'un entier SQL peut être indéfini (valeur NULL). Plus généralement, il y a un type spécifique pour les valeurs SQL, `'a Sql.t`, utilisant un paramètre fantôme. Ce paramètre est un type objet[2] décrivant la valeur. Dans les exemples de cette section, nous utiliserons des types de la forme `< t : 'a > Sql.t`, mais ce n'est qu'une simplification des types complets (présentés en section 3) qui contiennent des champs supplémentaires stockant des informations diverses. Des fonctions d'extraction permettent d'en obtenir des valeurs OCaml usuelles : par exemple, un entier SQL pouvant être indéfini produira une valeur de type `int32 option`.

Vues

Les vues représentent soit une table du serveur SQL, soit une sélection sur ces tables et sur d'autres vues définies préalablement. Le type d'une vue est de la forme `'a Sql.view`, où le paramètre est un type objet donnant les colonnes de la vue et leurs types, par exemple :
$< \text{nom} : < \text{t} : \text{string\_t} > \text{Sql.t}; \text{age} : < \text{t} : \text{int\_t} > \text{Sql.t} > \text{Sql.view}$

Requêtes

Les requêtes constituent les valeurs finales, qui seront envoyées au serveur SQL. Elles sont représentées par le type `'a Sql.query`, où `'a` représente la valeur OCaml qui sera renvoyée par Macaque après envoi de la requête : une vue de type `'a Sql.view` produira une `'a list query`, et les requêtes `INSERT`, `DELETE` et `UPDATE` correspondent à des `unit query`.

1.1. *Choix d'une syntaxe de compréhension pour les requêtes*

Nous avons choisi d'utiliser pour les requêtes de type `SELECT` une syntaxe basée sur les compréhensions.

---

2. Le type `< foo : 'a; bar : int >` est un type structurel qui décrit les objets ayant deux méthodes `foo` et `bar`, de type respectif `'a` et `int`.

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

Cette syntaxe, inspirée de la notation mathématique, est souvent familière aux utilisateurs de langages fonctionnels (Haskell, Erlang..), et a une signification proche de celle des requêtes de base de données : elle se décompose en une partie résultat (`{u.nom; ...}`), de générateurs (`u in ...`) et de gardes (`u.age >= 18`), qui correspondent respectivement aux parties SELECT, FROM et WHERE d'une requête SQL :

```
SELECT u.nom, u.adresse FROM utilisateurs AS u WHERE u.age >= 18
```

L'idée d'utiliser une syntaxe de compréhensions a été popularisée par l'étude formelle des compréhensions comme base d'un langage de programmation adapté aux bases de données [BLS+94], et mise en place par exemple dans le système de requêtes Kleisli [Won00].

1.2. *Utilisation du pré-processeur Camlp4*

Pour ajouter une syntaxe concrète à notre langage de compréhension embarqué, nous avons utilisé le pré-processeur Camlp4 [MdR94]. Camlp4 dispose d'un *parser* du langage OCaml qui est extensible : le concepteur d'un langage embarqué peut ajouter les constructions de son choix, s'il en spécifie la traduction vers le langage OCaml.

Camlp4 comporte en particulier un mécanisme de *quotations*, qui permet de restreindre la modification de la syntaxe à des fragments bien délimités (entre `<< .. >>` ou `<:foo< .. >>`), de façon standardisée et homogène entre différentes extensions. Macaque propose des quotations `value`, `view`, `select`, `insert`, `update` et `delete` correspondant aux valeurs, vues et requêtes. `<:view<` est la quotation par défaut, et peut donc se noter `<<`.

Au sein d'une quotation Camlp4, on peut utiliser (si l'auteur du parser de quotations l'a pris en compte) des *antiquotations*, qui sont

des fragments de code du langage hôte (OCaml), habituellement délimités par des signes `$`. Dans notre exemple précédent, le générateur `u in $utilisateurs$` fait référence à une variable `utilisateurs` déclarée côté OCaml, désignant une vue. Ce sont les antiquotations, permises à des endroits bien choisis, qui font la flexibilité de Macaque. On peut ainsi décrire une sélection dépendant d'un paramètre et s'effectuant sur une table elle-même en paramètre :

```
# let limite_age age_minimal t =
  << p | p in $t$; p.age > $age_minimal$ >> ;;
val limite_age :
  < t : 'a; .. > Sql.t ->
  (< age : < t : 'a > Sql.t; .. > as 'b) Sql.view ->
  'b Sql.view
```

Ce mécanisme ressemble aux "requêtes paramétrées" proposées par la plupart des serveurs SQL, mais avec plus de flexibilité (table variable), et surtout s'appuie directement sur le mécanisme d'abstraction du langage hôte, ce qui permet une meilleure intégration. Ainsi, c'est le typeur OCaml qui infère les types des paramètres.

Remarque

Le deuxième paramètre de la fonction `limite_age` est une valeur SQL, donc de type `.. Sql.t` : c'est au programmeur de construire une telle valeur à fournir à la fonction, en utilisant soit la quotation `value` (`<:value< 2 >>`), soit la fonction de conversion `Sql.Value.int32`. Si l'on veut passer directement un entier, on peut utiliser une antiquotation paramétrée `$int32:..$` qui insère implicitement la fonction de conversion :

```
let limite_age (age_minimal : int32) t =
  << p | p in $t$; p.age > $int32:age_minimal$ >>
```

1.3. *Panorama des principales fonctionnalités*

Jointures

Pour obtenir une sélection simultanément sur plusieurs tables/vues, on utilise plusieurs générateurs. Par exemple ici on construit un résultat à deux colonnes nommées `ingredient` et `recette` en faisant une triple jointure sur des tables du même nom et une table de correspondance :

```
<< {ingredient = i.nom; recette = r.nom} |
    i in $ingredient$; r in $recette$; l in $liste$;
    l.ingredient = i.id; l.recette = r.id >>
```

Valeurs composées

Macaque a une notion de *valeur composée* (n-uplets aux champs nommés). En particulier, les lignes des tables sont considérées comme des valeurs composées. Ce sont donc des valeurs comme les autres, que l'on peut placer dans une colonne d'une vue :

```
# 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
```

Cette fonctionnalité ajoute de l'homogénéité et de l'expressivité au langage SQL, qui propose quelques manipulations de types composés, mais de façon très *ad hoc*. Elle permet aussi de remplacer l'idiome SQL `ligne.*`, qui permet d'inclure tous les champs d'une ligne dans le résultat de la requête courante, mais n'est pas typable de façon satisfaisante en OCaml. En effet, une requête de la forme `SELECT a.*, b.* FROM a, b` demanderait la concaténation de deux types structurés (objets ou enregistrements), qui n'est pas exprimable dans le langage des types Caml. Certains auteurs de langages fonctionnels interagissant avec des bases de données ont choisi de partir d'un typage plus puissant (Ur/Web [Chl09] par exemple permet la concaténation de records), mais nous devons nous limiter au typage du langage hôte.

Les types composés ajoutent cependant une complexité sensible à notre implémentation : comme on ne peut pas reposer pour les traduire sur les types composés de SQL, trop peu expressifs (ils ne retiennent pas les noms des champs, par exemple), il faut passer par une phase d'aplatissage qui remplace chaque type composé par l'ensemble de ses champs, récursivement. Cette transformation ne peut pas être effectuée statiquement, car il faut savoir lesquels des champs sont des types composés, information à laquelle le pré-processeur n'a pas accès.

Filtrage de nullabilité

Un filtrage spécialisé permet de tester si une valeur vaut `NULL`, et sinon de l'utiliser avec le type "non nullable". Pour typer l'exemple, on introduit ici le champ `nul` du type `Sql.t`, que nous avions caché jusque là pour plus de clarté.

```
# let non_nullable_float x =
  <:value< match $x$ with null -> $float:nan$ | y -> y >>
val non_nullable_float :
  < nul : Sql.nullable; t : Sql.float_t; .. > Sql.t ->
  < nul : Sql.non_nullable; t : Sql.float_t > Sql.t
```

La traduction SQL est de la forme :

$$[\![\texttt{match}\ e_1\ \texttt{with null ->}\ e_2\ \texttt{|}\ v\ \texttt{->}\ e_3]\!] \longrightarrow \texttt{CASE WHEN (}[\![e_1]\!]\ \texttt{IS NULL) THEN}\ [\![e_2]\!]\ \texttt{ELSE}\ [\![e_3[v := e_1]]\!]\ \texttt{END}$$

Cette construction est la seule manière de passer d'une valeur de type nullable à une valeur non nullable. Elle pose par contre un léger problème d'implémentation, puisque dans le cas non nul elle effectue une liaison : le motif est un nom de variable, qui est lié à l'expression étudiée. Le langage SQL ne permet pas de lier des noms à des valeurs, et il est difficile de simuler cette fonctionnalité. Nous l'avons essayé, mais au vu de la complexité supplémentaire de l'implémentation (dans le cas par exemple où le filtrage rend une valeur composée) nous avons plutôt choisi d'effectuer une substitution ; c'est très simple à mettre en place[3], mais produit des expressions SQL avec potentiellement des calculs dupliqués. Nous pensons que cela ne produira pas en pratique de dégradation sensible des performances, car les utilisateurs adoptent en général un style proche du SQL et n'effectuent pas de calculs lourds au sein des requêtes.

Macaque propose aussi le simple test `if .. then .. else`.

$$[\![\texttt{if}\ e_1\ \texttt{then}\ e_2\ \texttt{else}\ e_3]\!] \longrightarrow \texttt{CASE WHEN}\ [\![e_1]\!]\ \texttt{THEN}\ [\![e_2]\!]\ \texttt{ELSE}\ [\![e_3]\!]\ \texttt{END}$$

3. on peut utiliser la β-réduction du langage hôte, à la manière HOAS [PE88]

Requêtes de modification

Nous avons dû choisir une syntaxe adaptée pour les opérations de modification, en essayant de conserver une certaine homogénéité avec les compréhensions de sélection.

```
<:insert< $utilisateurs$ :=
          {id = nextval $user_id_seq$; nom = "Theodule"} >>

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

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

<:delete< d in $meetings$ |
            jour in $calendrier$;
            d.date_rendu = jour.id;
            jour.periode = "vacances" >>
```

Tris, limites et groupes

Le traitement de la clause GROUP BY est détaillé en section 4.

```
<< u order by u.nom asc, u.age desc | u in $utilisateurs$ >>

<< u limit 2 offset 3 | u in $utilisateurs$ >>

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

Description de tables

L'utilisateur doit décrire la structure des tables de sa base de données à Macaque. Nous avons développé pour cela une syntaxe concrète de description, inspirée de la syntaxe SQL de création des tables.

```
let utilisateurs = <:table< my_db_schema.utilisateurs
( id integer NOT NULL,
  nom text NOT NULL,
  ville text,
  age integer NOT NULL ) >>
```

Pour éviter les problèmes de redondance, le module `Check` apporte une fonction de vérification des descriptions : elle utilise les capacités d'introspection des bases de données SQL pour vérifier (pendant l'exécution) que la structure décrite par l'utilisateur est cohérente avec celle présente dans la base de données. Cette fonction est présente dans la bibliothèque logicielle, et l'utilisateur a donc le plein contrôle de son utilisation ; un paramètre de ligne de commande (donné au préprocesseur de Macaque) permet d'insérer automatiquement un de ces tests à chaque description de table.

Communication avec le serveur SQL

Nous utilisons, pour la connexion avec le serveur SQL, les facilités de la bibliothèque PG'OCaml. Une fois qu'on a effectué la connexion (`PGOCaml.connect`), on donne la requête (ou la vue) au module `Query`, qui renvoie directement le résultat, sous forme d'une liste d'objets représentant les lignes de la vue, dont on peut extraire[4] les valeurs des colonnes pour les utiliser dans son application.

```
let parisiens = << u | u in $utilisateurs$; u.ville = "Paris" >>
let print_user u = Printf.printf "%s : %ld ans\n" u#!nom u#!age
let () =
  let dbh = PGOCaml.connect () in
  let result = Query.Simple.view dbh parisiens in
  print_endline "Liste des parisiens :";
  List.iter print_user result;
  PGOCaml.close dbh
```

2. Architecture de Macaque

Nous allons maintenant parler de ce que l'utilisateur ne voit pas : la machinerie qui lui permet d'exprimer ses requêtes et d'en utiliser les résultats.

4. Le sucre syntaxique `u#!nom` est détaillé en section 2.1.

2.1. *Transformations*

Un schéma des transformations qui ont lieu pendant le cycle de vie d'une requête SQL est donné par la figure 1.

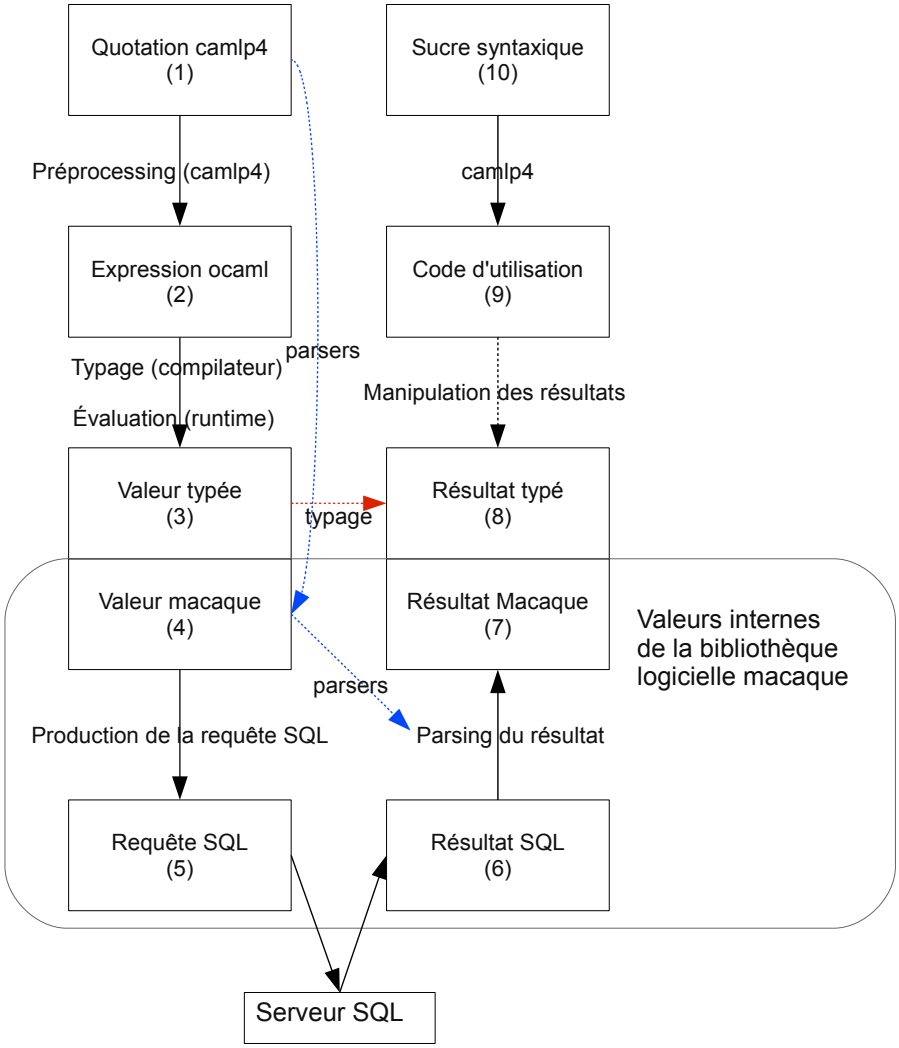

Figure 1 – Transformations successives d'une compréhension

Preprocessing, Typage

Un module Camlp4 se charge de traduire les quotations en code OCaml. Le code produit consiste essentiellement en une série d'appels à la bibliothèque logicielle de Macaque (le module `Sql`). Cette bibliothèque utilise des types abstraits expressifs et une interface restreinte pour imposer des contraintes de correction au code généré.

Cette partie est relativement "standard" : c'est une instance d'une méthodologie générale d'insertion d'un langage domaine-spécifique dans un langage hôte fonctionnel typé [LM99].

Le module Camlp4 abrite cependant une opération supplémentaire : il doit "compiler" les compréhensions vers un arbre syntaxique SQL. En effet, la bibliothèque logicielle utilise en interne une représentation proche du langage SQL, et non du langage des compréhensions. C'est donc le module Camlp4 qui porte la logique applicative faisant la conversion d'un langage à l'autre. Nous avons fait ce choix dans l'espoir de pouvoir réutiliser la bibliothèque logicielle avec d'autres syntaxes concrètes de requêtes, comme par exemple une syntaxe imitant fidèlement le langage SQL.

Évaluation du code généré

L'interface du module `Sql` joue un rôle crucial pendant la compilation, puisque c'est dans cette interface que sont encodés la plupart des critères de validité des requêtes[5]. Pendant l'exécution du programme, c'est dans la partie privée du module `Sql` que se trouve toute la logique applicative de Macaque.

Production de la requête

La production de la requête en format texte a lieu lorsque l'utilisateur demande l'exécution de la requête SQL. Elle ne pose pas de difficultés, puisque la représentation interne du module est déjà essentiellement un arbre syntaxique d'un sous-ensemble du langage SQL.

C'est par contre à ce moment que sont appliquées un certain nombre de transformations importantes. En effet, le préprocesseur n'a en général accès qu'à des fragments de requêtes. Certaines opérations

5. Dans des cas plus avancés, le typage seul ne suffit pas à garantir la correction et il faut jouer sur le code généré par Camlp4 ; cf. la section 4 concernant la construction GROUP BY.

(en particulier l'aplatissage lié aux types composés, décrit en 1.3) sont globales par essence et ne peuvent s'effectuer localement sur des fragments de requêtes.

Envoi de la requête au serveur

Pour communiquer avec le serveur SQL, nous utilisons la bibliothèque PG'OCaml. Celle-ci propose en effet une interface de haut niveau (évoquée en introduction, qui sera comparée à Macaque dans la section 5) mais aussi une interface de bas niveau, qui permet la création d'une connexion au serveur SQL, l'envoi de requête et la récupération des résultats en format texte.

L'avantage principal de ce choix est une compatibilité forte entre Macaque et PG'OCaml : comme nous utilisons les *connection handlers* de cette bibliothèque, il est possible de faire passer tour à tour des requêtes Macaque et des requêtes PG'OCaml par la même connexion au serveur SQL. PG'OCaml étant aujourd'hui la solution de requêtes SQL sûres la plus utilisée, il est important d'apporter une voie de migration en douceur : on peut convertir un projet requête par requête. En particulier, Macaque a été conçu suite à un besoin ressenti par les auteurs du logiciel Ocsigen (un *framework web* en OCaml [BVY09], auquel participe le second auteur), qui utilise actuellement PG'OCaml.

Parsing du résultat

Le serveur SQL renvoie les résultat d'une requête sous forme textuelle, une chaîne de caractères pour chaque champ de chaque ligne du résultat. Macaque construit à partir de ce résultat une liste d'objets OCaml (un par ligne), dont chaque méthode fournit la valeur (de type `Sql.t`) associée à une colonne. L'utilisateur ne verra donc que les formes typées (une requête typée en entrée, une liste d'objets typés en sortie de la requête) ; pour cela, Macaque doit parser les données textuelles pour produire les valeurs OCaml correspondantes.

Pour pouvoir choisir pendant l'exécution le parser approprié à chaque champ, Macaque transporte avec chaque fragment de requête SQL de type `'a Sql.t` une valeur décrivant son type. En d'autres termes, la représentation privée des valeurs dans le module `Sql` utilise une forme de typage dynamique.

En général, le type dynamique suffit à construire le parseur correspondant (`TInt` $\longrightarrow$ `int_of_string`). Cependant, les valeurs composées (représentant un ensemble de champs) posent problème : elles sont décrites par une liste de la forme `TRow [("nom", TString); ("age", TInt)]`, qui ne permet pas de construire à l'exécution un objet à deux méthodes `nom` et `age`. Nous utilisons donc une "astuce syntaxique" : c'est le préprocesseur qui construit le parser. C'est possible car les seules valeurs composées dans une requête sont celles construites explicitement par l'utilisateur, en utilisant une syntaxe concrète de la forme `{nom = ...; age = ...}`. Le préprocesseur a donc accès au nom des champs, et peut ajouter au code généré un morceau de programme décrivant le parser :

```
fun input -> object method nom = parser_of_value nom input.(0)
                            method age = parser_of_value age input.(1) end
```

Ce parser appelle récursivement les parseurs correspondants aux deux champs de l'objets, fournis par les variables `nom` et `age` définies en amont dans le code généré.

Cette difficulté semble anecdotique, mais elle illustre un aspect important du traitement de langages embarqués au sein d'un langage typé : quand le langage hôte n'est pas assez expressif pour exprimer certaines transformations, on peut souvent recourir à des manipulations syntaxiques au moment de la génération du code.

Manipulation des résultats, sucre syntaxique

L'utilisateur reçoit de Macaque une liste d'objets, dont chaque méthode représente un champ de la table, et contient une valeur de type `Sql.t`. Il peut en extraire une valeur OCaml en utilisant les fonctions `Sql.getn`[6] et `Sql.get`, selon que la valeur peut être `NULL` (auquel cas on récupère un type `option`) ou non :

```
let print_user u =
  let print_age = function
  | None -> "age inconnu"
  | Some n -> Printf.sprintf "%d ans" n in
  Printf.printf "%s : %s, %s\n"
    (Sql.get u#nom) (Sql.get u#metier) (print_age (Sql.getn u#age))
```

6. `getn` comme "get nullable"

L'inconvénient de cette pratique est la difficulté des accès à des champs composés imbriqués, qui sont de la forme `Sql.get (Sql.get user#conf)#login`. Nous avons ajouté une très fine couche de sucre syntaxique[7] : `a#!b` est traduit en `(Sql.get a#b)`, et `a#?b` en `(Sql.getn a#b)` : on peut alors écrire `user#!conf#!login`.

2.2. *Importance de l'interface fortement typée*

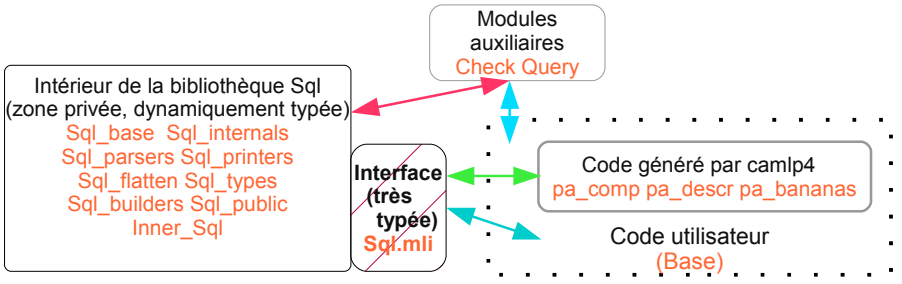

Figure 2 – Interfaces entre le code utilisateur et les composants logiciels (publics et privés) de Macaque

La communication entre les différentes parties de Macaque, le code utilisateur et l'interface du module `Sql` est schématisée dans la figure 2.

La différence principale entre `Sql` et une bibliothèque logicielle OCaml classique est la présence de code "ami" *en dehors* de l'interface : le code généré par Camlp4 partage avec la bibliothèque `Sql` certaines informations sur la manipulation des valeurs de la bibliothèque `Sql`, qui ne peuvent être exprimées par le typage de l'interface. Par exemple, la traduction de la syntaxe concrète construisant une ligne de plusieurs champs transmet à la fois une information de typage sur l'objet correspondant (de la forme `< nom : ...; id : ... > Sql.t`) et une valeur runtime correspondant au type dynamique, utilisé pour la génération de la requête en format texte : `[("nom", ...); ("id", ...)]`.

7. C'est la seule extension de la grammaire OCaml en tant que telle (tout le reste passant par le mécanisme standard des quotations), et elle est séparée et optionnelle.

Le type de la fonction correspondante est de la forme `'obj parser -> (string * ...) list -> ...`, avec l'hypothèse implicite que le type `'obj` est cohérent avec les valeurs de la liste associative qui l'accompagne ; cet invariant est respecté par notre module Camlp4, mais ne le serait pas nécessairement par un utilisateur disposant de cette fonction.

Il n'y a pas de solution véritablement satisfaisante à ce problème de sécurité : le code généré par Camlp4 a le même statut, pour le compilateur OCaml, que le code utilisateur ; la fonction *doit* être publiée dans l'interface du module. On doit donc se contenter d'une convention, en précisant dans la documentation de l'interface que cette fonction ne doit pas être utilisée, comme c'est une pratique courante pour les bibliothèques OCaml. Cette convention est rendue explicite par une annotation, le type `'a unsafe` :

type 'a unsafe
val unsafe : 'a −> 'a unsafe

val example_of_unsafe_fun : 'obj −> (string * ...) list unsafe −> ...

Tous les paramètres des fonctions exposées par l'interface qui reposent sur un invariant implicite sont marqués du type `unsafe`, et le code Camlp4 génère des appels à la fonction `Sql.unsafe`. Même si l'utilisateur peut toujours l'imiter et contourner la recommandation s'il le souhaite, le passage d'une convention présente dans la documentation du module à une convention signalée concrètement dans l'interface nous semble être un progrès sensible.

2.3. *Flexibilité des opérateurs*

Les quotations paramétrées apportent une syntaxe adaptée à la conversion d'une valeur OCaml en la valeur SQL lui correspondant : `<:value< 1 + $int32:2l$ >>`. Cette possibilité de paramétrisation de la forme `$foo: ... $` est une fonctionnalitée de Camlp4 : les auteurs d'une extension doivent seulement décrire les paramètres utilisés et leur comportement.

Plutôt que d'implémenter la gestion de ces paramètres directement dans l'extension syntaxique, nous avons transféré cette responsabilité à la bibliothèque logicielle `Sql` : une antiquotation paramétrée `$foo:bar$` génère le code `Sql.Value.foo bar`. Cette technique est utilisé à plusieurs reprises ; par exemple, les appels de fonction utilisent le module `Sql.Op` : `<:value< foo $bar$ >>` devient `Sql.Op.foo bar`. C'est aussi le cas des opérateurs infixes : `Sql.Op.(+)`.

Cette politique simplifie l'extension syntaxique, et réduit le temps d'adaptation de l'utilisateur car les différentes antiquotations sont directement documentées dans l'interface du module `Sql`.

3. Type des valeurs SQL

Le type des valeurs SQL, `'a Sql.t`, est un type qui permet de transmettre des informations riches, nécessaires à la sûreté des opérations sur ces valeurs. Le paramètre `'a` est un type fantôme : il sert uniquement à transporter des informations de typage, et n'indique pas la présence d'une valeur runtime. En particulier, l'interface du module `Sql` impose (par la restriction des constructeurs disponibles pour ce type abstrait) qu'il soit toujours un type objet, mais les valeurs de type `Sql.t` ne contiennent pas d'objet OCaml. Ces types objets contiennent au plus trois champs :

- `nul :` toujours présent, exprime la nullabilité de la valeur (3.1) ;
- `get :` optionnel, indique que la valeur est "immédiate" (3.2) ;
- `t :` toujours présent, donne les informations sur le type SQL de la valeur (3.4).

Le type `t` est lui-même un type objet. En particulier, son champ `typ` désigne le type OCaml correspondant à la valeur SQL représentée (le type de valeur OCaml que l'on peut récupérer en résultat de la requête). Par exemple, une chaîne de caractères pourrait avoir le type `< t : < typ : string > > Sql.t`.

3.1. *Nullabilité*

Le champ de `nul` peut être de deux types, `nullable` et `non_nullable`, qui sont deux types abstraits (et vides) du module Sql servant de témoins de nullabilité. On peut fixer la nullabilité d'un paramètre ou d'une valeur de retour d'une fonction, indiquer que la nullabilité est préservée par une opération (en partageant une variable de type entre le paramètre et le retour), et indiquer que la nullabilité d'un paramètre n'a aucune importance (en utilisant _ ou une variable de type non liée) :

```
val nullable :
  < t : 't; nul : non_nullable; .. > t ->
  < t : 't; nul : nullable > t
val not : < t : bool_t; nul : 'n; .. > t -> < t : bool_t; nul : 'n; > t
```

Une limitation de cette technique est qu'elle ne peut pas exprimer par le typage des propriétés telles que "cet opérateur prend des opérandes de n'importe quelle nullabilité, et son résultat est nullable si l'un de ses paramètres l'est". Il faudrait un opérateur de "maximum" au niveau des types, dont Caml ne dispose pas. À la place, on déclare un opérateur prenant deux opérandes de même nullabilité (par le partage d'une variable de typage), et si l'utilisateur veut utiliser une variable nullable et une non nullable, il convertit explicitement la deuxième avec l'opérateur `nullable` ci-dessus.

Une dernière utilisation légèrement plus surprenante du champ `nul` est la déclaration de valeurs nullable ou non nullable suivant le contexte, à l'aide de polymorphisme :

```
val int32 : int32 -> < t : int32_t; get : unit; nul : _ > t
```

Si l'on avait simplement choisi le type `non_nullable`, il aurait fallu écrire par exemple `<:value< nullable 1 + $n$ >>` dans le cas où `n` est nullable.

3.2. *Valeurs immédiates*

Macaque manipule deux sortes de valeurs SQL : les expressions, qui sont des fragments de requêtes qui seront envoyés au serveur SQL, et

les *valeurs immédiates*, qui sont le résultat des requêtes exécutées par le serveur. Elles ont toutes deux le type `'a Sql.t`. Cette uniformité a permis une conception simple et fiable des types des opérations SQL. En particulier, on peut directement réutiliser le résultat d'une requête SQL pour en construire une nouvelle, ce qui augmente nettement la composabilité des requêtes.

Pour utiliser les résultats de la requête, on dispose de fonctions d'*extraction*, qui renvoient la valeur OCaml correspondant au résultat. Elles opèrent sur les valeurs immédiates, mais n'ont pas de sens pour des expressions pas encore évaluées. On ajoute donc un champ `get` au type fantôme, pour effectuer la distinction : la valeur est immédiate si et seulement si le champ est présent. Les fonctions d'extraction ont donc le type suivant :

```
val get :
  < get : _; nul : non_nullable; t : < typ: 't; .. > > t -> 't
val getn :
  < get : _; nul : nullable; t : < typ : 't; .. > > t -> 't option
```

Les fonctions de construction de requête produisent des expressions non évaluées. On s'assure donc que leur type de retour ne contient pas le champ `get` :

```
val is_null :
  < nul : nullable; .. > t -> < t : bool_t; nul : non_nullable > t
```

3.3. *Lisibilité des types*

Le fonctionnement des champs `get` et `nul` est différent : le champ `get` est présent ou absent, alors que le champ `nul` est toujours présent, valant soit "vrai" (`nullable`) soit "faux" (`non_nullable`). On aurait pu choisir une interface homogène, avec les deux champs toujours présents, de valeur `true_t` ou `false_t`, mais l'absence du champ `get` raccourcit la description des expressions SQL, ce qui rend plus lisible les messages d'erreur de typage.

3.4. *Informations statiques de typage*

C'est la partie peut-être la plus croustillante, et sans doute la moins délicate conceptuellement, du type `Sql.t`. Elle n'est pas motivée par des raisons fondamentales de sûreté mais pour des considérations de facilité d'utilisation, liée à la surcharge d'opérateurs.

En SQL, il y a une surcharge des opérateurs : on peut utiliser le même opérateur `+` pour tous les types numériques. Et c'est assez heureux, parce que le SQL propose une grande quantité de types numériques : `int`, `smallint`, `bigint`, `real`, `double`... La traduction naturelle en OCaml serait de fournir une fonction d'addition différente pour chaque variante, mais ce n'est pas très gentil pour l'utilisateur.

Notre traitement de ce problème est très similaire à celui des valeurs immédiates : pour indiquer qu'une valeur "supporte l'opération d'accès", on a rajouté un champ `get` au type objet ; pour indiquer que le type "supporte l'addition", on lui ajoute un champ `numeric` :

```
val int32 :
  int32 -> < t : < typ : int32; numeric : unit > get : unit; nul : _ > t
```

Pour autoriser une opération arithmétique, il suffit alors de vérifier la présence de ce champ. On peut imaginer le même genre de marqueur pour les types "comparables" (qui supportent les opérations de comparaison, `min`, `max`, etc.), mais nous n'en avons pas eu besoin en pratique.

```
# let somme a b = <:value< $a$ + $b$ >> ;;
val somme : < nul : 'a; t : #Sql.numeric_t as 'b; .. > Sql.t ->
              < nul : 'a; t : 'b; .. > Sql.t -> < nul : 'a; t : 'b > Sql.t

# ( <:value< 1. + 2. >>, <:value< 1L + 2L >> ) ;;
- : < nul : 'b; t : Sql.float_t > Sql.t * < nul : 'c; t : Sql.int64_t > Sql.t
```

Le champ `t` du type `Sql.t` est donc un type objet contenant éventuellement ces marqueurs de surcharge, en plus du champ `typ` décrivant la valeur OCaml associée à cette valeur SQL. On peut imaginer rajouter d'autres informations de typage par la suite ; leur point commun est qu'elles concernent l'ensemble des valeurs SQL d'un type donnée, et pas des informations concernant une valeur particulière (est-elle nullable, immédiate, etc.). Leur factorisation au sein d'un sous-

type fantôme permet d'alléger les types manipulés, car la plupart des opérations n'accèdent pas au champ `t` et se contentent de propager son type.

4. Traitement des clauses GROUP BY

La traduction des clauses GROUP BY a été une partie délicate de la conception de Macaque. Sa sémantique côté SQL est compliquée, et, pour assurer la sûreté, le typage fort du module `Sql` ne suffit pas : il a fallu utiliser la génération de code pour garantir des propriétés supplémentaires de sûreté.

4.1. *GROUP BY en SQL*

La clause GROUP BY permet de moduler une requête SELECT pour regrouper les lignes du résultat selon la valeur d'une ou plusieurs de ses colonnes.

Imaginons par exemple une table SQL `ingredients` représentant l'ensemble des ingrédients présents dans un livre de cuisine, disposant des colonnes suivantes :

nom de l'ingrédient

recette dans laquelle il est utilisé

poids en grammes, de l'ingrédient pour la recette considérée

Avec cette représentation, un ingrédient présent dans deux recettes différentes sera présent deux fois, dans deux lignes différentes de la table. On peut s'intéresser aux ingrédients sans afficher les doublons, avec la requête suivante :

```
SELECT nom FROM ingredients GROUP BY nom
```

La clause GROUP BY considère l'ensemble des lignes ayant le même `nom`, et renvoie un résultat unique pour toutes ces lignes[8]. On

8. On peut considérer qu'on renvoie le quotient de l'ensemble des lignes par la relation d'équivalence "a le même nom".

peut aussi effectuer des opérations sur cet ensemble de lignes en utilisant des fonctions d'*agrégat*. Par exemple, on peut vouloir récupérer, avec chaque ingrédient, la quantité maximale nécessaire pour faire l'une des recettes du livre :

SELECT nom, **MAX**(poids) **FROM** ingredients **GROUP BY** nom

On utilise ici l'agrégateur MAX, qui renvoie, pour chaque ensemble de lignes groupées, la valeur maximale de la colonne demandée, ou plus généralement d'une expression dépendant des colonnes : `MAX(2 * poids)` est valide. On peut aussi par exemple compter le nombre d'ingrédients utilisés par chaque recette :

SELECT recette, **COUNT**(nom) **FROM** ingredients **GROUP BY** recette

Par contre, la requête suivante n'est pas bien formée, et le serveur SQL renvoie une erreur :

SELECT nom, recette **FROM** ingredients **GROUP BY** nom
ERROR: **column** "recette" must appear **in** the **GROUP BY** clause
or be used **in** an aggregate function

En effet, il n'y a pas en général une unique recette pour chaque nom. Pour qu'une requête ait du sens, il faut que les résultats demandés soient des expressions ne dépendant pas du choix d'une ligne en particulier dans le groupe (autrement dit, que ce soit des expressions constantes sur la classe d'équivalence). La "règle du GROUP BY" imposée par SQL exige que les expressions demandées soient l'une des expressions suivant le "GROUP BY" (qui par définition sont constantes au sein de chaque groupe), ou bien un agrégateur qui renvoie une valeur pour l'ensemble du groupe. C'est une approximation conservatrice : la requête suivante respecte les classes d'équivalences mais est refusée :

SELECT poids + 1 **FROM** ingredients **GROUP BY** poids $*$ 2;

4.2. *GROUP BY dans Macaque*

La difficulté de la traduction des clauses GROUP BY est l'impératif de vérification statique : il faut s'assurer que l'utilisateur ne rencontrera jamais, à l'exécution, un message d'erreur SQL provoqué par une requête mal formée. Il faut donc trouver une traduction qui impose le

respect de la “règle du GROUP BY”, et qui soit vérifiée statiquement par le compilateur OCaml.

Voici un exemple de requête GROUP BY incorrecte :

```
<< group {invalide = t.nom} by {recette = t.recette} |
    t in $ingredients$ >>
```

L'expression est séparée en deux parties : les expressions groupantes après le `by`, et les résultats supplémentaires directement après le `group`. On renvoie l'union de ces deux ensembles de champs : ici la vue a deux colonnes `recette` et `invalide`.

L'idée clé est la suivante : on empêche l'utilisateur d'utiliser, dans les champs résultats, des expressions violant la “règle du GROUP BY” en jouant sur la portée de ces expressions dans le code OCaml généré : les expressions groupantes sont évaluées sous la portée de la variable `t` représentant la ligne de la vue `ingredients`, et les résultats supplémentaires *hors* de cette portée : l'utilisation d'une colonne non constante passant nécessairement par une référence à son nom de ligne `t`, elle sera rejetée à la compilation.

On peut imaginer la sortie OCaml suivante (les accès aux colonnes restant à traduire) :

```
let recette =
  let t = Sql.row ingredients in <:value< t.recette >> in
let invalide = <:value< t.nom >> in
Sql.group ... (* reste de la comprehension *)
```

Avec cette sortie, la portée de la variable `t` est locale à la déclaration `let recette = ... in`, et ne s'étend donc pas à l'expression `<:value< t.nom >>`, dont la traduction provoquera donc un message d'erreur du compilateur (“Unbound value t”).

Cela ne suffit pas tout à fait, car il pourrait y avoir une variable `t` dans la portée globale de cette expression, par exemple déclarée par l'utilisateur. Plutôt que d'essayer de *retirer* la variable `t` de la portée, il faut l'*écraser* par une définition dont on est certain qu'elle provoquera une erreur à la compilation, par exemple :

```
let t = Sql.row ingredients in
let recette = ... in
let t = () in
let invalide = Sql.field t ... in
Sql.group ...
```

Ici, la liaison de `t` est écrasée et son type devient `unit`, ce qui provoque une erreur à la compilation, puisque toute utilisation de `t` en tant que table dans un résultat supplémentaire impose un type de ligne (différent de `unit`). Plutôt que `()`, notre implémentation utilise en fait une valeur du module `Sql`, du type abstrait `Sql.grouped_row`, pour obtenir des messages d'erreur plus lisibles.

On a donc empêché l'utilisation de la variable `t` dans les résultats supplémentaires. Ils ont par contre toujours accès aux expressions groupantes (puisqu'elles sont déclarées en amont dans le code généré), ce qui respecte la règle du GROUP BY :

```
<< group {double_poids = p * 2} by {p = t.poids} | t in $ingredients$ >>
```

On peut aussi effectuer des groupements dans des compréhensions à plusieurs générateurs (et éventuellement grouper selon plusieurs colonnes). Dans ce cas, toutes les tables liées par la requête sont masquées. Le traitement de GROUP BY est donc "non local", dans le sens où il demande de maintenir un environnement décrivant les lignes liées par des générateurs. Cela reste une transformation locale à la vue définie par la compréhension groupante : on ne considère que ses générateurs, qui ne dépendent pas de ses composants ou de son utilisation future. Cette opération sur le contexte de la compréhension est à rapprocher du traitement de GROUP BY dans [JW07].

La gestion de la portée doit encore être affinée pour accepter les accumulateurs. Voici une requête utilisant un accumulateur :

```
<< group {nb_ingr = count[t.nom]} by {recette = t.recette} |
    t in $ingredients$ >>
```

Avec la stratégie présentée ci-dessus, cette requête serait rejetée puisqu'elle utilise la ligne `t` dans les résultats supplémentaires. C'est justement pour corriger ce problème que nous imposons une syntaxe spéciale, utilisant des crochets (`[t.nom]`), aux paramètres des utilisateurs : pendant le parsing des résultats supplémentaires, Macaque

récolte les expressions entre crochets, et les déclare en amont, avant de masquer la variable `t` :

```
let t = Sql.row ingredients in
let recette = <:value< t.recette >> in
let accum_1 = Sql.accum <:value< t.nom >> in
let t = () in
let nb_ingr = Sql.Op.count (Sql.group_of_accum accum_1) in
Sql.group ...
```

Les deux marqueurs `Sql.accum` et `Sql.group_of_accum` servent à empêcher une utilisation invalide des accumulateurs et fonctions d'agrégat (comme `count`). Elles injectent les valeurs dans deux types abstraits, `'a Sql.accum` et `'a Sql.group`, qui sont en interne égales à `'a Sql.t` mais ne lui sont pas unifiables (hors du module Sql.t). Seules les fonctions d'agrégation, de type `'a group -> ...`, peuvent donc utiliser les accumulateurs.

Il reste cependant possible de capturer une de ces valeurs, par exemple dans une référence, pour la réutiliser plus tard dans une autre compréhension reproduisant le même contexte de typage. Macaque ne peut pas empêcher un utilisateur de se tirer volontairement (et malicieusement) dans le pied[9] !

5. Travaux connexes

La plupart des langages de programmation se sont confrontés au problème de l'intégration de bases de données relationnelles. Un grand nombre de programmeurs ont découvert le SQL dans le contexte de la programmation de sites webs dynamiques, généralement depuis PHP, langage qui apporte un support minimal du SQL : historiquement, il a popularisé le modèle de construction de requêtes par concaténation des chaînes. Cette méthode peu sûre, associée aux faibles exigences du langage en matière de typage, est au centre de multiples problèmes de sécurité (les failles par "injection SQL") qui ont sensibilisé les programmeurs à la question de la correction des requêtes.

9. C'est la même problématique pour les fonctions de la forme `with_open_file_in : file_name -> (in_channel -> 'a) -> 'a`.

Un bon exemple de sous-langage spécialisé forçant la production de requêtes syntaxiquement bien formées est **SchemeQL** [WSG02]. Développé pour le langage Scheme[10], SchemeQL reprend exactement la syntaxe du langage SQL, ce qui a l'avantage de faciliter l'adaptation d'un programmeur ayant une expérience du SQL, au détriment de la concision et de la (toute relative) homogénéité avec le reste du langage hôte permise par les compréhensions. Utilisant un langage à typage dynamique, SchemeQL ne met en place aucun typage statique, qui est à nos yeux l'intérêt principal de Macaque.

Haskell/DB [LM99] est plus proche de Macaque, car c'est une intégration d'un langage de base de données dans Haskell, un langage fonctionnel statiquement typé, plus proche de OCaml. Les types fantômes y sont utilisés pour transporter les informations de typage. Le typage des valeurs y est cependant moins travaillé, par exemple il n'y a pas de gestion statique de la nullabilité.

Pour typer les lignes d'une table et leur champ, Haskell/DB utilise Trex, une extension non standard du langage Haskell (et non supportée par l'implémentation *de facto* standard, GHC) ; cette particularité nuit à son utilisation par les programmeurs Haskell. Nous profitons des types objets de OCaml, qui apportent des fonctionnalités similaires.

Enfin, Haskell/DB n'utilise pas d'extension syntaxique, mais réutilise la syntaxe spécifique des monades (*do-notation*) dans le langage Haskell, qui n'est pas exactement une syntaxe de compréhensions. Nous pensons qu'une syntaxe concrète est plus agréable à utiliser, et nous avons profité de la génération de code pour un traitement sûr du GROUP BY.

PG'OCaml [Jon] est actuellement le projet le plus utilisé pour communiquer avec un serveur SQL depuis OCaml sans abandonner typage et sûreté. Comme précisé en 2.1, Macaque repose sur la couche bas niveau de PG'OCaml (une implémentation native en OCaml du protocole de communication du serveur PostgreSQL), ce qui lui permet d'assurer une forte compatibilité : il est possible de mélanger à volonté des requêtes conçues par PG'OCaml et par Macaque.

---

10. Les langages de la famille Lisp sont des hôtes populaires de sous-langages spécialisés, grâce à leurs facilités d'extension syntaxique (*macros*).

La couche haut niveau de PG'OCaml est très différente. Elle repose sur une idée simple : utiliser les capacités de vérification du serveur SQL, au moment de la compilation. C'est la grande force de PG'OCaml : ses requêtes sont très fiables car elles sont vérifiées directement par le serveur SQL. C'est aussi sa grande faiblesse, puisque cela impose que les requêtes définies statiquement soient suffisamment complètes pour être acceptées par la méthode relativement simpliste de vérification du serveur SQL, qui permet la paramétrisation des constantes et valeurs SQL, mais empêche la modularité et composabilité des requêtes. Cela demande aussi que le serveur SQL soit accessible pendant la compilation du programme utilisateur, ce qui peut être une contrainte désagréable quand les serveurs de développement et en production sont différents.

D'un point de vue d'ingénierie logicielle, l'approche de PG'OCaml a aussi un avantage certain : PG'OCaml ne gère explicitement aucune des fonctionnalités de SQL, il se contente de passer les requêtes au serveur SQL. À l'inverse, ajouter le support d'une fonctionnalité SQL dans Macaque demande un certain effort de spécification, un choix prudent des types des opérations, etc. En pratique, nous supportons actuellement un sous-ensemble relativement limité (mais qui va en s'élargissant) du langage SQL, alors que PG'OCaml a accès, "gratuitement", à l'ensemble du langage. Il est irréaliste d'espérer supporter un jour la totalité du langage SQL, et la compatibilité avec PG'OCaml apporte donc une solution de repli : si un utilisateur a localement besoin d'une fonctionnalité SQL avancée, il peut insérer une requête PG'OCaml au sein d'une application Macaque.

Une dernière limitation importante liée à PG'OCaml est la dépendance à un serveur SQL particulier, PostgreSQL. PG'OCaml repose sur les fonctionnalités "avancées" de PostgreSQL et utilise le protocole de communication natif de ce serveur, donc n'est pas portable à d'autres implémentations populaires de serveur SQL (par exemple MySQL). Macaque, qui utilise cette bibliothèque, hérite de la même restriction ; mais comme il n'a en fait pas l'usage des spécificités de PostgreSQL, il serait simple de le porter vers un autre serveur SQL. Un avantage de PostgreSQL est sa relativement bonne capacité d'optimisation : Macaque n'effectue aucun post-traitement des requêtes, plutôt

naïves, correspondant aux compréhensions, et repose à la place sur le travail d'optimisation du backend du serveur SQL.

Conclusion

Avant tout, Macaque constitue un témoignage de l'efficacité de la "recette" proposée par [LM99] : pour bien intégrer un sous-langage spécialisé dans un langage fonctionnel statiquement typé, il faut une bibliothèque logicielle avec une interface fortement typée, qui exprime par des informations de typage les contraintes de correction ; les types fantômes sont très utiles pour cela.

On peut aussi remarquer que nous avons pu accommoder le système de typage du langage OCaml pour obtenir les vérifications désirées. Nous sommes sensibles à la séduction des langages de types expressifs (GADT, *dependent types*...), mais quand on se force à rester dans le cadre des types OCaml, on peut en fait très souvent trouver chaussure à son pied.

Enfin, nous avons été surpris de constater l'utilité incontestable de la phase de génération syntaxique. Ce n'est pas une simple syntaxe de surface, mais aussi la possibilité de générer un code OCaml non trivial pour apporter certaines contraintes, non réalisables directement par des opérations typées au sein du langage. C'est à nos yeux l'enseignement le plus inattendu de Macaque.

Références

[BLS+94] Peter Buneman, Leonid Libkin, Dan Suciu, Val Tannen, and Limsoon Wong. Comprehension syntax. *SIGMOD Record*, 23 :87–96, 1994.

[BVY09] Vincent Balat, Jérôme Vouillon, and Boris Yakobowski. Experience report : Ocsigen, a web programming framework. In *ICFP*, pages 311–316, 2009.

[Chl09] Adam Chlipala. *The Ur/Web Manual*, October 2009. Available from : `http://www.impredicative.com/ur/manual.pdf`.

[Jon] Richard Jones. PG'OCaml. Available from : `http://pgocaml.berlios.de/`.

[JW07] Simon Peyton Jones and Philip Wadler. Comprehensive comprehensions. In *Haskell '07 : Proceedings of the ACM SIGPLAN workshop on Haskell workshop*, pages 61–72, 2007.

[LM99] Daan Leijen and Erik Meijer. Domain specific embedded compilers. In *DSL'99 : Proceedings of the 2nd conference on Conference on Domain-Specific Languages*, pages 9–9, 1999.

[MdR94] Michel Mauny and Daniel de Rauglaudre. A complete and realistic implementation of quotations for ML. In *Workshop on ML and its Applications*, June 1994.

[PE88] F. Pfenning and C. Elliot. Higher-order abstract syntax. *SIGPLAN Not.*, 23(7) :199–208, 1988.

[SV09] Gabriel Scherer and Jérôme Vouillon. Macaque, 2009. Available from : `http://macaque.forge.ocamlcore.org/`.

[Won00] Limsoon Wong. Kleisli, a functional query system. *J. Funct. Program.*, 10(1) :19–56, 2000.

[WSG02] Noel Welsh, Francisco Solsona, and Ian Glover. SchemeUnit and SchemeQL : Two little languages. In *Third Workshop on Scheme and Functional Programming*, 2002.