Rapport de projet — TQL

Antoni Boucher

Version 1.0, 2015-12-07

Table des matières

Introduction . . . 1
Sous-ensemble du SQL supporté . . . 2
Résumé général . . . 3
Résumé détaillé . . . 4
Support des types . . . 6
Extension syntaxique pour Rust . . . 7
Attribut . . . 7
Macros procédurales . . . 8
Schémas de traduction de code . . . 9
Optimisations . . . 17
Architecture du code . . . 20
Documentation . . . 22
Création du projet . . . 22
Squelette de base . . . 24
Création des structures pour les tables . . . 25
Utilisation de la macro `sql!()` . . . 25
Code complet . . . 27
Exemple d’erreur . . . 29
Conclusion . . . 30
Annexes . . . 33
Annexe A: Code source de la bibliothèque . . . 33
Annexe B: Code source de l’extension syntaxique . . . 34
Annexe C: Tests . . . 128
Annexe D: Exemples . . . 132
Glossaire . . . 138

Introduction

Ce rapport se présente sous la forme d'un manuel technique du projet Typed Query Language (TQL).

TQL est un module d'extension du compilateur Rust fournissant un langage dédié embarqué pour le langage SQL. Cela permet d'écrire du code Rust qui est converti en SQL à la compilation, tout en faisant la vérification de la validité des requêtes. Les erreurs de syntaxe et de types sont ainsi détectées à la compilation, contrairement au SQL, dans lequel les erreurs sont détectées à l'exécution. Cela permet de réduire le nombre d'erreurs à l'exécution. Par ailleurs, comme le SQL est généré à la compilation, il n'y a pas de dégradation de la performance à l'exécution causée par cette abstraction. Le module d'extension TQL fournit un attribut de même que des macros procédurales qui servent respectivement à la vérification des types/identifiants et à la transformation du code Rust en SQL.

Ce rapport présente le sous-ensemble du SQL supporté par ce module de même que les schémas de traduction de Rust vers SQL. Ensuite, il est question des optimisations et de l'architecture du code. Puis, le rapport se termine par une documentation avant de conclure sur la performance et la pertinence d'un tel projet.

Sous-ensemble du SQL supporté

Cette section décrit le sous-ensemble du SQL qui est supporté par le module d’extension TQL.

Ce module supporte les formes les plus courantes des requêtes suivantes :

- Création de table
- Suppression de table
- Sélection
- Insertion
- Mise à jour
- Suppression

Voici une partie des types de requêtes qui ne sont pas couverts par ce module :

- Création de fonction
- Création de procédure
- Création de déclencheur
- Modification de fonction
- Modification de procédure
- Modification de déclencheur
- Modification de table

Cela n’impacte pas vraiment l’utilisation du module TQL, car ces requêtes sont généralement exécutées une seule fois. Or, le but de ce module est de programmer des applications qui font des requêtes à une base de données. Et comme les requêtes effectuées par une application sont utilisées plusieurs fois, les requêtes non supportées présentées ci-dessus ne sont pas vraiment utiles dans ce contexte.

La fonctionnalité `DISTINCT` n’est pas supportée pour les requêtes de sélection, ce qui est parfois utile.

Parmi les fonctionnalités non supportées dans les requêtes de création de table, il y a `DEFAULT`, `UNIQUE` et `INDEXES` qui sont assez importantes.

Pour les requêtes de suppression de table, la fonctionnalité `CASCADE` n’est pas supportée, ce qui peut être utile de temps en temps.

Les types qui ne sont pas supportés incluent `Money`, `Point` et `UUID`. Les types non supportés ne sont pas couramment utilisés, donc l’impact sur l’utilisation de TQL est minime.

En bref, environ 25 % du SQL est implémenté, dont 80 % des fonctionnalités les plus courantes.

Résumé général

Fonctionnalité	Est supportée ?	Fonctionnalité	Est supportée ?
ALTER DOMAIN	✖	ALTER TABLE	✖
CLOSE	✖	COMMIT WORK	✖
CONNECT	✖	CREATE ASSERTION	✖
CREATE CHARACTER SET	✖	CREATE COLLATION	✖
CREATE DOMAIN	✖	CREATE FUNCTION	✖
CREATE PROCEDURE	✖	CREATE SCHEMA	✖
CREATE TABLE	✔	CREATE TRANSLATION	✖
CREATE TRIGGER	✖	CREATE VIEW	✖
DEALLOCATE PREPARE	✖	DELETE	✔
DESCRIBE	✖	DESCRIPTOR	✖
DISCONNECT	✖	DROP ASSERTION	✖
DROP CHARACTER SET	✖	DROP COLLATION	✖
DROP DOMAIN	✖	DROP FUNCTION	✖
DROP PROCEDURE	✖	DROP SCHEMA	✖
DROP TABLE	✔	DROP TRANSLATION	✖
DROP TRIGGER	✖	DROP VIEW	✖
EXECUTE	✖	FETCH	✖
GET DIAGNOSTICS	✖	GRANT	✖
INSERT	✔	MERGE	✖
OPEN	✖	PREPARE	✖
REVOKE	✖	ROLLBACK WORK	✖
SAVEPOINT	✖	SELECT	✔
SET	✖	UPDATE	✔

Résumé détaillé

Support du `INSERT`

Fonctionnalité	Est supportée ?	Fonctionnalité	Est supportée ?
DEFAULT VALUES	✖	RETURNING	✔

Support du `SELECT`

Fonctionnalité	Est supportée ?	Fonctionnalité	Est supportée ?
DISTINCT	✖	FROM	✔
WHERE	✔	GROUP BY	✔
HAVING	✔	WINDOW	✖
UNION	✖	INTERSECT	✖
EXCEPT	✖	ORDER BY	✔
LIMIT	✔	OFFSET	✔
FETCH	✖	FOR	✖

Support du `UPDATE`

Fonctionnalité	Est supportée ?	Fonctionnalité	Est supportée ?
ONLY	✖	SET	✔
FROM	✖	WHERE	✔
WHERE CURRENT OF	✖	RETURNING	✖

Support du `DELETE`

Fonctionnalité	Est supportée ?	Fonctionnalité	Est supportée ?
ONLY	✖	USING	✖
WHERE	✔	WHERE CURRENT OF	✖
RETURNING	✖		

Support du `CREATE TABLE`

Fonctionnalité	Est supportée ?	Fonctionnalité	Est supportée ?
GLOBAL	✖	LOCAL	✖
TEMPORARY	✖	UNLOGGED	✖
IF NOT EXISTS	✖	COLLATE	✖
INHERITS	✖	WITH	✖
ON COMMIT	✖	TABLESPACE	✖
NOT NULL	✔	NULL	✔
CHECK	✖	DEFAULT	✖
UNIQUE	✖	PRIMARY KEY	✔
REFERENCES	✔	MATCH FULL	✖
MATCH PARTIAL	✖	MATCH SIMPLE	✖
ON DELETE	✖	ON UPDATE	✖
DEFERRABLE	✖	INITIALLY	✖
EXCLUDE	✖	FOREIGN KEY	✔
INCLUDING	✖	EXCLUDING	✖
CONSTRAINTS	✖	INDEXES	✖

Support du `DROP TABLE`

Fonctionnalité	Est supportée ?	Fonctionnalité	Est supportée ?
IF EXISTS	✖	CASCADE	✖
RESTRICT	✖		

Support des types

Type	Est supporté ?	Type	Est supporté ?
BIGINT	✔	BIGSERIAL	✔
BIT	✘	BOOLEAN	✔
BOX	✘	BYTEA	✔
CHARACTER	✔	CIDR	✘
CIRCLE	✘	DATE	✔
DOUBLE PRECISION	✔	INET	✘
INTEGER	✔	INTERVAL	✘
LINE	✘	LSEG	✘
MACADDR	✘	MONEY	✘
NUMERIC	✘	PATH	✘
POINT	✘	POLYGON	✘
REAL	✔	SMALLINT	✔
SERIAL	✔	TEXT	✔
TIME	✔	TIMESTAMP	✔
TSQUERY	✘	TSVECTOR	✘
TXID_SNAPSHOT	✘	UUID	✘
XML	✘		

Extension syntaxique pour Rust

Cette section décrit l'attribut et les macros procédurales fournis par le module TQL.

Attribut

Ce module fournit un attribut `#[SqlTable]` permettant d'indiquer qu'une structure fait référence à une table SQL.

Les champs de la structure doivent avoir un des types supportés dont le tableau de correspondance se trouve à la section Traduction des types.

Par exemple, TQL convertit le tuple `(1, "Boucher", "Antoni")` vers la structure suivante :

```
#[SqlTable]
struct Personne {
    id: PrimaryKey,
    nom: String,
    prenom: String,
}
```

de cette façon :

```
Personne {
    id: ligne.get(0),
    nom: ligne.get(1),
    prenom: ligne.get(2),
}
```

(où `ligne` est le tuple et `ligne.get(x)` retourne l'élément à l'index `x` du tuple)

Cela suppose que la requête SQL exécutée est la suivante (car l'ordre des éléments du tuple est important) :

```
SELECT id, nom, prenom FROM Personne
```

Macros procédurales

TQL fournit les macros procédurales `sql!()` et `to_sql!()`. La première transforme la requête reçue en paramètre en SQL et génère du code Rust qui exécute cette requête. La seconde macro ne fait que transformer la requête en SQL.

Ces macros s'attendent à obtenir un paramètre qui respecte le schéma suivant :

```
NomDeLaTable.methode1(parametres1).methode2(parametres2)...
```

c'est-à-dire le nom de la table, suivi d'une nombre quelconque d'appels de méthode. La syntaxe est la même que celle de Rust, donc l'AST utilisé pour la conversion est celui du compilateur Rust.

Par exemple, avec la structure ci-dessus, une requête d'insertion ressemble à :

```
sql!(Personne.insert(nom = "Boucher", prenom = "Antoni"));
```

Les schémas de traduction de Rust vers SQL sont décrits dans la section Schémas de traduction de code.

Des exemples sont montrés dans la Documentation.

Schémas de traduction de code

Cette section décrit la traduction des différents types de requêtes de Rust vers postgresql. Elle montre également la traduction des méthodes de filtre, des fonctions d'agrégat et des types. Il s'agit donc d'une référence pour l'utilisation du module d'extension TQL.

Voici la structure des tables qui sont utilisées dans cette section :

```
#[SqlTable]
struct Personne {
    id: PrimaryKey,
    nom: String,
    prenom: String,
    age: i32,
    date_naissance: DateTime<UTC>,
    poids: Option<i32>,
    ecole: ForeignKey<Ecole>,
}

#[SqlTable]
struct Ecole {
    id: PrimaryKey,
    nom: String,
}
```

Traduction des requêtes

Rust	SQL

Personne.all()	SELECT Personne.id, Personne.nom,
	    Personne.prenom, Personne.age,
	    Personne.date_naissance, Personne.poids
	FROM Personne

Personne.filter(nom == "Boucher")	SELECT Personne.id, Personne.nom,
	    Personne.prenom, Personne.age,
	    Personne.date_naissance, Personne.poids
	FROM Personne
	WHERE nom = 'Boucher'

Personne.get(1)
 	
/// Raccourci pour :
Personne.filter(id == 1)[0]
 	
/// Raccourci pour :
Personne.filter(id == 1)[0..1].get()
	SELECT Personne.id, Personne.nom,
	    Personne.prenom, Personne.age,
	    Personne.date_naissance, Personne.poids
	FROM Personne
	WHERE id = 1

<table>
<tr><th>Rust</th><th>SQL</th></tr>
<tr><td>Personne.get(nom == "Boucher")

/// Raccourci pour :
Personne.filter(nom == "Boucher")[0]</td><td>SELECT Personne.id, Personne.nom,
    Personne.prenom, Personne.age,
    Personne.date_naissance, Personne.poids
FROM Personne
WHERE nom = 'Boucher'
LIMIT 1</td></tr>
<tr><td>Personne.filter(nom == "Boucher" && age < 30)</td><td>SELECT Personne.id, Personne.nom,
    Personne.prenom, Personne.age,
    Personne.date_naissance, Personne.poids
FROM Personne
WHERE nom = 'Boucher'
  AND age < 30</td></tr>
<tr><td>Personne.filter(nom == "Boucher" || age < 30)</td><td>SELECT Personne.id, Personne.nom,
    Personne.prenom, Personne.age,
    Personne.date_naissance, Personne.poids
FROM Personne
WHERE nom = 'Boucher'
  OR age < 30</td></tr>
<tr><td>Personne
    .filter(!(nom == "Boucher" || age < 30))</td><td>SELECT Personne.id, Personne.nom,
    Personne.prenom, Personne.age,
    Personne.date_naissance, Personne.poids
FROM Personne
WHERE NOT (nom = 'Boucher'
  OR age < 30)</td></tr>
<tr><td>Personne.sort(age)</td><td>SELECT Personne.id, Personne.nom,
    Personne.prenom, Personne.age,
    Personne.date_naissance, Personne.poids
FROM Personne
ORDER BY age</td></tr>
<tr><td>Personne.sort(-age)</td><td>SELECT Personne.id, Personne.nom,
    Personne.prenom, Personne.age,
    Personne.date_naissance, Personne.poids
FROM Personne
ORDER BY age DESC</td></tr>
<tr><td>Personne[0..20]</td><td>SELECT Personne.id, Personne.nom,
    Personne.prenom, Personne.age,
    Personne.date_naissance, Personne.poids
FROM Personne
LIMIT 20 OFFSET 0</td></tr>
<tr><td>Personne
    .filter(nom == "Boucher" && age < 30)
    .sort(-age)[10..20]</td><td>SELECT Personne.id, Personne.nom,
    Personne.prenom, Personne.age,
    Personne.date_naissance, Personne.poids
FROM Personne
WHERE nom = 'Boucher'
  AND age < 30
ORDER BY age DESC
LIMIT 20 OFFSET 10</td></tr>
<tr><td>Personne.aggregate(avg(age))</td><td>SELECT AVG(Personne.age)
FROM Personne</td></tr>
</table>

<table>
<tr><th>Rust</th><th>SQL</th></tr>
<tr><td>Personne.values(nom).aggregate(avg(age))</td><td>SELECT AVG(Personne.age)
FROM Personne
GROUP BY nom</td></tr>
<tr><td>Personne.values(nom)
    .aggregate(age_moyen = avg(age))
    .filter(age_moyen > 30)</td><td>SELECT AVG(Personne.age) AS age_moyen
FROM Personne
GROUP BY nom
HAVING AVG(age) > 30</td></tr>
<tr><td>Personne.filter(age < 50).values(nom)
    .aggregate(age_moyen = avg(age))
    .filter(age_moyen > 30)</td><td>SELECT AVG(Personne.age) AS age_moyen
FROM Personne
WHERE age < 50
GROUP BY nom
HAVING AVG(age) > 30</td></tr>
<tr><td>Personne.join(ecole)</td><td>SELECT Personne.id, Personne.nom,
    Personne.prenom, Personne.age,
    Personne.date_naissance, Personne.poids,
    Ecole.id, Ecole.nom
FROM Personne
INNER JOIN Ecole
  ON Personne.ecole = Ecole.id</td></tr>
<tr><td>Personne
    .filter(date_naissance.year() == 2000)</td><td>SELECT Personne.id, Personne.nom,
    Personne.prenom, Personne.age,
    Personne.date_naissance, Personne.poids
FROM Personne
WHERE EXTRACT(YEAR FROM date_naissance) =
2000</td></tr>
<tr><td>Personne.filter(nom.contains("ouc"))</td><td>SELECT Personne.id, Personne.nom,
    Personne.prenom, Personne.age,
    Personne.date_naissance, Personne.poids
FROM Personne
WHERE nom LIKE '%' || 'ouc' || '%'</td></tr>
<tr><td>Personne.filter(poids.is_none())</td><td>SELECT Personne.id, Personne.nom,
    Personne.prenom, Personne.age,
    Personne.date_naissance, Personne.poids
FROM Personne
WHERE poids IS NULL</td></tr>
<tr><td>Personne.filter(nom.match("%ou_h%"))</td><td>SELECT Personne.id, Personne.nom,
    Personne.prenom, Personne.age,
    Personne.date_naissance, Personne.poids
FROM Personne
WHERE nom LIKE '%ou_h%'</td></tr>
<tr><td>Personne.insert(
    nom = "Boucher",
    prenom = "Antoni",
    age = 22,
    date_naissance = UTC::now(),
    ecole = ecole
)</td><td>INSERT INTO Personne
(nom, prenom, age, date_naissance, ecole)
VALUES('Boucher', 'Antoni', 22, $1, $2)
RETURNING id

(où $x indique le paramètre numéro x envoyé à la requête préparée)</td></tr>
</table>

Rust	SQL

`Personne.get(1)`
`    .update(nom = "Boucher", age = 22)`
	`UPDATE Personne`
	`SET nom = 'Boucher', age = 22`
	`WHERE id = 1`

`Personne.get(1).delete()`	`DELETE FROM Personne` `WHERE id = 1`

`Ecole.create()`	`CREATE TABLE Ecole (`
	`  id SERIAL NOT NULL PRIMARY KEY,`
	`  nom VARCHAR NOT NULL`
	`)`

`Ecole.drop()`	`DROP TABLE Ecole`

Traduction des méthodes de filtre

Voici la liste des méthodes supportées par TQL, chacune suivie de leur traduction en postgresql :

Date::year()	Date::month()
Date::day()	Time::hour()
Time::minute()	Time::second()
String::contains(String)	String::ends_with(String)
String::starts_with(String)	String::len()
String::match(String)	String::imatch(String)
Option<T>::is_some()	Option<T>::is_none()

<table>
<tr><th>Rust</th><th>SQL</th></tr>
<tr><td>date_naissance.year()</td><td>EXTRACT(YEAR FROM date_naissance)</td></tr>
<tr><td>date_naissance.month()</td><td>EXTRACT(MONTH FROM date_naissance)</td></tr>
<tr><td>date_naissance.day()</td><td>EXTRACT(DAY FROM date_naissance)</td></tr>
<tr><td>date_naissance.hour()</td><td>EXTRACT(HOUR FROM date_naissance)</td></tr>
<tr><td>date_naissance.minute()</td><td>EXTRACT(MINUTE FROM date_naissance)</td></tr>
<tr><td>date_naissance.second()</td><td>EXTRACT(SECOND FROM date_naissance)</td></tr>
<tr><td>nom.contains("ouc")</td><td>nom LIKE '%' || 'ouc' || '%'</td></tr>
<tr><td>nom.ends_with("cher")</td><td>nom LIKE '%' || 'cher'</td></tr>
<tr><td>nom.starts_with("Bou")</td><td>nom LIKE 'Bou' || '%'</td></tr>
<tr><td>nom.len()</td><td>CHAR_LENGTH(nom)</td></tr>
<tr><td>nom.match("Bou%")</td><td>nom LIKE 'Bou%'</td></tr>
<tr><td>nom.imatch("bou%")</td><td>nom ILIKE 'bou%'</td></tr>
<tr><td>poids.is_some()</td><td>poids IS NOT NULL</td></tr>
<tr><td>poids.is_none()</td><td>poids IS NULL</td></tr>
</table>

Traduction des fonctions d’agrégat

Pour le moment, une seule fonction d’agrégat est supportée :

- avg()

dont voici sa traduction en SQL :

Rust	SQL
avg(age)	AVG(age)

Traduction des types

Rust	SQL
bool	BOOLEAN
Vec< u8 >	BYTEA
char	CHARACTER(1)
ForeignKey<Ecole>	INTEGER REFERENCES Ecole(id)
f32	REAL
f64	DOUBLE PRECISION
i16	SMALLINT
i32	INTEGER
i64	BIGINT
DateTime<Local>	TIMESTAMP WITH TIME ZONE
NaiveDate	DATE
NaiveDateTime	TIMESTAMP
NaiveTime	TIME
PrimaryKey	SERIAL PRIMARY KEY
String (&str)	CHARACTER VARYING
DateTime<UTC>	TIMESTAMP WITH TIME ZONE

Traduction de l’exécution de la requête

Pour conclure, voici la traduction du code des macros vers Rust. Cela concerne l’exécution des requêtes et a pour but d’éviter d’avoir à écrire du code redondant.

<table>
<tr>
<td>

```
let moyennes_age_personne = sql!(Personne
    .values(nom)
    .aggregate(avg(age))
);
```

</td>
<td>

```
let moyennes_age_personne = {
    let result = connection.prepare(
        "SELECT AVG(Personne.age)
         FROM Personne
         GROUP BY nom"
    ).unwrap();
    result.query(&[]).unwrap().iter().map(
        |row|
        {
          Aggregate {
              age_avg = row.get(0),
          }
        }
    ).collect::<Vec<_>>()
};
```

</td>
</tr>
<tr>
<td>

```
let moyenne_age_personne = sql!(
    Personne.aggregate(avg(age))
);
```

</td>
<td>

```
let moyenne_age_personne = {
    let result = connection.prepare(
        "SELECT AVG(Personne.age)
         FROM Personne"
    ).unwrap();
    result.query(&[]).unwrap().iter().next()
        .map(|row| {
            Aggregate {
                age_avg = row.get(0),
            }
        }
    )
};
```

</td>
</tr>
<tr>
<td>

```
let personne_id = sql!(Personne.insert(
    nom = "Boucher",
    prenom = "Antoni",
    age = 22,
    date_naissance = UTC::now(),
    ecole = ecole
));
```

</td>
<td>

```
let personne_id = {
    connection.prepare(
        "INSERT INTO Personne
(nom, prenom, age, date_naissance, ecole)
VALUES('Boucher', 'Antoni', 22, $1, $2)
RETURNING id")
        .and_then(|result| {
            let rows = result.query(
            &[UTC::now(), ecole]).unwrap();
            let row = rows.iter().next()
                .unwrap();
            let id: i32 = row.get(0);
            Ok(id)
        })
};
```

</td>
</tr>
</table>

<table>
<tr>
<td>

```
let personnes = sql!(Personne.all());
```

</td>
<td>

```
let personnes = {
    let result = connection.prepare(
        "SELECT Personne.id, Personne.nom,
         Personne.prenom, Personne.age,
         Personne.date_naissance,
         Personne.poids
         FROM Personne"
    ).unwrap();
    result.query(&[]).unwrap().iter().map(
        |row| {
            Personne {
                id: row.get(0),
                nom: row.get(1),
                prenom: row.get(2),
                age: row.get(3),
                date_naissance: row.get(4),
                poids: row.get(5),
                ecole: None,
            }
        }
    ).collect::<Vec<_>>()
};
```

</td>
</tr>
<tr>
<td>

```
let personne = sql!(Personne.get(1));
```

</td>
<td>

```
let personne = {
    let result = connection.prepare(
        "SELECT Personne.id, Personne.nom,
         Personne.prenom, Personne.age,
         Personne.date_naissance,
         Personne.poids
         FROM Personne
         WHERE id = 1"
    ).unwrap();
    result.query(&[]).unwrap().iter().next()
        .map(|row| {
            Personne {
                id: row.get(0),
                nom: row.get(1),
                prenom: row.get(2),
                age: row.get(3),
                date_naissance: row.get(4),
                poids: row.get(5),
                ecole: None,
            }
        }
    )
};
```

</td>
</tr>
<tr>
<td>

```
let nombre_personnes_mises_a_jour =
    sql!(Personne.get(1).update(
        nom = "Boucher",
        age = 22
    ));
```

</td>
<td>

```
let nombre_personnes_mises_a_jour = {
    connection.prepare(
        "UPDATE Personne SET
         nom = 'Boucher', age = 22
         WHERE id = 1"
    )
        .and_then(|result|
            result.execute(&[])
        )
};
```

</td>
</tr>
</table>

Optimisations

TQL ne supporte qu'une seule optimisation pour le moment. Le module d'extension simplifie les opérations composées uniquement de littéraux, ce qui évite d'avoir à passer un paramètre inutile à la requête préparée à l'exécution.

Par exemple, le code suivant :

```
Personne.limit[0 + 10 - 2 .. 50 - (4 + 2)]
```

compile en :

```
SELECT Personne.id, Personne.nom, Personne.prenom, Personne.age, Personne.date_naissance,
Personne.poids
FROM Personne
LIMIT 44
OFFSET 8
```

D'autres optimisations sont envisageables. Par exemple, TQL pourrait déterminer quels sont les champs qui sont nécessaires à l'exécution et seulement inclure ceux-ci dans la requête.

Donc, si le seul champ utilisé est `age`, la requête suivante :

```
Personne.all()
```

pourrait être compilée en :

```
SELECT Personne.age
FROM Personne
```

ce qui diminue le transfert d'informations entre le système de gestion de bases de données et l'application.

Par ailleurs, certaines méthodes peuvent être optimisées lorsqu'elles sont utilisées avec des littéraux.

En effet, cette requête :

```
Personne.filter(nom.contains("ouc"))
```

compile en :

```
SELECT Personne.age
FROM Personne
WHERE nom LIKE '%' || 'ouc' || '%'
```

mais pourrait être compilée en :

```
SELECT Personne.age
FROM Personne
WHERE nom LIKE '%ouc%'
```

Enfin, la génération de code Rust pourrait être optimisée pour ne pas préparer une requête plusieurs fois, dans le cas où elle est exécutée plusieurs fois de suite (par exemple, dans une boucle).

Par exemple, ce code

```
for i in (0..5) {
    let personnes = sql!(Personne.all());
}
```

compile en :

```
for i in (0 .. 5) {
    let personnes = {
        let result = connection.prepare("SELECT Personne.id, Personne.nom,
Personne.prenom, Personne.age, Personne.date_naissance, Personne.poids FROM
Personne").unwrap();
        result.query(&[]).unwrap().iter().map(|row| {
            Personne {
                id: row.get(0),
                nom: row.get(1),
                prenom: row.get(2),
                age: row.get(3),
                date_naissance: row.get(4),
                poids: row.get(5),
                ecole: None,
            }
        }).collect::<Vec<_>>()
    };
}
```

mais pourrait être compilé en :

```
let result = connection.prepare("SELECT Personne.id, Personne.nom, Personne.prenom,
Personne.age, Personne.date_naissance, Personne.poids FROM Personne").unwrap();
for i in (0 .. 5) {
    let personnes = {
        result.query(&[]).unwrap().iter().map(|row| {
            Personne {
                id: row.get(0),
                nom: row.get(1),
                prenom: row.get(2),
                age: row.get(3),
                date_naissance: row.get(4),
                poids: row.get(5),
                ecole: None,
            }
        }).collect::<Vec<_>>()
    };
}
```

Architecture du code

Le code du module d'extension débute son exécution dans le module `lib`. Ce module définit les macros et l'attribut fournis par TQL. Il appelle également les autres modules pour exécuter les diverses tâches (conversion de l'AST Rust vers l'AST utilisé par TQL, analyse sémantique et génération de code).

Le module `attribute` fournit le code utilisé par l'attribut `#[SqlTable]`. Cet attribut sert à indiquer qu'une structure Rust fait référence à une table SQL. Cet attribut enregistre donc les champs de chaque structure annotée afin de pouvoir effectuer la vérification des types lors de l'analyse sémantique. Une conversion des types Rust vers ceux requis par l'AST de TQL est aussi effectuée.

Le module `gen` effectue la génération du code SQL à partir de l'AST.

Le module `parser` transforme l'AST de Rust vers une structure intermédiaire (sous forme d'une suite d'appels de méthode) qui est convertie plus tard vers l'AST de TQL. Comme TQL n'admet qu'un sous-ensemble de Rust (une suite d'appels de méthode), cela simplifie grandement cette conversion.

Le module `state` contient les différents états globaux nécessaires à TQL :

- les fonctions d'agrégat
- les méthodes de filtre
- les tables définies via l'attribut `#[SqlTable]`
- les arguments passés à la macro `sql!()` (explications sur la nécessité de cet état global plus loin)

Le module `types` définit les types supportés dans une structure annotée via l'attribut `#[SqlTable]` de même que des fonctions pour effectuer diverses tâches :

- afficher ces types (pour les messages d'erreur)
- convertir ces types vers leur représentation SQL
- convertir un type Rust vers un type TQL
- comparer un type Rust avec un type TQL

Le module `ast` définit la structure de l'arbre syntaxique abstrait. Il contient un type `Query` qui est une énumération pouvant contenir un des différents types de requête.

Le module `string` contient, entre autres, une fonction permettant de trouver une chaîne de caractères proche d'une autre (pour proposer des corrections dans les messages d'erreur).

Le module `methods` définit les méthodes de filtre et les fonctions d'agrégation pouvant être utilisées dans la macro `sql!()`.

Le module `arguments` permet d'extraire les arguments envoyés à la macro `sql!()` à partir de l'AST de TQL. Cela est nécessaire pour les envoyer à la méthode `postgres::stmt::Statement::query()`.

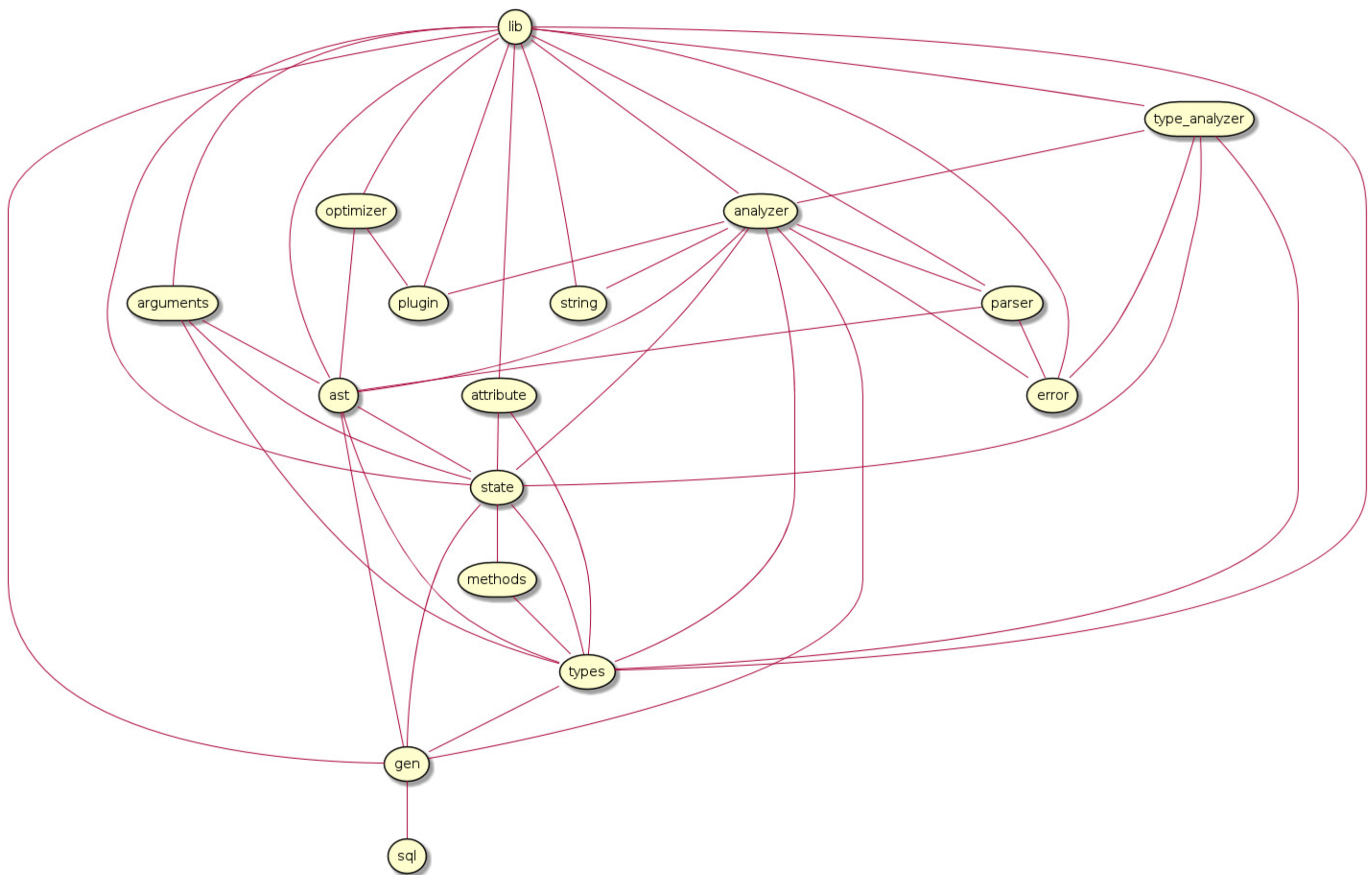
Le module `optimizer` optimise les expressions composées uniquement de littéraux en simplifiant celles-ci.

Le module `analyzer` effectue l'analyse sémantique. Il vérifie que les noms de champ utilisés existent et que le type des expressions littérales correspond au type des champs. Ce module est divisé en plusieurs sous-modules.

Le module `type_analyzer` effectue aussi l'analyse sémantique. Il vérifie le nom des tables et le type des expressions non littérales. Cette distinction entre les modules `analyzer` et `type_analyzer` est nécessaire, car un module d'extension en Rust peut soit faire de la transformation d'AST, soit de l'analyse de types. Il ne peut pas faire les deux en même temps. C'est pourquoi il y a plusieurs sortes de module d'extension dans TQL. Cela explique aussi pourquoi les arguments sont enregistrés dans l'état global : parce que la conversion du code est effectuée avant l'analyse du type des expressions.

Le module `error` définit la structure des erreurs.

Voici un diagramme montrant les relations entre les modules :



Documentation

Cette section se présente sous la forme d'un tutoriel montrant comment utiliser le module d'extension du compilateur Rust TQL.

Création du projet

Premièrement, il faut créer un nouveau projet Rust. Pour se faire, il suffit de lancer la commande suivante :

```
cargo new nom_du_projet --bin
```

Ce projet nécessite l'usage de la version instable du compilateur (« nightly ») pour fonctionner.

Ensuite, la dépendance à la bibliothèque TQL doit être ajoutée dans le fichier `Cargo.toml` qui vient d'être généré dans le nouveau dossier `nom_du_projet` :

```
[package]
name = "nom_du_projet"
version = "0.1.0"
authors = ["Antoni Boucher <Antoni.Boucher@USherbrooke.ca>"]

[dependencies]
chrono = "0.2.16"
postgres = { version = "0.10.1", features = ["chrono"] }
tql = "0.1.0"
tql_macros = "0.1.0"
```

Les cinq dernières lignes ont été ajoutées pour spécifier les nouvelles dépendances du projet.

Si les types de date et heure (par exemple, `DateTime`) ne sont pas utilisés, les lignes suivantes :

```
chrono = "0.2.16"
postgres = { version = "0.10.1", features = ["chrono"] }
```

peuvent être remplacées par la ligne suivante :

```
postgres = "0.10.1"
```

(`chrono` étant une bibliothèque de dates et heures pour Rust.)

Ceci indique de ne pas compiler la bibliothèque `postgres` avec le support de ces types.

Puis, il suffit de lancer la commande suivante pour installer les dépendances :

```
cargo build
```

Squelette de base

Après avoir installé les dépendances, le code du fichier `src/main.rs` peut être remplacé par le suivant, qui constitue le squelette de base d'une application utilisant `tql` :

Exemple 1. Squelette de base

```
#![feature(plugin)] ①
#![plugin(tql_macros)] ②

extern crate chrono;
extern crate postgres;
extern crate tql; ③

use chrono::datetime::DateTime;
use chrono::offset::utc::UTC;
use postgres::{Connection, SslMode};
use tql::{ForeignKey, PrimaryKey};

fn main() {
    let connection = get_connection();
}

fn get_connection() -> Connection {
    Connection::connect(
        "postgres://nom_d_utilisateur:mot_de_passe@serveur/base_de_donnees",
        &SslMode::None
    ).unwrap()
}
```

① Comme pour tous les programmes Rust qui utilisent un module d'extension du compilateur, il faut indiquer au compilateur que cette fonctionnalité instable est utilisée dans le code.

② Cette ligne charge le module d'extension `tql_macros`, qui contient, entre autres, les macros `sql!()` et `to_sql!()`.

③ Cette ligne indique au compilateur de compiler et lier la bibliothèque `tql`.

Le reste du code utilise la bibliothèque `postgres` en ouvrant une connexion au serveur PostgreSQL.

Le programme peut être lancé avec la commande suivante :

```
cargo run
```

mais il ne fait rien à cette étape.

Création des structures pour les tables

La prochaine étape consiste à créer les structures Rust définissant les tables SQL. Pour cela, le module d'extension fournit l'attribut `#[SqlTable]`.

Voici un exemple utilisant cet attribut :

Exemple 2. Structures des tables

```
#[SqlTable] ①
struct Article {
    id: PrimaryKey, ②
    auteur: String,
    titre: String,
    revision: i32,
    date_ajout: DateTime<UTC>, ③
}

#[SqlTable]
struct Commentaire {
    id: PrimaryKey,
    auteur: String,
    message: String,
    date_post: DateTime<UTC>,
    article: ForeignKey<Article>, ④
}
```

① Utilisation de l'attribut `#[SqlTable]`.

② Définition du champ `id` comme étant la clé primaire.

③ Un champ de type date heure.

④ Une clé étrangère vers la table `Article`.

Utilisation de la macro `sql!()`

La macro `sql!()` permet d'exécuter des requêtes SQL. Cette macro s'attend à ce qu'une variable nommée `connection` existe, car elle l'utilise afin de communiquer avec le serveur de la base de données.

Il existe aussi la macro `to_sql!()` qui ne fait que convertir le code Rust en SQL.

Voici un exemple utilisant cette macro :

Exemple 3. Utilisation de la macro `sql!()`

```
fn main() {
    let connection = get_connection();

    sql!(Article.create()); ①
    sql!(Commentaire.create());

    let auteur = "Antoni Boucher";
    let article_id = ②
        sql!(Article.insert(
            auteur = auteur,
            titre = "Introduction à TQL",
            revision = 1,
            date_ajout = UTC::now()
        )).unwrap();

    let article = sql!(Article.get(article_id)).unwrap(); ③

    println!("{}", article.titre); ④

    sql!(Commentaire.insert(
        auteur = "Antoni Boucher",
        message = "Cet article présente brièvement TQL.",
        date_post = UTC::now(),
        article = article,
    ));

    sql!(Commentaire.drop()); ⑤
    sql!(Article.drop());
}
```

① Création de la table `Article`.

② La méthode `insert()` retourne la clé primaire (l'id) de l'objet nouvellement inséré.

③ Obtenir l'article par sa clé primaire.

④ L'accès au champ d'un objet se fait comme pour une structure normale en Rust, c'est-à-dire `objet.champ`.

⑤ Suppression de la table `Commentaire`.

Voir Schémas de traduction de code pour la liste complète des méthodes et fonctions supportées par TQL.

Code complet

Voici le code complet :

```
#![feature(plugin)]
#![plugin(tql_macros)]

extern crate chrono;
extern crate postgres;
extern crate tql;

use chrono::datetime::DateTime;
use chrono::offset::utc::UTC;
use postgres::{Connection, SslMode};
use tql::{ForeignKey, PrimaryKey};

#[SqlTable]
struct Article {
    id: PrimaryKey,
    auteur: String,
    titre: String,
    revision: i32,
    date_ajout: DateTime<UTC>,
}

#[SqlTable]
struct Commentaire {
    id: PrimaryKey,
    auteur: String,
    message: String,
    date_post: DateTime<UTC>,
    article: ForeignKey<Article>,
}

fn main() {
    let connection = get_connection();

    sql!(Article.create());
    sql!(Commentaire.create());

    let auteur = "Antoni Boucher";
    let article_id =
        sql!(Article.insert(
            auteur = auteur,
            titre = "Introduction à TQL",
            revision = 1,
            date_ajout = UTC::now()
```

```
        )).unwrap();

    let article = sql!(Article.get(article_id)).unwrap();

    println!("{}", article.titre);

    sql!(Commentaire.insert(
        auteur = "Antoni Boucher",
        message = "Cet article présente brièvement TQL.",
        date_post = UTC::now(),
        article = article,
    ));

    sql!(Commentaire.drop());
    sql!(Article.drop());
}

fn get_connection() -> Connection {
    Connection::connect(
        "postgres://nom_d_utilisateur:mot_de_passe@serveur/base_de_donnees",
        &SslMode::None
    ).unwrap()
}
```

Exemple d’erreur

Voici un exemple de message d’erreur généré par TQL.

Si la ligne :

```
let article = sql!(Article.get(article_id)).unwrap();
```

est remplacée par :

```
let article = sql!(Article.get("article_id")).unwrap();
```

Le message suivant est affiché :

```
code.rs:46:36: 46:47 error: mismatched types:
 expected `i32`,
    found `String` [E0308]
code.rs:46     let article = sql!(Article.get("article_id")).unwrap();
                                                ^~
code.rs:46:36: 46:47 help: run `rustc --explain E0308` to see a detailed explanation
code.rs:46:36: 46:47 note: in this expansion of sql! (defined in TQL)
```

Conclusion

Somme toute, ce projet démontre qu'il est possible de bénéficier grandement d'une extension syntaxique pour convertir du code en SQL à la compilation. En effet, cela permet de détecter les erreurs de syntaxe à la compilation tout en fournissant une abstraction du SQL. Par ailleurs, les performances d'un tel module sont très bonnes, contre le montre le graphique suivant :

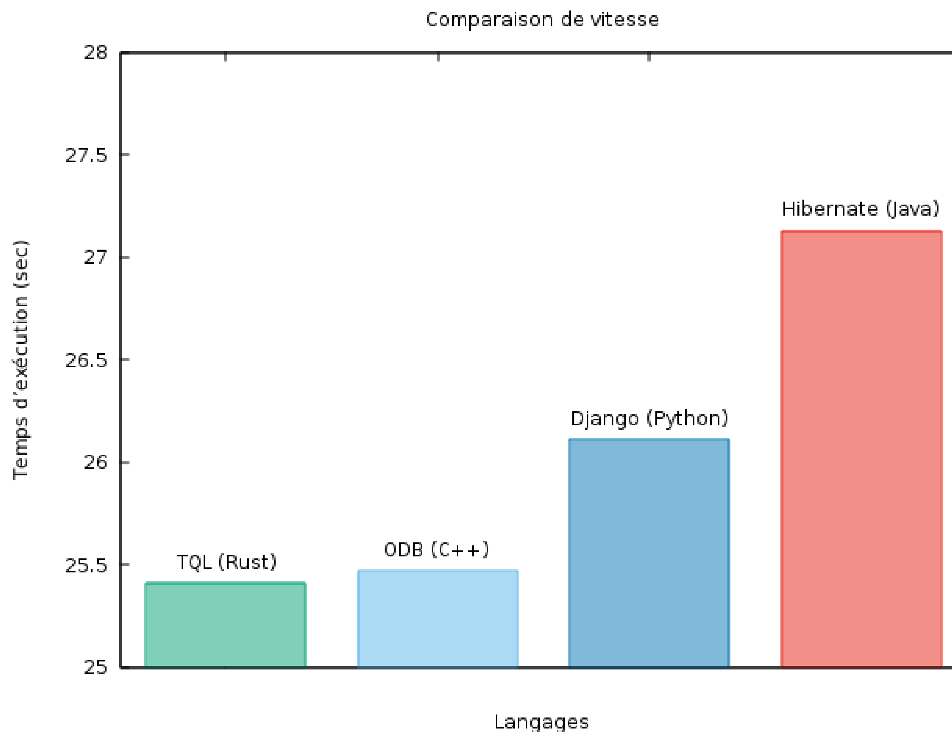


Figure 1. Vitesse d'exécution en secondes de codes utilisant différentes abstractions de bases de données

Sans utiliser beaucoup d'optimisations, TQL se retrouve premier de ce test de performance, suivi de près par la bibliothèque C++ `ODB` qui génère une partie du SQL à la compilation (contrairement à TQL qui génère tout le SQL à la compilation). `Django` et `Hibernate` génère le SQL à l'exécution, ce qui les rend plus lent.

Ces tests de performance ont été écrits par moi et exécutent 4000 requêtes SQL. Les différents programmes ont été exécutés trois fois et le meilleur temps a été retenu.

En outre, le code à écrire par le programmeur n'est pas plus long comparé aux autres bibliothèques, comme le montrent les graphiques suivants :

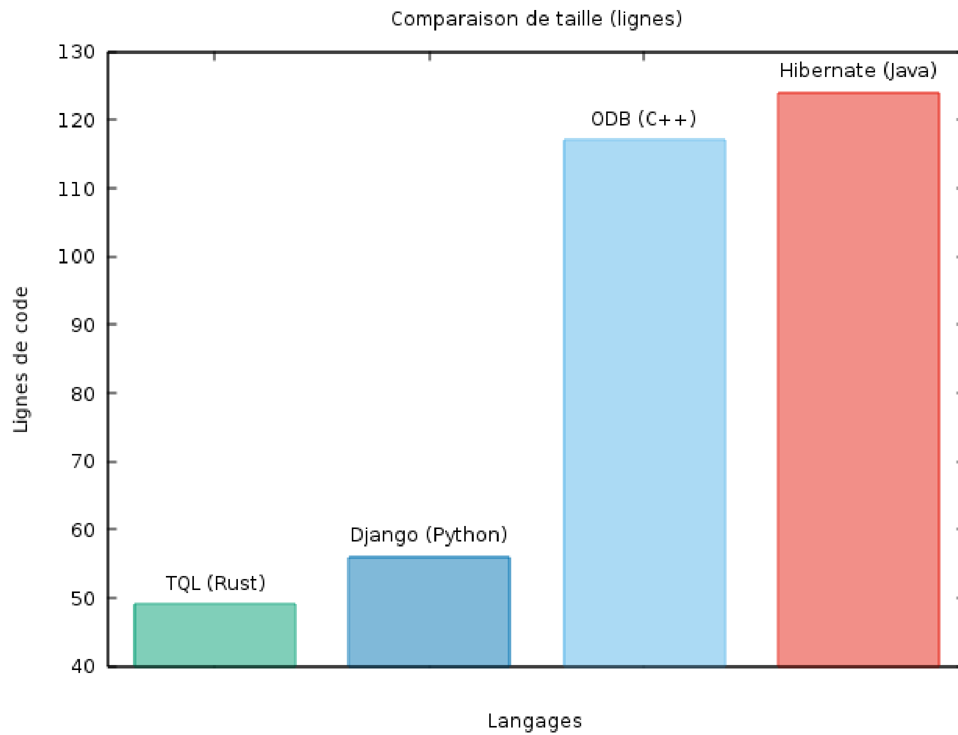


Figure 2. Nombre de lignes dans les codes utilisés pour le test de performance

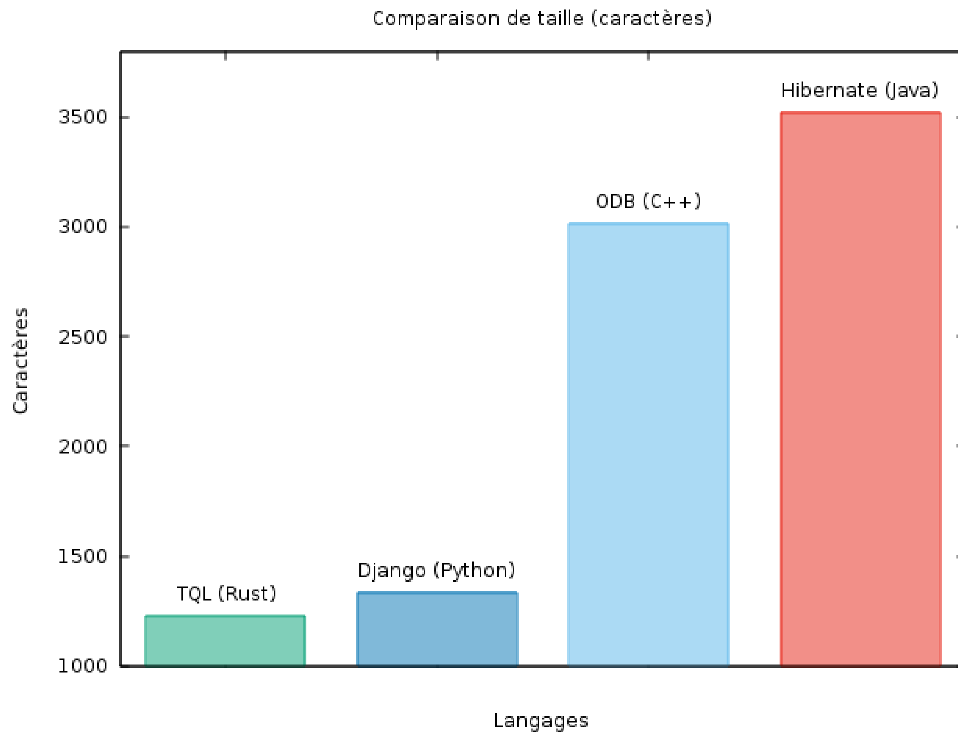


Figure 3. Nombre de caractères dans les codes utilisés pour le test de performance

Grâce à un usage judicieux des macros procédurales, TQL arrive à fournir une syntaxe concise et connue d'un programmeur Rust.

Finalement, il est tout à fait possible d’utiliser TQL pour un projet de programmation nécessitant l’usage d’une base de données, sauf pour les cas qui ne sont pas encore gérés par ce module d’extension. Dans ce cas, il est toujours possible d’utiliser SQL directement, sans toutefois bénéficier de l’analyse sémantique. Pour obtenir des erreurs à la compilation lors de l’utilisation directe du SQL, il est possible d’utiliser le module d’extension rust-postgres-macros.

Annexes

Annexe A: Code source de la bibliothèque

TQL fournit une bibliothèque, composée d'un unique module, fournissant les types `ForeignKey<T>` et `PrimaryKey` afin que l'utilisateur puisse ajouter des clés étrangères et des clés primaires dans les structures de table qu'il définit.

src/lib.rs

```
//! TQL is a Rust compiler plugin providing an SQL DSL.
//!
//! It type check your expression at compile time and converts it to SQL.

/// The `ForeignKey` is optional.
///
/// There is no value when the `join()` method is not called.
pub type ForeignKey<T> = Option<T>;

/// A `PrimaryKey` is a 4-byte integer.
pub type PrimaryKey = i32;
```

Annexe B: Code source de l’extension syntaxique

Le module `analyzer` vérifie que la suite d’appels de méthode est valide (par exemple, il est interdit d’appeler `update()` et `delete()` dans une même requête, car cela ne fait aucun sens). De surcroît, ce module vérifie que les méthodes sont appelées avec le bon nombre d’arguments. Il vérifie également que ces méthodes existent. Il fournit des méthodes utilitaires pour ses sous-modules. De plus, ce module vérifie si une requête de suppression a un filtre : dans le cas contraire, un avertissement est généré (car cela détruit toutes les données d’une table et c’est rarement le comportement voulu). Un avertissement est également généré dans le cas d’une table qui n’a pas de clé primaire. Enfin, ce module construit l’AST à partir de la suite d’appels de méthode.

tql_macros/src/analyzer/mod.rs

```
//! Semantic analyzer.

use std::borrow::Cow;
use std::fmt::Display;

use syntax::ast::Expr;
use syntax::ast::Expr_::{ExprLit, ExprPath};
use syntax::ast::FloatTy;
use syntax::ast::IntTy;
use syntax::ast::Lit_::{LitBool, LitByte, LitByteStr, LitChar, LitFloat,
LitFloatUnsuffixed, LitInt, LitStr};
use syntax::ast::LitIntType::{SignedIntLit, UnsignedIntLit, UnsuffixedIntLit};
use syntax::ast::UintTy;
use syntax::codemap::{Span, Spanned};
use syntax::ptr::P;

mod aggregate;
mod assignment;
mod filter;
mod get;
mod insert;
mod join;
mod limit;
mod sort;

use ast::{self, Aggregate, AggregateFilterExpression, Assignment, Expression, FieldList,
FilterExpression, FilterValue, Groups, Identifier, Join, Limit, Order, Query,
TypedField};
use error::{SqlError, SqlResult, res};
use gen::ToSql;
use parser::{MethodCall, MethodCalls};
use plugin::number_literal;
use self::aggregate::{argument_to_aggregate, argument_to_group,
expression_to_aggregate_filter_expression};
```

```rust
use self::assignment::{analyze_assignments_types, argument_to_assignment};
use self::filter::{analyze_filter_types, expression_to_filter_expression};
use self::get::get_expression_to_filter_expression;
use self::insert::check_insert_arguments;
use self::join::argument_to_join;
use self::limit::{analyze_limit_types, argument_to_limit};
use self::sort::argument_to_order;
use state::{SqlTable, SqlTables, get_field_type, methods_singleton, tables_singleton};
use string::{find_near, plural_verb};
use types::Type;

/// The type of the SQL query.
enum SqlQueryType {
    Aggregate,
    CreateTable,
    Delete,
    Drop,
    Insert,
    Select,
    Update,

}

impl Default for SqlQueryType {
    fn default() -> SqlQueryType {
        SqlQueryType::Select
    }
}

/// The query data gathered during the analysis.
#[derive(Default)]
struct QueryData {
    // Aggregate
    aggregate_filter: AggregateFilterExpression,
    aggregates: Vec<Aggregate>,
    groups: Groups,
    // Aggregate, Delete, Select, Update
    filter: FilterExpression,
    // Aggregate / Select
    joins: Vec<Join>,
    // Create
    fields_to_create: Vec<TypedField>,
    // Insert / Update
    assignments: Vec<Assignment>,
    // Select
    fields: FieldList,
    limit: Limit,
    order: Vec<Order>,
```

```
    // All
    query_type: SqlQueryType,
}

/// Analyze and transform the AST.
pub fn analyze(method_calls: MethodCalls, sql_tables: &SqlTables) -> SqlResult<Query> {
    let mut errors = vec![];

    // Check if the table exists.
    let table_name = method_calls.name.clone();
    if !sql_tables.contains_key(&table_name) {
        unknown_table_error(&table_name, method_calls.position, sql_tables, &mut errors);
    }

    check_methods(&method_calls, &mut errors);
    check_method_calls_validity(&method_calls, &mut errors);

    let table = sql_tables.get(&table_name);
    let calls = &method_calls.calls;
    let mut delete_position = None;

    // Get all the data from the query.
    let query_data =
        match table {
            Some(table) => {
                let mut query_data = try!(process_methods(&calls, table, &mut
delete_position));
                let fields = get_query_fields(table, &query_data.joins, sql_tables);
                query_data.fields = fields;
                query_data

            },
            None => QueryData::default(),
        };

    let query = new_query(query_data, table_name);

    check_delete_without_filters(&query, delete_position, &mut errors);

    res(query, errors)
}

/// Analyze the literal types in the `Query`.
pub fn analyze_types(query: Query) -> SqlResult<Query> {
    let mut errors = vec![];
    match query {
        Query::Aggregate { ref filter, ref table, .. } => {
            analyze_filter_types(filter, &table, &mut errors);
```

```
        },
        Query::CreateTable { .. } => (), // Nothing to analyze.
        Query::Delete { ref filter, ref table } => {
            analyze_filter_types(filter, &table, &mut errors);
        },
        Query::Drop { .. } => (), // Nothing to analyze.
        Query::Insert { ref assignments, ref table } => {
            analyze_assignments_types(assignments, &table, &mut errors);
        },
        Query::Select { ref filter, ref limit, ref table, .. } => {
            analyze_filter_types(filter, &table, &mut errors);
            analyze_limit_types(limit, &mut errors);
        },
        Query::Update { ref assignments, ref filter, ref table } => {
            analyze_filter_types(filter, &table, &mut errors);
            analyze_assignments_types(assignments, &table, &mut errors);
        },
    }
    res(query, errors)
}

/// Check that the `arguments` vector contains `expected_count` elements.
/// If this is not the case, add an error to `errors`.
fn check_argument_count(arguments: &[Expression], expected_count: usize, position: Span,
errors: &mut Vec<SqlError>) -> bool {
    if arguments.len() == expected_count {
        true
    }
    else {
        let length = arguments.len();
        errors.push(SqlError::new_with_code(
            &format!("this function takes 1 parameter but {} parameter{} supplied",
length, plural_verb(length)),
            position,
            "E0061",
        ));
        false
    }
}

/// Check that `Delete` `Query` contains a filter.
fn check_delete_without_filters(query: &Query, delete_position: Option<Span>, errors:
&mut Vec<SqlError>) {
    if let Query::Delete { ref filter, .. } = *query {
        if let FilterExpression::NoFilters = *filter {
            errors.push(SqlError::new_warning(
                "delete() without filters",
                delete_position.unwrap(), // There is always a delete position when the
```

```
query is of type Delete.
            ));
        }
    }
}

/// Check if the `identifier` is a field in the struct `table_name`.
pub fn check_field(identifier: &str, position: Span, table: &SqlTable, errors: &mut
Vec<SqlError>) {
    if !table.fields.contains_key(identifier) {
        errors.push(SqlError::new(
            &format!("attempted access of field `{field}` on type `{table}`, but no field
with that name was found",
                    field = identifier,
                    table = table.name
                   ),
            position
        ));
        let field_names = table.fields.keys();
        propose_similar_name(identifier, field_names, position, errors);
    }
}

/// Check if the type of `identifier` matches the type of the `value` expression.
fn check_field_type(table_name: &str, filter_value: &FilterValue, value: &Expression,
errors: &mut Vec<SqlError>) {
    let field_type = get_field_type_by_filter_value(table_name, filter_value);
    check_type(field_type, value, errors);
}

/// Check if the method calls sequence is valid.
/// For instance, one cannot call both insert() and delete() methods in the same query.
fn check_method_calls_validity(method_calls: &MethodCalls, errors: &mut Vec<SqlError>) {
    let method_map =
        hashmap!{
            "aggregate" => vec!["filter", "join", "values"],
            "all" => vec!["filter", "get", "join", "limit", "sort"],
            "create" => vec![],
            "delete" => vec!["filter", "get"],
            "drop" => vec![],
            "insert" => vec![],
            "update" => vec!["filter", "get"],
        };

    let main_method = method_calls.calls.iter()
        .filter(|call| method_map.contains_key(&*call.name) )
        .next()
        .map(|call| call.name.as_str())
```

```
        .unwrap_or("all");

    let mut valid_methods = vec![main_method];
    valid_methods.append(&mut method_map[&main_method].clone());

    let methods = get_methods();
    let invalid_methods = method_calls.calls.iter()
        .filter(|call| methods.contains(&call.name) &&
!valid_methods.contains(&&*call.name));

    for method in invalid_methods {
        errors.push(SqlError::new(
            &format!("cannot call the {method}() method with the {main_method}() method",
                method = method.name,
                main_method = main_method
            ),
            method.position,
        ));
    }
}

/// Check if the method `calls` exist.
fn check_methods(method_calls: &MethodCalls, errors: &mut Vec<SqlError>) {
    let methods = get_methods();
    for method_call in &method_calls.calls {
        if !methods.contains(&method_call.name) {
            errors.push(SqlError::new(
                &format!("no method named `{method}` found in tql",
                        method = method_call.name
                       ),
                method_call.position,
            ));
            propose_similar_name(&method_call.name, methods.iter(), method_call.position,
errors);
        }
    }

    if method_calls.calls.is_empty() {
        let table_name = &method_calls.name;
        errors.push(SqlError::new_with_code(
            &format!("`{table}` is the name of a struct, but this expression uses it like
a method name",
                    table = table_name
                   ),
            method_calls.position, "E0423"
        ));
        errors.push(SqlError::new_help(
            &format!("did you mean to write `{table}.method()`?",
```

```
                table = table_name
            ),
            method_calls.position,
        ));
    }
}

/// Check that the specified method call did not received any arguments.
fn check_no_arguments(method_call: &MethodCall, errors: &mut Vec<SqlError>) {
    if !method_call.arguments.is_empty() {
        let length = method_call.arguments.len();
        errors.push(SqlError::new_with_code(
            &format!("this method takes 0 parameters but {param_count} parameter{plural}
supplied",
                    param_count = length,
                    plural = plural_verb(length)
                   ),
            method_call.position, "E0061"
       ));
    }
}

/// Check if the `field_type` is compatible with the `expression`'s type.
pub fn check_type(field_type: &Type, expression: &Expression, errors: &mut Vec<SqlError>)
{
    if field_type != expression {
        let literal_type = get_type(expression);
        mismatched_types(field_type, &literal_type, expression.span, errors);
    }
}

/// Check if the `field_type` is compatible with the `filter_value`'s type.
fn check_type_filter_value(expected_type: &Type, filter_value: &Spanned<FilterValue>,
table_name: &str, errors: &mut Vec<SqlError>) {
    let field_type = get_field_type_by_filter_value(table_name, &filter_value.node);
    if *field_type != *expected_type {
        mismatched_types(expected_type, &field_type, filter_value.span, errors);
    }
}

/// Convert the `arguments` to the `Type`.
fn convert_arguments<F, Type>(arguments: &[P<Expr>], table: &SqlTable, convert_argument:
F) -> SqlResult<Vec<Type>>
    where F: Fn(&Expression, &SqlTable) -> SqlResult<Type>
{
    let mut items = vec![];
    let mut errors = vec![];
```

```
    for arg in arguments {
        try(convert_argument(arg, table), &mut errors, |item| {
            items.push(item);
        });
    }

    res(items, errors)
}

/// Get the type of the field if it exists from an `FilterValue`.
fn get_field_type_by_filter_value<'a>(table_name: &'a str, filter_value: &FilterValue) ->
&'a Type {
    // NOTE: At this stage (type analysis), the field exists, hence unwrap().
    match *filter_value {
        FilterValue::Identifier(ref identifier) => {
            get_field_type(table_name, identifier).unwrap()
        },
        FilterValue::MethodCall(ast::MethodCall { ref method_name, ref object_name, .. })
=> {
            let tables = tables_singleton();
            let table = tables.get(table_name).unwrap();
            let methods = methods_singleton();
            let typ = table.fields.get(object_name).unwrap();
            let typ =
                match typ.node {
                    // NOTE: return a Generic Type because Option methods work
independently from
                    // the nullable type (for instance, is_some()).
                    Type::Nullable(_) => Cow::Owned(Type::Nullable(box Type::Generic)),
                    ref typ => Cow::Borrowed(typ),
                };
            let type_methods = methods.get(&typ).unwrap();
            let method = type_methods.get(method_name).unwrap();
            &method.return_type
        },
    }
}

/// Get all the existing methods.
fn get_methods() -> Vec<String> {
    vec![
        "aggregate".to_owned(),
        "all".to_owned(),
        "create".to_owned(),
        "delete".to_owned(),
        "drop".to_owned(),
        "filter".to_owned(),
        "get".to_owned(),
```

```
        "insert".to_owned(),
        "join".to_owned(),
        "limit".to_owned(),
        "sort".to_owned(),
        "update".to_owned(),
        "values".to_owned(),
    ]
}

/// Get the query field fully qualified names.
fn get_query_fields(table: &SqlTable, joins: &[Join], sql_tables: &SqlTables) ->
Vec<Identifier> {
    let mut fields = vec![];
    for (field, typ) in &table.fields {
        match typ.node {
            Type::Custom(ref foreign_table) => {
                let table_name = foreign_table;
                if let Some(foreign_table) = sql_tables.get(foreign_table) {
                    if has_joins(&joins, &field) {
                        for (field, typ) in &foreign_table.fields {
                            match typ.node {
                                Type::Custom(_) | Type::UnsupportedType(_) => (), //
NOTE: Do not add foreign key recursively.
                                _ => {
                                    fields.push(table_name.clone() + "." + &field);
                                },
                            }
                        }
                    }
                }
            },
            Type::UnsupportedType(_) => (),
            _ => {
                fields.push(table.name.to_owned() + "." + &field);
            },
        }
    }
    fields
}

/// Get the string representation of an literal `Expression` type.
/// Useful to show in an error.
fn get_type(expression: &Expression) -> &str {
    match expression.node {
        ExprLit(ref literal) => {
            match literal.node {
                LitBool(_) => "bool",
                LitByte(_) => "u8",
```

```
                LitByteStr(_) => "Vec<u8>",
                LitChar(_) => "char",
                LitFloat(_, FloatTy::TyF32) => "f32",
                LitFloat(_, FloatTy::TyF64) => "f64",
                LitFloatUnsuffixed(_) => "floating-point variable",
                LitInt(_, int_type) =>
                    match int_type {
                        SignedIntLit(IntTy::TyIs, _) => "isize",
                        SignedIntLit(IntTy::TyI8, _) => "i8",
                        SignedIntLit(IntTy::TyI16, _) => "i16",
                        SignedIntLit(IntTy::TyI32, _) => "i32",
                        SignedIntLit(IntTy::TyI64, _) => "i64",
                        UnsignedIntLit(UintTy::TyUs) => "usize",
                        UnsignedIntLit(UintTy::TyU8) => "u8",
                        UnsignedIntLit(UintTy::TyU16) => "u16",
                        UnsignedIntLit(UintTy::TyU32) => "u32",
                        UnsignedIntLit(UintTy::TyU64) => "u64",
                        UnsuffixedIntLit(_) => "integral variable",
                    }
                ,
                LitStr(_, _) => "String",
            }
        }
        _ => panic!("expression needs to be a literal"),
    }
}

/// Check if there is a join in `joins` on a field named `name`.
pub fn has_joins(joins: &[Join], name: &str) -> bool {
    joins.iter()
        .map(|join| &join.base_field)
        .any(|field_name| field_name == name)
}

/// Add a mismatched types error to `errors`.
fn mismatched_types<S: Display, T: Display>(expected_type: S, actual_type: &T, position:
Span, errors: &mut Vec<SqlError>) {
    errors.push(SqlError::new_with_code(
        &format!("mismatched types:\n expected `{expected_type}`,\n    found
`{actual_type}`",
            expected_type = expected_type,
            actual_type = actual_type
        ),
        position,
        "E0308",
    ));
    errors.push(SqlError::new_note(
        "in this expansion of sql! (defined in tql)",
```

```
        position,
    ));
}

/// Create a new query from all the data gathered by the method calls.
fn new_query(QueryData { fields, filter, joins, limit, order, assignments,
fields_to_create, aggregates, groups, aggregate_filter, query_type }: QueryData,
table_name: String) -> Query {
    match query_type {
        SqlQueryType::Aggregate =>
            Query::Aggregate {
                aggregates: aggregates,
                aggregate_filter: aggregate_filter,
                filter: filter,
                groups: groups,
                joins: joins,
                table: table_name,
            },
        SqlQueryType::CreateTable =>
            Query::CreateTable {
                fields: fields_to_create,
                table: table_name,
            },
        SqlQueryType::Delete =>
            Query::Delete {
                filter: filter,
                table: table_name,
            },
        SqlQueryType::Drop =>
            Query::Drop {
                table: table_name,
            },
        SqlQueryType::Insert =>
            Query::Insert {
                assignments: assignments,
                table: table_name,
            },
        SqlQueryType::Select =>
            Query::Select {
                fields: fields,
                filter: filter,
                joins: joins,
                limit: limit,
                order: order,
                table: table_name,
            },
        SqlQueryType::Update =>
            Query::Update {
```

```
                assignments: assignments,
                filter: filter,
                table: table_name,
            },
    }
}

/// Create an error about a table not having a primary key.
pub fn no_primary_key(table_name: &str, position: Span) -> SqlError {
    SqlError::new(
        &format!("Table {table} does not have a primary key",
            table = table_name
        ),
        position
    )
}

/// Convert an `Expression` to an `Identifier` if `expression` is an `ExprPath`.
/// It adds an error to `errors` if `expression` is not an `ExprPath`.
fn path_expr_to_identifier(expression: &Expression, errors: &mut Vec<SqlError>) ->
Option<Identifier> {
    if let ExprPath(_, ref path) = expression.node {
        let identifier = path.segments[0].identifier.to_string();
        Some(identifier)
    }
    else {
        errors.push(SqlError::new(
            "Expected identifier",
            expression.span,
        ));
        None
    }
}

/// Gather data about the query in the method `calls`.
/// Also analyze the types.
fn process_methods(calls: &[MethodCall], table: &SqlTable, delete_position: &mut
Option<Span>) -> SqlResult<QueryData> {
    let mut errors = vec![];
    let mut query_data = QueryData::default();

    for method_call in calls {
        match &method_call.name[..] {
            "aggregate" => {
                try(convert_arguments(&method_call.arguments, table,
argument_to_aggregate), &mut errors, |aggrs| {
                    query_data.aggregates = aggrs;
                });
```

```
                query_data.query_type = SqlQueryType::Aggregate;
            },
            "all" => {
                check_no_arguments(&method_call, &mut errors);
            },
            "create" => {
                check_no_arguments(&method_call, &mut errors);
                query_data.query_type = SqlQueryType::CreateTable;
                for (field, typ) in &table.fields {
                    query_data.fields_to_create.push(TypedField {
                        identifier: field.clone(),
                        typ: typ.node.to_sql(),
                    });
                }
            },
            "delete" => {
                check_no_arguments(&method_call, &mut errors);
                query_data.query_type = SqlQueryType::Delete;
                *delete_position = Some(method_call.position);
            },
            "drop" => {
                check_no_arguments(&method_call, &mut errors);
                query_data.query_type = SqlQueryType::Drop;
            },
            "filter" => {
                if query_data.aggregates.is_empty() {
                    // If the aggregate() method was not called, filter() filters on the
values
                    // (WHERE).
                    try(expression_to_filter_expression(&method_call.arguments[0],
table), &mut errors, |filter| {
                        query_data.filter = filter;
                    });
                }
                else {
                    // If the aggregate() method was called, filter() filters on the
aggregated
                    // values (HAVING).

try(expression_to_aggregate_filter_expression(&method_call.arguments[0],
&query_data.aggregates, table), &mut errors, |filter| {
                        query_data.aggregate_filter = filter;
                    });
                }
            },
            "get" => {
                if method_call.arguments.is_empty() {
                    query_data.limit = Limit::Index(number_literal(0));
```

```
                }
                else {
                    try(get_expression_to_filter_expression(&method_call.arguments[0],
table), &mut errors, |(filter, new_limit)| {
                        query_data.filter = filter;
                        query_data.limit = new_limit;
                    });
                }
            },
            "insert" => {
                try(convert_arguments(&method_call.arguments, table,
argument_to_assignment), &mut errors, |assigns| {
                    query_data.assignments = assigns;
                });
                if !query_data.assignments.is_empty() {
                    check_insert_arguments(&query_data.assignments, method_call.position,
&table, &mut errors);
                }
                query_data.query_type = SqlQueryType::Insert;
            },
            "join" => {
                try(convert_arguments(&method_call.arguments, table, argument_to_join),
&mut errors, |mut new_joins| {
                    query_data.joins.append(&mut new_joins);
                });
            },
            "limit" => {
                try(argument_to_limit(&method_call.arguments[0]), &mut errors,
|new_limit| {
                    query_data.limit = new_limit;
                });
            },
            "sort" => {
                try(convert_arguments(&method_call.arguments, table, argument_to_order),
&mut errors, |new_order| {
                    query_data.order = new_order;
                });
            },
            "update" => {
                try(convert_arguments(&method_call.arguments, table,
argument_to_assignment), &mut errors, |assigns| {
                    query_data.assignments = assigns;
                });
                query_data.query_type = SqlQueryType::Update;
            },
            "values" => {
                try(convert_arguments(&method_call.arguments, table, argument_to_group),
&mut errors, |new_groups| {
```

```
                    query_data.groups = new_groups;
                });
            },
            _ => (), // NOTE: Nothing to do since check_methods() check for unknown
method.
        }
    }
    res(query_data, errors)
}

/// Check if a name similar to `identifier` exists in `choices` and show a message if one
exists.
/// Returns true if a similar name was found.
pub fn propose_similar_name<'a, T>(identifier: &str, choices: T, position: Span, errors:
&mut Vec<SqlError>) -> bool
    where T: Iterator<Item = &'a String>
{
    if let Some(name) = find_near(&identifier, choices) {
        errors.push(SqlError::new_help(
            &format!("did you mean {}?", name),
            position,
        ));
        true
    }
    else {
        false
    }
}

/// If `result` is an `Err`, add the errors to `errors`.
/// Otherwise, execute the closure.
fn try<F: FnMut(T), T>(mut result: Result<T, Vec<SqlError>>, errors: &mut Vec<SqlError>,
mut fn_using_result: F) {
    match result {
        Ok(value) => fn_using_result(value),
        Err(ref mut errs) => errors.append(errs),
    }
}

/// Add an error to the vector `errors` about an unknown SQL table.
/// It suggests a similar name if there is one.
pub fn unknown_table_error(table_name: &str, position: Span, sql_tables: &SqlTables,
errors: &mut Vec<SqlError>) {
    errors.push(SqlError::new_with_code(
        &format!("`{table}` does not name an SQL table",
                table = table_name
        ),
        position,
```

```
            "E0422",
        ));
        let tables = sql_tables.keys();
        if !propose_similar_name(&table_name, tables, position, errors) {
            errors.push(SqlError::new_help(
                &format!("did you forget to add the #[SqlTable] attribute on the {table}
struct?",
                        table = table_name
                ),
                position,
            ));
        }
}
```

Le module `analyzer::aggregate` effectue la conversion d'une expression Rust dans la structure `Aggregate`, vérifie que l'expression est valide et que la fonction d'agrégat existe.

tql_macros/src/analyzer/aggregate.rs

```
/// Analyzer for the aggregate() method.

use syntax::ast::{BinOp_, ExprAssign, ExprCall, ExprParen, ExprPath, ExprUnary};
use syntax::ast::Expr_::ExprBinary;
use syntax::ast::UnOp;
use syntax::codemap::{Span, Spanned};

use ast::{Aggregate, AggregateFilter, AggregateFilterExpression, AggregateFilters,
Expression, Identifier};
use error::{SqlError, SqlResult, res};
use state::{SqlTable, aggregates_singleton};
use super::{check_argument_count, check_field, path_expr_to_identifier,
propose_similar_name};
use super::filter::{binop_to_logical_operator, binop_to_relational_operator,
is_logical_operator, is_relational_operator};

/// Convert an `Expression` to an `Aggregate`.
pub fn argument_to_aggregate(arg: &Expression, _table: &SqlTable) -> SqlResult<Aggregate>
{
    let mut errors = vec![];
    let mut aggregate = Aggregate::default();
    let aggregates = aggregates_singleton();

    let call = get_call_from_aggregate(arg, &mut aggregate, &mut errors);

    if let ExprCall(ref function, ref arguments) = call.node {
        if let Some(identifier) = path_expr_to_identifier(function, &mut errors) {
            if let Some(sql_function) = aggregates.get(&identifier) {
```

```
                aggregate.function = sql_function.clone();
            }
            else {
                errors.push(SqlError::new_with_code(
                    &format!("unresolved name `{}`", identifier),
                    arg.span,
                    "E0425",
                ));
                propose_similar_name(&identifier, aggregates.keys(), arg.span, &mut
errors);
            }
        }

        if check_argument_count(arguments, 1, arg.span, &mut errors) {
            if let ExprPath(_, ref path) = arguments[0].node {
                aggregate.field = path.segments[0].identifier.to_string();

                if aggregate.result_name.is_empty() {
                    aggregate.result_name = aggregate.field.clone() + "_" +
&aggregate.function.to_lowercase();
                }
            }
        }
    }
    else {
        errors.push(SqlError::new(
            "Expected function call",
            arg.span,
        ));
    }

    res(aggregate, errors)
}

/// Convert an `Expression` to a group `Identifier`.
pub fn argument_to_group(arg: &Expression, table: &SqlTable) -> SqlResult<Identifier> {
    let mut errors = vec![];
    let mut group = "".to_owned();

    if let Some(identifier) = path_expr_to_identifier(arg, &mut errors) {
        check_field(&identifier, arg.span, table, &mut errors);
        group = identifier;
    }

    res(group, errors)
}

/// Convert a Rust binary expression to an `AggregateFilterExpression` for an aggregate
```

```
filter.
fn binary_expression_to_aggregate_filter_expression(expr1: &Expression, op: BinOp_,
expr2: &Expression, aggregates: &[Aggregate], table: &SqlTable) ->
SqlResult<AggregateFilterExpression> {
    let filter1 = try!(expression_to_aggregate_filter_expression(expr1, aggregates,
table));
    let dummy = AggregateFilterExpression::NoFilters;

    let filter =
        if is_logical_operator(op) {
            let filter2 = try!(expression_to_aggregate_filter_expression(expr2,
aggregates, table));
            AggregateFilterExpression::Filters(AggregateFilters {
                operand1: Box::new(filter1),
                operator: binop_to_logical_operator(op),
                operand2: Box::new(filter2),
            })
        }
        else if is_relational_operator(op) {
            if let AggregateFilterExpression::FilterValue(filter1) = filter1 {
                AggregateFilterExpression::Filter(AggregateFilter {
                    operand1: filter1.node,
                    operator: binop_to_relational_operator(op),
                    operand2: expr2.clone(),
                })
            }
            else {
                dummy
            }
        }
        else {
            dummy
        };
    Ok(filter)
}

/// Check that an aggregate field exists.
fn check_aggregate_field<'a>(identifier: &str, aggregates: &'a [Aggregate], position:
Span, errors: &mut Vec<SqlError>) -> Option<&'a Aggregate> {
    let result = aggregates.iter().find(|aggr| aggr.result_name == identifier);
    if let None = result {
        errors.push(SqlError::new(
            &format!("no aggregate field named `{}` found", identifier),
            position
        ));
    }
    result
}
```

```rust
/// Convert a Rust expression to an `AggregateFilterExpression` for an aggregate filter.
pub fn expression_to_aggregate_filter_expression(arg: &Expression, aggregates:
&[Aggregate], table: &SqlTable) -> SqlResult<AggregateFilterExpression> {
    let mut errors = vec![];

    let filter =
        match arg.node {
            ExprBinary(Spanned { node: op, .. }, ref expr1, ref expr2) => {
                try!(binary_expression_to_aggregate_filter_expression(expr1, op, expr2,
aggregates, table))
            },
            ExprPath(None, ref path) => {
                let identifier = path.segments[0].identifier.to_string();
                let aggregate = check_aggregate_field(&identifier, aggregates, path.span,
&mut errors);
                if let Some(aggregate) = aggregate {
                    // Transform the aggregate field name to its SQL representation.
                    AggregateFilterExpression::FilterValue(Spanned {
                        node: aggregate.clone(),
                        span: arg.span,
                    })
                }
                else {
                    AggregateFilterExpression::NoFilters
                }
            },
            ExprParen(ref expr) => {
                let filter = try!(expression_to_aggregate_filter_expression(expr,
aggregates, table));
                AggregateFilterExpression::ParenFilter(box filter)
            },
            ExprUnary(UnOp::UnNot, ref expr) => {
                let filter = try!(expression_to_aggregate_filter_expression(expr,
aggregates, table));
                AggregateFilterExpression::NegFilter(box filter)
            },
            _ => {
                errors.push(SqlError::new(
                    "Expected binary operation",
                    arg.span,
                ));
                AggregateFilterExpression::NoFilters
            },
        };

    res(filter, errors)
}
```

```
/// Get the call expression from an `arg` expression.
fn get_call_from_aggregate<'a>(arg: &'a Expression, aggregate: &mut Aggregate, errors:
&mut Vec<SqlError>) -> &'a Expression {
    // If the `arg` expression is an assignment, the call is on the right side.
    if let ExprAssign(ref left_value, ref right_value) = arg.node {
        if let Some(identifier) = path_expr_to_identifier(left_value, errors) {
            aggregate.result_name = identifier;
        }
        right_value
    }
    else {
        arg
    }
}
```

Le module `analyzer::assignment` effectue la conversion d'une expression Rust dans la structure `Assignment` (qui est utilisée pour les méthodes `insert()` et `update()`) et vérifie que les expressions utilisées ont un type compatible avec le champ.

tql_macros/src/analyzer/assignment.rs

```
/// Argument to assignment converter.

use syntax::ast::BinOp_;
use syntax::ast::Expr_::{ExprAssign, ExprAssignOp};
use syntax::codemap::Spanned;

use ast::{Assignment, AssignementOperator, Expression, FilterValue};
use error::{SqlError, SqlResult, res};
use plugin::number_literal;
use state::SqlTable;
use super::{check_field, check_field_type, path_expr_to_identifier};

/// Analyze the types of the `Assignment`s.
pub fn analyze_assignments_types(assignments: &[Assignment], table_name: &str, errors:
&mut Vec<SqlError>) {
    for assignment in assignments {
        check_field_type(table_name,
&FilterValue::Identifier(assignment.identifier.clone()), &assignment.value, errors);
    }
}

/// Convert an `Expression` to an `Assignment`.
pub fn argument_to_assignment(arg: &Expression, table: &SqlTable) ->
SqlResult<Assignment> {
    fn assign_values(assignment: &mut Assignment, expr1: &Expression, expr2: &Expression,
```

```rust
table: &SqlTable, errors: &mut Vec<SqlError>) {
        assignment.value = expr2.clone();
        if let Some(identifier) = path_expr_to_identifier(expr1, errors) {
            assignment.identifier = identifier;
            check_field(&assignment.identifier, expr1.span, table, errors);
        }
    }

    let mut errors = vec![];
    let mut assignment = Assignment {
        identifier: "".to_owned(),
        operator: Spanned {
            node: AssignementOperator::Equal,
            span: arg.span,
        },
        value: number_literal(0),
    };
    match arg.node {
        ExprAssign(ref expr1, ref expr2) => {
            assign_values(&mut assignment, expr1, expr2, table, &mut errors);
        },
        ExprAssignOp(ref binop, ref expr1, ref expr2) => {
            assignment.operator = Spanned {
                node: binop_to_assignment_operator(binop.node),
                span: binop.span,
            };
            assign_values(&mut assignment, expr1, expr2, table, &mut errors);
        },
        _ => {
            errors.push(SqlError::new(
                "Expected assignment",
                arg.span,
            ));
        },
    }
    res(assignment, errors)
}

/// Convert a `BinOp_` to an SQL `AssignmentOperator`.
fn binop_to_assignment_operator(binop: BinOp_) -> AssignementOperator {
    match binop {
        BinOp_::BiAdd => AssignementOperator::Add,
        BinOp_::BiSub => AssignementOperator::Sub,
        BinOp_::BiMul => AssignementOperator::Mul,
        BinOp_::BiDiv => AssignementOperator::Divide,
        BinOp_::BiRem => AssignementOperator::Modulo,
        BinOp_::BiAnd => unreachable!(),
        BinOp_::BiOr => unreachable!(),
```

```
            BinOp_::BiBitXor => unreachable!(),
            BinOp_::BiBitAnd => unreachable!(),
            BinOp_::BiBitOr => unreachable!(),
            BinOp_::BiShl => unreachable!(),
            BinOp_::BiShr => unreachable!(),
            BinOp_::BiEq => AssignementOperator::Equal,
            BinOp_::BiLt => unreachable!(),
            BinOp_::BiLe => unreachable!(),
            BinOp_::BiNe => unreachable!(),
            BinOp_::BiGe => unreachable!(),
            BinOp_::BiGt => unreachable!(),
        }
}
```

Le module `analyzer::filter` effectue la conversion d'une expression Rust dans la structure `FilterExpression` et vérifie que les expressions utilisées ont un type compatible avec le champ.

tql_macros/src/analyzer/filter.rs

```
/// Analyzer for the filter() method.

use syntax::ast::{BinOp_, Expr, Path, SpannedIdent};
use syntax::ast::Expr_::{ExprBinary, ExprMethodCall, ExprParen, ExprPath, ExprUnary};
use syntax::ast::UnOp;
use syntax::codemap::{Span, Spanned};
use syntax::ptr::P;

use ast::{self, Expression, Filter, FilterExpression, Filters, FilterValue,
LogicalOperator, RelationalOperator};
use error::{SqlError, SqlResult, res};
use state::{SqlMethod, SqlMethodTypes, SqlTable, methods_singleton};
use super::{check_field, check_field_type, check_type, check_type_filter_value,
propose_similar_name};
use types::Type;

/// Analyze the types of the `FilterExpression`.
pub fn analyze_filter_types(filter: &FilterExpression, table_name: &str, errors: &mut
Vec<SqlError>) {
    match *filter {
        FilterExpression::Filter(ref filter) => {
            check_field_type(table_name, &filter.operand1, &filter.operand2, errors);
        },
        FilterExpression::Filters(ref filters) => {
            analyze_filter_types(&*filters.operand1, table_name, errors);
            analyze_filter_types(&*filters.operand2, table_name, errors);
        },
        FilterExpression::NegFilter(ref filter) => {
```

```
            analyze_filter_types(filter, table_name, errors);
        },
        FilterExpression::NoFilters => (),
        FilterExpression::ParenFilter(ref filter) => {
            analyze_filter_types(filter, table_name, errors);
        },
        FilterExpression::FilterValue(ref filter_value) => {
            check_type_filter_value(&Type::Bool, filter_value, table_name, errors);
        },
    }
}

/// Convert a Rust binary expression to a `FilterExpression`.
fn binary_expression_to_filter_expression(expr1: &Expression, op: BinOp_, expr2:
&Expression, table: &SqlTable) -> SqlResult<FilterExpression> {
    let filter1 = try!(expression_to_filter_expression(expr1, table));
    let dummy = FilterExpression::NoFilters;

    let filter =
        if is_logical_operator(op) {
            let filter2 = try!(expression_to_filter_expression(expr2, table));
            FilterExpression::Filters(Filters {
                operand1: Box::new(filter1),
                operator: binop_to_logical_operator(op),
                operand2: Box::new(filter2),
            })
        }
        else if is_relational_operator(op) {
            if let FilterExpression::FilterValue(filter1) = filter1 {
                FilterExpression::Filter(Filter {
                    operand1: filter1.node,
                    operator: binop_to_relational_operator(op),
                    operand2: expr2.clone(),
                })
            }
            else {
                dummy
            }
        }
        else {
            dummy
        };
    Ok(filter)
}

/// Convert a `BinOp_` to an SQL `LogicalOperator`.
pub fn binop_to_logical_operator(binop: BinOp_) -> LogicalOperator {
    match binop {
```

```
        BinOp_::BiAdd => unreachable!(),
        BinOp_::BiSub => unreachable!(),
        BinOp_::BiMul => unreachable!(),
        BinOp_::BiDiv => unreachable!(),
        BinOp_::BiRem => unreachable!(),
        BinOp_::BiAnd => LogicalOperator::And,
        BinOp_::BiOr => LogicalOperator::Or,
        BinOp_::BiBitXor => unreachable!(),
        BinOp_::BiBitAnd => unreachable!(),
        BinOp_::BiBitOr => unreachable!(),
        BinOp_::BiShl => unreachable!(),
        BinOp_::BiShr => unreachable!(),
        BinOp_::BiEq => unreachable!(),
        BinOp_::BiLt => unreachable!(),
        BinOp_::BiLe => unreachable!(),
        BinOp_::BiNe => unreachable!(),
        BinOp_::BiGe => unreachable!(),
        BinOp_::BiGt => unreachable!(),
    }
}

/// Convert a `BinOp_` to an SQL `RelationalOperator`.
pub fn binop_to_relational_operator(binop: BinOp_) -> RelationalOperator {
    match binop {
        BinOp_::BiAdd => unreachable!(),
        BinOp_::BiSub => unreachable!(),
        BinOp_::BiMul => unreachable!(),
        BinOp_::BiDiv => unreachable!(),
        BinOp_::BiRem => unreachable!(),
        BinOp_::BiAnd => unreachable!(),
        BinOp_::BiOr => unreachable!(),
        BinOp_::BiBitXor => unreachable!(),
        BinOp_::BiBitAnd => unreachable!(),
        BinOp_::BiBitOr => unreachable!(),
        BinOp_::BiShl => unreachable!(),
        BinOp_::BiShr => unreachable!(),
        BinOp_::BiEq => RelationalOperator::Equal,
        BinOp_::BiLt => RelationalOperator::LesserThan,
        BinOp_::BiLe => RelationalOperator::LesserThanEqual,
        BinOp_::BiNe => RelationalOperator::NotEqual,
        BinOp_::BiGe => RelationalOperator::GreaterThan,
        BinOp_::BiGt => RelationalOperator::GreaterThanEqual,
    }
}

/// Check the type of the arguments of the method.
fn check_method_arguments(arguments: &[Expression], argument_types: &[Type], errors: &mut
Vec<SqlError>) {
```

```
    for (argument, argument_type) in arguments.iter().zip(argument_types) {
        check_type(argument_type, argument, errors)
    }
}

/// Convert a Rust expression to a `FilterExpression`.
pub fn expression_to_filter_expression(arg: &P<Expr>, table: &SqlTable) ->
SqlResult<FilterExpression> {
    let mut errors = vec![];

    let filter =
        match arg.node {
            ExprBinary(Spanned { node: op, .. }, ref expr1, ref expr2) => {
                try!(binary_expression_to_filter_expression(expr1, op, expr2, table))
            },
            ExprMethodCall(identifier, _, ref exprs) => {
                FilterExpression::FilterValue(Spanned {
                    node: method_call_expression_to_filter_expression(identifier, &exprs,
table, &mut errors),
                    span: arg.span,
                })
            },
            ExprPath(None, ref path) => {
                let identifier = path.segments[0].identifier.to_string();
                check_field(&identifier, path.span, table, &mut errors);
                FilterExpression::FilterValue(Spanned {
                    node: FilterValue::Identifier(identifier),
                    span: arg.span,
                })
            },
            ExprParen(ref expr) => {
                let filter = try!(expression_to_filter_expression(expr, table));
                FilterExpression::ParenFilter(box filter)
            },
            ExprUnary(UnOp::UnNot, ref expr) => {
                let filter = try!(expression_to_filter_expression(expr, table));
                FilterExpression::NegFilter(box filter)
            },
            _ => {
                errors.push(SqlError::new(
                    "Expected binary operation",
                    arg.span,
                ));
                FilterExpression::NoFilters
            },
        };

    res(filter, errors)
```

```
}

/// Get an SQL method and arguments by type and name.
fn get_method<'a>(object_type: &'a Spanned<Type>, exprs: &[Expression], method_name: &
str, identifier: SpannedIdent, errors: &mut Vec<SqlError>) -> Option<(&'a SqlMethodTypes,
Vec<Expression>)> {
    let methods = methods_singleton();
    let type_methods =
        if let Type::Nullable(_) = object_type.node {
            methods.get(&Type::Nullable(box Type::Generic))
        }
        else {
            methods.get(&object_type.node)
        };
    match type_methods {
        Some(type_methods) => {
            match type_methods.get(method_name) {
                Some(sql_method) => {
                    let arguments: Vec<Expression> = exprs[1..].iter().map(
Clone::clone).collect();
                    check_method_arguments(&arguments, &sql_method.argument_types,
errors);
                    Some((sql_method, arguments))
                },
                None => {
                    unknown_method(identifier.span, &object_type.node, method_name,
Some(type_methods), errors);
                    None
                },
            }
        },
        None => {
            unknown_method(identifier.span, &object_type.node, method_name, None,
errors);
            None
        },
    }
}

/// Check if a `BinOp_` is a `LogicalOperator`.
pub fn is_logical_operator(binop: BinOp_) -> bool {
    match binop {
        BinOp_::BiAnd | BinOp_::BiOr => true,
        _ => false,
    }
}

/// Check if a `BinOp_` is a `RelationalOperator`.
```

```
pub fn is_relational_operator(binop: BinOp_) -> bool {
    match binop {
        BinOp_::BiEq | BinOp_::BiLt | BinOp_::BiLe | BinOp_::BiNe | BinOp_::BiGe |
BinOp_::BiGt => true,
        _ => false,
    }
}

/// Convert a method call expression to a filter expression.
fn method_call_expression_to_filter_expression(identifier: SpannedIdent, exprs:
&[Expression], table: &SqlTable, errors: &mut Vec<SqlError>) -> FilterValue {
    let method_name = identifier.node.name.to_string();
    let dummy = FilterValue::Identifier("".to_owned());
    match exprs[0].node {
        ExprPath(_, ref path) => {
            path_method_call_to_filter(path, identifier, &method_name, exprs, table,
errors)
        },
        _ => {
            errors.push(SqlError::new(
                "expected identifier",
                exprs[0].span,
            ));
            dummy
        },
    }
}

/// Convert a method call where the object is an identifier to a filter expression.
fn path_method_call_to_filter(path: &Path, identifier: SpannedIdent, method_name: &str,
exprs: &[Expression], table: &SqlTable, errors: &mut Vec<SqlError>) -> FilterValue {
    let dummy = FilterValue::Identifier("".to_owned());
    let object_name = path.segments[0].identifier.name.to_string();
    match table.fields.get(&object_name) {
        Some(object_type) => {
            let type_method = get_method(object_type, exprs, method_name, identifier,
errors);

            if let Some((&SqlMethodTypes { ref template, .. }, ref arguments)) =
type_method {
                FilterValue::MethodCall(ast::MethodCall {
                    arguments: arguments.clone(),
                    method_name: method_name.to_owned(),
                    object_name: object_name,
                    template: template.clone(),
                })
            }
            else {
```

```
                // NOTE: An error is emitted in the get_method() function.
                dummy
            }
        },
        None => {
            check_field(&object_name, path.span, table, errors);
            dummy
        },
    }
}

/// Add an error to the vector `errors` about an unknown SQL method.
/// It suggests a similar name if there is one.
fn unknown_method(position: Span, object_type: &Type, method_name: &str, type_methods:
Option<&SqlMethod>, errors: &mut Vec<SqlError>) {
    errors.push(SqlError::new(
        &format!("no method named `{}` found for type `{}`", method_name, object_type),
        position,
    ));
    if let Some(type_methods) = type_methods {
        propose_similar_name(method_name, type_methods.keys(), position, errors);
    }
}
```

Le module `analyzer::get` effectue la conversion d’une expression Rust dans les structures `FilterExpression` et `Limit` et vérifie que les expressions utilisées ont un type compatible avec le champ.

tql_macros/src/analyzer/get.rs

```
/// Analyzer for the get() method.

use syntax::ast::Expr;
use syntax::ast::Expr_::{ExprLit, ExprPath};
use syntax::ptr::P;

use ast::{Filter, FilterExpression, FilterValue, Limit, RelationalOperator};
use error::{SqlResult, res};
use plugin::number_literal;
use state::{SqlTable, get_primary_key_field};
use super::no_primary_key;
use super::filter::expression_to_filter_expression;

/// Convert an expression from a `get()` method to a FilterExpression and a Limit.
pub fn get_expression_to_filter_expression(arg: &P<Expr>, table: &SqlTable) ->
SqlResult<(FilterExpression, Limit)> {
    let primary_key_field = get_primary_key_field(table);
    match primary_key_field {
        Some(primary_key_field) =>
            match arg.node {
                ExprLit(_) | ExprPath(_, _) => {
                    let filter = FilterExpression::Filter(Filter {
                        operand1: FilterValue::Identifier(primary_key_field),
                        operator: RelationalOperator::Equal,
                        operand2: arg.clone(),
                    });
                    res((filter, Limit::NoLimit), vec![])
                },
                _ => expression_to_filter_expression(arg, table)
                        .and_then(|filter| Ok((filter, Limit::Index(number_literal(
0))))),
            },
        None => Err(vec![no_primary_key(&table.name, table.position)]),
    }
}
```

Le module `analyzer::insert` vérifie que la méthode `insert()` est appelée comme il faut avec tous les champs obligatoires.

tql_macros/src/analyzer/insert.rs

```
/// Analyzer for the insert() method.

use std::collections::HashSet;

use syntax::codemap::{Span, Spanned};

use ast::{Assignment, AssignementOperator};
use error::SqlError;
use state::{SqlTable, get_primary_key_field};
use types::Type;

/// Check that the method call contains all the fields from the `table` and that all
assignments
/// does not use an operation (e.g. +=).
pub fn check_insert_arguments(assignments: &[Assignment], position: Span, table:
&SqlTable, errors: &mut Vec<SqlError>) {
    let mut fields = HashSet::new();
    let mut missing_fields: Vec<&str> = vec![];

    // Check the assignment operators.
    for assignment in assignments {
        fields.insert(assignment.identifier.clone());
        let operator = &assignment.operator.node;
        if *operator != AssignementOperator::Equal {
            errors.push(SqlError::new(&format!("expected = but got {}", *operator),
assignment.operator.span));
        }
    }
    let primary_key = get_primary_key_field(&table);

    for field in table.fields.keys() {
        if !fields.contains(field) && Some(field) != primary_key.as_ref() {
            if let Some(&Spanned { node: Type::Nullable(_), .. }) =
table.fields.get(field) {
                // Do not err about missing nullable field.
            }
            else {
                missing_fields.push(&field);
            }
        }
    }

    if !missing_fields.is_empty() {
```

```
        let fields = "`".to_owned() + &missing_fields.join("`, `") + "`";
        errors.push(SqlError::new_with_code(&format!("missing fields: {}", fields),
position, "E0063"));
    }
}
```

Le module `analyzer::join` convertit une expression dans la structure `Join`, qui est utilisée pour la génération de la jointure.

tql_macros/src/analyzer/join.rs

```
/// Analyzer for the join() method.

use syntax::codemap::Spanned;

use ast::{Expression, Join};
use error::{SqlResult, res};
use state::{SqlTable, get_primary_key_field, tables_singleton};
use super::{check_field, mismatched_types, no_primary_key, path_expr_to_identifier};
use types::Type;

/// Convert an `Expression` to a `Join`
pub fn argument_to_join(arg: &Expression, table: &SqlTable) -> SqlResult<Join> {
    let mut errors = vec![];
    let mut join = Join::default();

    if let Some(identifier) = path_expr_to_identifier(arg, &mut errors) {
        check_field(&identifier, arg.span, table, &mut errors);
        match table.fields.get(&identifier) {
            Some(&Spanned { node: ref field_type, .. }) => {
                if let &Type::Custom(ref related_table_name) = field_type {
                    let sql_tables = tables_singleton();
                    if let Some(related_table) = sql_tables.get(related_table_name) {
                        match get_primary_key_field(related_table) {
                            Some(primary_key_field) =>
                                join = Join {
                                    base_field: identifier,
                                    base_table: table.name.clone(),
                                    joined_field: primary_key_field,
                                    joined_table: related_table_name.clone(),
                                },
                            None => errors.push(no_primary_key(related_table_name,
related_table.position)),
                        }
                    }
                    // NOTE: if the field type is not an SQL table, an error is thrown by
the
```

```
                    // linter.
                }
                else {
                    mismatched_types("ForeignKey<_>", field_type, arg.span, &mut errors);
                }
            },
            None => (), // NOTE: This case is handled by the check_field() call above.
        }
    }
    res(join, errors)
}
```

Le module `analyzer::limit` convertit une expression dans la structure `Limit` et vérifie le type des arguments utilisés pour limiter le nombre de résultats d'une requête.

tql_macros/src/analyzer/limit.rs

```
/// Analyzer for the limit() method.

use syntax::ast::Expr;
use syntax::ast::Expr_::{ExprBinary, ExprCall, ExprCast, ExprLit, ExprMethodCall,
ExprPath, ExprRange, ExprUnary};
use syntax::ptr::P;

use ast::Limit;
use error::{SqlError, SqlResult, res};
use super::check_type;
use types::Type;

/// Analyze the types of the `Limit`.
pub fn analyze_limit_types(limit: &Limit, errors: &mut Vec<SqlError>) {
    match *limit {
        Limit::EndRange(ref expression) => check_type(&Type::I64, expression, errors),
        Limit::Index(ref expression) => check_type(&Type::I64, expression, errors),
        Limit::LimitOffset(ref expression1, ref expression2) => {
            check_type(&Type::I64, expression1, errors);
            check_type(&Type::I64, expression2, errors);
        },
        Limit::NoLimit => (),
        Limit::Range(ref expression1, ref expression2) => {
            check_type(&Type::I64, expression1, errors);
            check_type(&Type::I64, expression2, errors);
        },
        Limit::StartRange(ref expression) => check_type(&Type::I64, expression, errors),
    }
}
```

```
/// Convert an `Expression` to a `Limit`.
pub fn argument_to_limit(expression: &P<Expr>) -> SqlResult<Limit> {
    let mut errors = vec![];
    let limit =
        match expression.node {
            ExprRange(None, Some(ref range_end)) => {
                Limit::EndRange(range_end.clone())
            }
            ExprRange(Some(ref range_start), None) => {
                Limit::StartRange(range_start.clone())
            }
            ExprRange(Some(ref range_start), Some(ref range_end)) => {
                Limit::Range(range_start.clone(), range_end.clone())
            }
            ExprLit(_) | ExprPath(_, _) | ExprCall(_, _) | ExprMethodCall(_, _, _) |
ExprBinary(_, _, _) | ExprUnary(_, _) | ExprCast(_, _)  => {
                Limit::Index(expression.clone())
            }
            _ => {
                errors.push(SqlError::new(
                    "Expected index range or number expression",
                    expression.span,
                ));
                Limit::NoLimit
            }
        };

    res(limit, errors)
}
```

Le module `analyzer::sort` convertit une expression dans la structure `Order` et vérifie que l’expression passée en paramètre est bien un champ de la table.

tql_macros/src/analyzer/sort.rs

```
/// Analyzer for the sort() method.

use syntax::ast::Expr_::{ExprPath, ExprUnary};
use syntax::ast::UnOp;

use ast::{Expression, Order};
use error::{SqlError, SqlResult, res};
use state::SqlTable;
use super::{check_field, path_expr_to_identifier};

/// Convert an `Expression` to an `Order`.
pub fn argument_to_order(arg: &Expression, table: &SqlTable) -> SqlResult<Order> {
```

```
    let mut errors = vec![];
    let order =
        match arg.node {
            ExprUnary(UnOp::UnNeg, ref expr) => {
                let ident = try!(get_identifier(expr, table));
                Order::Descending(ident)
            }
            ExprPath(None, ref path) => {
                let identifier = path.segments[0].identifier.to_string();
                check_field(&identifier, path.span, table, &mut errors);
                Order::Ascending(identifier)
            }
            _ => {
                errors.push(SqlError::new(
                    "Expected - or identifier",
                    arg.span,
                ));
                Order::Ascending("".to_owned())
            }
        };
    res(order, errors)
}

/// Get the `String` indentifying the identifier from an `Expression`.
fn get_identifier(identifier_expr: &Expression, table: &SqlTable) -> SqlResult<String> {
    let mut errors = vec![];
    if let Some(identifier) = path_expr_to_identifier(identifier_expr, &mut errors) {
        check_field(&identifier, identifier_expr.span, table, &mut errors);
        res(identifier, errors)
    }
    else {
        Err(errors)
    }
}
```

Le module `arguments` extrait les arguments de l'AST d'une requête en fonction de son type.

tql_macros/src/arguments.rs

```
//! Query arguments extractor.

use syntax::ast::Expr_::ExprLit;
use syntax::ext::base::ExtCtxt;

use ast::{Aggregate, AggregateFilterExpression, Assignment, Expression, FilterExpression,
FilterValue, Identifier, Limit, MethodCall, Query, query_table};
use state::{get_field_type, get_method_types};
```

```rust
use types::Type;

macro_rules! add_filter_arguments {
    ( $name:ident, $typ:ident, $func:ident ) => {
        /// Create arguments from the `filter` and add them to `arguments`.
        fn $name(filter: $typ, args: &mut Args, table_name: &str) {
            match filter {
                $typ::Filter(filter) => {
                    $func(&filter.operand1, args, table_name, Some(filter.operand2));
                },
                $typ::Filters(filters) => {
                    $name(*filters.operand1, args, table_name);
                    $name(*filters.operand2, args, table_name);
                },
                $typ::NegFilter(box filter) => {
                    $name(filter, args, table_name);
                },
                $typ::NoFilters => (),
                $typ::ParenFilter(box filter) => {
                    $name(filter, args, table_name);
                },
                $typ::FilterValue(filter_value) => {
                    $func(&filter_value.node, args, table_name, None);
                },
            }
        }
    };
}

/// A Rust expression to be send as a parameter to the SQL query function.
#[derive(Clone, Debug)]
pub struct Arg {
    pub expression: Expression,
    pub field_name: Option<Identifier>,
    pub typ: Type,
}

/// A collection of `Arg`s.
pub type Args = Vec<Arg>;

/// Create an argument from the parameters and add it to `arguments`.
fn add(arguments: &mut Args, field_name: Option<Identifier>, typ: Type, expr: Expression)
{
    add_expr(arguments, Arg {
        expression: expr,
        field_name: field_name,
        typ: typ,
    });
```

```
}

/// Create arguments from the `assignments` and add them to `arguments`.
fn add_assignments(assignments: Vec<Assignment>, arguments: &mut Args, table_name: &str)
{
    for assign in assignments {
        // NOTE: At this stage (code generation), the field exists, hence unwrap().
        let field_type = get_field_type(table_name, &assign.identifier).unwrap();
        add(arguments, Some(assign.identifier), field_type.clone(), assign.value);
    }
}

/// Add an argument to `arguments`.
fn add_expr(arguments: &mut Args, arg: Arg) {
    // Do not add literal.
    if let ExprLit(_) = arg.expression.node {
        return;
    }
    arguments.push(arg);
}

add_filter_arguments!(add_filter_arguments, FilterExpression,
add_filter_value_arguments);

add_filter_arguments!(add_aggregate_filter_arguments, AggregateFilterExpression,
add_aggregate_filter_value_arguments);

/// Create arguments from the `limit` and add them to `arguments`.
fn add_limit_arguments(cx: &mut ExtCtxt, limit: Limit, arguments: &mut Args) {
    match limit {
        Limit::EndRange(expression) => add(arguments, None, Type::I64, expression),
        Limit::Index(expression) => add(arguments, None, Type::I64, expression),
        Limit::LimitOffset(_, _) => (), // NOTE: there are no arguments to add for a
`LimitOffset` because it is always using literals.
        Limit::NoLimit => (),
        Limit::Range(expression1, expression2) => {
            let offset = expression1.clone();
            add(arguments, None, Type::I64, expression1);
            let expr2 = expression2;
            add_expr(arguments, Arg {
                expression: quote_expr!(cx, $expr2 - $offset),
                field_name: None,
                typ: Type::I64,
            });
        },
        Limit::StartRange(expression) => add(arguments, None, Type::I64, expression),
    }
}
```

```
/// Construct an argument from the method and add it to `args`.
fn add_with_method(args: &mut Args, method_name: &str, object_name: &str, index: usize,
expr: Expression, table_name: &str) {
    // NOTE: At this stage (code generation), the method exists, hence unwrap().
    let method_types = get_method_types(table_name, object_name, method_name).unwrap();
    add_expr(args, Arg {
        expression: expr,
        field_name: None,
        typ: method_types.argument_types[index].clone(),
    });
}

fn add_aggregate_filter_value_arguments(aggregate: &Aggregate, args: &mut Args,
_table_name: &str, expression: Option<Expression>) {
    if let Some(expr) = expression {
        add(args, Some(aggregate.field.clone()), Type::I32, expr);
    }
}

fn add_filter_value_arguments(filter_value: &FilterValue, args: &mut Args, table_name:
&str, expression: Option<Expression>) {
    match *filter_value {
        FilterValue::Identifier(ref identifier) => {
            // It is possible to have an identifier without expression, when the
identifier is a
            // boolean field name, hence this condition.
            if let Some(expr) = expression {
                // NOTE: At this stage (code generation), the field exists, hence
unwrap().
                let field_type = get_field_type(table_name, identifier).unwrap();
                add(args, Some(identifier.clone()), field_type.clone(), expr);
            }
        },
        FilterValue::MethodCall(MethodCall { ref arguments, ref method_name, ref
object_name, .. }) => {
            for (index, arg) in arguments.iter().enumerate() {
                add_with_method(args, method_name, object_name, index, arg.clone(),
table_name);
            }
        },
    }
}

/// Extract the Rust `Expression`s from the `Query`.
pub fn arguments(cx: &mut ExtCtxt, query: Query) -> Args {
    let mut arguments = vec![];
    let table_name = query_table(&query);
```

```
    match query {
        Query::Aggregate { aggregate_filter, filter, .. } => {
            add_filter_arguments(filter, &mut arguments, &table_name);
            add_aggregate_filter_arguments(aggregate_filter, &mut arguments,
&table_name);
        },
        Query::CreateTable { .. } => (), // No arguments.
        Query::Delete { filter, .. } => {
            add_filter_arguments(filter, &mut arguments, &table_name);
        },
        Query::Drop { .. } => (), // No arguments.
        Query::Insert { assignments, .. } => {
            add_assignments(assignments, &mut arguments, &table_name);
        },
        Query::Select { filter, limit, ..} => {
            add_filter_arguments(filter, &mut arguments, &table_name);
            add_limit_arguments(cx, limit, &mut arguments);
        },
        Query::Update { assignments, filter, .. } => {
            add_assignments(assignments, &mut arguments, &table_name);
            add_filter_arguments(filter, &mut arguments, &table_name);
        },
    }

    arguments
}
```

Le module `ast` fournit toutes les structures et énumérations définissant l’AST.

tql_macros/src/ast.rs

```
//! Abstract syntax tree for SQL generation.

use std::fmt::{Display, Error, Formatter};

use syntax::ast::Expr;
use syntax::codemap::Spanned;
use syntax::ptr::P;

use state::tables_singleton;
use types::Type;

pub type Expression = P<Expr>;
pub type FieldList = Vec<Identifier>;
pub type Groups = Vec<Identifier>;
pub type Identifier = String;
```

```rust
/// Macro generating a struct for a filter type.
macro_rules! filter {
    ( $name:ident, $ty:ty ) => {
        #[derive(Debug)]
        pub struct $name {
            /// The filter value to be compared to `operand2`.
            pub operand1: $ty,
            /// The `operator` used to compare `operand1` to `operand2`.
            pub operator: RelationalOperator,
            /// The expression to be compared to `operand1`.
            pub operand2: Expression,
        }
    };
}

/// Macro generating an enum for a filter expression type.
macro_rules! filter_expression {
    ( $name:ident, $filter_name:ty, $filters_name:ty, $filter_expression_name:ty,
$filter_value_name:ty ) => {
        #[derive(Debug)]
        pub enum $name {
            Filter($filter_name),
            Filters($filters_name),
            NegFilter(Box<$filter_expression_name>),
            NoFilters,
            ParenFilter(Box<$filter_expression_name>),
            FilterValue(Spanned<$filter_value_name>),
        }

        impl Default for $name {
            fn default() -> $name {
                $name::NoFilters
            }
        }
    };
}

/// Macro generating a struct for a filters type.
macro_rules! filters {
    ( $name:ident, $ty:ty ) => {
        #[derive(Debug)]
        pub struct $name {
            /// The `T` to be combined with `operand2`.
            pub operand1: Box<$ty>,
            /// The `LogicalOperator` used to combine the `FilterExpression`s.
            pub operator: LogicalOperator,
            /// The `T` to be combined with `operand1`.
```

```
            pub operand2: Box<$ty>,
        }
    };
}

/// `Aggregate` for une in SQL Aggregate `Query`.
#[derive(Clone, Debug, Default)]
pub struct Aggregate {
    pub field: Identifier,
    pub function: Identifier,
    pub result_name: Identifier,
}

/// `AggregateFilter` for SQL `Query` (HAVING clause).
filter!(AggregateFilter, Aggregate);

/// Aggregate filter expression.
filter_expression!(AggregateFilterExpression, AggregateFilter, AggregateFilters,
AggregateFilterExpression, Aggregate);

/// A `Filters` is used to combine `AggregateFilterExpression`s with a `LogicalOperator`.
filters!(AggregateFilters, AggregateFilterExpression);

/// `Assignment` for use in SQL Insert and Update `Query`.
#[derive(Debug)]
pub struct Assignment {
    pub identifier: Identifier,
    pub operator: Spanned<AssignementOperator>,
    pub value: Expression,
}

/// `AssignementOperator` for use in SQL Insert and Update `Query`.
#[derive(Debug, PartialEq)]
pub enum AssignementOperator {
    Add,
    Divide,
    Equal,
    Modulo,
    Mul,
    Sub,
}

impl Display for AssignementOperator {
    fn fmt(&self, formatter: &mut Formatter) -> Result<(), Error> {
        let op =
            match *self {
                AssignementOperator::Add => "+=",
                AssignementOperator::Divide => "/=",
```

```
                AssignementOperator::Equal => "=",
                AssignementOperator::Modulo => "%=",
                AssignementOperator::Mul => "*=",
                AssignementOperator::Sub => "-=",
            };
        write!(formatter, "{}", op).unwrap();
        Ok(())
    }
}

/// `Filter` for SQL `Query` (WHERE clause).
filter!(Filter, FilterValue);

/// Either a single `Filter`, `Filters`, `NegFilter`, `NoFilters`, `ParenFilter` or a
`FilterValue`.
filter_expression!(FilterExpression, Filter, Filters, FilterExpression, FilterValue);

/// A `Filters` is used to combine `FilterExpression`s with a `LogicalOperator`.
filters!(Filters, FilterExpression);

/// Either an identifier or a method call.
#[derive(Debug)]
pub enum FilterValue {
    Identifier(Identifier),
    MethodCall(MethodCall),
}

/// A `Join` with another `joined_table` via a specific `joined_field`.
#[derive(Clone, Debug, Default)]
pub struct Join {
    pub base_field: Identifier,
    pub base_table: Identifier,
    pub joined_field: Identifier,
    pub joined_table: Identifier,
}

/// An SQL LIMIT clause.
#[derive(Clone, Debug)]
pub enum Limit {
    /// [..end]
    EndRange(Expression),
    /// [index]
    Index(Expression),
    /// Not created from a query. It is converted from a `Range`.
    LimitOffset(Expression, Expression),
    /// No limit was specified.
    NoLimit,
    /// [start..end]
```

```rust
    Range(Expression, Expression),
    /// [start..]
    StartRange(Expression),
}

impl Default for Limit {
    fn default() -> Limit {
        Limit::NoLimit
    }
}

/// `LogicalOperator` to combine `Filter`s.
#[derive(Debug, PartialEq)]
pub enum LogicalOperator {
    And,
    Not,
    Or,
}

/// A method call is an abstraction of SQL function call.
#[derive(Debug)]
pub struct MethodCall {
    pub arguments: Vec<Expression>,
    pub method_name: Identifier,
    pub object_name: Identifier,
    pub template: String,
}

/// An SQL ORDER BY clause.
#[derive(Debug)]
pub enum Order {
    /// Comes from `sort(field)`.
    Ascending(Identifier),
    /// Comes from `sort(-field)`.
    Descending(Identifier),
}

/// `RelationalOperator` to be used in a `Filter`.
#[derive(Debug)]
pub enum RelationalOperator {
    Equal,
    LesserThan,
    LesserThanEqual,
    NotEqual,
    GreaterThan,
    GreaterThanEqual,
}
```

```rust
/// An SQL `Query`.
#[derive(Debug)]
pub enum Query {
    Aggregate {
        aggregates: Vec<Aggregate>,
        aggregate_filter: AggregateFilterExpression,
        filter: FilterExpression,
        groups: Groups,
        joins: Vec<Join>,
        table: Identifier,
    },
    CreateTable {
        fields: Vec<TypedField>,
        table: Identifier,
    },
    Delete {
        filter: FilterExpression,
        table: Identifier,
    },
    Drop {
        table: Identifier,
    },
    Insert {
        assignments: Vec<Assignment>,
        table: Identifier,
    },
    Select {
        fields: FieldList,
        filter: FilterExpression,
        joins: Vec<Join>,
        limit: Limit,
        order: Vec<Order>,
        table: Identifier,
    },
    Update {
        assignments: Vec<Assignment>,
        filter: FilterExpression,
        table: Identifier,
    },
}

/// The type of the query.
pub enum QueryType {
    AggregateMulti,
    AggregateOne,
    Exec,
    InsertOne,
    SelectMulti,
```

```
    SelectOne,
}

/// An SQL field with its type.
#[derive(Debug)]
pub struct TypedField {
    pub identifier: Identifier,
    pub typ: String,
}

/// Get the query table name.
pub fn query_table(query: &Query) -> Identifier {
    let table_name =
        match *query {
            Query::Aggregate { ref table, .. } => table,
            Query::CreateTable { ref table, .. } => table,
            Query::Delete { ref table, .. } => table,
            Query::Drop { ref table, .. } => table,
            Query::Insert { ref table, .. } => table,
            Query::Select { ref table, .. } => table,
            Query::Update { ref table, .. } => table,
        };
    table_name.clone()
}

/// Get the query type.
pub fn query_type(query: &Query) -> QueryType {
    match *query {
        Query::Aggregate { ref groups, .. } => {
            if !groups.is_empty() {
                QueryType::AggregateMulti
            }
            else {
                QueryType::AggregateOne
            }
        },
        Query::Insert { .. } => QueryType::InsertOne,
        Query::Select { ref filter, ref limit, ref table, .. } => {
            let mut typ = QueryType::SelectMulti;
            if let FilterExpression::Filter(ref filter) = *filter {
                let tables = tables_singleton();
                // NOTE: At this stage (code generation), the table and the field exist,
hence unwrap().
                let table = tables.get(table).unwrap();
                if let FilterValue::Identifier(ref identifier) = filter.operand1 {
                    if table.fields.get(identifier).unwrap().node == Type::Serial {
                        typ = QueryType::SelectOne;
                    }
```

```
            }
        }
        if let Limit::Index(_) = *limit {
            typ = QueryType::SelectOne;
        }
        typ
    },
    Query::CreateTable { .. } | Query::Delete { .. } | Query::Drop { .. } |
Query::Update { .. } => QueryType::Exec,
  }
}
```

Le module `attribute` fournit une fonction qui convertit une structure Rust dans la structure `SqlFields` qui indique le type des champs d'une table.

tql_macros/src/attribute.rs

```
//! A conversion function for the #[SqlTable] attribute.

use std::collections::BTreeMap;
use std::fmt::Write;

use syntax::ast::{AngleBracketedParameters, AngleBracketedParameterData, StructField,
StructFieldKind, Ty};
use syntax::ast::Ty_::TyPath;
use syntax::codemap::Spanned;

use state::SqlFields;
use types::Type;

/// Convert a type from the Rust AST to the SQL `Type`.
#[allow(cmp_owned)]
fn field_ty_to_type(ty: &Ty) -> Spanned<Type> {
    let typ =
        if let TyPath(None, ref path) = ty.node {
            Type::from(path)
        }
        else {
            let mut type_string = String::new();
            let _ = write!(type_string, "{:?}", ty);
            Type::UnsupportedType(type_string[5..type_string.len() - 1].to_owned())
        };
    let mut position = ty.span;
    if let TyPath(_, ref path) =  ty.node {
        if path.segments[0].identifier.to_string() == "Option" {
            if let AngleBracketedParameters(AngleBracketedParameterData { ref types, ..
}) = path.segments[0].parameters {
```

```
                if let Some(typ) = types.first() {
                    position = typ.span
                }
            }
        }
    }
    Spanned {
        node: typ,
        span: position,
    }
}

/// Convert a vector of Rust struct fields to a collection of fields.
pub fn fields_vec_to_hashmap(fields: &[StructField]) -> SqlFields {
    let mut sql_fields = BTreeMap::new();
    for field in fields {
        if let StructFieldKind::NamedField(ident, _) = field.node.kind {
            if !sql_fields.contains_key(&ident.to_string()) {
                sql_fields.insert(ident.to_string(), field_ty_to_type(&*field.node.ty));
            }
            // NOTE: do not override the field type. Rust will show an error if the same
field name
            // is used twice.
        }
    }
    sql_fields
}
```

Le module `error` fournit une structure définissant un message d'erreur de même que diverses méthodes pour créer une telle structure.

tql_macros/src/error.rs

```
//! Error handling with the `Result` and `SqlError` types.
//!
//! `SqlResult<T>` is a `Result<T, Vec<SqlError>>` synonym and is used for returning and
propagating
//! multiple compile errors.

use syntax::codemap::Span;

/// `SqlError` is a type that represents an error with its position.
#[derive(Debug)]
pub struct SqlError {
    pub code: Option<String>,
    pub kind: ErrorType,
    pub message: String,
```

```rust
    pub position: Span,
}

/// `ErrorType` is an `SqlError` type.
#[derive(Debug)]
pub enum ErrorType {
    Error,
    Help,
    Note,
    Warning,
}

/// `SqlResult<T>` is a type that represents either a success (`Ok`) or failure (`Err`).
/// The failure may be represented by multiple `SqlError`s.
pub type SqlResult<T> = Result<T, Vec<SqlError>>;

impl SqlError {
    /// Returns a new `SqlError`.
    ///
    /// This is a shortcut for:
    ///
    /// ```
    /// SqlError {
    ///     code: None,
    ///     kind: ErrorType::Error,
    ///     message: message,
    ///     position: position,
    /// }
    /// ```
    pub fn new(message: &str, position: Span) -> SqlError {
        SqlError {
            code: None,
            kind: ErrorType::Error,
            message: message.to_owned(),
            position: position,
        }
    }

    /// Returns a new `SqlError` of type help.
    ///
    /// This is a shortcut for:
    ///
    /// ```
    /// SqlError {
    ///     code: None,
    ///     kind: ErrorType::Note,
    ///     message: message,
    ///     position: position,
```

```
    /// }
    pub fn new_help(message: &str, position: Span) -> SqlError {
        SqlError {
            code: None,
            kind: ErrorType::Help,
            message: message.to_owned(),
            position: position,
        }
    }

    /// Returns a new `SqlError` of type note.
    ///
    /// This is a shortcut for:
    ///
    /// ```
    /// SqlError {
    ///     code: None,
    ///     kind: ErrorType::Note,
    ///     message: message,
    ///     position: position,
    /// }
    pub fn new_note(message: &str, position: Span) -> SqlError {
        SqlError {
            code: None,
            kind: ErrorType::Note,
            message: message.to_owned(),
            position: position,
        }
    }

    /// Returns a new `SqlError` of type warning.
    ///
    /// This is a shortcut for:
    ///
    /// ```
    /// SqlError {
    ///     code: None,
    ///     kind: ErrorType::Warning,
    ///     message: message,
    ///     position: position,
    /// }
    pub fn new_warning(message: &str, position: Span) -> SqlError {
        SqlError {
            code: None,
            kind: ErrorType::Warning,
            message: message.to_owned(),
            position: position,
        }
```

```
    }

    /// Returns a new `SqlError` with a code.
    ///
    /// This is a shortcut for:
    ///
    /// ```
    /// SqlError {
    ///     code: Some(code.to_owned()),
    ///     kind: ErrorType::Error,
    ///     message: message,
    ///     position: position,
    /// }
    /// ```
    pub fn new_with_code(message: &str, position: Span, code: &str) -> SqlError {
        SqlError {
            code: Some(code.to_owned()),
            kind: ErrorType::Error,
            message: message.to_owned(),
            position: position,
        }
    }
}

/// Returns an `SqlResult<T>` from potential result and errors.
/// Returns `Err` if there are at least one error.
/// Otherwise, returns `Ok`.
pub fn res<T>(result: T, errors: Vec<SqlError>) -> SqlResult<T> {
    if !errors.is_empty() {
        Err(errors)
    }
    else {
        Ok(result)
    }
}
```

Le module `gen` transforme l’AST en SQL.

tql_macros/src/gen.rs

```
//! The PostgreSQL code generator.

use std::str::from_utf8;

use syntax::ast::Expr_::ExprLit;
use syntax::ast::Lit_::{LitBool, LitByte, LitByteStr, LitChar, LitFloat,
LitFloatUnsuffixed, LitInt, LitStr};
```

```rust
use ast::{Aggregate, AggregateFilter, AggregateFilterExpression, AggregateFilters,
Assignment, AssignementOperator, Expression, FieldList, Filter, Filters,
FilterExpression, FilterValue, Identifier, Join, Limit, LogicalOperator, MethodCall,
Order, RelationalOperator, Query, TypedField};
use ast::Limit::{EndRange, Index, LimitOffset, NoLimit, Range, StartRange};
use sql::escape;
use state::get_primary_key_field_by_table_name;

/// Macro used to generate a ToSql implementation for a filter (for use in WHERE or
HAVING).
macro_rules! filter_to_sql {
    ( $name:ident ) => {
        impl ToSql for $name {
            fn to_sql(&self) -> String {
                self.operand1.to_sql() + " " +
                    &self.operator.to_sql() + " " +
                    &self.operand2.to_sql()
            }
        }
    };
}

/// Macro used to generate a ToSql implementation for a filter expression.
macro_rules! filter_expression_to_sql {
    ( $name:ident ) => {
        impl ToSql for $name {
            fn to_sql(&self) -> String {
                match *self {
                    $name::Filter(ref filter) => filter.to_sql(),
                    $name::Filters(ref filters) => filters.to_sql(),
                    $name::NegFilter(ref filter) =>
                        "NOT ".to_owned() +
                        &filter.to_sql(),
                    $name::NoFilters => "".to_owned(),
                    $name::ParenFilter(ref filter) =>
                        "(".to_owned() +
                        &filter.to_sql() +
                        ")",
                    $name::FilterValue(ref filter_value) => filter_value.node.to_sql(),
                }
            }
        }
    };
}

/// Macro used to generate a ToSql implementation for a slice.
macro_rules! slice_to_sql {
```

```
    ( $name:ty, $sep:expr ) => {
        impl ToSql for [$name] {
            fn to_sql(&self) -> String {
                self.iter().map(ToSql::to_sql).collect::<Vec<_>>().join($sep)
            }
        }
    };
}

/// A generic trait for converting a value to SQL.
pub trait ToSql {
    fn to_sql(&self) -> String;
}

impl ToSql for Aggregate {
    fn to_sql(&self) -> String {
        "CAST(".to_owned() + &self.function.to_sql() + "(" + &self.field.to_sql() + ") AS
INT)"
    }
}

slice_to_sql!(Aggregate, ", ");

filter_to_sql!(AggregateFilter);

filter_expression_to_sql!(AggregateFilterExpression);

filter_to_sql!(AggregateFilters);

impl ToSql for Assignment {
    fn to_sql(&self) -> String {
        if let AssignementOperator::Equal = self.operator.node {
            self.identifier.to_sql() +
                &self.operator.node.to_sql() +
                &self.value.to_sql()
        }
        else {
            let identifier = self.identifier.to_sql();
            identifier.clone() +
                &self.operator.node.to_sql().replace("{}", &identifier) +
                &self.value.to_sql()
        }
    }
}

slice_to_sql!(Assignment, ", ");

impl ToSql for AssignementOperator {
```

```
    fn to_sql(&self) -> String {
        match *self {
            AssignementOperator::Add => " = {} + ",
            AssignementOperator::Divide => " = {} / ",
            AssignementOperator::Equal => " = ",
            AssignementOperator::Modulo => " = {} % ",
            AssignementOperator::Mul => " = {} * ",
            AssignementOperator::Sub => " = {} - ",
        }.to_owned()
    }
}

/// Convert a literal expression to its SQL representation.
/// A non-literal is converted to ? for use with query parameters.
impl ToSql for Expression {
    fn to_sql(&self) -> String {
        match self.node {
            ExprLit(ref literal) => {
                match literal.node {
                    LitBool(boolean) => boolean.to_string().to_uppercase(),
                    LitByte(byte) =>
                        "'".to_owned() +
                        &escape((byte as char).to_string()) +
                        "'",
                    LitByteStr(ref bytestring) =>
                        "'".to_owned() +
                        &escape(from_utf8(&bytestring[..]).unwrap().to_owned()) +
                        "'",
                    LitChar(character) =>
                        "'".to_owned() +
                        &escape(character.to_string()) +
                        "'",
                    LitFloat(ref float, _) => float.to_string(),
                    LitFloatUnsuffixed(ref float) => float.to_string(),
                    LitInt(number, _) => number.to_string(),
                    LitStr(ref string, _) =>
                        "'".to_owned() +
                        &escape(string.to_string()) +
                        "'",
                }
            },
            _ => "?".to_owned(),
        }
    }
}

slice_to_sql!(Expression, ", ");
```

```
impl ToSql for FieldList {
    fn to_sql(&self) -> String {
        self.join(", ")
    }
}

filter_to_sql!(Filter);

filter_expression_to_sql!(FilterExpression);

filter_to_sql!(Filters);

impl ToSql for FilterValue {
    fn to_sql(&self) -> String {
        match *self {
            FilterValue::Identifier(ref identifier) => identifier.to_sql(),
            FilterValue::MethodCall(MethodCall { ref arguments, ref object_name, ref
template, ..  }) => {
                // In the template, $0 represents the object identifier and $1, $2, ...
the
                // arguments.
                let mut sql = template.replace("$0", object_name);
                let mut index = 1;
                for argument in arguments {
                    sql = sql.replace(&format!("${}", index), &argument.to_sql());
                    index += 1;
                }
                sql
            },
        }
    }
}

impl ToSql for Join {
    fn to_sql(&self) -> String {
        " INNER JOIN ".to_owned() + &self.joined_table +
            " ON " + &self.base_table + "." + &self.base_field + " = "
            + &self.joined_table + "." + &self.joined_field
    }
}

slice_to_sql!(Join, " ");

impl ToSql for Identifier {
    fn to_sql(&self) -> String {
        self.clone()
    }
}
```

```
impl ToSql for Limit {
    fn to_sql(&self) -> String {
        match *self {
            EndRange(ref expression) => " LIMIT ".to_owned() + &expression.to_sql(),
            Index(ref expression) =>
                " OFFSET ".to_owned() + &expression.to_sql() +
                " LIMIT 1",
            LimitOffset(ref expression1, ref expression2) =>
                " OFFSET ".to_owned() + &expression2.to_sql() +
                " LIMIT " + &expression1.to_sql(),
            NoLimit => "".to_owned(),
            Range(ref expression1, ref expression2) =>
                " OFFSET ".to_owned() + &expression1.to_sql() +
                " LIMIT " + &expression2.to_sql(),
            StartRange(ref expression) => " OFFSET ".to_owned() + &expression.to_sql(),
        }
    }
}

impl ToSql for LogicalOperator {
    fn to_sql(&self) -> String {
        match *self {
            LogicalOperator::And => "AND",
            LogicalOperator::Not => "NOT",
            LogicalOperator::Or => "OR",
        }.to_owned()
    }
}

impl ToSql for Order {
    fn to_sql(&self) -> String {
        match *self {
            Order::Ascending(ref field) => field.to_sql(),
            Order::Descending(ref field) => field.to_sql() + " DESC",
        }
    }
}

slice_to_sql!(Order, ", ");

/// Convert a whole `Query` to SQL.
impl ToSql for Query {
    fn to_sql(&self) -> String {
        match *self {
            Query::Aggregate{ref aggregates, ref aggregate_filter, ref filter, ref
groups, ref joins, ref table} => {
                let where_clause = filter_to_where_clause(filter);
```

```
                let group_clause =
                    if !groups.is_empty() {
                        " GROUP BY "
                    }
                    else {
                        ""
                    };
                let having_clause =
                    if let AggregateFilterExpression::NoFilters = *aggregate_filter {
                        ""
                    }
                    else {
                        " HAVING "
                    };
                replace_placeholder(format!(
                    "SELECT {aggregates} FROM
{table_name}{joins}{where_clause}{filter}{group_clause}{groups}{having_clause}{aggregate_
filter}",
                    aggregates = aggregates.to_sql(),
                    table_name = table,
                    joins = joins.to_sql(),
                    where_clause = where_clause,
                    filter = filter.to_sql(),
                    group_clause = group_clause,
                    groups = groups.to_sql(),
                    having_clause = having_clause,
                    aggregate_filter = aggregate_filter.to_sql()
                ))
            },
            Query::CreateTable { ref fields, ref table } => {
                format!("CREATE TABLE {table} ({fields})",
                    table = table,
                    fields = fields.to_sql()
                )
            },
            Query::Delete { ref filter, ref table } => {
                let where_clause = filter_to_where_clause(filter);
                replace_placeholder(format!("DELETE FROM {table}{where_clause}{filter}",
                                            table = table,
                                            where_clause = where_clause,
                                            filter = filter.to_sql()
                                           )
                                   )
            },
            Query::Drop { ref table } => {
                format!("DROP TABLE {table}", table = table)
            },
            Query::Insert { ref assignments, ref table } => {
```

```
                let fields: Vec<_> = assignments.iter().map(|assign|
assign.identifier.to_sql()).collect();
                let values: Vec<_> = assignments.iter().map(|assign|
assign.value.to_sql()).collect();
                // Add the SQL code to get the inserted primary key.
                let return_value = get_primary_key_field_by_table_name(table)
                    .map(|primary_key| " RETURNING ".to_owned() + &primary_key)
                    .unwrap_or("".to_owned());
                replace_placeholder(format!(
                    "INSERT INTO {table}({fields}) VALUES({values}){return_value}",
                    table = table,
                    fields = fields.to_sql(),
                    values = values.to_sql(),
                    return_value = return_value
                ))
            },
            Query::Select{ref fields, ref filter, ref joins, ref limit, ref order, ref
table} => {
                let where_clause = filter_to_where_clause(filter);
                let order_clause =
                    if !order.is_empty() {
                        " ORDER BY "
                    }
                    else {
                        ""
                    };
                replace_placeholder(format!(
                    "SELECT {fields} FROM
{table}{joins}{where_clause}{filter}{order_clause}{order}{limit}",
                    fields = fields.to_sql(),
                    table = table,
                    joins = joins.to_sql(),
                    where_clause = where_clause,
                    filter = filter.to_sql(),
                    order_clause = order_clause,
                    order = order.to_sql(),
                    limit = limit.to_sql()
                ))
            },
            Query::Update { ref assignments, ref filter, ref table } => {
                let where_clause = filter_to_where_clause(filter);
                replace_placeholder(format!(
                    "UPDATE {table} SET {assignments}{where_clause}{filter}",
                    table = table,
                    assignments = assignments.to_sql(),
                    where_clause = where_clause,
                    filter = filter.to_sql()
                ))
```

```
            },
        }
    }
}

impl ToSql for RelationalOperator {
    fn to_sql(&self) -> String {
        match *self {
            RelationalOperator::Equal => "=",
            RelationalOperator::LesserThan => "<",
            RelationalOperator::LesserThanEqual => "<=",
            RelationalOperator::NotEqual => "<>",
            RelationalOperator::GreaterThan => ">=",
            RelationalOperator::GreaterThanEqual => ">",
        }.to_owned()
    }
}

impl ToSql for TypedField {
    fn to_sql(&self) -> String {
        self.identifier.to_sql() + " " + &self.typ
    }
}

slice_to_sql!(TypedField, ", ");

/// Convert a `FilterExpression` to either " WHERE " or the empty string if there are no
filters.
fn filter_to_where_clause(filter: &FilterExpression) -> &str {
    match *filter {
        FilterExpression::Filter(_) | FilterExpression::Filters(_) |
FilterExpression::NegFilter(_) | FilterExpression::ParenFilter(_) |
FilterExpression::FilterValue(_) => " WHERE ",
        FilterExpression::NoFilters => "",
    }
}

/// Replace the placeholders `{}` by $# by # where # is the index of the placeholder.
fn replace_placeholder(string: String) -> String {
    let mut result = "".to_owned();
    let mut in_string = false;
    let mut skip_next = false;
    let mut index = 1;
    for character in string.chars() {
        if character == '?' && !in_string {
            result.push('$');
            result.push_str(&index.to_string());
            index = index + 1;
```

```
        }
        else {
            if character == '\\' {
                skip_next = true;
            }
            else if character == '\'' && !skip_next {
                skip_next = false;
                in_string = !in_string;
            }
            else {
                skip_next = false;
            }
            result.push(character);
        }
    }
    result
}
```

Le module `hashmap` fournit une macro pour créer des `HashMap` de façon plus concise.

tql_macros/src/hashmap.rs

```
/// An hashmap macro.
///
/// # Examples
///
/// ```
/// let hashmap = hashmap! {
///     "one" => 1,
///     "two" => 2,
/// };
/// ```
macro_rules! hashmap {
    { $($k:expr => $v:expr),* $(,)* } => {{
        let mut hashmap = ::std::collections::HashMap::new();
        $(hashmap.insert($k, $v);)*
        hashmap
    }};
}
```

Le module `lib` crée les macros `sql!()` et `to_sql!()` de même que l'attribut `#[SqlTable]`. En outre, il ajoute une implémentation du trait `postgres::types::ToSql` à chaque structure de table, car cela est requis pour envoyer un argument d'un nouveau type à la méthode `query()` (utile pour les clés étrangères). De plus, ce module affiche les messages d'erreur. Enfin, ce code génère le code Rust qui remplace la macro `sql!()` en fonction du type de requête.

tql_macros/src/lib.rs

```
//! The TQL library provide macros and attribute useful to generate SQL.
//!
//! The SQL is generated at compile time via a procedural macro.

#![feature(box_patterns, box_syntax, convert, plugin, plugin_registrar, quote,
rustc_private)]
#![plugin(clippy)]
#![allow(ptr_arg)]

#[macro_use]
extern crate rustc;
extern crate syntax;

use std::error::Error;

use rustc::lint::{EarlyLintPassObject, LateLintPassObject};
use rustc::plugin::Registry;
use syntax::ast::{AngleBracketedParameters, AngleBracketedParameterData, Block, Field,
Ident, MetaItem, Path, PathSegment, StructField_, StructFieldKind, TokenTree, Ty, Ty_,
VariantData, Visibility};
use syntax::ast::Expr_::ExprLit;
use syntax::ast::Item_::ItemStruct;
use syntax::ast::MetaItem_::MetaWord;
use syntax::codemap::{BytePos, Span, Spanned};
use syntax::ext::base::{Annotatable, DummyResult, ExtCtxt, MacEager, MacResult};
use syntax::ext::base::Annotatable::Item;
use syntax::ext::base::SyntaxExtension::MultiDecorator;
use syntax::ext::build::AstBuilder;
use syntax::ext::deriving::debug::expand_deriving_debug;
use syntax::owned_slice::OwnedSlice;
use syntax::parse::token::{InternedString, Token, intern, str_to_ident};
use syntax::ptr::P;

#[macro_use]
pub mod hashmap;
pub mod analyzer;
pub mod arguments;
pub mod ast;
pub mod attribute;
```

```
pub mod error;
pub mod gen;
pub mod methods;
pub mod optimizer;
pub mod parser;
pub mod plugin;
pub mod sql;
pub mod state;
pub mod string;
pub mod type_analyzer;
pub mod types;

pub type SqlQueryWithArgs = (String, QueryType, Args, Vec<Join>, Vec<Aggregate>);

use analyzer::{analyze, analyze_types, has_joins};
use arguments::{Args, arguments};
use ast::{Aggregate, Expression, Join, Query, QueryType, query_type};
use attribute::fields_vec_to_hashmap;
use error::{ErrorType, SqlError, SqlResult};
use gen::ToSql;
use optimizer::optimize;
use parser::parse;
use plugin::NODE_ID;
use state::{SqlArg, SqlArgs, SqlFields, SqlTable, SqlTables,
get_primary_key_field_by_table_name, lint_singleton, tables_singleton};
use type_analyzer::{SqlAttrError, SqlErrorLint};
use types::Type;

/// Add the #[derive(Debug)] attribute to the `annotatable` item if needed.
/// It won't be added if it is already present.
#[allow(cmp_owned)]
fn add_derive_debug(cx: &mut ExtCtxt, sp: Span, meta_item: &MetaItem, annotatable:
&Annotatable, push: &mut FnMut(Annotatable)) {
    let attrs = annotatable.attrs();
    if let &Item(_) = annotatable {
        let has_derive_debug_attribute = attrs.iter().all(|item| {
            if let MetaWord(ref word) = item.node.value.node {
                return word.to_string() != "derive_Debug"
            }
            true
        });
        if has_derive_debug_attribute {
            // Add the #[derive(Debug)] attribute.
            expand_deriving_debug(cx, sp, meta_item, annotatable, push);
        }
    }
}
```

```
/// Add a `Field` to `fields` made with the `expr`, identified by `name` at `position`.
/// This is used to generate a struct expression.
fn add_field(fields: &mut Vec<Field>, expr: Expression, name: &str, position: Span) {
    fields.push(Field {
        expr: expr,
        ident: Spanned {
            node: str_to_ident(name),
            span: position,
        },
        span: position,
    });
}

/// Add the postgres::types::ToSql implementation on the struct.
/// Its SQL representation is the same as the primary key SQL representation.
fn add_tosql_impl(cx: &mut ExtCtxt, push: &mut FnMut(Annotatable), table_name: &str) {
    match get_primary_key_field_by_table_name(table_name) {
        Some(primary_key_field) => {
            let table_ident = str_to_ident(table_name);
            let primary_key_ident = str_to_ident(&primary_key_field);
            let implementation = quote_item!(cx,
                impl postgres::types::ToSql for $table_ident {
                    fn to_sql<W: std::io::Write + ?Sized>(
                        &self,
                        ty: &postgres::types::Type,
                        out: &mut W,
                        ctx: &postgres::types::SessionInfo
                    ) -> postgres::Result<postgres::types::IsNull>
                    {
                        self.$primary_key_ident.to_sql(ty, out, ctx)
                    }

                    fn accepts(ty: &postgres::types::Type) -> bool {
                        match *ty {
                            postgres::types::Type::Int4 => true,
                            _ => false,
                        }
                    }

                    fn to_sql_checked(
                        &self,
                        ty: &postgres::types::Type,
                        out: &mut ::std::io::Write,
                        ctx: &postgres::types::SessionInfo
                    ) -> postgres::Result<postgres::types::IsNull>
                    {
                        if !<Self as postgres::types::ToSql>::accepts(ty) {
                            return Err(postgres::error::Error::WrongType(ty.clone()));
```

```
                        }
                        self.to_sql(ty, out, ctx)
                    }
                }
            );
            push(Annotatable::Item(implementation.unwrap()));
        },
        None => (), // NOTE: Do not add the implementation when there is no primary key.
    }
}

/// Create an aggregate field definition to be added to a struct definition.
fn create_aggregate_field_def(field_name: &str, sp: Span) -> Spanned<StructField_> {
    Spanned {
        node: StructField_ {
            kind: StructFieldKind::NamedField(str_to_ident(field_name),
Visibility::Inherited),
            id: NODE_ID,
            ty: P(Ty {
                id: NODE_ID,
                node: Ty_::TyPath(None, Path {
                    span: sp,
                    global: false,
                    segments: vec![PathSegment {
                        identifier: str_to_ident("i32"),
                        parameters: AngleBracketedParameters(AngleBracketedParameterData
{
                            bindings: OwnedSlice::empty(),
                            lifetimes: vec![],
                            types: OwnedSlice::empty(),
                        }),
                    }],
                }),
                span: sp,
            }),
            attrs: vec![],
        },
        span: sp,
    }
}

/// Expand the `sql!()` macro.
/// This macro converts the Rust code provided as argument to SQL and outputs Rust code
using the
/// `postgres` library.
fn expand_sql(cx: &mut ExtCtxt, sp: Span, args: &[TokenTree]) -> Box<MacResult + 'static>
{
    let sql_result = to_sql(cx, args);
```

```
    match sql_result {
        Ok(sql_query_with_args) => {
            if let TokenTree::Token(_, Token::Ident(ident, _)) = args[0] {
                gen_query(cx, sp, ident, sql_query_with_args)
            }
            else {
                cx.span_err(sp, "Expected table identifier");
                DummyResult::any(sp)
            }
        },
        Err(errors) => {
            span_errors(errors, cx);
            DummyResult::any(sp)
        },
    }
}

/// Expand the `#[SqlTable]` attribute.
/// This attribute must be used on structs to tell tql that it represents an SQL table.
fn expand_sql_table(cx: &mut ExtCtxt, sp: Span, meta_item: &MetaItem, annotatable:
&Annotatable, push: &mut FnMut(Annotatable)) {
    // Add to sql_tables.
    let mut sql_tables = tables_singleton();

    add_derive_debug(cx, sp, meta_item, annotatable, push);

    if let &Annotatable::Item(ref item) = annotatable {
        if let ItemStruct(ref struct_def, _) = item.node {
            let table_name = item.ident.to_string();
            if !sql_tables.contains_key(&table_name) {
                let fields = get_struct_fields(cx, struct_def);

                sql_tables.insert(table_name.clone(), SqlTable {
                    fields: fields,
                    name: table_name.clone(),
                    position: item.span,
                });

                add_tosql_impl(cx, push, &table_name);
            }
            else {
                // NOTE: This error is needed because the code could have two table
structs in
                // different modules.
                cx.parse_sess.span_diagnostic.span_err_with_code(item.span,
&format!("duplicate definition of table `{}`", table_name), "E0428");
            }
        }
```

```
        else {
            cx.span_err(item.span, "Expected struct but found");
        }
    }
    else {
        cx.span_err(sp, "Expected struct item");
    }
}

/// Expand the `to_sql!()` macro.
/// This macro converts the Rust code provided as argument to SQL and ouputs it as a
string
/// expression.
fn expand_to_sql(cx: &mut ExtCtxt, sp: Span, args: &[TokenTree]) -> Box<MacResult +
'static> {
    let sql_result = to_sql(cx, args);
    match sql_result {
        Ok((sql, _, _, _, _)) => {
            let string_literal = intern(&sql);
            MacEager::expr(cx.expr_str(sp,
InternedString::new_from_name(string_literal)))
        },
        Err(errors) => {
            span_errors(errors, cx);
            DummyResult::any(sp)
        },
    }
}

/// Generate the aggregate struct and struct expression.
fn gen_aggregate_struct(cx: &mut ExtCtxt, sp: Span, aggregates: &[Aggregate]) -> P<Block>
{
    let mut aggregate_fields = vec![];
    let mut fields = vec![];
    for (index, aggregate) in aggregates.iter().enumerate() {
        let field_name = aggregate.result_name.clone();
        add_field(&mut aggregate_fields, quote_expr!(cx, row.get($index)), &field_name,
sp);
        fields.push(create_aggregate_field_def(&field_name, sp));
    }
    let struct_ident = str_to_ident("Aggregate");
    let aggregate_struct = cx.item_struct(sp, struct_ident, VariantData::Struct(fields,
NODE_ID));
    let aggregate_stmt = cx.stmt_item(sp, aggregate_struct);
    let instance = cx.expr_struct(sp, cx.path_ident(sp, struct_ident), aggregate_fields);
    cx.block(sp, vec![aggregate_stmt], Some(instance))
}
```

```
/// Generate the Rust code from the SQL query.
fn gen_query(cx: &mut ExtCtxt, sp: Span, table_ident: Ident, sql_query_with_args:
SqlQueryWithArgs) -> Box<MacResult + 'static> {
    let (sql, query_type, arguments, joins, aggregates) = sql_query_with_args;
    let string_literal = intern(&sql);
    let sql_query = cx.expr_str(sp, InternedString::new_from_name(string_literal));
    let ident = Ident::new(intern("connection"), table_ident.ctxt);
    let sql_tables = tables_singleton();
    let table_name = table_ident.to_string();
    match sql_tables.get(&table_name) {
        Some(table) => {
            let fields = get_query_fields(cx, sp, &table.fields, sql_tables, joins);
            let struct_expr = cx.expr_struct(sp, cx.path_ident(sp, table_ident), fields);
            let aggregate_struct = gen_aggregate_struct(cx, sp, &aggregates);
            let args_expr = get_query_arguments(cx, sp, table_name, arguments);
            let expr = gen_query_expr(cx, ident, sql_query, args_expr, struct_expr,
aggregate_struct, query_type);
            MacEager::expr(expr)
        },
        None => DummyResult::any(sp),
    }
}

/// Generate the Rust code using the `postgres` library depending on the `QueryType`.
fn gen_query_expr(cx: &mut ExtCtxt, ident: Ident, sql_query: Expression, args_expr:
Expression, struct_expr: Expression, aggregate_struct: P<Block>, query_type: QueryType)
-> Expression {
    match query_type {
        QueryType::AggregateMulti => {
            quote_expr!(cx, {
                let result = $ident.prepare($sql_query).unwrap();
                result.query(&$args_expr).unwrap().iter().map(|row| {
                    $aggregate_struct
                }).collect::<Vec<_>>()
            })
        },
        QueryType::AggregateOne => {
            quote_expr!(cx, {
                let result = $ident.prepare($sql_query).unwrap();
                result.query(&$args_expr).unwrap().iter().next().map(|row| {
                    $aggregate_struct
                })
            })
        },
        QueryType::InsertOne => {
            quote_expr!(cx, {
                $ident.prepare($sql_query)
                    .and_then(|result| {
```

```
                        // NOTE: The query is not supposed to fail, hence unwrap().
                        let rows = result.query(&$args_expr).unwrap();
                        // NOTE: There is always one result (the inserted id), hence
unwrap().
                        let row = rows.iter().next().unwrap();
                        let count: i32 = row.get(0);
                        Ok(count)
                    })
            })
        },
        QueryType::SelectMulti => {
            quote_expr!(cx, {
                let result = $ident.prepare($sql_query).unwrap();
                result.query(&$args_expr).unwrap().iter().map(|row| {
                    $struct_expr
                }).collect::<Vec<_>>()
            })
        },
        QueryType::SelectOne => {
            quote_expr!(cx, {
                let result = $ident.prepare($sql_query).unwrap();
                result.query(&$args_expr).unwrap().iter().next().map(|row| {
                    $struct_expr
                })
            })
        },
        QueryType::Exec => {
            quote_expr!(cx, {
                $ident.prepare($sql_query)
                    .and_then(|result| result.execute(&$args_expr))
            })
        },
    }
}

/// Get the arguments to send to the `postgres::stmt::Statement::query` or
/// `postgres::stmt::Statement::execute` method.
fn get_query_arguments(cx: &mut ExtCtxt, sp: Span, table_name: String, arguments: Args)
-> Expression {
    let mut arg_refs = vec![];
    let mut sql_args = vec![];
    let calls = lint_singleton();

    for arg in arguments {
        let pos = arg.expression.span;

        let (low, high) =
            match (pos.lo, pos.hi) {
```

```
                (BytePos(low), BytePos(high)) => (low, high),
            };
        sql_args.push(SqlArg {
            high: high,
            low: low,
            typ: arg.typ,
        });

        match arg.expression.node {
            // Do not add literal arguments as they are in the final string literal.
            ExprLit(_) => (),
            _ => {
                arg_refs.push(cx.expr_addr_of(sp, arg.expression));
            },
        }
    }

    // Add the arguments to the lint state so that they can be type-checked by the lint.
    let BytePos(low) = sp.lo;
    calls.insert(low, SqlArgs {
        arguments: sql_args,
        table_name: table_name.to_owned(),
    });

    cx.expr_vec(sp, arg_refs)
}

/// Get the fully qualified field names for the struct expression needed by the generated
code.
fn get_query_fields(cx: &mut ExtCtxt, sp: Span, table: &SqlFields, sql_tables:
&SqlTables, joins: Vec<Join>) -> Vec<Field> {
    let mut fields = vec![];
    let mut index = 0usize;
    for (name, typ) in table {
        match typ.node {
            Type::Custom(ref foreign_table) => {
                let table_name = foreign_table;
                if let Some(foreign_table) = sql_tables.get(foreign_table) {
                    if has_joins(&joins, name) {
                        // If there is a join, fetch the joined fields.
                        let mut foreign_fields = vec![];
                        for (field, typ) in &foreign_table.fields {
                            match typ.node {
                                Type::Custom(_) | Type::UnsupportedType(_) => (), // Do
not add foreign key recursively.
                                _ => {
                                    add_field(&mut foreign_fields, quote_expr!(cx,
row.get($index)), &field, sp);
```

```
                                    index += 1;
                                },
                            }
                        }
                        let related_struct = cx.expr_struct(sp, cx.path_ident(sp,
str_to_ident(table_name)), foreign_fields);
                        add_field(&mut fields, quote_expr!(cx, Some($related_struct)),
name, sp);
                    }
                    else {
                        // Since a `ForeignKey` is an `Option`, we output `None` when the
field
                        // is not `join`ed.
                        add_field(&mut fields, quote_expr!(cx, None), name, sp);
                    }
                }
                // NOTE: if the field type is not an SQL table, an error is thrown by the
linter.
            },
            Type::UnsupportedType(_) => (),
            _ => {
                add_field(&mut fields, quote_expr!(cx, row.get($index)), name, sp);
                index += 1;
            },
        }
    }
    fields
}

/// Get the fields from the struct.
/// Also check if the field types from the struct are supported types.
fn get_struct_fields(cx: &mut ExtCtxt, struct_def: &VariantData) -> SqlFields {
    let fields = fields_vec_to_hashmap(struct_def.fields());
    for field in fields.values() {
        match field.node {
            Type::UnsupportedType(ref typ) | Type::Nullable(box Type::UnsupportedType(ref
typ)) =>
                cx.parse_sess.span_diagnostic.span_err_with_code(
                    field.span,
                    &format!("use of unsupported type name `{}`", typ), "E0412"
                ),
            _ => (), // NOTE: Other types are supported.
        }
    }
    fields
}

/// Show the compilation errors.
```

```
fn span_errors(errors: Vec<SqlError>, cx: &mut ExtCtxt) {
    for &SqlError {ref code, ref message, position, ref kind} in &errors {
        match *kind {
            ErrorType::Error => {
                match *code {
                    Some(ref code) =>
cx.parse_sess.span_diagnostic.span_err_with_code(position, &message, code),
                    None => cx.span_err(position, &message),
                }
            },
            ErrorType::Help => {
                cx.parse_sess.span_diagnostic.fileline_help(position, &message);
            },
            ErrorType::Note => {
                cx.parse_sess.span_diagnostic.fileline_note(position, &message);
            },
            ErrorType::Warning => {
                cx.span_warn(position, &message);
            },
        }
    }
}

/// Convert the Rust code to an SQL string with its type, arguments, joins, and aggregate
fields.
fn to_sql(cx: &mut ExtCtxt, args: &[TokenTree]) -> SqlResult<SqlQueryWithArgs> {
    if args.is_empty() {
        return Err(vec![SqlError::new_with_code("this macro takes 1 parameter but 0
parameters were supplied", cx.call_site(), "E0061")]);
    }

    let mut parser = cx.new_parser_from_tts(args);
    let expression = try!(parser.parse_expr()
        .map_err(|error| vec![SqlError::new(error.description(), cx.call_site())]));
    let sql_tables = tables_singleton();
    let method_calls = try!(parse(expression));
    let mut query = try!(analyze(method_calls, sql_tables));
    optimize(&mut query);
    query = try!(analyze_types(query));
    let sql = query.to_sql();
    let joins =
        match query {
            Query::Select { ref joins, .. } => joins.clone(),
            _ => vec![],
        };
    let aggrs: Vec<Aggregate> =
        match query {
            Query::Aggregate { ref aggregates, .. } => aggregates.clone(),
```

```
            _ => vec![],
        };
    Ok((sql, query_type(&query), arguments(cx, query), joins, aggrs))
}

#[plugin_registrar]
pub fn plugin_registrar(reg: &mut Registry) {
    reg.register_macro("to_sql", expand_to_sql);
    reg.register_macro("sql", expand_sql);
    reg.register_syntax_extension(intern("SqlTable"), MultiDecorator(box
expand_sql_table));
    reg.register_early_lint_pass(box SqlAttrError as EarlyLintPassObject);
    reg.register_late_lint_pass(box SqlErrorLint as LateLintPassObject);
}
```

Le module `methods` définit les méthodes de filtre et les fonctions d'agrégat.

tql_macros/src/methods.rs

```
//! Methods definition for use in filters.

use std::collections::HashMap;

use state::{SqlMethodTypes, aggregates_singleton, methods_singleton};
use types::Type;

/// Add a new `method` on `object_type` of type `argument_types` -> `return_type`.
/// The template is the resulting SQL with `$0` as a placeholder for `self` and `$1`,
`$2`, … as
/// placeholders for the arguments.
pub fn add_method(object_type: &Type, return_type: Type, argument_types: Vec<Type>,
method: &str, template: &str) {
    let methods = methods_singleton();
    let type_methods = methods.entry(object_type.clone()).or_insert(HashMap::new());
    type_methods.insert(method.to_owned(), SqlMethodTypes {
        argument_types: argument_types,
        return_type: return_type,
        template: template.to_owned(),
    });
}

/// Add a new aggregate `rust_function` mapping to `sql_function`.
pub fn add_aggregate(rust_function: &str, sql_function: &str) {
    let aggregates = aggregates_singleton();
    aggregates.insert(rust_function.to_owned(), sql_function.to_owned());
}
```

```
/// Add the default SQL aggregate functions.
pub fn add_initial_aggregates() {
    add_aggregate("avg", "AVG");
}

/// Add the default SQL methods.
pub fn add_initial_methods() {
    // Date methods.
    let date_types = [Type::LocalDateTime, Type::NaiveDate, Type::NaiveDateTime,
Type::UTCDateTime];
    for date_type in &date_types {
        add_method(date_type, Type::I32, vec![], "year", "EXTRACT(YEAR FROM $0)");
        add_method(date_type, Type::I32, vec![], "month", "EXTRACT(MONTH FROM $0)");
        add_method(date_type, Type::I32, vec![], "day", "EXTRACT(DAY FROM $0)");
    }

    // Time methods.
    let time_types = [Type::LocalDateTime, Type::NaiveDateTime, Type::NaiveTime,
Type::UTCDateTime];
    for time_type in &time_types {
        add_method(time_type, Type::I32, vec![], "hour", "EXTRACT(HOUR FROM $0)");
        add_method(time_type, Type::I32, vec![], "minute", "EXTRACT(MINUTE FROM $0)");
        add_method(time_type, Type::I32, vec![], "second", "EXTRACT(SECOND FROM $0)");
    }

    // String methods.
    add_method(&Type::String, Type::Bool, vec![Type::String], "contains", "$0 LIKE '%' ||
$1 || '%'");
    add_method(&Type::String, Type::Bool, vec![Type::String], "ends_with", "$0 LIKE '%'
|| $1");
    add_method(&Type::String, Type::Bool, vec![Type::String], "starts_with", "$0 LIKE $1
|| '%'");
    add_method(&Type::String, Type::I32, vec![], "len", "CHAR_LENGTH($0)");
    add_method(&Type::String, Type::Bool, vec![Type::String], "match", "$0 LIKE $1");
    add_method(&Type::String, Type::Bool, vec![Type::String], "imatch", "$0 ILIKE $1");

    // Option methods.
    add_method(&Type::Nullable(box Type::Generic), Type::Bool, vec![], "is_some", "$0 IS
NOT NULL");
    add_method(&Type::Nullable(box Type::Generic), Type::Bool, vec![], "is_none", "$0 IS
NULL");
}
```

Le module `optimizer` vérifie s'il y a des expressions composées uniquement de littéraux et simplifie celles-ci.

tql_macros/src/optimizer.rs

```
/// A Query optimizer.

use syntax::ast::BinOp_::{BiAdd, BiSub};
use syntax::ast::Expr_::{ExprBinary, ExprLit};
use syntax::ast::Lit_::LitInt;

use ast::{Expression, Limit, Query};
use ast::Limit::{EndRange, Index, LimitOffset, Range, StartRange};
use plugin::number_literal;

/// Check that all the expressions in `expression` are literal.
fn all_integer_literal(expression: &Expression) -> bool {
    match expression.node {
        ExprLit(ref literal) => {
            match literal.node {
                LitInt(_, _) => true,
                _ => false,
            }
        },
        ExprBinary(_, ref expr1, ref expr2) => all_integer_literal(expr1) &&
all_integer_literal(expr2),
        _ => false,
    }
}

/// Reduce an `expression` containing only literals to a mere literal.
fn evaluate(expression: &Expression) -> u64 {
    match expression.node {
        ExprLit(ref literal) => {
            match literal.node {
                LitInt(number, _) => number,
                _ => 0,
            }
        },
        ExprBinary(op, ref expr1, ref expr2) =>
            match op.node {
                BiAdd => evaluate(expr1) + evaluate(expr2),
                BiSub => evaluate(expr1) - evaluate(expr2),
                _ => 0,
            },
        _ => 0,
    }
}
```

```rust
/// Optimize the query.
pub fn optimize(query: &mut Query) {
    match *query {
        Query::Aggregate { .. } => (),
        Query::CreateTable { .. } => (), // Nothing to optimize.
        Query::Delete { .. } => (),
        Query::Drop { .. } => (), // Nothing to optimize.
        Query::Insert { .. } => (),
        Query::Select { ref mut limit, .. } => {
            *limit = optimize_limit(limit);
        },
        Query::Update { .. } => (),
    }
}

/// Optimize the limit by simplifying the expressions containing only literal.
fn optimize_limit(limit: &Limit) -> Limit {
    match *limit {
        EndRange(ref expression) => {
            EndRange(try_simplify(expression))
        },
        Index(ref expression) => {
            Index(try_simplify(expression))
        },
        Range(ref expression1, ref expression2) => {
            if all_integer_literal(expression1) && all_integer_literal(expression2) {
                let offset = evaluate(expression1);
                let expr2 = evaluate(expression2);
                let limit = expr2 - offset;
                LimitOffset(number_literal(limit), number_literal(offset))
            }
            else {
                Range(expression1.clone(), expression2.clone())
            }
        },
        StartRange(ref expression) => {
            StartRange(try_simplify(expression))
        },
        ref limit => limit.clone(),
    }
}

/// If `expression` only contains literal, simplify this expression.
/// Otherwise returns it as is.
fn try_simplify(expression: &Expression) -> Expression {
    if all_integer_literal(expression) {
        number_literal(evaluate(expression))
```

```
    }
    else {
        expression.clone()
    }
}
```

Le module `parser` transforme une expression Rust en une structure `MethodCalls` qui désigne une suite d'appels de méthode.

tql_macros/src/parser.rs

```
//! Rust parsing.

use syntax::ast::Expr;
use syntax::ast::Expr_::{ExprIndex, ExprMethodCall, ExprPath};
use syntax::codemap::{Span, Spanned};
use syntax::ptr::P;

use ast::Expression;
use error::{SqlError, SqlResult, res};

/// A method call.
#[derive(Debug)]
pub struct MethodCall {
    pub arguments: Vec<P<Expr>>,
    pub name: String,
    pub position: Span,
}

/// A collection of method calls.
#[derive(Debug)]
pub struct MethodCalls {
    pub calls: Vec<MethodCall>,
    /// The identifier at the start of the calls chain.
    pub name: String,
    pub position: Span,
}

impl MethodCalls {
    /// Add a call to the method calls `Vec`.
    fn push(&mut self, call: MethodCall) {
        self.calls.push(call);
    }
}

/// Convert a method call expression to a simpler vector-based structure.
pub fn parse(expression: Expression) -> SqlResult<MethodCalls> {
```

```
    let mut errors = vec![];
    let mut calls = MethodCalls {
        calls: vec![],
        name:  "".to_owned(),
        position: expression.span,
    };

    /// Add the calls from the `expression` into the `calls` `Vec`.
    fn add_calls(expression: &Expression, calls: &mut MethodCalls, errors: &mut
Vec<SqlError>) {
        match expression.node {
            ExprMethodCall(Spanned { node: object, span: method_span}, _, ref arguments)
=> {
                add_calls(&arguments[0], calls, errors);

                let mut arguments = arguments.clone();
                arguments.remove(0);

                calls.push(MethodCall {
                    name: object.to_string(),
                    arguments: arguments,
                    position: method_span,
                });
            }
            ExprPath(_, ref path) => {
                if path.segments.len() == 1 {
                    calls.name = path.segments[0].identifier.to_string();
                }
            }
            ExprIndex(ref expr1, ref expr2) => {
                add_calls(expr1, calls, errors);
                calls.push(MethodCall {
                    name: "limit".to_owned(),
                    arguments: vec![expr2.clone()],
                    position: expr2.span,
                });
            }
            _ => {
                errors.push(SqlError::new(
                    "Expected method call",
                    expression.span,
                ));
            }
        }
    }

    add_calls(&expression, &mut calls, &mut errors);
    res(calls, errors)
```

```
}
```

Le module `plugin` fournit des fonctions utilitaires pour créer une expression littérale de nombre ou de type accès à un champ d'une structure.

tql_macros/src/plugin.rs

```
//! Rust compiler plugin functions.

use syntax::ast::{Expr, Ident, Path};
use syntax::ast::Expr_::{ExprField, ExprLit};
use syntax::ast::Lit_::LitInt;
use syntax::ast::LitIntType::SignedIntLit;
use syntax::ast::IntTy::TyI64;
use syntax::ast::Sign;
use syntax::codemap::{Span, Spanned, DUMMY_SP};
use syntax::parse::token::intern;
use syntax::ptr::P;

pub static NODE_ID: u32 = 4294967295;

/// Create the `ExprField` expression `expr`.`field_name` (struct field access).
pub fn field_access(expr: P<Expr>, path: &Path, position: Span, field_name: String) ->
P<Expr> {
    let syntax_context = path.segments[0].identifier.ctxt;
    let ident = Ident::new(intern(&field_name), syntax_context);
    P(Expr {
        id: NODE_ID,
        node: ExprField(expr, Spanned {
            node: ident,
            span: position,
        }),
        span: position,
    })
}

/// Converts a number to an `P<Expr>`.
pub fn number_literal(number: u64) -> P<Expr> {
    P(Expr {
        id: NODE_ID,
        node: ExprLit(P(Spanned {
            node: LitInt(number, SignedIntLit(TyI64, Sign::Plus)),
            span: DUMMY_SP,
        })),
        span: DUMMY_SP,
    })
}
```

Le module `sql` fournit une fonction pour échapper des caractères spéciaux dans une chaîne de caractères.

tql_macros/src/sql.rs

```
//! A module providing SQL related functions.

/// Escape the character '.
pub fn escape(string: String) -> String {
    string.replace("'", "''")
}
```

Le module `state` fournit quatre états globaux (décrits dans la section Architecture du code).

tql_macros/src/state.rs

```
//! Global mutable states handling.
//!
//! There are four global states:
//!
//! The aggregates global state contains the existing aggregate functions.
//!
//! The lint global state contains the arguments at every sql!() macro call site to be
able to
//! analyze their types in the lint plugin.
//!
//! The methods global state contains the existing aggregate functions.
//!
//! The tables global state contains the SQL tables gathered by the `SqlTable` attribute
with their
//! fields.
//! A field is an identifier and a type.

use std::collections::BTreeMap;
use std::collections::HashMap;
use std::mem;

use syntax::codemap::{Span, Spanned};

use methods::{add_initial_aggregates, add_initial_methods};
use types::Type;

/// A collection of tql aggregate functions.
pub type SqlAggregates = HashMap<String, String>;

/// An SQL query argument type.
#[derive(Debug)]
```

```
pub struct SqlArg {
    pub high: u32,
    pub low: u32,
    pub typ: Type,
}

/// A collection of SQL query argument types.
#[derive(Debug)]
pub struct SqlArgs {
    pub arguments: Vec<SqlArg>,
    pub table_name: String,
}

/// A collection of query calls (with their arguments).
/// A map from the call position to its arguments.
/// The position is used to get the right call from the position in the lint plugin.
pub type SqlCalls = HashMap<u32, SqlArgs>;

/// A collection of fields from an `SqlTable`.
pub type SqlFields = BTreeMap<String, Spanned<Type>>;

/// A tql method that can be used in filters.
pub type SqlMethod = HashMap<String, SqlMethodTypes>;

/// A collection mapping types to the methods that can be used on them.
pub type SqlMethods = HashMap<Type, SqlMethod>;

/// Tql method return type, argument types and template.
pub struct SqlMethodTypes {
    pub argument_types: Vec<Type>,
    pub return_type: Type,
    pub template: String,
}

/// An `SqlTable` has a name, a position and some `SqlFields`.
pub struct SqlTable {
    pub fields: SqlFields,
    pub name: String,
    pub position: Span,
}

/// A collection of SQL tables.
/// A map from table name to `SqlTable`.
pub type SqlTables = HashMap<String, SqlTable>;

/// Get the type of the field if it exists.
pub fn get_field_type<'a, 'b>(table_name: &'a str, identifier: &'b str) -> Option<&'a
Type> {
```

```
    let tables = tables_singleton();
    tables.get(table_name)
        .and_then(|table| table.fields.get(identifier))
        .map(|field_type| &field_type.node)
}

/// Get method types by field name.
pub fn get_method_types<'a>(table_name: &str, field_name: &str, method_name: &str) ->
Option<&'a SqlMethodTypes> {
    let tables = tables_singleton();
    let methods = methods_singleton();
    tables.get(table_name)
        .and_then(|table| table.fields.get(field_name))
        .and_then(move |field_type|
            methods.get(&field_type.node)
                .and_then(|type_methods| type_methods.get(method_name))
        )
}

/// Get the name of the primary key field.
pub fn get_primary_key_field(table: &SqlTable) -> Option<String> {
    table.fields.iter()
        .find(|&(_, typ)| typ.node == Type::Serial)
        .map(|(field, _)| field.clone())
}

/// Get the name of the primary key field by table name.
pub fn get_primary_key_field_by_table_name(table_name: &str) -> Option<String> {
    let tables = tables_singleton();
    tables.get(table_name).and_then(|table| get_primary_key_field(table))
}

/// Returns the global aggregate state.
pub fn aggregates_singleton() -> &'static mut SqlAggregates {
    static mut hash_map: *mut SqlAggregates = 0 as *mut SqlAggregates;

    let map: SqlAggregates = HashMap::new();
    unsafe {
        if hash_map == 0 as *mut SqlAggregates {
            hash_map = mem::transmute(Box::new(map));
            add_initial_aggregates();
        }
        &mut *hash_map
    }
}

/// Returns the global lint state.
pub fn lint_singleton() -> &'static mut SqlCalls {
```

```
    static mut hash_map: *mut SqlCalls = 0 as *mut SqlCalls;

    let map: SqlCalls = HashMap::new();
    unsafe {
        if hash_map == 0 as *mut SqlCalls {
            hash_map = mem::transmute(Box::new(map));
        }
        &mut *hash_map
    }
}

/// Returns the global methods state.
pub fn methods_singleton() -> &'static mut SqlMethods {
    static mut hash_map: *mut SqlMethods = 0 as *mut SqlMethods;

    let map: SqlMethods = HashMap::new();
    unsafe {
        if hash_map == 0 as *mut SqlMethods {
            hash_map = mem::transmute(Box::new(map));
            add_initial_methods();
        }
        &mut *hash_map
    }
}

/// Returns the global state.
pub fn tables_singleton() -> &'static mut SqlTables {
    static mut hash_map: *mut SqlTables = 0 as *mut SqlTables;

    let map: SqlTables = HashMap::new();
    unsafe {
        if hash_map == 0 as *mut SqlTables {
            hash_map = mem::transmute(Box::new(map));
        }
        &mut *hash_map
    }
}
```

Le module `string` fournit une fonction pour trouver une chaîne de caractères proche d'une autre et une fonction pour afficher correctement le pluriel dans un message d'erreur.

tql_macros/src/string.rs

```
//! String proximity lookup function.

use std::cmp;
```

```
/// Variadic minimum macro. It returns the minimum of its arguments.
macro_rules! min {
    ( $e:expr ) => {
        $e
    };
    ( $e:expr, $( $rest:expr ),* ) => {
        cmp::min($e, min!($( $rest ),*))
    };
}

/// Finds a near match of `str_to_check` in `strings`.
#[allow(needless_lifetimes)]
pub fn find_near<'a, T>(str_to_check: &str, strings: T) -> Option<&'a String>
    where T: Iterator<Item = &'a String>
{
    let mut result = None;
    let mut best_distance = str_to_check.len();
    for string in strings {
        let distance = levenshtein_distance(&string, str_to_check);
        if distance < best_distance {
            best_distance = distance;
            if distance < 3 {
                result = Some(string);
            }
        }
    }
    result
}

/// Returns the Levensthein distance between `string1` and `string2`.
#[allow(needless_range_loop)]
fn levenshtein_distance(string1: &str, string2: &str) -> usize {
    fn distance(i: usize, j: usize, d: &[Vec<usize>], string1: &str, string2: &str) ->
usize {
        match (i, j) {
            (i, 0) => i,
            (0, j) => j,
            (i, j) => {
                let delta =
                    if string1.chars().nth(i - 1) == string2.chars().nth(j - 1) {
                        0
                    }
                    else {
                        1
                    };
                min!( d[i - 1][j] + 1
                    , d[i][j - 1] + 1
                    , d[i - 1][j - 1] + delta
```

```
                      )
              },
          }
      }

      let mut d = vec![];
      for i in 0 .. string1.len() + 1 {
          d.push(vec![]);
          for j in 0 .. string2.len() + 1 {
              let dist = distance(i, j, &d, string1, string2);
              d[i].push(dist);
          }
      }
      d[string1.len()][string2.len()]
  }

  /// Returns " was" if count equals 1, "s were" otherwise.
  pub fn plural_verb<'a>(count: usize) -> &'a str {
      if count == 1 {
          " was"
      }
      else {
          "s were"
      }
  }
```

Le module `type_analyzer` effectue l'analyse des structures de table et vérifie le type des expressions dans les requêtes.

tql_macros/src/type_analyzer.rs

```
//! Expression type analyzer.

extern crate rustc_front;

use rustc::lint::{EarlyContext, EarlyLintPass, LateContext, LateLintPass, LintArray,
LintContext, LintPass};
use rustc::middle::ty::{Ty, TyS};
use self::rustc_front::hir::Expr;
use self::rustc_front::hir::Expr_::{self, ExprAddrOf, ExprMethodCall, ExprVec};
use syntax::ast::Attribute;
use syntax::codemap::{NO_EXPANSION, BytePos, Span};

use analyzer::unknown_table_error;
use error::{SqlError, ErrorType, SqlResult, res};
use state::{SqlTable, SqlTables, lint_singleton, tables_singleton};
use types::Type;
```

```
declare_lint!(SQL_LINT, Forbid, "Err about SQL type errors");
declare_lint!(SQL_ATTR_LINT, Forbid, "Err about SQL table errors");

pub struct SqlErrorLint;
pub struct SqlAttrError;

impl LintPass for SqlErrorLint {
    fn get_lints(&self) -> LintArray {
        lint_array!(SQL_LINT)
    }
}

impl LintPass for SqlAttrError {
    fn get_lints(&self) -> LintArray {
        lint_array!(SQL_ATTR_LINT)
    }
}

/// Analyze the types of the SQL table struct.
fn analyze_table_types(table: &SqlTable, sql_tables: &SqlTables) -> SqlResult<()> {
    let mut errors = vec![];
    let mut primary_key_count = 0u32;
    for field in table.fields.values() {
        match field.node {
            Type::Custom(ref related_table_name) =>
                if let None = sql_tables.get(related_table_name) {
                    unknown_table_error(related_table_name, field.span, sql_tables, &mut
errors);
                },
            Type::Serial => {
                primary_key_count += 1;
            }
            _ => (),
        }
    }
    match primary_key_count {
        0 => errors.insert(0, SqlError::new_warning("No primary key found",
table.position)),
        1 => (), // One primary key is OK.
        _ => errors.insert(0, SqlError::new_warning("More than one primary key is
currently not supported", table.position)),
    }
    res((), errors)
}

/// Get the types of the elements in a `Vec`.
fn argument_types<'a>(cx: &'a LateContext, arguments: &'a Expr_) -> Vec<Ty<'a>> {
```

```
    let mut types = vec![];
    if let ExprAddrOf(_, ref argument) = *arguments {
        if let ExprVec(ref vector) = argument.node {
            for element in vector {
                if let ExprAddrOf(_, ref field) = element.node {
                    types.push(cx.tcx.node_id_to_type(field.id));
                }
                else {
                    panic!("Argument should be a `&_`");
                }
            }
        }
        else {
            panic!("Arguments should be a `&Vec<_>`");
        }
    }
    else {
        panic!("Arguments should be a `&Vec<_>`");
    }
    types
}

impl EarlyLintPass for SqlAttrError {
    /// Check the ForeignKey types at the end because the order of the declarations does
not matter
    /// in Rust.
    fn exit_lint_attrs(&mut self, cx: &EarlyContext, _: &[Attribute]) {
        static mut analyze_done: bool = false;
        let done = unsafe { analyze_done };
        if !done {
            let sql_tables = tables_singleton();
            for table in sql_tables.values() {
                if let Err(errors) = analyze_table_types(&table, &sql_tables) {
                    span_errors(errors, cx);
                }
            }
        }
        unsafe {
            analyze_done = true;
        }
    }
}

impl LateLintPass for SqlErrorLint {
    /// Check the types of the `Vec` argument of the `postgres::stmt::Statement::query`
and `postgres::stmt::Statement::execute` methods.
    fn check_expr(&mut self, cx: &LateContext, expr: &Expr) {
        if let ExprMethodCall(name, _, ref arguments) = expr.node {
```

```
            let method_name = name.node.to_string();
            if method_name == "query" || method_name == "execute" {
                let types = argument_types(cx, &arguments[1].node);
                let calls = lint_singleton();
                let BytePos(low) = expr.span.lo;
                match calls.get(&low) {
                    Some(fields) => {
                        for (i, typ) in types.iter().enumerate() {
                            let field = &fields.arguments[i];
                            let position = Span {
                                lo: BytePos(field.low),
                                hi: BytePos(field.high),
                                expn_id: NO_EXPANSION,
                            };
                            check_type(&field.typ, typ, position, expr.span, cx);
                        }
                    },
                    None => (),
                }
            }
        }
    }
}

/// Check that the `field_type` is the same as the `actual_type`.
/// If not, show an error message.
fn check_type(field_type: &Type, actual_type: &TyS, position: Span, note_position: Span,
cx: &LateContext) {
    if field_type != actual_type {
        cx.sess().span_err_with_code(position,
            &format!(
                "mismatched types:\n expected `{expected_type}`,    found
`{actual_type:?}`",
                expected_type = field_type,
                actual_type = actual_type
            ), "E0308"
        );
        cx.sess().fileline_note(
            note_position,
            "in this expansion of sql! (defined in tql)"
        );
    }
}

/// Show the compilation errors.
fn span_errors(errors: Vec<SqlError>, cx: &EarlyContext) {
    for &SqlError {ref code, ref message, position, ref kind} in &errors {
        match *kind {
```

```
            ErrorType::Error => {
                match *code {
                    Some(ref code) => cx.sess().span_err_with_code(position, &message,
code),
                    None => cx.sess().span_err(position, &message),
                }
            },
            ErrorType::Help => {
                cx.sess().fileline_help(position, &message);
            },
            ErrorType::Note => {
                cx.sess().fileline_note(position, &message);
            },
            ErrorType::Warning => {
                cx.sess().span_warn(position, &message);
            },
        }
    }
}
```

Le module `types` définit les types permis dans une structure de table et fournit diverses fonctions décrites dans la section Architecture du code.

tql_macros/src/types.rs

```
//! SQL types.

use std::fmt::{self, Display, Formatter};

use rustc::middle::ty::{TypeAndMut, TyS, TypeVariants};
use syntax::ast::{AngleBracketedParameterData, FloatTy, IntTy, Path, PathParameters};
use syntax::ast::Expr_::ExprLit;
use syntax::ast::LitIntType::{SignedIntLit, UnsignedIntLit, UnsuffixedIntLit};
use syntax::ast::Lit_::{LitBool, LitByte, LitByteStr, LitChar, LitFloat,
LitFloatUnsuffixed, LitInt, LitStr};
use syntax::ast::PathParameters::AngleBracketedParameters;
use syntax::ast::Ty_::TyPath;

use ast::Expression;
use gen::ToSql;
use state::{get_primary_key_field, tables_singleton};

/// A field type.
#[derive(Clone, Debug, Eq, Hash, PartialEq)]
pub enum Type {
    Bool,
    ByteString,
```

```
    Char,
    Custom(String),
    F32,
    F64,
    Generic,
    I8,
    I16,
    I32,
    I64,
    LocalDateTime,
    NaiveDate,
    NaiveDateTime,
    NaiveTime,
    Nullable(Box<Type>),
    Serial,
    String,
    UnsupportedType(String),
    UTCDateTime,
}

impl Display for Type {
    /// Get a string representation of the SQL `Type` for display in error messages.
    fn fmt(&self, f: &mut Formatter) -> fmt::Result {
        let typ = match *self {
            Type::Bool => "bool".to_owned(),
            Type::ByteString => "Vec<u8>".to_owned(),
            Type::Char => "char".to_owned(),
            Type::Custom(ref typ) => typ.clone(),
            Type::F32 => "f32".to_owned(),
            Type::F64 => "f64".to_owned(),
            Type::Generic => "".to_owned(),
            Type::I8 => "i8".to_owned(),
            Type::I16 => "i16".to_owned(),
            Type::I32 => "i32".to_owned(),
            Type::I64 => "i64".to_owned(),
            Type::LocalDateTime =>
"chrono::datetime::DateTime<chrono::offset::local::Local>".to_owned(),
            Type::NaiveDate => "chrono::naive::datetime::NaiveDate".to_owned(),
            Type::NaiveDateTime => "chrono::naive::datetime::NaiveDateTime".to_owned(),
            Type::NaiveTime => "chrono::naive::datetime::NaiveTime".to_owned(),
            Type::Nullable(ref typ) => "Option<".to_owned() + &typ.to_string() + ">",
            Type::Serial => "i32".to_owned(),
            Type::String => "String".to_owned(),
            Type::UnsupportedType(_) => "".to_owned(),
            Type::UTCDateTime =>
"chrono::datetime::DateTime<chrono::offset::utc::UTC>".to_owned(),
        };
        write!(f, "{}", typ)
```

```
    }
}

impl<'a> From<&'a Path> for Type {
    /// Convert a `Path` to a `Type`.
    fn from(&Path { ref segments, .. }: &Path) -> Type {
        let unsupported = Type::UnsupportedType("".to_owned());
        if segments.len() == 1 {
            let ident = segments[0].identifier.to_string();
            match &ident[..] {
                "bool" => Type::Bool,
                "char" => Type::Char,
                "DateTime" => match get_type_parameter(&segments[0].parameters) {
                    Some(ty) => match ty.as_ref() {
                        "Local" => Type::LocalDateTime,
                        "UTC" => Type::UTCDateTime,
                        parameter_type => Type::UnsupportedType("DateTime<".to_owned() +
parameter_type + ">"),
                    },
                    None => Type::UnsupportedType("DateTime".to_owned()),
                },
                "f32" => Type::F32,
                "f64" => Type::F64,
                "i8" => Type::I8,
                "i16" => Type::I16,
                "i32" => Type::I32,
                "i64" => Type::I64,
                "ForeignKey" => match get_type_parameter(&segments[0].parameters) {
                    Some(ty) => Type::Custom(ty),
                    None => Type::UnsupportedType("ForeignKey".to_owned()),
                },
                "NaiveDate" => Type::NaiveDate,
                "NaiveDateTime" => Type::NaiveDateTime,
                "NaiveTime" => Type::NaiveTime,
                "Option" =>
                    match get_type_parameter_as_path(&segments[0].parameters) {
                        Some(ty) => {
                            let result = From::from(ty);
                            let typ =
                                if let Type::Nullable(_) = result {
                                    Type::UnsupportedType(result.to_string())
                                }
                                else {
                                    From::from(ty)
                                };
                            Type::Nullable(box typ)
                        },
                        None => Type::UnsupportedType("Option".to_owned()),
```

```
                },
            "PrimaryKey" => {
                Type::Serial
            },
            "String" => {
                Type::String
            },
            "Vec" => match get_type_parameter(&segments[0].parameters) {
                Some(ty) => match ty.as_ref() {
                    "u8" => Type::ByteString,
                    parameter_type => Type::UnsupportedType("Vec<".to_owned() +
parameter_type + ">"),
                },
                None => Type::UnsupportedType("Vec".to_owned()),
            },
            typ => Type::UnsupportedType(typ.to_owned()),
        }
    }
    else {
        unsupported
    }
  }
}

impl PartialEq<Expression> for Type {
    /// Check if an literal `expression` is equal to a `Type`.
    fn eq(&self, expression: &Expression) -> bool {
        // If the field type is `Nullable`, `expected_type` needs not to be an `Option`.
        let typ =
            match *self {
                Type::Nullable(box ref typ) => typ,
                ref typ => typ,
            };
        match expression.node {
            ExprLit(ref literal) => {
                match literal.node {
                    LitBool(_) => *typ == Type::Bool,
                    LitByte(_) => false,
                    LitByteStr(_) => *typ == Type::ByteString,
                    LitChar(_) => *typ == Type::Char,
                    LitFloat(_, FloatTy::TyF32) => *typ == Type::F32,
                    LitFloat(_, FloatTy::TyF64) => *typ == Type::F64,
                    LitFloatUnsuffixed(_) => *typ == Type::F32 || *typ == Type::F64,
                    LitInt(_, int_type) =>
                        match int_type {
                            SignedIntLit(IntTy::TyIs, _) => false,
                            SignedIntLit(IntTy::TyI8, _) => *typ == Type::I8,
                            SignedIntLit(IntTy::TyI16, _) => *typ == Type::I16,
```

```
                            SignedIntLit(IntTy::TyI32, _) => *typ == Type::I32 || *typ ==
Type::Serial,
                            SignedIntLit(IntTy::TyI64, _) => *typ == Type::I64,
                            UnsignedIntLit(_) => false,
                            UnsuffixedIntLit(_) =>
                                *typ == Type::I8 ||
                                *typ == Type::I16 ||
                                *typ == Type::I32 ||
                                *typ == Type::I64 ||
                                *typ == Type::Serial,
                        }
                    ,
                    LitStr(_, _) => *typ == Type::String,
                }
            }
            _ => true, // Returns true, because the type checking for non-literal is done
later.
        }
    }
}

impl<'tcx> PartialEq<TyS<'tcx>> for Type {
    /// Compare the `expected_type` with `Type`.
    fn eq(&self, expected_type: &TyS<'tcx>) -> bool {
        // If the field type is `Nullable`, `expected_type` needs not to be an `Option`.
        let typ =
            match *self {
                Type::Nullable(box ref typ) => typ,
                ref typ => typ,
            };
        match expected_type.sty {
            TypeVariants::TyBool => {
                *typ == Type::Bool
            },
            TypeVariants::TyFloat(float_type) => {
                match float_type {
                    FloatTy::TyF32 => *typ == Type::F32,
                    FloatTy::TyF64 => *typ == Type::F64,
                }
            },
            TypeVariants::TyInt(int_type) => {
                match int_type {
                    IntTy::TyIs => false, // NOTE: system integer does not make sense for
DBMS.
                    IntTy::TyI8 => *typ == Type::I8,
                    IntTy::TyI16 => *typ == Type::I16,
                    IntTy::TyI32 => *typ == Type::I32 || *typ == Type::Serial,
                    IntTy::TyI64 => *typ == Type::I64,
```

```
                }
            },
            TypeVariants::TyRef(_, TypeAndMut { ty, .. }) => {
                match ty.sty {
                    TypeVariants::TyStr => {
                        *typ == Type::String
                    },
                    _ => false,
                }
            },
            TypeVariants::TyStruct(def, sub) => {
                match def.struct_variant().name.to_string().as_str() {
                    "DateTime" => {
                        match sub.types.iter().next() {
                            Some(inner_type) => {
                                match get_type_parameter_from_type(&inner_type.sty) {
                                    Some(generic_type) =>
                                        match generic_type.as_str() {
                                            "UTC" => *typ == Type::UTCDateTime,
                                            "Local" => *typ == Type::LocalDateTime,
                                            _ => false,
                                        },
                                    _ => false,
                                }
                            },
                            None => false,
                        }
                    },
                    "NaiveDate" => *typ == Type::NaiveDate,
                    "NaiveDateTime" => *typ == Type::NaiveDateTime,
                    "NaiveTime" => *typ == Type::NaiveTime,
                    "String" => *typ == Type::String,
                    struct_type => *typ == Type::Custom(struct_type.to_owned()),
                }
            },
            _ => false,
        }
    }
}

/// Get the type between < and > as a String.
fn get_type_parameter(parameters: &PathParameters) -> Option<String> {
    get_type_parameter_as_path(parameters).map(|path| path.segments[
0].identifier.to_string())
}

/// Get the type between < and > as a Path.
fn get_type_parameter_as_path(parameters: &PathParameters) -> Option<&Path> {
```

```
    if let AngleBracketedParameters(AngleBracketedParameterData { ref types, .. }) =
*parameters {
        types.first()
            .and_then(|ty| {
                if let TyPath(None, ref path) = ty.node {
                    Some(path)
                }
                else {
                    None
                }
            })
    }
    else {
        None
    }
}

/// Get the type between < and > as a String.
fn get_type_parameter_from_type(typ: &TypeVariants) -> Option<String> {
    if let TypeVariants::TyStruct(def, _) = *typ {
        Some(def.struct_variant().name.to_string())
    }
    else {
        None
    }
}

/// Convert a `Type` to its SQL representation.
fn type_to_sql(typ: &Type, mut nullable: bool) -> String {
    let sql_type =
        match *typ {
            Type::Bool => "BOOLEAN".to_owned(),
            Type::ByteString => "BYTEA".to_owned(),
            Type::I8 | Type::Char => "CHARACTER(1)".to_owned(),
            Type::Custom(ref related_table_name) => {
                let tables = tables_singleton();
                if let Some(table) = tables.get(related_table_name) {
                    let primary_key_field = get_primary_key_field(table).unwrap();
                    "INTEGER REFERENCES ".to_owned() + &related_table_name + "(" +
&primary_key_field + ")"
                }
                else {
                    "".to_owned()
                }
                // NOTE: if the field type is not an SQL table, an error is thrown by the
linter.
            },
            Type::F32 => "REAL".to_owned(),
```

```
            Type::F64 => "DOUBLE PRECISION".to_owned(),
            Type::Generic => "".to_owned(),
            Type::I16 => "SMALLINT".to_owned(),
            Type::I32 => "INTEGER".to_owned(),
            Type::I64 => "BIGINT".to_owned(),
            Type::LocalDateTime => "TIMESTAMP WITH TIME ZONE".to_owned(),
            Type::NaiveDate => "DATE".to_owned(),
            Type::NaiveDateTime => "TIMESTAMP".to_owned(),
            Type::NaiveTime => "TIME".to_owned(),
            Type::Nullable(ref typ) => {
                nullable = true;
                type_to_sql(&*typ, true)
            },
            Type::Serial => "SERIAL PRIMARY KEY".to_owned(),
            Type::String => "CHARACTER VARYING".to_owned(),
            Type::UnsupportedType(_) => "".to_owned(),
            Type::UTCDateTime => "TIMESTAMP WITH TIME ZONE".to_owned(),
        };

    if nullable {
        sql_type
    }
    else {
        sql_type + " NOT NULL"
    }
}

impl ToSql for Type {
    fn to_sql(&self) -> String {
        type_to_sql(self, false)
    }
}
```

Annexe C: Tests

Le test suivant vérifie que la méthode `insert()` fonctionne comme prévu. Il vérifie que l'insertion a bien eu lieu dans la base de données et que la clé primaire est bien retournée.

tql_macros/tests/insert_expr.rs

```
#![feature(box_patterns, plugin)]
#![plugin(tql_macros)]

extern crate postgres;
extern crate tql;

use postgres::{Connection, SslMode};
use postgres::error::Error::DbError;
use postgres::error::SqlState::UndefinedTable;
use tql::{ForeignKey, PrimaryKey};

mod teardown;

use teardown::TearDown;

#[SqlTable]
struct TableInsertExpr {
    primary_key: PrimaryKey,
    field1: String,
    field2: i32,
    related_field: ForeignKey<RelatedTableInsertExpr>,
    optional_field: Option<i32>,
    boolean: Option<bool>,
    float32: Option<f32>,
    float64: Option<f64>,
    int16: Option<i16>,
    int64: Option<i64>,
}

#[SqlTable]
struct RelatedTableInsertExpr {
    id: PrimaryKey,
    field1: i32,
}

fn get_connection() -> Connection {
    Connection::connect(
        "postgres://nom_d_utilisateur:mot_de_passe@serveur/base_de_donnees",
        &SslMode::None
    ).unwrap()
```

```
}

#[test]
fn test_insert() {
    let connection = get_connection();

    let _teardown = TearDown::new(|| {
        let _ = sql!(TableInsertExpr.drop());
        let _ = sql!(RelatedTableInsertExpr.drop());
    });

    let _ = sql!(RelatedTableInsertExpr.create());
    let _ = sql!(TableInsertExpr.drop());

    let related_id = sql!(RelatedTableInsertExpr.insert(field1 = 42)).unwrap();
    let related_field = sql!(RelatedTableInsertExpr.get(related_id)).unwrap();

    let result = sql!(TableInsertExpr.insert(
        field1 = "value1",
        field2 = 55,
        related_field = related_field
    ));
    match result {
        Err(DbError(box db_error)) => assert_eq!(UndefinedTable, *db_error.code()),
        Ok(_) => assert!(false),
        Err(_) => assert!(false),
    }

    let _ = sql!(TableInsertExpr.create());

    let id = sql!(TableInsertExpr.insert(
        field1 = "value1",
        field2 = 55,
        related_field = related_field
    )).unwrap();
    assert_eq!(1, id);

    let table = sql!(TableInsertExpr.get(id)).unwrap();
    assert_eq!("value1", table.field1);
    assert_eq!(55, table.field2);
    assert!(table.related_field.is_none());
    assert!(table.optional_field.is_none());

    let table = sql!(TableInsertExpr.get(id).join(related_field)).unwrap();
    assert_eq!("value1", table.field1);
    assert_eq!(55, table.field2);
    let related_table = table.related_field.unwrap();
    assert_eq!(related_id, related_table.id);
```

```
    assert_eq!(42, related_table.field1);
    assert!(table.optional_field.is_none());

    let new_field2 = 42;
    let id = sql!(TableInsertExpr.insert(
        field1 = "value2",
        field2 = new_field2,
        related_field = related_field
    )).unwrap();
    assert_eq!(2, id);

    let table = sql!(TableInsertExpr.get(id)).unwrap();
    assert_eq!("value2", table.field1);
    assert_eq!(42, table.field2);
    assert!(table.related_field.is_none());
    assert!(table.optional_field.is_none());

    let new_field1 = "value3".to_owned();
    let new_field2 = 24;
    let id = sql!(TableInsertExpr.insert(
        field1 = new_field1,
        field2 = new_field2,
        related_field = related_field,
        optional_field = 12
    )).unwrap();
    assert_eq!(3, id);

    let table = sql!(TableInsertExpr.get(id)).unwrap();
    assert_eq!("value3", table.field1);
    assert_eq!(24, table.field2);
    assert!(table.related_field.is_none());
    assert_eq!(Some(12), table.optional_field);

    let boolean_value = true;
    let float32 = 3.14f32;
    let float64 = 3.14f64;
    let int16 = 42i16;
    let int64 = 42i64;
    let id = sql!(TableInsertExpr.insert(
        field1 = new_field1,
        field2 = new_field2,
        related_field = related_field,
        optional_field = 12,
        boolean = boolean_value,
        float32 = float32,
        float64 = float64,
        int16 = int16,
        int64 = int64
```

```
    )).unwrap();
    assert_eq!(4, id);
}
```

Voici un tableau résumant tous les tests qui sont effectués :

Fonctionnalité	**Génération de code**	**Exécution**	**Erreur de compilation**
Requêtes d’agrégation	✔	✔	✔
Requêtes de création de table	✔	✔	✖
Requêtes de suppression	✔	✔	✔
Requêtes de suppression de table	✔	✔	✖
Requêtes d’insertion	✔	✔	✔
Méthodes de filtre	✔	✔	✔
Erreur de l’analyse syntaxique	✖	✖	✔
Requêtes de sélection	✔	✔	✔
Attribut `#[SqlTable]`	✔	✖	✔
Requêtes de mise à jour	✔	✔	✔

Annexe D: Exemples

Cette section montre deux exemples de programme utilisant l’extension syntaxique TQL.

Le premier exemple montre un programme en ligne de commande qui permet de gérer une liste de tâches à effectuer.

examples/todo.rs

```
#![feature(plugin)]
#![plugin(tql_macros)]

extern crate chrono;
extern crate postgres;
extern crate tql;

use std::env;

use chrono::datetime::DateTime;
use chrono::offset::utc::UTC;
use postgres::{Connection, SslMode};
use tql::PrimaryKey;

// A TodoItem is a table containing a text, an added date and a done boolean.
#[SqlTable]
struct TodoItem {
    id: PrimaryKey,
    text: String,
    date_added: DateTime<UTC>,
    done: bool,
}

fn add_todo_item(connection: Connection, text: String) {
    // Insert the new item.
    let result = sql!(TodoItem.insert(
        text = text,
        date_added = UTC::now(),
        done = false
    ));
    if let Err(err) = result {
        println!("Failed to add the item ({})", err);
    }
    else {
        println!("Item added");
    }
}
```

```
fn delete_todo_item(connection: Connection, id: i32) {
    // Delete the item.
    let result = sql!(TodoItem.get(id).delete());
    if let Err(err) = result {
        println!("Failed to delete the item ({})", err);
    }
    else {
        println!("Item deleted");
    }
}

fn do_todo_item(connection: Connection, id: i32) {
    // Update the item to make it done.
    let result = sql!(TodoItem.get(id).update(done = true));
    if let Err(err) = result {
        println!("Failed to do the item ({})", err);
    }
    else {
        println!("Item done");
    }
}

fn get_id(args: &mut env::Args) -> Option<i32> {
    if let Some(arg) = args.next() {
        if let Ok(id) = arg.parse() {
            return Some(id);
        }
        else {
            println!("Please provide a valid id");
        }
    }
    else {
        println!("Missing argument: id");
    }
    None
}

fn list_todo_items(connection: Connection, show_done: bool) {
    let items =
        if show_done {
            // Show the last 10 todo items.
            sql!(TodoItem.sort(-date_added)[..10])
        }
        else {
            // Show the last 10 todo items that are not done.
            sql!(TodoItem.filter(done == false).sort(-date_added)[..10])
        };
```

```
    for item in items {
        let done_text =
            if item.done {
                "(✓)"
            }
            else {
                ""
            };
        println!("{}. {} {}", item.id, item.text, done_text);
    }
}

fn main() {
    let connection = get_connection();

    // Create the table.
    let _ = sql!(TodoItem.create());

    let mut args = env::args();
    args.next();

    let command = args.next().unwrap_or("list".to_owned());
    match command.as_ref() {
        "add" => {
            if let Some(item_text) = args.next() {
                add_todo_item(connection, item_text);
            }
            else {
                println!("Missing argument: task");
            }
        },
        "delete" => {
            if let Some(id) = get_id(&mut args) {
                delete_todo_item(connection, id);
            }
        },
        "do" => {
            if let Some(id) = get_id(&mut args) {
                do_todo_item(connection, id);
            }
        },
        "list" => {
            let show_done = args.next() == Some("--show-done".to_owned());
            list_todo_items(connection, show_done);
        },
        command => println!("Unknown command {}", command),
    }
}
```

```
fn get_connection() -> Connection {
    Connection::connect(
        "postgres://test:test@localhost/database", &SslMode::None
    ).unwrap()
}
```

Le second exemple montre un site web de bavardage (« chat »).

examples/chat.rs

```
#![feature(plugin)]
#![plugin(tql_macros)]

extern crate chrono;
extern crate handlebars_iron as hbs;
extern crate iron;
extern crate params;
extern crate persistent;
extern crate postgres;
extern crate r2d2;
extern crate r2d2_postgres;
extern crate rustc_serialize;
extern crate tql;

use std::collections::BTreeMap;

use chrono::datetime::DateTime;
use chrono::offset::utc::UTC;
use hbs::{HandlebarsEngine, Template};
use iron::{Iron, IronResult, Plugin, status};
use iron::middleware::Chain;
use iron::method::Method;
use iron::modifier::Set;
use iron::modifiers::Redirect;
use iron::request::Request;
use iron::response::Response;
use iron::typemap::Key;
use params::{FromValue, Params};
use postgres::SslMode;
use r2d2::Pool;
use r2d2_postgres::PostgresConnectionManager;
use rustc_serialize::json::{Json, ToJson};
use tql::PrimaryKey;

struct AppDb;
```

```
impl Key for AppDb {
    type Value = Pool<PostgresConnectionManager>;
}

// A Message is a table containing a username, a text and an added date.
#[SqlTable]
struct Message {
    id: PrimaryKey,
    username: String,
    message: String,
    date_added: DateTime<UTC>,
}

impl ToJson for Message {
    fn to_json(&self) -> Json {
        let mut map = BTreeMap::new();
        map.insert("username".to_owned(), self.username.to_json());
        map.insert("message".to_owned(), self.message.to_json());
        map.to_json()
    }
}

fn chat(req: &mut Request) -> IronResult<Response> {
    let pool = req.get::<persistent::Read<AppDb>>().unwrap();
    let connection = pool.get().unwrap();
    let mut resp = Response::new();

    let mut data = BTreeMap::new();
    if req.method == Method::Post {
        {
            let params = req.get_ref::<Params>();
            if let Ok(params) = params {
                let username: String = FromValue::from_value(&params[
"username"]).unwrap();
                let message: String = FromValue::from_value(&params["message"]).unwrap();

                // Insert a new message.
                let _ = sql!(Message.insert(
                            username = username,
                            message = message,
                            date_added = UTC::now()
                        ));
            }
        }

        Ok(Response::with((status::Found, Redirect(req.url.clone()))))
    }
    else {
```

```
        // Get the last 10 messages by date.
        let messages: Vec<Message> = sql!(Message.sort(-date_added)[..10]);

        data.insert("messages".to_owned(), messages.to_json());

        resp.set_mut(Template::new("chat", data))
            .set_mut(status::Ok);
        Ok(resp)
    }
}

fn main() {
    let pool = get_connection_pool();

    {
        let connection = pool.get().unwrap();

        // Create the Message table.
        let _ = sql!(Message.create());
    }

    let mut chain = Chain::new(chat);
    chain.link(persistent::Read::<AppDb>::both(pool));
    chain.link_after(HandlebarsEngine::new("./examples/templates/", ".hbs"));
    println!("Running on http://localhost:3000");
    Iron::new(chain).http("localhost:3000").unwrap();
}

fn get_connection_pool() -> Pool<PostgresConnectionManager> {
    let manager = r2d2_postgres::PostgresConnectionManager::new(
        "postgres://test:test@localhost/database", SslMode::None
    ).unwrap();
    let config = r2d2::Config::builder().pool_size(1).build();
    r2d2::Pool::new(config, manager).unwrap()
}
```

Glossaire

AST
: Arbre syntaxique abstrait.

Attribut
: Un attribut est une méta donnée associée à un objet en Rust (module, fonction, structure, …).

Énumération
: En Rust, une énumération est un type permettant de représenter les données d'une de plusieurs possibilité, à l'instar d'une union étiquetée en C.

Langage dédié
: Un langage dédié est un langage de programmation répondant à un besoin spécifique (en l'occurrence effectuer des requêtes SQL).

Langage dédié embarqué
: Un langage dédié embarqué utilise la syntaxe du langage hôte (en l'occurrence Rust) et est implémenté en tant que bibliothèque de ce langage.

Macro procédurale
: Une macro procédurale permet de manipuler l'arbre syntaxique abstrait à souhait, contrairement à une macro qui est un raccourci pour une forme syntaxique.

Rust
: Langage de programmation système compilé, développé par Mozilla, qui supporte les paradigmes impératif et fonctionnel. Il est une alternative sécuritaire au C++.

SQL
: Langage de requête structurée permettant de communiquer avec des bases de données relationnelles.