coq-of-ocaml & Tezos

Application à la vérification formelle de programmes OCaml

Guillaume Claret

Foobar.land `https://foobar.land/`

Novembre 2021

Language OCaml

- langage de programmation fonctionnel et typé
- traits impératifs et objet
- utilisation industrielle (Jane Street, Tezos, ...)

Langage Coq

- un langage de preuve formelle
- on peut écrire des tests qui sont vérifiés pour toutes les entrées possibles
- $=>$ très haut niveau de fiabilité
- on doit écrire les preuves à la main

Langage Coq

On peut écrire des programmes :

```
Fixpoint sum (tree : tree int) : int :=
  match tree with
  | Leaf n => n
  | Node tree1 tree2 =>
    Z.add (sum tree1) (sum tree2)
  end.
```

Langage Coq

On peut écrire des lemmes (comme des tests mais pour toutes les entrées possibles) :

```
Lemma positive_sum (t : tree int)
  : is_positive t -> sum t > 0.
```

Langage Coq

On peut écrire des preuves :

```
intro H; induction tree; simpl;
  inversion H; trivial.
apply positive_plus; now apply positive_sum.
```

Langage Coq

- populaire pour la recherche en langages de programmation
- utilisation pour vérifier des outils critiques
- aussi utilisé pour vérifier des théorèmes mathématiques

Utilisation de Coq

- code écrit en Coq directement puis compilé (CompCert)
- modélisation en Coq
- traduction OCaml vers Coq automatisée : coq-of-ocaml

coq-of-ocaml

- `https://github.com/foobar-land/coq-of-ocaml`
- traduction OCaml -> Coq
- open-source (MIT)
- développé principalement pour Tezos (Nomadic Labs)
- ouvert pour tous les projets OCaml

Exemple

Code OCaml en entrée :

```
type 'a tree =
| Leaf of 'a
| Node of 'a tree * 'a tree
```

Exemple

Code Coq généré :

```
Inductive tree (a : Set) : Set :=
| Leaf : a -> tree a
| Node : tree a -> tree a -> tree a.

Arguments Leaf {_}.
Arguments Node {_}.
```

Exemple

Code OCaml en entrée :

```
let rec sum tree =
  match tree with
  | Leaf n -> n
  | Node (tree1, tree2) -> sum tree1 + sum tree2
```

Exemple

Code Coq généré :

```
Fixpoint sum (tree : tree int) : int :=
  match tree with
  | Leaf n => n
  | Node tree1 tree2 =>
    Z.add (sum tree1) (sum tree2)
  end.
```

Exemple

Ensuite on peut faire les lemmes / preuves.

Idée générale

- générer du code similaire
- avoir du code simple pour les preuves
- ne pas forcément tout gérer dans OCaml

Application

- crypto-monnaie Tezos
- 40.000 lignes d'OCaml traduits en Coq
- `https://nomadic-labs.gitlab.io/coq-tezos-of-ocaml/`

Exploration du site du projet

...online...

Avantages

- traduction maintenue à jour du code OCaml en Coq
- possibilité de vérifier des tests dans tous les cas
- quelques bugs trouvés

Travail restant

- intégrer plus de monde au projet
- vérifier tous les tests écrits en OCaml
- vérifier d'autres parties d'OCaml

Contribution

- `https://nomadic-labs.gitlab.io/coq-tezos-of-ocaml/docs/contribute`
- avec des pull-requests
- avec des issues
- possibilité d'invitation sur le Slack de Tezos

Fonctionnement de coq-of-ocaml

- compilateur OCaml : donne syntaxe typée
- coq-of-ocaml : syntaxe typée -> syntaxe Coq
- pretty-printing pour bien formatter le code

Fonctionnement de coq-of-ocaml

- assez simple (une passe + quelques passe d'analyse)
- message de warning si termes mal gérés
- produit toujours une sortie

Code OCaml traduit

- cœur d'OCaml (fonctions, lets, match, ...)
- types (record, algébrique, synonymes, mutuels)
- modules (modules, foncteurs, signatures, first-class modules)
- type existentiels
- projets en plusieurs fichiers
- `.ml` et `.mli`

Partiellement traduit

- GADTs
- variants polymorphes `‘Constr`
- types extensible +=

Non traduit

- orienté-objet (peu utilisé)
- effets de bord (beaucoup utilisé) :
  - exceptions
  - références
  - I/O (printing, réseau, ...)

Difficultés

Fonctions récursive : on peut désactiver la vérification de terminaison en Coq.

Difficultés

Modules OCaml :

```
module type Source = sig
  type t
  val x : t
end

module M : Source = struct
  type t = int
  let x = 12
end
```

Difficultés

- traduction vers des records Coq
- plus souples pour les preuves que les foncteurs Coq
- support des modules first-class

Difficultés

En Coq :

```
Module Source.
  Record signature {t : Set} : Set := {
    t := t;
    x : t;
  }.
End Source.
Definition Source := @Source.signature.
Arguments Source {_}.
```

Difficultés

En Coq :

```
Module M.
  Definition t : Set := int.

  Definition x : int := 12.

  Definition module :=
    {|
      Source.x := x
    |}.
End M.
Definition M : Source (t := _) := M.module.
```

Code avec effets

- seulement gestion du code monadique
- monades nécessaire en Coq
- projets pour la traduction des exceptions / reférences, ...
- demande de l'inférence

Types

- GADTs
- types extensibles

Merci

Merci
Questions ?