`coqffi`: Génération automatique de FFI Coq/OCaml

Thomas Letan[1] and Li-yao Xia[2]

[1] Agence nationale de la sécurité des systèmes d'information (ANSSI)
Paris, France
thomas.letan@ssi.gouv.fr
[2] University of Pennsylvania, Philadelphia, Pennsylvania, U.S.A.
xialiyao@seas.upenn.edu

Résumé

Dans cet article, nous présentons `coqffi`, un outil pour générer automatiquement des FFI Coq à partir d'interfaces OCaml. Plus précisément, `coqffi` génère le code bureaucratique permettant la liaison à ces interfaces OCaml via le mécanisme d'extraction de Coq. Utilisé conjointement avec le système de compilation `dune`, dont les versions récentes permettent de compiler et d'extraire des théories Coq, `coqffi` permet de mélanger de manière transparente des modules OCaml et Coq au sein d'une même base de code.

1 Introduction

Le mécanisme d'extraction de Coq [5] permet de *transpiler* du Gallina en OCaml, afin de tirer bénéfice du compilateur optimisant OCaml pour exécuter efficacement des programmes certifiés en Coq. Gallina étant un langage de programmation fonctionnel *pur*, c'est-à-dire sans effets de bord, l'utilisation de Coq a tendance à rester confinée à des composants très spécifiques — mais aussi critiques — de logiciels. CompCert [2], un compilateur pour le langage C, est sans doute l'exemple le plus emblématique de cet usage du mécanisme d'extraction. Ses passes de compilation certifiées sont des fonctions pures écrites en Gallina.

Depuis plusieurs années, de nombreux travaux de recheche se sont penchés sur l'écriture et la vérification de code impur en Gallina. C'est le cas de `Coq.io` [1], FreeSpec [3] et Interaction Tree [6] par exemple. Ces trois cadriciels sont compatibles avec le mécanisme d'extraction de Coq et permettent d'envisager un développement logiciel où Gallina est le langage « principal » et où OCaml est cantonné à l'implémentation de bibliothèques logicielles « bas-niveau ».

Dans les deux cas, le mécanisme d'extraction est malheureusement lourd à utiliser, faisant appel à des commandes vernaculaires rébarbatives et sujettes à erreurs. Mal utilisé, il peut ainsi être la source de bogues et vulnérabilités dans le programme OCaml généré. Dans cet article, nous présentons `coqffi`, un outil permettant de générer automatiquement des FFI Coq-OCaml. L'objectif de cet outil est de réduire drastiquement la difficulté d'utilisation du mécanisme d'extraction de Coq. `coqffi` a été publié au sein de l'organisation `coq-community` sur GitHub afin de favoriser son adoption par la communauté Coq [4] [1].

2 Présentation des fonctionnalités

`coqffi` prend en entrée une interface de module OCaml compilée (`.cmi`) et génère en sortie un module Coq (`.v`). Pour chaque déclaration présente dans l'interface de l'entrée, le module généré en sortie contient deux commandes vernaculaires Coq : une définition Coq possédant un type « compatible » (du point de vue du mécanisme d'extraction) et la configuration du

1. https://github.com/coq-community/coqffi

```
val send_msg :
    string -> unit [@@ impure]
```

(**1**) Interface OCaml

```
Axiom ml_send_msg : string -> IO unit.
Extract Constant ml_send_msg =>
    "(fun x k -> k (MyLib.send_msg x))".
```

(**2**) Interface Coq générée

mécanisme d'extraction. `coqffi` traduit récursivement les types OCaml en types Coq. Un type polymorphique OCaml est ainsi introduit avec l'opérateur `forall` Coq. Les fonctions OCaml sont traduites par des fonctions en Gallina. `coqffi` traduit un ensemble de « types primitifs » OCaml bien identifiés vers des types « compatibles » Coq. Enfin, les types OCaml définis par l'interface sont laissés inchangés.

Par ailleurs, `coqffi` distingue deux types de valeurs OCaml, selon si elles sont déclarées pures (par défaut) ou impures (par le biais d'une annotation `[@@impure]`). En ce qui concerne les valeurs impures, `coqffi` axiomatise une monade `IO` et des directives d'extraction en conséquence. Afin de demeurer générique et de laisser la porte ouverte à l'utilisation d'autres monades, `coqffi` génère par ailleurs une classe de types et une instance de cette classe pour le type `IO`.

Par défaut, les valeurs et types OCaml sont exposés en Coq sous la forme d'axiomes. Les utilisateurs de `coqffi` peuvent par la suite introduire des hypothèses sur les comportements de ces définitions, si leur but est de conduire un travail de vérification formelle sur le logiciel qu'ils sont en train d'implémenter. Il est aussi possible d'assigner à une valeur pure OCaml un modèle implémenté en Coq (par le biais de l'annotation `[@@coq_model <model>]`). Dans ce cas là, le terme Coq défini pour la valeur concernée sera définie comme un alias vers son modèle, plutôt que comme un axiome. `coqffi` est aussi capable de générer des types inductifs pour un sous-ensemble de types OCaml [2]. Dans ce cas, le mécanisme d'extraction est configuré afin de projeter les types inductifs ainsi définis vers les types OCaml.

Enfin, `coqffi` s'utilise facilement avec `dune`, ce qui permet de mélanger au sein du même dossier du code Coq et OCaml. Nous souhaitons nous rapprocher des développeurs de `dune` afin de déterminer s'il est envisageable de proposer une *stanza* dédiée, similaire à `coq.theory` et `coq.extraction`.

3 Conclusion

Le principal objectif de `coqffi` est de faciliter l'utilisation du mécanisme d'extraction de Coq, dont la configuration est rapidement lourde et sujette à erreurs. Couplé avec `dune`, il permet de mélanger facilement, au sein de la même base de code, des modules Coq et OCaml. Nous espérons que `coqffi` saura trouver son public au sein de la communauté et favorisera l'utilisation de Coq dans des développement logiciels souhaitant tirer parti de ses fonctionnalités d'assistant de preuves.

Références

[1] Guillaume Claret and Yann Régis-Gianas. Mechanical Verification of Interactive Programs Specified by Use Cases. In *Proceedings of the Third FME Workshop on Formal Methods in Software Engineering*, pages 61–67. IEEE Press, 2015.

[2] Xavier Leroy et al. The CompCert Verified Compiler. *Documentation and user's manual. INRIA Paris-Rocquencourt*, 2012.

2. Il s'agit d'une fonctionnalité expérimentale, désactivée par défaut.

[3] Thomas Letan, Yann Régis-Gianas, Pierre Chifflier, and Guillaume Hiet. Modular Verification of Programs with Effects and Effects Handlers in Coq. In *22st International Symposium on Formal Methods (FM 2018)*, volume 10951. Springer, 2018.

[4] Thomas Letan, Yann Régis-Gianas, Li-yao Xia, and Yannick Zakowski. coqffi : Coq to ocaml ffi made easy. https ://coq.inria.fr/.

[5] Pierre Letouzey. A new extraction for coq. In *International Workshop on Types for Proofs and Programs*, pages 200–219. Springer, 2002.

[6] Li-yao Xia, Yannick Zakowski, Paul He, Chung-Kil Hur, Gregory Malecha, Benjamin C Pierce, and Steve Zdancewic. Interaction Trees : Representing Recursive and Impure Programs in Coq. *Proceedings of the ACM on Programming Languages*, 4(POPL) :1–32, 2019.