Extraction dans le CCI

Loïc Pottier[1]

1: 2004 route des Lucioles, B.P. 93
06902 SOPHIA ANTIPOLIS Cedex FRANCE
`Loic.Pottier@sophia.inria.fr`

1. Introduction

L'extraction [5] dans le système `Coq` [2] est un procédé automatique qui permet de transformer certains termes du calcul des constructions inductives (CCI) en un programme `ocaml` [3]. En gros, pour extraire d'un terme un programme `ocaml`, on efface les sous-termes logiques (ceux qui sont dans la sorte Prop), on garde ceux de la sorte Set (ceux qui ont un contenu constructif), on oublie les dépendances de valeurs dans les types, les types inductifs deviennent des types concrets de `ocaml`, les abstractions, les applications et les définitions récursives sont gardées telles quelles. Le code `ocaml` obtenu est alors très proche du terme `Coq` dont il provient, surtout si ce dernier ne comporte pas de sous-termes logiques (i.e. des preuves).

Malheureusement, l'extraction telle qu'elle est définie dans `Coq` ne permet pas de traiter la sorte Type. Elle peut aussi échouer sur des termes de la sorte Set, les termes `ocaml` obtenus n'étant pas typables dans le système de type de `ocaml`. Le fait de ne pas pouvoir extraire de la sorte Type n'est pas vraiment grave si on n'utilise pas de structure mathématiques abstraites, la sorte Set de Coq étant suffisamment riche pour représenter tout type de données informatique. Mais la sorte Type est nécessaire si on veut pouvoir mélanger données, propositions et preuves dans une même structure. Un exemple est la structure de *Sétoïde*, qu'on représente par un Record de Coq, et qui contient un type (par exemple dans la sorte Set), une relation binaire sur ce type, et une preuve que cette relaton est d'équivalence: dans ce cas un sétoïde doit se situer dans la sorte Type (si on ne veut pas perdre les projections qui donnent ses constituants). Un autre exemple est l'axiomatisation actuelle des réels de Coq, qui sont dans la sorte Type.

On présente ici une méthode permettant d'extraire de n'importe quel terme du CCI (toutes sortes confondues) un programme `ocaml` acceptable par `ocaml`, et dont l'évaluation est efficace. Elle est basée sur trois idées:

- La première idée est de ne pas traduire les types du CCI vers des types `ocaml`, mais vers des termes `ocaml`. En effet, dans `Coq`, les types sont des objets du premier ordre, il est donc nécessaire de les traduire vers des objets similaires de `ocaml`.
- La deuxième idée est de s'affranchir des types et du typage de `ocaml` en utilisant au besoin la fonction *Obj.magic* qui permet de rendre un objet acceptable par n'importe quelle fonction, ou, de manière équivalente, de lui associer le type que l'on veut.
- La troisième idée, proposée par Christine Paulin (réunion CFC de juin 98): tous les termes de la sorte Prop sont identifiés à une unique et nouvelle constante.

Le terme obtenu est alors évaluable par `ocaml` en un terme qui est celui qui aurait été extrait de sa forme normale en `Coq`.

La suite de cet article est organisée ainsi:

- on commence par présenter une méthode de traduction des termes du CCI dans une structure de termes intermédiaire, qu'on illustre par un exemple simple.
- On montre ensuite comment les termes de cette structure intermédiaire se traduisent naturellement en programmes `ocaml`, avec de nouveau un exemple.
- Puis on décrit une implémentation en `Coq` de la méthode d'extraction complète, qu'on teste sur deux exemples, un simple, et un autre où l'extraction standard de `Coq` échoue.
- Enfin on conclue, et on mentionne quelques extensions et applications possibles du travail présenté, par exemple à la compilation des termes du CCI ou à leur évaluation rapide.

2. Extraction "abstraite"

2.1. Structure intermédiaire

Les termes du CCI sont traduits dans une structure de termes intermédiaire, appelée $\mathcal{T}$, qui n'est autre qu'un lambda-calcul avec constantes, récurseurs, constructeurs et pattern-matching, donc très proche de `ocaml` non typé.

Ces termes sont des formes suivantes (où les termes $t, t_1, ..., t_n$ sont dans $\mathcal{T}$):

- $R(n)$: indice de De Bruijn, n est un entier non nul.
- $L(t)$: abstraction, traduit un terme $[x : u]t$ ou $(x : u)t$ du CCI.
- $A(f, t_1, ..., t_n)$: application $f(t_1, ..., t_n)$ du CCI.
- $Cs(s)$: constante de nom s (par exemple les sortes `Prop`, `Set`, `Type`, mais aussi les constantes et définitions du contexte).
- $Rec(k, t_1, ..., t_n)$: définitions mutuellement récursives de fonctions, désigne la $k$ième fonction, correspond à la construction Fix du CCI.
- $Ci(s, i, a_1, a_2)$: type inductif ou constructeur d'un type inductif, de nom s; son indice est i (numéro de constructeur ou rang dans une définition mutuellement récursive, à partir de 0); ses arités sont a_1(paramètres de type inductif inclus) et a_2 (paramètres non inclus). On distingue ces deux arités car dans les constructions *Cases* de Coq, les paramètres du type inductif ambiant ne sont pas donnés comme arguments aux constructeurs des *patterns*. Alors qu'on les utilise dans la structure $\mathcal{T}$.
- $Case(x, t, c_1, ..., c_n)$: branchement par cas sur la valeur x de type inductif t avec n constructeurs.

Il est assez clair que les mécanismes de réduction du CCI (β, δ, ι) se traduisent de manière compatible sur les termes de $\mathcal{T}$: la forme normale de la traduction dans $\mathcal{T}$ d'un terme du CCI est alors égale à la traduction de sa forme normale.

Il peut paraître étrange de traduire les produits dépendants de `Coq` par des abstractions (on aurait pu en effet ajouter une construction de produits à $\mathcal{T}$). La seule interférence qui pourrait être gênante concerne la β-réduction: quel sens donner à l'application d'un produit à un terme, qui se traduit en une abstraction appliquée à un terme? Dans le CCI aucun: les règles de typage interdisent à un tel terme (application d'un produit à un terme) d'être bien typé. Pourtant, un lieur est par nature fait pour être réduit par instanciation de sa variable liée. Alors, dans $\mathcal{T}$ on peut réduire les applications de produits dépendants du CCI, mais à ses risques et périls (rien ne dit *priori* que de telles réductions terminent, par exemple).

Enfin, on remarquera qu'on ne parle plus de types dans $\mathcal{T}$. En effet, un terme de $\mathcal{T}$ n'étant intéressant qu'à partir du moment où il a été traduit d'un terme du CCI bien typé, plus besoin de le typer de nouveau pour s'assurer de son bon comportement vis-à-vis des calculs (β, δ, ι).

En `ocaml`, on peut représenter $\mathcal{T}$ par le type `_T` suivante:

```
type _T =
     R of int
   | L of _T
   | A of (_T array)
   | Cs of string
   | Rec of int*(_T array)
   | Ci of string*int*int*int
   | Case of _T * _T *(_T array)
;;
```

2.2. Exemple

Par exemple, le terme `plus` de l'addition des entiers naturels a pour définition dans `Coq`:

```
Fix plus
  {plus [n:nat] : nat->nat :=
     [m:nat]Cases n of
              O => m
            | (S p) => (S (plus p m))
            end}
```

Traduit dans le type `_T` cela donne:

```
Rec
 (0,
  [|L
     (L
       (Case
         (R 2, Ci ("nat", 0, 0, 0),
            [|R 1;
              L (A [|Ci ("S", 2, 1, 1);
                     A [|R 4; R 1; R 2|]|])|])))|])
```

2.3. Effondrement des termes logiques

Lors de la traduction CCI -> $\mathcal{T}$, tous les termes dont la sorte est `Prop` sont traduits vers le terme `Cs "Prop"`. L'idéal aurait été de les supprimer purement et simplement, comme cela est fait dans l'extraction standard de `Coq`. Mais, en raison de la coercion Prop -> Type, il n'est pas possible de supprimer un terme de Prop qui serait utilisé dans un contexte où il est vu dans Type. Supprimer ou non un terme logique dépend alors du contexte de son utilisation. Devant la complexité de cette opération, on a choisi l'idée proposée par Christine Paulin, qui consiste à identifier tous les termes de Prop. Notons que la sorte d'un sous-terme est calculée au vol en fonction de ce sous-terme et des variables de son environnement, et non en fonction de son contexte (par exemple s'il est argument d'une fonction qui attend un argument de sorte Type, ceci ne sera pas pris en compte).

Prenons par exemple le terme `plus1`, qui ajoute 1 à son argument, et dont le type `nat->{x:nat | (lt O x)}`:

```
[n:nat](exist nat [x:nat](lt O x) (S n) (le_n_S O n (le_O_n n)))
```

Dans _T, son extraction est:

```
L
 (A
   [|Ci ("exist", 1, 4, 2);
       Ci ("nat", 0, 0, 0);
       Cs "Prop";
       A [|Ci ("S", 2, 1, 1); R 1|];
       Cs "Prop"|])
```

On voit que les termes `[x:nat](lt O x)` et `(le_n_S O n (le_O_n n))` ont été effondrés en `Cs "Prop"`.

3. Extraction en `ocaml`

La traduction des termes de $\mathcal{T}$ en `ocaml` est assez directe:

- les indices de De Bruijn sont transformés en variables, selon leur contexte de liaison (les lieurs étant les constructeurs `L, Rec`).
- les abstractions sont traduites en formes `(fun x -> t)`.
- une application $A(f, t_1, ..., t_n)$ est traduite en application curryfiée $f\ t_1\ ...\ t_n$.
- une constante $Cs(s)$ est traduite en définition globale `let s=def;;` lorsqu'elle en a une dans l'environnement courant de `Coq`.
- les définitions récursives $Rec(k, t_1, ..., t_n)$ sont traduites en formes `let rec f1 = ... and f2 = ... and fn = ... in fk;;`.
- les types inductifs et leurs constructeurs sont tous regroupés dans un type concret global de `ocaml` dont ils deviennent les constructeurs (un exemple sera plus clair).
- les branchements $Case(x, t, t_1, ..., t_n)$ sont traduits en pattern matching `match x with ...`.

3.1. Exemple

Prenons l'exemple de la fonction `plus` dáddition sur les entiers naturels de `Coq`:

```
plus =
Fix plus
  {plus [n:nat] : nat->nat :=
     [m:nat]Cases n of
              O => m
            | (S p) => (S (plus p m))
            end}
```

Traduit dans $\mathcal{T}$, ce terme done:

```
Rec
 (0,
  [|L
     (L
       (Case
         (R 2,
```

```
          Ci (nat, 0, 0, 0),
          [|R 1;
            L
             (A
               [|Ci (S, 2, 1, 1);
                 A
                  [|R 4; R 1;
                    R 2|]|])|])))|])
```

Enfin, le code ocaml produit à partir de ce terme de $\mathcal{T}$ est le suivant:

```
type tml = Type | Set | Prop
   | Nat
   | O
   | S of tml
;;

let plus =
 (let rec funrec1 =
   (fun x1 ->
    (fun x2 ->
     (match x1 with
      | O -> x2
      | S a2_1 -> (S (funrec1 a2_1 x2))) ))
   in funrec1)
 ;;
```

On voit qu'il y a deux parties dans ce code:

- la définition du type tml qui va regrouper les types inductifs mis en jeu dans la définition de plus1, i.e. nat [1], ainsi que leurs constructeurs O et S. Les autres constructeurs sont toujours présents et servent à représenter les autres objets de Coq (sortes).
- l'extraction de plus proprement dite, qui reste proche de ce quón aurait écrit à la main en ocaml.

Dans cet exemple simple, le code ocaml produit ne pose pas de problème de typage à ocaml. Dans des exemples plus compliqués, ce code peut ne pas être typable. Dans ce cas la solution est d'encapsuler les termes qui ne sont pas des applications de constructeurs du type tml par la fonction Obj.magic de ocaml, qui donne le type attendu par le contexte à son argument. Malheureusement, comme on le verra sur des tests, le temps de calcul peut être presque doublé par cette opération.

4. Implémentation, tests et comparaisons

4.1. Commande Coq

La méthode d'extraction présentée est implémentée sous la forme d'une commande que l'on peut utiliser sous Coq. Pour cela il faut d'abord charger le module concerné:

```
Require FullExtraction.
```

puis utiliser la commande avec la syntaxe suivante:

[1] les majuscules ont été ajoutées car les constructeurs des types de ocaml doivent en comporter une.

```
Extract "<fichier>" [<identificateur1> <identificateur2> ... ].
```

Pour assurer que `ocaml` typera bien le code produit, ajouter l'argument `exact`:

```
Extract "<fichier>" [<identificateur1> <identificateur2> ... ] exact.
```

Le code complet est disponible à l'adresse Web suivante:
`http://www-sop.inria.fr/lemme/Loic.Pottier/Coq/extraction.html`.
Traitons maintenant deux exemples.

4.2. Division euclidienne

Dans la librairie de `Coq` est définie une division euclidienne sur les entiers, dont le théorème principal est:

```
Theorem div2 : (b:nat)(gt b O)->(a:nat)(diveucl a b).
```

Sachant que le prédicat `diveucl` est:

```
Inductive diveucl [a,b:nat] : Set
      := divex : (q,r:nat)(gt b r)->(a=(plus (mult q b) r))->(diveucl a b).
```

Le code `ocaml` extrait de `div2` est le suivant, et se type bien par `ocaml`:

```
type tml = Type | Set | Prop
   | Sumbool of tml*tml
   | Left of tml*tml*tml
   | Right of tml*tml*tml
   | Nat
   | O
   | S of tml
   | Diveucl of tml*tml
   | Divex of tml*tml*tml*tml*tml*tml
;;

let div2 =
 (let rec funrec1 =
   (fun x1 ->
    (fun x2 ->
     (fun x3 ->
      (match x3 with
       | O -> (Divex(O,x1,O,O,Prop,Prop))
       | S a2_1 ->
        (match (funrec1 x1 Prop a2_1) with
         | Divex(a4_1,a4_2,a4_3,a4_4,a4_5,a4_6) ->
          (match
            ((let rec funrec8 =
               (fun x4 ->
                (fun x5 ->
                 (match x4 with
                  | O -> (Left(Prop,Prop,Prop))
                  | S a9_1 ->
                   (match x5 with
                    | O -> (Right(Prop,Prop,Prop))
                    | S a11_1 ->
```

```
                    (match (funrec8 a9_1 a11_1) with
                     | Left(a13_1,a13_2,a13_3) ->
                      (Left(Prop,Prop,Prop))
                     | Right(a13_1,a13_2,a13_3) ->
                      (Right(Prop,Prop,Prop)) ) ) ) ))
              in funrec8)  x1 (S a4_4)) with
           | Left(a5_1,a5_2,a5_3) ->
            (Divex((S a2_1),x1,(S a4_3),O,Prop,Prop))
           | Right(a5_1,a5_2,a5_3) ->
            (Divex((S a2_1),x1,a4_3,(S a4_4),Prop,Prop)) ) ) ) )))
   in funrec1)
 ;;
```

En utilisant l'extraction standard de `Coq`, on obtient:

```
type nat =
    O
  | S of nat


type sumbool =
    Left_ren
  | Right_ren


type diveucl =
    Divex of nat * nat

let div2 =
  let rec div1 b = function
    O -> Divex (O, O)
  | S n ->
      (match div1 b n with
         Divex (q, r) ->
           (match let rec inf_dec n0 m =
                    match n0 with
                      O -> Left_ren
                    | S n' ->
                        (match m with
                           O -> Right_ren
                         | S m' ->
                             (match inf_dec n' m' with
                                Left_ren -> Left_ren
                              | Right_ren -> Right_ren))
                  in inf_dec b (S r) with
              Left_ren -> Divex ((S q), O)
            | Right_ren -> Divex (q, (S r))))
  in div1
```

Maintenant comparons les temps de calcul, dans le cas de la division euclidienne de 30000 par 31 (sur un portable DELL 350MHz sous Windows 98):

extraction standard de `Coq`	1,1s
extraction complète sans Obj.magic	1,3s
extraction complète avec Obj.magic	2 s

Le temps légèrement supérieur de l'extraction complète sans *Obj.magic* par rapport à l'extraction standard s'explique ici par le fait que les fonctions ont plus d'arguments, puisque les preuves restent en arguments sous la forme de la constante *Prop*, alors qu'elles disparaissent dans l'extraction standard.

Enfin, le fait que l'utilisation de *Obj.magic* ralentisse tant le code produit reste mystérieux pour nous, bien qu'il soit normal qu'un compilateur utilise des informations de type pour produire du code plus efficace.

4.3. Lemme de Higman

Il s'agit d'une des contributions au système `Coq`: `Higman.v`[4]. Il s''agit du fait que dans toute suite de mots $(w_i)_{i \in N}$ sur un alphabet à deux lettres, il existe deux indices $i < j$ tels que w_i est un sous-mot de w_j. La preuve de cette propriété est constructive, et fournit un algorithme permettant de calculer les indices en question.

L'extraction standard échoue sur cet exemple:

```
Coq < Write Caml File "higman" [Higman].
The axiom False_rec is translated into an exception.
Error during interpretation of command:
Write Caml File "higman" [Higman].
Error: Constant Minbad does not correspond to an ML type
```

Notre méthode donne un code qui n'est pas typable par ML si on n'utilise pas l'encapsulation par la fonction `Obj.magic`. Sinon, comme prévu, le code produit (2572 lignes) est typable par `ocaml`. La fonction principale est `_Higman`, elle prend en argument une suite de mots $(w_i)_{i \in N}$ sur un alphabet à deux lettres, et rend deux indices $i < j$ tels que w_i est un sous-mot de w_j. On peut l'utiliser en `ocaml` comme le montre cet exemple:

```
#use "higman.ml";;
(* Une fonction d'impression partielle des termes de tml *)
let rec ptml t = match t with
   | A_intro (_,i,j,_,_) -> "("^(ptml i)^","^(ptml j)^")"
   | O -> "O"
   | (S n) -> "(S "^(ptml n)^")"
   | Emptyword -> ""
   | Cons (a,w)-> (ptml a)^(ptml w)
   | A0 -> "0"
   | A1 -> "1"
;;
(* une suite aléatoire *)
let s2 a n =
    Random.init 1;
    let l = Random.int (a+ (nat_to_int n)) in
    let s = ref Emptyword in
    for i=1 to l do
       s:=Cons ((if (Random.int 2)=0 then A0 else A1),
                !s);
    done;
    !s
;;
(* Une fonction d'impression des suites *)
let ps s n =
    let a=ref "\n" in
    for i=0 to n do
      a:=(!a)^"\n"^(ptml (s (int_to_nat i)));
    done;
```

```
    print_string (!a)
;;
(* les 5 premiers termes de (s2 10) *)
ps (s2 10) 5;;
10100001
00001
100001
0001
10100001
(* le calcul proprement dit *)

ptml (_Higman (s2 10));;
- : string = "((S 0),(S (S 0)))"
```

Le dernier résultat indique que le mot d'indice 1 de la suite (`00001`) est un sous-mot du mot d'indice 2 (`100001`). En effet.

5. Conclusion

On a montré comment on peut extraire un programme `ocaml` raisonnablement efficace et parfaitement sûr à partir de n'importe quel terme de `Coq`, en oubliant autant que possible les parties logiques et donc non constructives de ce terme. La méthode proposée est implémentée et conduit à des résultats qui nous semblent probants, sur des exemples de taille conséquente.

Bien sûr, il reste des améliorations à apporter. D'une part au code généré, qui peut gagner en lisibilité, en concision, en efficacité, et en modularité. D'autre part à la méthode elle-même. En effet on peut sans doute mener une analyse plus poussée pour déterminer en fonction du contexte si un argument logique d'une fonction peut être éliminé plutôt qu'effondré. D'une manière plus générale, sans doute peut-on utiliser des résultats comme ceux de [6] pour extraire d'un terme fonctionnel uniquement les sous-termes nécessaires au calcul de son résultat.

Une autre piste à explorer consisterait à utiliser `ocaml` comme compilateur de `Coq`, pour effectuer des calculs dont le résultat est à priori simple (i.e. sans abstractions ni produits). Dans ce cas on extrairait vers `ocaml` l'intégralité d'un terme `Coq`, on évaluerait le code `ocaml` produit et compilé, et on remonterait le résultat dans `Coq` (essentiellement si celui-ci est constitué de constructeurs de types inductifs et de constantes opaques). Sans doute améliorerait-on de manière sensible l'efficacité de certains calculs de forme normale dans `Coq` en profitant du compilateur très efficace de `ocaml`.

Comme nous l'a indiqué un des rapporteur de ce papier, `ocaml` limite le nombre de constructeurs non constants par type à environ 250. Dans le cas de gros développements, cela pourrait être gênant. Une solution dans ce cas peut être de ne pas faire de pattern matching sur les constructeurs dans le code extrait, mais plutôt d'associer un indice et une liste d'arguments à chaque constructeur et de faire le pattern matching sur cet indice et la liste des arguments. Mais sans doute perdrait-on légèrement en efficacité, et beaucoup en lisibilité du code produit.

Enfin, notons que la structure intermédiaire $\mathcal{T}$ a été utilisée dans une expérience d'extraction de `Coq` vers `Java` [7].

Note: ce travail a été réalisé principalement dans le cadre de l'action de recherche "Calcul formel certifié" durant 1998 et 1999 [1].

Remerciements: nous tenons à remercier les rapporteurs anonymes de la première version de ce papier pour les remarques qu'ils nous ont faites.

Bibliographie

[1] Calcul formel certifié. *Action de recherche coordonnée de la direction scientifique de l'INRIA*, pages Web `http://www-sop.inria.fr/croap/CFC/`.

[2] B. Barras et al. The coq proof assistant, reference manual. *Projet Coq, INRIA*, pages Web `http://pauillac.inria.fr/coq/assis-fra.html`, 1999.

[3] X. Leroy et al. The objective caml system. *INRIA*, page Web `http://caml.inria.fr/ocaml/htmlman/`, 2000.

[4] H. Herbelin. A program from an A-translated impredicative proof of Higman's Lemma . *Contributions to the system Coq*, page Web `http://pauillac.inria.fr/coq/contribs/higman.html`.

[5] C. Paulin-Mohring and B. Werner. Synthesis of ML programs in the system Coq. *Journal of Symbolic Computation*, 15:607–640, 1993.

[6] F. Prost. *Interprétation de l'analyse statique en théorie des types.* PhD thesis, École Normale Supérieure de Lyon, december 1999.

[7] V. Prévosto. Extraction de Coq vers Java. *Rapport de stage dans le projet Lemme, INRIA Sophia Antipolis*, 1999.