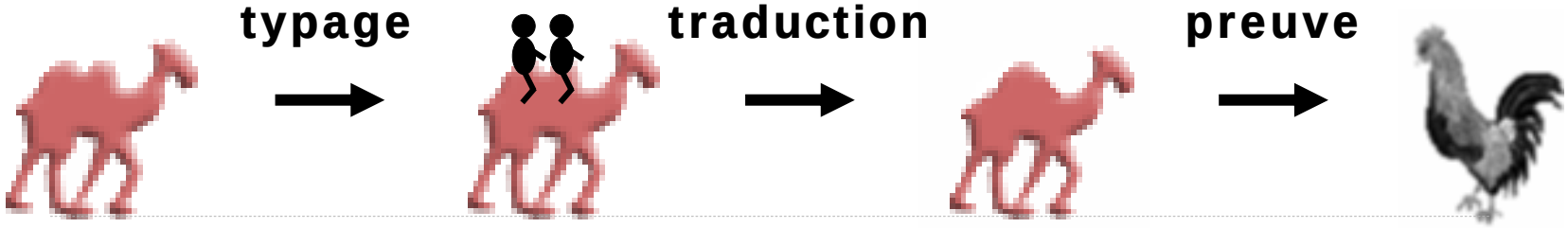
Vérification Interactive de Programmes Caml Purs

Arthur Charguéraud

INRIA – Projet Gallium

Séminaire Typical École Polytechnique, 02/04/2009

Projet de thèse



CAML impératif	CAML bien typé	CAML fonctionnel	ML décrit dans Coq

1) Typage du source avec régions et capacités
2) Traduction vers un programme fonctionnel équivalent (traduction dirigée par le typage)
3) Raisonnement sur le programme fonctionnel

Référence: *Functional Translation of a Calculus of Capabilities*
Arthur Charguéraud & François Pottier, *ICFP'08*

Preuve de programmes

1) Hoare logic + Verification Condition Generator

– annoter le programme source avec des invariants
– extraire des obligations de preuves
– les décharger automatiquement ou interactivement
– ex: Why (Filliâtre), Pangolin (Pottier & Régis-Gianas)

2) Shallow embedding

– programmer dans la logique d'un assistant de preuve
– ex: programmation en Coq, Ynot (Morrisett *et al*)

3) Deep embedding

– décrire la syntaxe et la sémantique d'un langage dans la logique d'un assistant de preuve
– ex: mini-langage procédural (Mehta & Nipkow)
– ex: code machine (Zhong Shao *et al*)
– ex: code Caml pur (dans ce travail)

Deep Embedding : pros & cons

Intérêts de cette approche:

– pas de contraintes sur la forme de code source
– pas de limitation sur le langage de spécification
– contrôle total sur le déroulement de la preuve
– inutile d'apprendre un nouveau langage
(on utilise juste Caml et Coq, plus quelques tactiques)

Limitations potentielles de cette approche:

– risque d'avoir des obligations de preuves illisibles
– risque d'avoir des scripts de preuves très longs
– risque de ne pas pouvoir automatiser les preuves

Objectif : montrer qu'un deep embedding est utilisable en pratique, et pas juste en théorie.

Shéma général

Programmes CAML		**Développement Coq**
		déf. syntaxe et sémantique
définitions de types	→	déf. de type correspondante
définitions top-level	→	déf. de termes embeddés
		spécification des termes sous forme de lemmes
		vérification de (code,spec) via la preuve de ces lemmes

En pratique, on utilise un outil externe qui parse des sources Caml et produit des définitions Coq.

Quatre ingrédients

1) Description de la syntaxe et de la sémantique

$$t : \mathsf{Trm} \qquad t \longrightarrow t'$$

2) Définition de prédicats de comportement

$$t \rhd B \qquad t \rhd \mid P$$

3) Réflection des valeurs dans la logique

$$\mathsf{val}\ (\mathsf{vconstr}_2\ \mathsf{cons}\ (\mathsf{vint}\ 4)\ (\mathsf{vconstr}_0\ \mathsf{nil}))$$
$$= \mathsf{\_List}\ \mathsf{\_Int}\ (4 :: \mathsf{Nil})$$

4) Règle de raisonnement en style big-step

$$\frac{t_1 \rhd \mid P \qquad \forall x.\ (P\,x) \Rightarrow t_2 \rhd B}{(\mathsf{let}\, x = t_1\, \mathsf{in}\, t_2) \rhd B}$$

(presque correct)

Représentation de la syntaxe

```
Inductive trm : Type :=
| trm_val : val -> trm
| trm_var : var -> trm
| trm_app : trm -> trm -> trm
| trm_con : con -> list trm -> trm
| trm_abs : pat -> trm -> trm -> trm
with val : Type :=
| val_int : int -> val
| val_prim : prim -> val
| val_con : con -> list val -> val
| val_abs : pat -> trm -> trm -> val
with pat : Type :=
| pat_var : var -> pat
| pat_int : int -> pat
| pat_con : con -> list pat -> pat
| pat_wild : pat
| pat_alias : var -> pat -> pat.
```

– 3 types: termes, valeurs closes, et patterns

– la substitution est l'identité sur les valeurs closes

– une valeur **v** peut être vu comme un terme, via **trm_val**

– **var** et **con** sont des constantes

\+ exceptions et fonctions récursives

Affichage du code embeddé

Le code source de List.map (VerifList.ml) :

```
let rec map f = function
  | [] -> []
  | a::l -> let r = f a in r :: map f l
```

Le code embeddé correspondant (VerifList_ml.v) :

```
Definition map : val :=
  'let_rec_fun 'map '=
    'fun 'f '->
    'function '| '[] '-> '[]
              '| 'a ':: 'l  '-> ('let 'r '= 'f ' 'a 'in
                                 'r ':: 'map ' 'f ' 'l)
```

Le symbole quote est utilisée pour tous les mots clés du langage objet, ainsi que pour dénoter l'application. Les variables, comme **'a**, sont aussi des notations.

Définition de la sémantique

```
Inductive red: trm -> trm -> Prop :=
| red_app_2 : forall t1 t2 t2r,
    t2 --> t2r ->
    (t1 ' t2) --> (t1 ' t2r)
| red_app_1 : forall t1 t1r v2,
    t1 --> t1r ->
    (t1 ' v2) --> (t1r ' v2)
| red_beta : forall p t1 t2 v,
    (val_abs p t1 t2) ' v -->
       match matching p v with
       | None => t2 ' v
       | Some m => subst m t1
       end
| red_val : forall v t,
    trm_to_val t = Some v ->
    t --> v
where "t1 --> t2" := (red t1 t2)
```

La relation :

`t --> t'`

définit une sémantique

– small-step,

– call-by-value,

– déterministe.

On définit ensuite la clôture transitive :

`t -->* t'`

+ exceptions, récursion, primitives

Spécification des termes

Quatres comportements big-step considérés :

$$B \quad := \quad (|\, P) \quad | \quad !\, v \quad | \quad \Uparrow \quad | \quad ?$$

Prédicat : "le terme *t* admet le comportement *B*"

$$t \rhd B$$

t retourne une valeur	*t* lance une exception	*t* diverge	*t* n'est pas spécifié

$$\frac{P\, v}{t \longrightarrow^* \mathsf{val}\, v \quad t \rhd (|\, P)} \qquad \frac{t \longrightarrow^* \mathsf{exn}\, v}{t \rhd (!\, v)} \qquad \frac{\mathsf{diverges}\, t}{t \rhd (\Uparrow)} \qquad \frac{}{t \rhd (?)}$$

où $\mathsf{diverges}\, t \quad \equiv \quad \forall t'.\ (t \longrightarrow^* t') \Rightarrow \exists t''.\ (t' \longrightarrow t'')$

Spécification des fonctions

Spécification avec pré / post-conditions *P* et *Q* :

$$\forall v. \quad (P\, v) \quad \Rightarrow \quad \exists v'.\ (\mathsf{App}\, f\, v) \longrightarrow^{*} (\mathsf{val}\, v')\ \wedge\ (Q\, v\, v')$$

La même chose, avec le prédicat *comportement* :

$$\forall v. \quad (P\, v) \quad \Rightarrow \quad (\mathsf{App}\, f\, v) \rhd \mid (Q\, v)$$

Plus généralement, la spécification d'une fonction *f* est une proposition *K* dépendant de l'argument *v* et du terme "*App f v*".

$$\mathsf{spec}\, f\, K \quad \equiv \quad \forall v.\ K\, v\, (\mathsf{App}\, f\, v)$$

où $K : \mathsf{Val} \to \mathsf{Trm} \to \mathsf{Prop}$

Réflection dans la logique

	Valeur	**Type**
Programme ML	4::nil	int list
Embedding	vconstr_2 cons (vint 4) (vconstr_0 nil)	Val
Logique	4::Nil	List Int

Formalisation à l'aide de fonctions de traduction des valeurs logiques vers les valeurs embeddés.

Note: on ne considère pas les types récursifs. On verra plus tard comment traiter les fonctions de 1ère classe.

Réflection des types

Types ML :

```
type bool =
  | true
  | false
type list 'a =
  | nil
  | cons of 'a * list 'a
type bitlist = list bool
```

Types Coq :

```
Inductive Bool : Type :=
  | True : Bool
  | False : Bool.
Inductive List (A:Type) : Type :=
  | Nil : List A
  | Cons : A -> List A -> List A.
Definition Bitlist := List Bool.
```

Cette première étape consiste à traduire la syntaxe des définitions de types ML vers Coq.

Note : comme on ne traite pas les types des fonctions ni les types récursifs, il n'y pas de problèmes d'occurence négative.

Définition des encoders

Les *encoders* traduisent des valeurs logiques vers les valeurs embeddées correspondantes.

```
Definition _Bool (b:Bool) : Val :=
   match b with
   | True => vconstr_0 true
   | False => vconstr_0 false
   end.
Fixpoint _List (A:Type) (_A:A->Val) (l:List A) : Val :=
  match l with
  | Nil => vconstr_0 nil
  | Cons h t => vconstr_2 cons (_A h) (_List A _A t)
  end.
Definition _Bitlist : List Bool -> Val :=
  _List Bool _Bool.
```

Avec les arguments implicites, on écrit "**_List _Bool**".

Cas général

Pour chaque type ML, on définit :
– le type logique *A* correspondant,
– un encoder pour les valeurs de ce type :
une fonction nommée *_A* et de type *A → Val*.

Si *(X : A)* est une valeur logique, alors
***(_A X : Val)* est la valeur embeddée associée.**

Les fonctions n'ont pas d'équivalent logique.
Ainsi *list (int → int)* est reflété par *List Val*.
L'encoder associé est *_List Val _Val*, où *_Val* est la fonction identité sur les valeurs de type *Val*.

La génération des définitions de types et des encoders est effectuée par un programme Caml.

Spécification des termes

Le comportement "*t* retourne l'encodage par _*A* d'une valeur logique de type *A* satisfiant *P*" :

$$\frac{t \longrightarrow^{*} \mathsf{val}\,(\_\mathsf{A}\,V) \qquad P\,V}{t \rhd \_\mathsf{A} \mid P}$$

Les spécifications sont ainsi montés au niveau logique (*P* est de type "$A \rightarrow Prop$", où *A* est un type logique).

En pratique :

```
t >> _Bool | (fun b => b = True)
t >> _Bool | = True
t >> _Int | (fun n => 0 < n < 10)
t >> [n:_Int] | 0 < n < 10
```

Spécification des fonctions

Le prédicat "l'application de *f* à l'encodage *_A V* d'une valeur *V* est un terme satisfiant *(K V)*" :

$$\mathsf{Spec}\,(f : \mathsf{Val})\,(A : \mathsf{Type})\,(\_A : A \to \mathsf{Val})\,(K : A \to \mathsf{Trm} \to \mathsf{Prop})$$
$$\equiv \quad \forall\,(V : A).\ K\,V\,(\mathsf{App}\,f\,(\_A\,V))$$

Exemple :

$$\mathsf{Spec}\ \ \mathsf{neg}\ \ \mathrm{Int\_Int}\ \ (\lambda n.\,\lambda t.\,(t \rhd \mathrm{\_Int}\,|\,{=}\,{-}n)\,)$$
$$= \forall\,n.\,\exists\,m.\ \ (\mathsf{App}\,\mathsf{neg}\,(\mathrm{\_Int}\,n)) \longrightarrow^{*} (\mathrm{\_Int}\,m) \ \ \wedge \ \ m = -n$$

En pratique :

```
spec neg [n:_Int] = t is t >> [m:_Int]| m = -n
spec neg [n:_Int] = t is t >> _Int | = -n
spec neg [n:_Int] is >> _Int | = -n
```

Règles de raisonnement

Ces règles sont des lemmes prouvés corrects vis à vis de la sémantique à petit pas.

Pour les termes :

$$\frac{t \longrightarrow^* t' \qquad t' \rhd B}{t \rhd B} \qquad \frac{t_1 \rhd \_A \mid P \qquad \forall X.\ (P\,X) \Rightarrow ([x \rightarrow \mathsf{val}\,(\_A\,X)]\,t_2) \rhd B}{(\mathsf{let}\,x = t_1\,\mathsf{in}\,t_2) \rhd B}$$

Pour les fonctions :

SPEC-WEAKEN

$$\frac{\mathsf{Spec}\,f\,A\,\_A\,K' \qquad \forall X\,t.\ (K'\,X\,t) \Rightarrow (K\,X\,t)}{\mathsf{Spec}\,f\,A\,\_A\,K}$$

SPEC-INDUCTION

$$\frac{\mathsf{Spec}\,f\,A\,\_A\,(\lambda X t.\ H \Rightarrow K\,X\,t) \quad \text{where} \quad H \equiv \mathsf{Spec}\,f\,A\,\_A\,(\lambda X' t'.\ X' \prec X \Rightarrow K\,X'\,t')}{\mathsf{Spec}\,f\,A\,\_A\,(\lambda X t.\ K\,X\,t)}$$

Plutôt que d'appliquer ces lemmes à la main avec **`apply`**, on utilise des tactiques pour aider à instancier les prémisses et pour décharger les buts triviaux.

Exemple : List.length

```
let rec length_aux len = function
  | [] -> len
  | a::l -> length_aux (len + 1) l
```

```
Lemma length_aux_spec :
  spec Caml.length_aux [n:_Int] [l:_List _A] is
    >> _Int | = n + length l.
Proof.
  xinduction (unproj22 int (@list_sub A)).
  xintros n l. intros IH. xcase.
    xreturns. calc_list~.
    #xapplys. #xapplys~. xreturns. calc_length~.
Qed.
```

```
let length l = length_aux 0 l
```

```
Lemma length_spec :
  spec Caml.length [l:_List _A] is
    >> _Int | = length l.
Proof.
  xintros l. xred. #xapplys length_aux_spec. xreturns~.
Qed.
```

Exemple : List.map

Pour commencer, on suppose que l'argument f de map implémente une fonction logique F.

```
Lemma map_spec : forall A _A B _B,
  spec map [f:_Val] [l:_List _A] = t is
    forall (F:A->B),
    (spec f [x:_A] is >> _B | = F x) ->
    t >> _List _B | = (Coq.map F l).
Proof.
  xinduction (unproj22 val (@list_sub A)).
  xintros f l. introv IH Sf. xcase.
    #xreturns*.
    xapplys*. xreds. xapplys*. #xreturns*.
Qed.
```

```
let rec map f =
 function
 | [] -> []
 | a::l ->
    let r = f a in
    r :: map f l
```

Dans le cas général, la fonction f est décrite en terme de la post-condition vérifiée sur les éléments de la liste.

Exemple : List.map plus général

Spécification de *f* avec une post-condition *Q* :

```
spec f [x:_A] = t' is
  Coq.mem x l ->
  t' >> [y:_B] | (Q x y)
```

Nouvelle spécification de List.map :

```
Lemma map_spec' : forall A _A B _B,
  spec map [f:_Val] [l:_List _A] = t is
   forall (Q:A->B->Prop),
   (spec f [x:_A] = t' is (Coq.mem x l) -> (t' >> _B | Q x)) ->
   t >> [l':_List _B] | (Coq.for_all2 Q l l')
```

Note : (**`Coq.for_all2 Q l l'`**) indique que **`l`** et **`l'`** sont des listes de même taille dont les paires d'éléments à indices correspondants sont reliés par le prédicat *Q*.

Exemple : List.fold_right

Spécification à l'aide d'un invariant *I* :

```
Lemma fold_right_spec : forall A _A B _B,
  spec fold_right [f:_Val] [l:_List _A] [a:_B] = t is
    forall (I : List A -> B -> Prop),
    I nil a ->
    (spec f [x:_A] [b:_B] = t' is
      forall l', I l' b -> t' >> _B | I (x::l')) ->
    t >> _B | (I l)
```

Si l'invariant *I* est vrai de la liste vide et de *a*, et si l'invariant est préservé par chaque application de *f*, alors l'invariant est vrai de la liste entière et du résultat.

Exemple : merge_sort

Spécification d'une fonction de comparaison :

```
Definition comparator cmp A _A (le:relation A) :=
  spec cmp [x1:_A] [x2:_A]
    is >> [n:_Int] | (n <= 0 <-> le x1 x2).
```

Spécification de merge_sort :

```
Lemma merge_sort_spec : forall A _A,
  spec merge_sort [cmp:_Val] [l:_List _A] = t is
    forall le, total_pre_order.relation le ->
    comparator cmp _A le ->
    t >> _List _A | sorts le l.
```

Avec `sorts le l l'  :=  permut l l'  /\  sorted le l'`

+démo+

Statistiques pour le module List

Nombre de lignes non vides pour la vérification du module List de Caml :

	33 fonctions	**+ merge_sort**
Code	137	65
Spécification	138	8
Vérification	258	300
Compilation	28 s	31 s

Note : ces chiffres ont été calculés sur des scripts n'utilisant pas **omega**.
Maintenant, les scripts sont un peu plus courts, plus jolis, et plus lents.

Interpréteur bytecode pour mini-ML

Compile un λ-terme en bytecode et exécute le bytecode.

```
let rec compile k = function
  | Tvar i -> (Ivar i)::k
  | Tint n -> (Iint n)::k
  | Tfun t1 -> (Iclo (compile [Iret] t1))::k
  | Tapp (t1,t2) -> compile (compile (Iapp::k) t2) t1

let rec run e s = function
  | [] -> let (Sval v)::_ = s in v
  | i::k -> match i with
     | Ivar n -> run e (Sval(List.nth e n)::s) k
     | Iint n -> run e (Sval(Mint(n))::s) k
     | Iclo c -> run e (Sval(Mclo(c,e))::s) k
     | Iapp -> let Sval(v)::Sval(Mclo(k2,e2))::s2 = s in
               run (v::e2) (Sret(k,e)::s2) k2
     | Iret -> let (Sval(v)::Sret(k2,e2)::s2) = s in
               run e2 (Sval(v)::s2) k2

let exec t =
  let k = compile [] t in
  let Mint n = run [] [] k in
  n
```

Code de Xavier Leroy

Interpréteur bytecode pour mini-ML

Spécification de la fonction d'exécution en termes de la sémantique à grands pas du terme fourni en argument.

```
Lemma exec_spec :
  spec exec [t:_Term] = r is
    forall n, (reds t (Vint n)) ->
    r >> _Int st = n.
```

Raisonnement au niveau logique :

– 71 lignes de définitions et de lemmes (indépendant du framework)

Vérification du code proprement dit :

– 8 lignes de spécifications

– 24 lignes de preuves

La preuve de terminaison de la machine s'effectue par induction sur une séquence finie de transitions.

Statistiques générales

Générateur de déf. Coq ≈ 1000 lignes de Caml

Librairie Coq réutilisable ≈ 3000 lignes de Coq

Développements formalisés ≈ 1000 lignes de Coq

	Source code	Specification	Verification
List.length	3	1	4
List.map	3	4	4
List.fold_right	4	6	4
List.split	4	2	5
List.merge_sort	65	5	298
OCaml's list library	201	143	558
ML virtual machine	43	8	95

Nb d'étapes admin. ≈ nb de noeuds syntaxiques

Programmes simples : vérification ≈ code + spec.

Conclusion

L'approche "deep embedding" :

– est assez souple vis-à-vis du langage embeddé,

– supporte un langage de specification très expressif,

– permet des preuves robustes et relativement courtes.

Quatre ingrédients clés :

– système de notations pour afficher le code embeddé,

– exploiter la correspondance avec les valeurs logiques,

– exprimer des lemmes de raisonnement en big-step,

– utiliser des tactiques pour instancier ces lemmes.

Résultat pratique :

– on peut vérifier des programmes Caml purs existants,

– en écrivant exactement les spécifications attendues,

– en les prouvant corrects en un temps raisonable.

À compléter

Petites extensions :

– support des valeurs mutuellement récursives,

– utilisation des type classes pour cacher les encoders,

– génération de lemmes pour les pattern-matching,

– support de modules en paramétrant par des records.

Extensions un peu plus ambitieuses :

– ajouter un vrai support pour prouver la divergence,

– généralisation à un langage non déterministe,

– supporter des relations générales (A $\rightarrow$ Val $\rightarrow$ Prop) dont les encodeurs ne serait qu'un cas particulier (par exemple, la relation "telle valeur Caml implémente tel ensemble fini" n'est pas bijective).

À méditer

– Parsing / printing des programmes embeddés :

⇒ idéalement, serait fait avec du code Caml dédié. Serait potentiellement utile à tous les embeddings.

– Lenteur de la procédure omega^{+bool} :

⇒ idéalement, omega saurait gérer en natif les opérateurs de comparaison booléens sur nat et int.

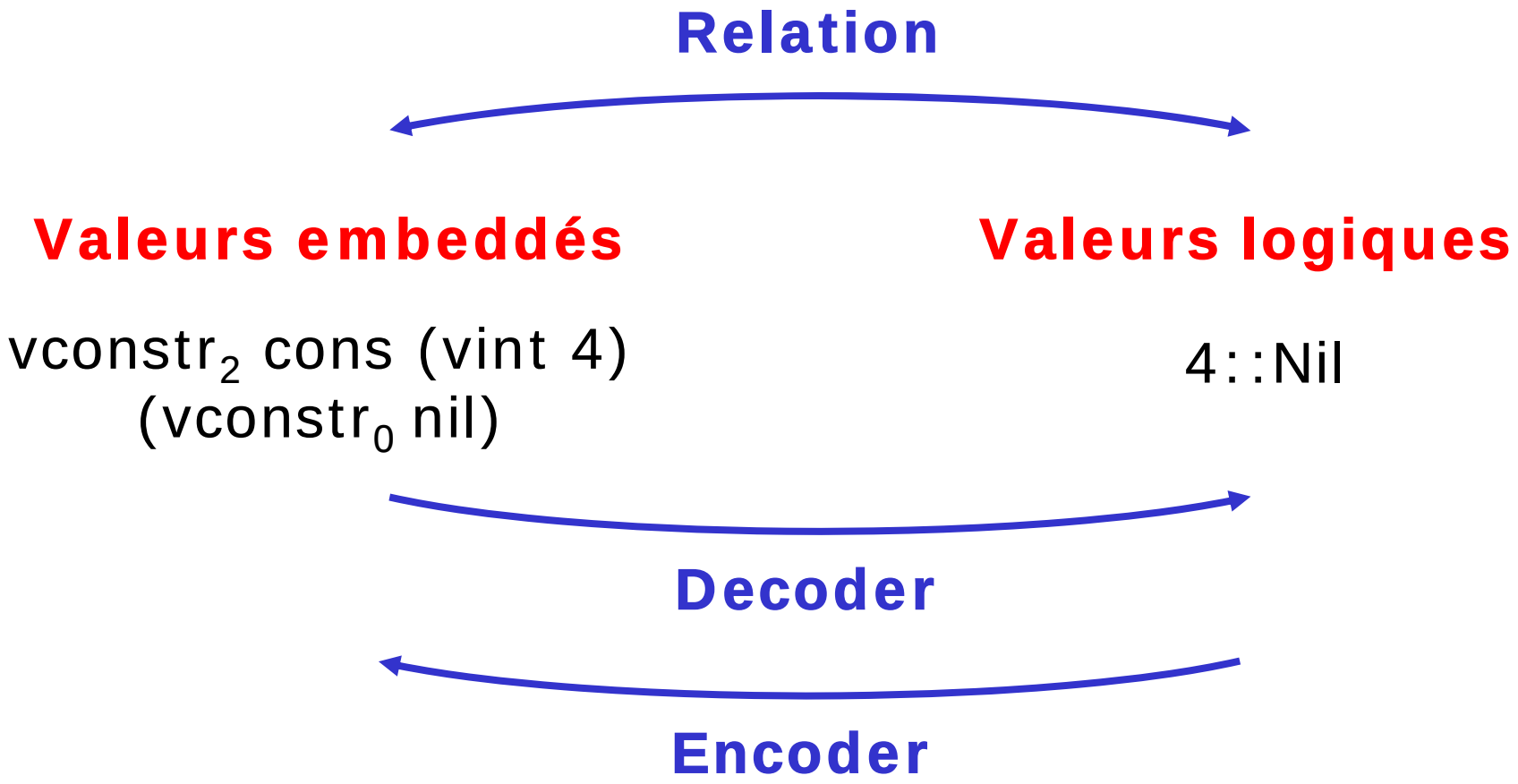
– Taille quadratique des termes de preuves :

⇒ comment éviter d'avoir des termes de preuve quadratiques en la taille des programmes sources ?

Bonus 1

Implémentations de la réflection

Implémentation de la réflection



Remarque : plusieurs valeurs logiques peuvent correspondre à une même valeur embeddée (ex: nil).

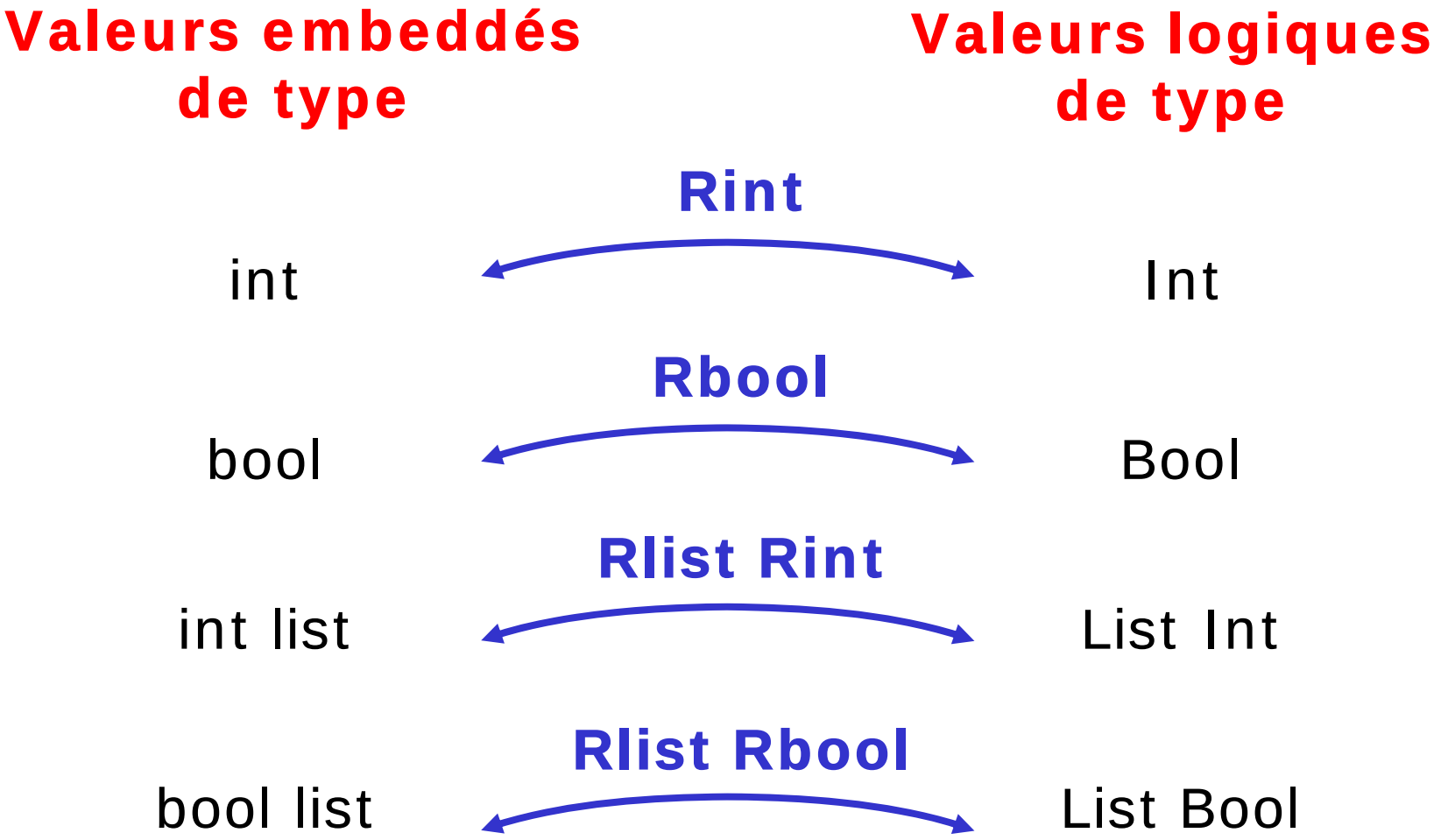
Typage d'opérateurs de réflection

Relation : $\forall$ A, A $\rightarrow$ Val $\rightarrow$ Prop
Decoder : $\forall$ A, Val $\rightarrow$ option A
Encoder : $\forall$ A, A $\rightarrow$ Val

Deux problèmes :
– il faudrait connaître tous les types de données utilisées au moment de la définition de ces opérateurs;
– l'encoder et le decoder ne peuvent être définit dans Coq, à cause de la paramétricité de la logique.

Solution : utiliser non pas une seule relation (ou fonction), mais plutôt une famille de telles relations, c'est-à-dire une relation (ou fonction) pour chaque type de donnée utilisé.

Famille d'opérateurs de réflection



Note : chaque relation est maintenant une bijection.

Choix pratique d'implémentation

Fixons un type A. Il y a trois manière de coder la bijection entre A et les valeurs embeddés associées :

Relation : $A \to Val \to Prop$
Decoder : $val \to option\ A$ $\Rightarrow$ **trop lourd à utiliser**
Encoder : $A \to Val$ $\Rightarrow$ **le plus pratique**

Conclusion : on choisit d'utiliser une famille de fonctions qui encodent des valeurs logiques.

Ainsi, on a _Int, _Bool et _List tels que :
_Int : $Int \to Val$
_Bool : $Bool \to Val$
_List _Int : $List\ Int \to Val$
_List _Bool : $List\ Bool \to Val$

Bonus 2

Anti-extraction

Principe de l'anti-extraction

Objectif : définir la fonction Coq correspondant à une fonction Caml totale prouvée correcte.

Idée de l'implémentation :

1) prendre un argument (valeur logique), et l'encoder;

2) evaluer l'application de la fonction embeddée à cette valeur encodée (ça termine, c'est prouvé!)

3) décoder le résultat de cette évaluation.

(Note: travail en cours)

Implémentation de l'anti-extraction

```
Definition anti_extract
  (f : Val)
  (A : Type) (B : inhabitedType)
  (_A : A -> Val) (__B : Val -> B)
  (Q : A -> B -> Prop)
  (H : spec f [x:_A] is >> [y:_B] | Q x y)
  : A -> B
  := fun (x : A) =>
     __B (proj1_sig (eval (_A x) (returns_ends (H x)))))
```

$\forall x.\ \exists y.\ (\text{app } f\ x) \dashrightarrow {}^{*}y \ \wedge\ Q\ x\ y$

```
Definition ends t := exists v, t -->* v.
Definition returns_of t := { v | t -->* v }.
Definition eval : forall t, ends t -> returns_of t.
Lemma returns_ends : forall t P, (t >> |P) -> ends t.
```

```
Lemma anti_extract_spec : forall ..,
  forall x, Q x (anti_extract f .. x).
```

Fonction d'évaluation : idéalement

```
Definition ends t := exists v, t -->* v.
Definition returns_of t := { v | t -->* v }.
```

```
Definition eval (t:Trm) (H:ends t) : (returns_of t) :=
  match t with
  | trm_app t1 t2 =>
     let (t1',H1) := @eval t1 _ in
     let (t2',H2) := @eval t2 _ in
     match t1' with
     | trm_abs x t3 =>
         let (t',H3) := @eval (subst x t2' t3) _ in
         returns t'
     | _ => returns (trm_app t1' t2')
     end
  | _ => returns t
  end.
```

```
Notation "'returns' v" := (@exist _ _ v _).
```

Décroissance... oui mais sur quoi ?

Intuitivement : on veut faire une récursion sur le prédicat de réduction big-step.

```
Inductive reds : Trm -> Trm -> Prop :=
  | reds_val : forall t1,
      value t1 ->
      reds t1 t1
  | reds_red : forall x t3 v2 v3 t1 t2,
      reds t1 (trm_abs x t3) ->
      reds t2 v2 ->
      reds (subst x v2 t3) v3 ->
      reds (trm_app t1 t2) v3.
```

Problème : cet inductif a deux constructeurs, donc pas de principe de récursion associé. Est-il possible de construire une relation bien fondé au-dessus de reds ?

Récursion sur prédicat ad-hoc

Solution : on va faire une récursion sur une proposition paramétrée a un unique constructeur.

```
Inductive ends' (t:Trm) : Prop :=
  | ends'_intro :
     (forall t1 t2, t = trm_app t1 t2 -> ends' t1) ->
     (forall t1 t2, t = trm_app t1 t2 -> ends' t2) ->
     (forall t1 t2 x t3 v2, t = trm_app t1 t2 ->
        reds t1 (trm_abs x t3) ->
        reds t2 v2 ->
        ends' (subst x v2 t3)) ->
     ends' t.
```

en exploitant un petit lemme qui relie `ends` et `ends'`.

```
Lemma ends_ends' : forall t,
  ends t -> ends' t.
```

Définition avec PROGRAM

```
Program Definition
  eval (t:Trm) (H:ends' t) {struct H} : (returns_of t) :=
  match t with
  | trm_app t1 t2 =>
     let (t1',H1) := @eval t1 _ in
     let (t2',H2) := @eval t2 _ in
     match t1' with
     | trm_abs x t3 =>
         let (t',H3) := @eval (subst x t2' t3) _ in
         returns t'
     | _ => returns (trm_app t1' t2')
     end
  | _ => returns t
  end.
Next Obligation. [...]  Defined. (* 15 obligations prouvées *)
```

Si près du but...

Un peu trop dur pour PROGRAM

```
Error:
Recursive definition of Eval is ill-formed.
In environment [...]
Recursive call to Eval has principal argument equal to
"Eval_obligation_1 H T Heq_t"
instead of a subterm of H.
```

Problème : PROGRAM a l'air de générer un terme d'une forme telle que le noyau n'arrive pas à reconnaître la décroissance structurelle de la récursion. Simple problème de unfold ou problème plus profond ?

Solution : programmer la fonction avec des tactiques (ou bien en donnant le terme de preuve directement).

Conclusion sur l'anti-extraction

En théorie, on devrait pouvoir définir des fonctions Coq à partir de leur équivalent Caml.

En pratique, il reste à :

– étendre la fonction "eval" au langage embeddé,
– fixer PROGRAM pour qu'on puisse le faire facilement,
– générer des définitions de decoders (type "Val $\rightarrow$ A"),
– combiner le tout, et vérifier que ça marche bien.
– trouve une utilité à l'anti-extraction...

Merci !

Pour plus de détails, lire le papier :

Interactive Verification of Call-by-Value Functional Programs

disponible sur http://arthur.chargueraud.org

Deep embedding d'une logique

Considérons le deep embedding d'une logique **L**:

– **p** dénote la représentation d'une proposition de **L**,

– **|- p** un jugement de validité des propositions de **L**,

– **[| · |]** une fonction d'interprétation dans Prop.

Il est naturel d'énoncer en Coq le théorème suivant :

$$\forall\ \mathbf{p},\ \ (\mathbf{|\text{-}\ p})\ \ \rightarrow\ \ \mathbf{[|\ p\ |]}$$

La specification de l'anti-extraction est l'analogue de ce théorème, pour le deep embedding d'un language de programmation : si une fonction vérifie une certaine spécification, son évaluation également.