Vérification Interactive de Programmes Fonctionnels

Arthur Charguéraud

INRIA – Projet Gallium

Séminaire Gallium INRIA Rocquencourt, 11/05/2009

Introduction

Objectif : spécifier et prouver formellement des propriétés de correction totale pour des programmes purement fonctionnels en évaluation call-by-value.

Moyen : utiliser un *deep embedding*, ce qui consiste à décrire la syntaxe et la sémantique du langage source dans la logique d'un assistant de preuve.

Résultat : un deep embedding de pur-Caml dans Coq, avec une infrastructure permettant de manipuler et de raisonner sur des programmes Caml purs.

Exposé :
⇒ la vérification de programmes Caml pur par cette approche est raisonable,
⇒ montrer les ingrédients d'un deep embedding.

Partie 1 : Description des ingrédients théorique

Shéma général

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

Les traductions correspondant aux flèches $\rightarrow$ *sont implémentées par un outil externe codé en Caml.*

Quatre ingrédients

1) Description de la syntaxe et de la sémantique

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

2) Définition de prédicats de comportement

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

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 se comporte comme l'identitée sur les valeurs closes

– quelques petites duplications entre termes et valeurs

\+ exceptions

\+ récursion

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'`

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

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

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* :

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

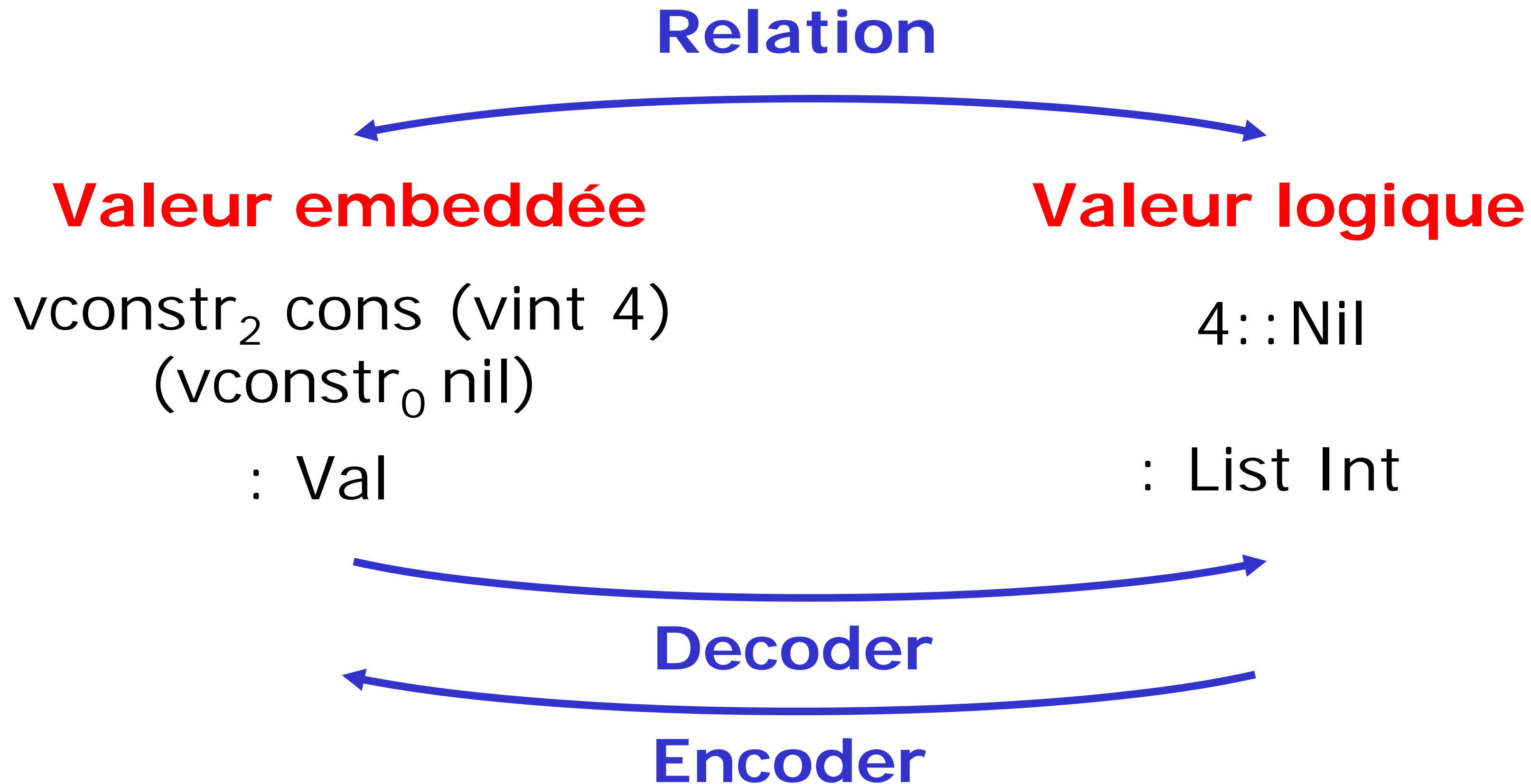
Plus généralement, la spécification d'une fonction *f* est une proposition *K* dépendant de l'argument *v* et du terme "*App f v*".

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

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

Réflection des valeurs

Description de la valeur ML "4::nil", de type "int list".



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

Définition des types reflétés

Types ML :

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

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

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.
```

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`".

À retenir sur la réflection

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 $A \rightarrow Val$.

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

On parlera plus tard des fonctions de première classe.

Spécification des termes

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

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

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

En pratique :

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

Spécification des fonctions

Le prédicat "l'application de *f* à l'encodage "_*A x*" d'une valeur *x* est un terme *t* tel que, si la pré-condition "*P x*" est vrai, alors t retourne une valeur y satisfiant la post-condition "*Q x y*".

```
spec f [x:_A] = t is
  P x ->
  t >> [y:_B] st Q x y
```

Exemple :

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

Règles de raisonnement

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

Un exemple :

$$\frac{t_1 \;\rhd\; _A \mid P \qquad \forall X.\,(P\,X) \;\Rightarrow\; ([x \to \mathsf{val}\,(_A\;X)]\,t_2) \;\rhd\; B}{(\mathsf{let}\;x = t_1\;\mathsf{in}\;t_2) \;\rhd\; B}$$

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.

Partie 2 : Pratique

Calculs

– **xintros introduit les arguments d'une fonction**

```
spec f [x:_A] = t is  P x  ->  t >> _B st Q x
```

est transformé par la tactique "xintros a" en

```
P a  ->  (f ' _A a) >> _B st Q a
```

– **xred permet de beta-réduire un terme**

```
C[ t1 ] >> _A st P
```

se réduit, lorsque `t1 --> t2`, vers

```
C[ t2 ] >> _A st P
```

– **xreturns prouve le comportement d'une valeur**

```
_A V >> _A st P
```

est transformé par "xreturns" en

```
P V
```

Décodage

– **xdecode retrouve la valeur logique associée**

```
C [ (_A a) ':: (_List _A q) ]
```

est réécrit par la tactique "xdecode" en

```
C [ _List _A (a::q) ]
```

Parfois il faut spécifier des types non inférables, ex :

```
C [ nil# ]
```

est réécrit par la tactique "xdecode _A" en

```
C [ _List _A (@Nil A) ]
```

Applications

– **xapply as X raisonne sur une application**

Considérons une spécification

```
spec f [x:_A] = t is
   P x  ->  t >> [y:_B] st Q x y
```

et raisonnons sur une application

```
C[ f ' (_A v) ] >> _D st Q
```

alors l'application de "xapply as y" va produire

```
C[ _B y ] >> _D st Q
```

pour un y frais (y de type B), sous l'hypothèse

```
Q v y
```

et va générer un sous-but pour la pré-condition

```
P v
```

Applications avec substitution

– **xapplys substitue le résultat d'une application**

Considérons une spécification produisant une égalité

```
spec f [x:_A] = t is
   P x  ->  t >> _B st = R x
```

et raisonnons sur une application

```
C[ f ' (_A v) ] >> _D st Q
```

alors l'application de "xapplys" va produire

```
C[ _B (R v) ] >> _D st Q
```

et va générer un sous-but

```
P v
```

Note : "xapplys" équivaut à "xapply as X; subst X"

Pattern matching

– **xred réduit aussi les patterns**

⇒ xred n'est à utiliser que si l'argument du pattern est suffisamment précis pour pouvoir déterminer si le pattern s'applique ou non

– **xpat réduit jusqu'au premier cas qui match**

⇒ c'est ainsi une version spécialisée de xreds

– **xcase applique un destruct puis appelle xpat**

⇒ raccourci très pratique pour les analyses de cas

Note : il manque pour l'instant la génération d'un lemme décrivant finement les disjonctions de cas.

Fonctions n-aires

Pour une fonction currifiée à deux arguments

```
spec f [x:_A] [y:_B] = t is t >> _C st P
```

est, par définition, équivalent à :

```
spec f [x:_A] = r is r >> [g:_Val] st
   (spec g [y:_B] = t is t >> _C st P)
```

Cette spécification indique que l'application partielle de la fonction f à un seul argument termine toujours.

– **xintros x1 .. xN introduit plusieurs arguments**

Contextes

– **xin se concentre sur le sous-terme à réduire**

Soit un terme de comportement arbitraire B0

```
C[t] >- B0
```

l'application de "xin B" donne deux sous-buts

```
1)   t >- B
2)   ∀t, (t >- B) -> (C[t] >- B0)
```

– **xout exploite le comportement du sous-terme**

Il simplifie par exemple l'instance de (2) suivante

```
∀t, (t >> _A st P) -> (C[t] >- B0)
```

en

```
∀X, (P X) -> (C[_A X] >- B0)
```

Contextes

– **xinout combine xin et xout**

Soit un terme de comportement arbitraire B0

```
   C[t] >- B0
```

l'application de "xin (>> _A st P)" donne

```
1)   t >> _A st P
2)   ∀x, (P x) -> (C[_A x] >- B0)
```

– **xin tout seul est utile pour l'inférence**

```
   C[t] >- B0
```

l'application de "xin" donne deux sous-buts

```
1)   t >- ?B
2)   ∀t, (t >- ?B) -> (C[t] >- B0)
```

permet au comportement ?B d'être inféré dans (1)

Affaiblissement

– **xweakens permet d'affaiblir une spécification**

$\Rightarrow$ variante : xweakens S permet de spécifier explicitement la spécification que l'on veut exploiter

– **xweaken permet d'affaiblir un comportement**

$\Rightarrow$ même chose, mais au niveau d'un terme

$\Rightarrow$ variante : xweaken H permet de spécifier explicitement l'hypothèse que l'on veut exploiter

– **ximpl prouve une implication de comportement**

$\Rightarrow$ prouve les buts "`B1 ==> B2`", qui sont définis comme "`∀t, t >- B1 => t >- B2`"

Induction

– xinduction pour prouver une fonction récursive

Considérons une spécification

```
spec f [x:_A] = t is t >> _B st Q x
```

L'application de "xinduction R" donne

```
spec f [x:_A] = t is H -> t >> _B st Q x
```

où l'hypothèse d'induciton H est :

```
spec f [x':_A] = t' is R x' x -> t' >> _B st Q x'
```

⇒ il faut fournir un ordre bien fondé ou une mesure, en syntaxe currifiée ou décurrifée, ou alors une preuve de bonne fondaison directement.

⇒ afin que la bonne fondaison soit prouvée facilement, on utilisera des combinateurs adaptés.

Partie 3 : Conclusions

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.

Limitations

– Lourdeur du parsing/printing du code embeddé

⇒ ce serait mieux fait avec du code Caml dédié, mais Coq n'offre pas (encore) cette possibilité facilement.

– Taille quadratique des termes de preuves :

⇒ en gros, il ne faut pas espérer vérifier une définition top-level de plus de 100 lignes de code sans la couper en plusieurs définitions plus petites.

– Lenteur des procédures de décisions (omega)

⇒ la lenteur de omega devient pénible dès que l'on fait beaucoup d'arithmétique; cela devrait s'améliorer avec le temps.

Extensions

– support des valeurs **mutuellement récursives**,

– utiliser des type classes pour **cacher les encoders**,

– choix automatique des **noms des variables** locales,

– génération de lemmes pour les **pattern-matching**,

– support des **modules**, encodable avec des records,

– supporter des **relations de réflection** plus générales, de type "A $\rightarrow$ Val $\rightarrow$ Prop", dont les encodeurs ne serait qu'un cas particulier (ex: la relation "telle valeur Caml implémente tel ensemble fini", qui n'est pas bijective).

Merci !

Pour plus de détails, voir le papier :

Interactive Verification of Call-by-Value Functional Programs

disponible sur http://arthur.chargueraud.org

Bonus : Un exemple

Interpréteur bytecode pour mini-ML

Termes sources :

```
type term =
   | Tvar of int
   | Tint of int
   | Tfun of term
   | Tapp of term * term

type value =
   | Vint of int
   | Vclo of term * value list

type env = value list
```

Termes compilés :

```
type instr =
   | Ivar of int
   | Iint of int
   | Iclo of instr list
   | Iapp
   | Iret

type mcode = instr list

type mvalue =
   | Mint of int
   | Mclo of mcode * mvalue list

type menv = mvalue list

type slot =
   | Sval of mvalue
   | Sret of mcode * menv

type mstack = slot list
```

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.