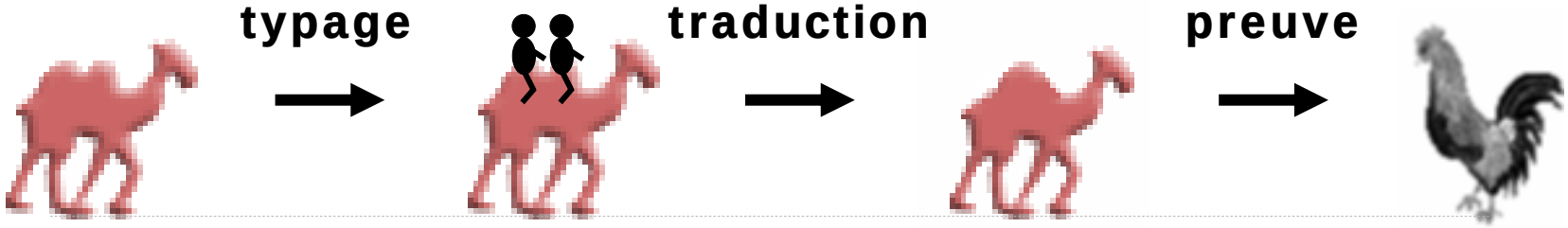
Vérification Interactive de Programmes Fonctionnels

Arthur Charguéraud

INRIA – Projet Gallium

Réunion "Langage, Type et Preuves" Évry, 28/11/2008

Projet de thèse



CAML impératif	+ Types	CAML fonctionnel	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 purs

1) Logique de Hoare + VCG

– 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 (Morrisset *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 ML (ici)

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, + quelques tactiques)

Limitations potentielles de cette approche:

– risque d'avoir des obligations de preuves illisibles
– risque d'avoir des scripts de preuves très longues
– 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

Source CAML		**Développement Coq**
		déf. syntaxe et sémantique
définitions de types	→	déf. de type correspondante
définitions top-level	→	déf. de termes embeddés
		spécification des termes
		vérification de (code,spec)

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

Overview des 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

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

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 vrai)

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_builtin : builtin -> 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_wildcard : 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

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	list int
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 réfléchit pas les types récursifs. Dans un premier temps, on ignore les fonctions de 1ère classe.

Traduction 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.

Traduction des valeurs

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

```
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.map de Caml (1/2)

Le code source (VerifList.ml) :

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

Le code embeddé (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.

Exemple : List.map de Caml (2/2)

Spécification (VerifList.v) **:**

```
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).
```

Vérification (VerifList.v) **:**

```
Proof.
  xinduction (unproj22 val (@list_sub A)).
  xintros f l. introv IH Sf. xcase.
    xreturns*.
    xapplys*. xreds. xapplys*. xreturns*.
Qed.
```

Rappel du code :

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

+Démo

Spécification de 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.

Spécification d'un tri fusion

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 st sorts le l.
```

Avec

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

Statistiques pour les listes Caml

(nombre de lignes non vides)

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

Interpréteur bytecode pour mini-ML

Programme source (compile des λ-termes vers du bytecode puis exécute le bytecode obtenu) :

– 24 lignes de déclarations de types (9 types en tout)

– 19 ligne de code bien denses (code de Xavier Leroy)

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 très expressive et relativement souple,

– n'est pas forcément trop lourde en pratique.

À compléter :

– support des valeurs mutuellement récursives,

– utilisation des type-classes pour cacher les encoders,

– optimisation de l'efficacité des tactiques,

– génération de lemmes pour les pattern-matching.

À l'étude :

– définir des fonctions Coq correspondantes à des fonctions totales écrites en Caml et prouvés correctes.

Merci !

Référence: *Interactive Verification of Call-by-Value Functional Programs*
Arthur Charguéraud, *Draft*, http://arthur.chargueraud.org