Master Parisien de Recherche en Informatique
cours 2-7-2 : assistants de preuve

Preuve de programmes fonctionnels (5)

Jean-Christophe Filliâtre

2005–2006

Preuve de programmes impératifs

on a vu que Coq est un outil de choix pour la preuve de programmes purement applicatifs

qu'en est-il de la preuve de programmes **impératifs** ?

Plan

1. Les monades

utilisées en Haskell pour simuler les traits impératifs

une monade est la donnée

- d'un opérateur de type $\texttt{mon} : \texttt{Set} \rightarrow \texttt{Set}$
  $\texttt{mon}\ A$ représente les calculs de type A
- d'une opération $\texttt{return} : A \rightarrow \texttt{mon}\ A$
  $\texttt{return}\ v$ désigne la valeur v vue comme un calcul
- d'une opération $\texttt{bind} : \texttt{mon}\ A \rightarrow (A \rightarrow \texttt{mon}\ B) \rightarrow \texttt{mon}\ B$
  bind séquence deux calculs

syntaxe Haskell : `do p <- e1 ; e2` dénote $\texttt{bind}\ e_1\ \lambda x.\, e_2$

Monades : axiomes

les opérateurs doivent vérifier **les trois lois** suivantes :

$$\texttt{bind}\ (\texttt{return}\ x)\ f = f\ x$$

$$\texttt{bind}\ m\ \texttt{return} = m$$

$$\texttt{bind}\ m\ (\lambda x.\, \texttt{bind}\ (f\ x)\ g) = \texttt{bind}\ (\texttt{bind}\ m\ f)\ g$$

Exemple : la monade d'erreur

- $\texttt{mon}\ A = \texttt{Error} \mid \texttt{Value of}\ A$ (type option)
- $\texttt{return}\ x = \texttt{Value}\ x$
- $\texttt{bind}\ m\ f = \texttt{match}\ m\ \texttt{with}$
  $\mid \texttt{Error} \Rightarrow \texttt{Error}$
  $\mid \texttt{Value}\ v \Rightarrow f\ v$

Exemple : la monade d'état

- $\texttt{mon}\ A = S \rightarrow S \times A$
- $\texttt{return}\ x = \texttt{fun}\ s \rightarrow (s,\ x)$
- $\texttt{bind}\ m\ f = \texttt{fun}\ s \rightarrow \texttt{let}\ (s', v) = m\ s\ \texttt{in}\ f\ v\ s'$

Combinaison : la monade d'état et d'erreur

en général, on ne sait pas combiner deux monades pour en faire une troisième

dans la cas des monades d'erreur et d'état, on peut

on a même deux solutions :

- $\texttt{mon}\ A = S \rightarrow \texttt{Error} \mid \texttt{Value of}\ (S \times A)$

ou

- $\texttt{mon}\ A = S \rightarrow S \times (\texttt{Error} \mid \texttt{Value of}\ A)$

les opérateurs se définissent aisément dans les deux cas

La monade d'état et d'erreur dans Coq

type de la monade (pour un certain type d'état state)

```
Set Implicit Arguments.

Inductive res (A: Set) : Set :=
  | Error: res A
  | OK: A → state → res A.

Definition mon (A: Set) : Set := state → res A.
```


La monade d'état et d'erreur dans Coq

opérations de la monade

```
Definition ret (A: Set) (x: A) : mon A :=
  fun (s: state) ⇒ OK x s.

Definition error (A: Set) : mon A :=
  fun (s: state) ⇒ Error A.

Definition bind (A B: Set) (f: mon A) (g: A→mon B): mon B
  := fun (s: state) ⇒
       match f s with
       | Error ⇒ Error B
       | OK a s' ⇒ g a s'
       end.

Notation "'do' X <- A ; B" := (bind A (fun X ⇒ B))
  (at level 200, X ident, A at level 100, B at level 200).
```


Application : un compteur

soit un compteur utilisant une référence (*gensym*) i.e.

```
let fresh = let r = ref 0 in fun () → incr r; !r
```

```
Definition state := nat.
...
Definition fresh : mon nat := fun s ⇒ OK s (S s).
```

```
Fixpoint list_n (n:nat) { struct n } : mon (list nat) :=
  match n with
  | O ⇒ ret nil
  | S n' ⇒ do k <- fresh; do l <- list_n n'; ret (k :: l)
  end.
```


Preuve de programmes

le programme utilisant les monades peut être prouvé dans Coq par les techniques déjà vues

2. Logique de Hoare

petit langage impératif très simplifié

$$\begin{array}{rcl} e & ::= & n \mid x \mid e \ op \ e \\ op & ::= & + \mid - \mid \times \mid / \mid = \mid \neq \mid < \mid > \mid \leq \mid \geq \mid \mathit{and} \mid \mathit{or} \\ i & ::= & \texttt{skip} \mid x := e \mid i;\ i \mid \texttt{if}\ e\ \texttt{then}\ i\ \texttt{else}\ i \mid \texttt{while}\ e\ \texttt{do}\ i\ \texttt{done} \end{array}$$

exemple : soit ISQRT le programme

```
count := 0; sum := 1;
while sum <= n do
  count := count + 1; sum := sum + 2 * count + 1
done
```

affirmation : à la fin du programme, count contient $\lfloor \sqrt{n} \rfloor$

Sémantique opérationnelle

état = fonction E associant à chaque variable sa valeur

$$\begin{array}{rcl} E(n) & = & n \\ E(x) & = & E(x) \\ E(e_1 \ op \ e_2) & = & E(e_1) \ op \ E(e_2) \end{array}$$

$$\begin{array}{lcl} E & \xrightarrow[x:=e]{} & E\{x = E(e)\} \\ E_1 & \xrightarrow[i_1;i_2]{} & E_3 \text{ si } E_1 \xrightarrow[i_1]{} E_2 \text{ et } E_2 \xrightarrow[i_2]{} E_3 \\ E_1 & \xrightarrow[\mathit{if}\ e\ \mathit{then}\ i_1\ \mathit{else}\ i_2]{} & E_2 \text{ si } E_1(e) = \mathit{true} \text{ et } E_1 \xrightarrow[i_1]{} E_2 \\ E_1 & \xrightarrow[\mathit{if}\ e\ \mathit{then}\ i_1\ \mathit{else}\ i_2]{} & E_2 \text{ si } E_1(e) = \mathit{false} \text{ et } E_1 \xrightarrow[i_2]{} E_2 \\ E_1 & \xrightarrow[\mathit{while}\ e\ \mathit{do}\ i]{} & E_3 \text{ si } E_1(e) = \mathit{true},\ E_1 \xrightarrow[i]{} E_2 \text{ et } E_2 \xrightarrow[\mathit{while}\ e\ \mathit{do}\ i]{} E_3 \\ E & \xrightarrow[\mathit{while}\ e\ \mathit{do}\ i]{} & E \text{ si } E(e) = \mathit{false} \end{array}$$

Triplets de Hoare

un **triplet de Hoare** est un triplet noté

$$\{P\}\ i\ \{Q\}$$

où P et Q sont des formules du premier ordre partageant leurs expressions (et donc leurs variables) avec les programmes

validité : le triplet $\{P\}\ i\ \{Q\}$ est valide si pour tous états E_1 et E_2 tels que $E_1(P)$ est vrai et $E_1 \xrightarrow[i]{} E_2$, alors $E_2(Q)$ est vrai

exemple : on souhaite montrer la validité du triplet

$$\{n \geq 0\}\ \mathit{ISQRT}\ \{\mathit{count} \times \mathit{count} \leq n \wedge n < (\mathit{count}+1) \times (\mathit{count}+1)\}$$

Règles de la logique de Hoare

logique de Hoare = ensemble de règles de déduction sur les triplets

$$\frac{}{\{P\}\ \texttt{skip}\ \{P\}} \qquad \frac{\{P \wedge e = \mathit{true}\}\ i_1\ \{Q\} \qquad \{P \wedge e = \mathit{false}\}\ i_2\ \{Q\}}{\{P\}\ \texttt{if}\ e\ \texttt{then}\ i_1\ \texttt{else}\ i_2\ \{Q\}}$$

$$\frac{}{\{P[x \leftarrow e]\}\ x := e\ \{P\}} \qquad \frac{\{I \wedge e = \mathit{true}\}\ i\ \{I\}}{\{I\}\ \texttt{while}\ e\ \texttt{do}\ i\ \texttt{done}\ \{I \wedge e = \mathit{false}\}}$$

$$\frac{\{P\}\ i_1\ \{Q\} \qquad \{Q\}\ i_2\ \{R\}}{\{P\}\ i_1;\ i_2\ \{R\}} \qquad \frac{\{P'\}\ i\ \{Q'\} \qquad P \Rightarrow P' \qquad Q' \Rightarrow Q}{\{P\}\ i\ \{Q\}}$$

Correction et complétude

correction : cet ensemble de règles est correct i.e. tout triplet dérivable est correct

difficulté : il faut deviner les bonnes annotations intermédiaires (pour ISQRT par exemple, il faut exhiber un **invariant de boucle**)

point de vue théorique : a-t-on la **complétude** de la logique de Hoare i.e. peut-on prouver tous les triplets valides ?

Complétude et calcul de plus faibles préconditions

pour i et Q donnés, l'ensemble des P tel que $\{P\}\ i\ \{Q\}$ est valide possède, s'il est non vide, un élément **minimal** P_0, i.e. tel que si $\{P\}\ i\ \{Q\}$ est valide alors P implique P_0

notons $\mathbf{WP}(i, Q)$ cet élément minimal ; il se calcule par récurrence sur i

$$\begin{array}{rcl} WP(x := e, Q) & = & Q\{x \leftarrow e\} \\ WP(i_1; i_2, Q) & = & WP(i_1, WP(i_2, Q)) \\ WP(\mathit{if}\ e\ \mathit{then}\ i_1\ \mathit{else}\ i_2, Q) & = & (e = \mathit{true} \Rightarrow WP(i_1, Q)) \wedge \\ & & (e = \mathit{false} \Rightarrow WP(i_2, Q)) \\ WP(\mathit{while}\ e\ \mathit{do}\ i, Q) & = & \text{pas de formule simple !} \end{array}$$

$WP(i, Q)$ s'appelle la **plus faible précondition** de i et Q

Complétude

le calcul des WP donne les annotations intermédiaires

complétude : l'ensemble des règles de la logique de Hoare est *relativement* complet i.e. tout triplet $\{P\}\ i\ \{Q\}$ valide est dérivable (en particulier, on peut trouver des invariants pour les boucles `while` de i)

hypothèse : la logique dans laquelle on exprime les annotations est suffisamment expressive (en particulier pour exprimer les invariants de boucle)

Exemple : racine carrée

```
let isqrt (n:int) =
  { n >= 0 }
  let count = ref 0 in
  let sum = ref 1 in
  while !sum <= n do
    { invariant count >= 0 and n >= count*count
            and sum = (count+1)*(count+1)
      variant n - sum }
    count := !count + 1;
    sum := !sum + 2 * !count + 1
  done;
  !count
  { result >= 0 and
    result * result <= n < (result+1)*(result+1) }
```


3. Mise en œuvre dans Coq

comment mettre en œuvre la logique de Hoare dans Coq ?

deux solutions

- **deep embedding**
  on formalise la logique de Hoare i.e. on internalise les notions de termes, instructions, règles de Hoare, etc.
  ⇒ utile pour montrer la correction ou la complétude (méta-théorie) mais lourd sur des exemples concrets
- **shallow embedding**
  un programme est directement interprété par sa sémantique et les règles de la logique de Hoare correspondent à des tactiques
  ⇒ utilisable sur des exemples concrets mais la méta-théorie n'est plus possible

En pratique

en pratique, on apprécie les quelques extensions suivantes :

- avoir des effets de bord dans les expressions
- désigner dans une annotation la valeur d'une variable à un instant antérieur (par exemple au début du programme)
- traiter l'appel de sous-programmes (modularité)
- prouver la terminaison des programmes
- supporter des constructions comme `break` ou `continue` (et plus généralement des exceptions)
- **avoir des structures de données complexes : tableaux, enregistrements, pointeurs, objets, etc.**

4. Structures de données complexes

la règle d'affectation de la logique de Hoare

$$\frac{}{\{P[x \leftarrow e]\}\ x := e\ \{P\}}$$

contient implicitement le fait que

- les expressions comme e sont partagées entre les programmes et la logique
- il n'y a **pas d'alias** entre les variables du programmes

pour traiter des structures de données complexes avec la logique de Hoare traditionnelle, il va falloir les **modéliser**

Exemple des tableaux : le drapeau hollandais

le drapeau hollandais de Dijkstra : trier un tableau dont les éléments ne prennent que trois valeurs différentes (bleu, blanc, rouge)

0	b	i	r	n
BLUE	WHITE	. . . à faire. . .	RED	

Quelques lignes de C

```
typedef enum { BLUE, WHITE, RED } color;

void swap(int t[], int i, int j) {
  color c = t[i]; t[i] = t[j]; t[j] = c;
}

void flag(int t[], int n) {
  int b = 0, i = 0, r = n;
  while (i < r) {
    switch (t[i]) {
    case BLUE: swap(t, b++, i++); break;
    case WHITE: i++; break;
    case RED: swap(t, --r, i); break;
    }
  }
}
```


Modélisation

on ne va pas vérifier le code C, mais une **modélisation**

on modélise

- les couleurs par un **type abstrait**
- les tableaux en utilisant des références contenant des **tableaux applicatifs**

Un type abstrait pour les couleurs

```
type color

logic blue : color
logic white : color
logic red : color

predicate is_color(c:color) = c=blue or c=white or c=red

parameter eq_color  :
  c1:color → c2:color →
    {} bool { if result then c1=c2 else c1≠c2 }
```


Des tableaux applicatifs

```
type color_array

logic acc : color_array, int → color
logic upd : color_array, int, color → color_array

axiom acc_upd_eq :
  ∀a:color_array. ∀i:int. ∀c:color.
    acc(upd(a,i,c),i) = c

axiom acc_upd_neq :
  ∀a:color_array. ∀i,j:int. ∀c:color.
    i ≠ j → acc(upd(a,j,c),i) = acc(a,i)
```


Accès aux tableaux dans les bornes

```
logic length : color_array → int

axiom length_update :
  ∀a:color_array. ∀i:int. ∀c:color.
    length(upd(a,i,c)) = length(a)

parameter get :
  t:color_array ref → i:int →
    { 0<=i<length(t) } color reads t { result=acc(t,i) }

parameter set :
  t:color_array ref → i:int → c:color →
    { 0<=i<length(t) } unit writes t { t=upd(t@,i,c) }
```


La fonction swap

```
let swap (t:color_array ref) (i:int) (j:int) =
  { 0 <= i < length(t) and 0 <= j < length(t) }
  let u = get t i in
  set t i (get t j);
  set t j u
  { t = upd(upd(t@,i,acc(t@,j)), j, acc(t@,i)) }
```

5 obligations de preuve

- 3 automatiquement prouvées par WHY
- 2 laissées à l'utilisateur (et automatiquement prouvées par Simplify)

Le code de la fonction de tri

```
let dutch_flag (t:color_array ref) (n:int) =
  let b = ref 0 in
  let i = ref 0 in
  let r = ref n in
  while !i < !r do
     if eq_color (get t !i) blue then begin
       swap t !b !i;
       b := !b + 1;
       i := !i + 1
     end else if eq_color (get t !i) white then
       i := !i + 1
     else begin
       r := !r - 1;
       swap t !r !i
     end
  done
```


Spécification de la fonction de tri

```
predicate monochrome(t:color_array,i:int,j:int,c:color) =
  ∀k:int. i<=k<j → acc(t,k)=c

let dutch_flag (t:color_array ref) (n:int) =
  { 0 <= n and length(t) = n and
    ∀k:int. 0 <= k < n → is_color(acc(t,k)) }
  :
  { ∃b:int. ∃r:int.
      monochrome(t,0,b,blue) and
      monochrome(t,b,r,white) and
      monochrome(t,r,n,red) }
```

Invariant de boucle

```
:
while !i < !r do
  { invariant 0 <= b <= i and i <= r <= n and
              monochrome(t,0,b,blue) and
              monochrome(t,b,i,white) and
              monochrome(t,r,n,red) and
              length(t) = n and
              ∀k:int. 0 <= k < n → is_color(acc(t,k))
    variant r - i }
  :
done
```


Obligations de preuve

11 obligations de preuve

- l'invariant est vrai initialement
- l'invariant est préservé et le variant diminue (3 cas)
- la précondition de swap (2 fois)
- l'accès au tableau dans les bornes (2 fois)
- la postcondition est satisfaite à l'issue de la fonction

Toutes automatiquement prouvées par Simplify !

Remarque : pour être complet, il faut montrer que le multi-ensemble des éléments n'a pas changé

Preuve de programmes C

modèle simple : la mémoire vue comme un grand tableau
mais les obligations de preuve sont rapidement **trop complexes** (car tous les pointeurs sont potentiellement identiques)

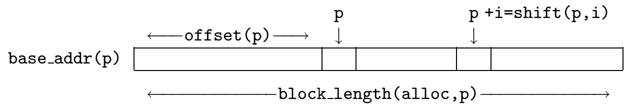
modèle plus fin : on exploite une information statique pour **séparer** la mémoire en différentes composantes

Le modèle de Burstall-Bornat

la mémoire est séparée selon les **champs de structures** (pour le C, ou selon les attributs pour des programmes Java, etc.)

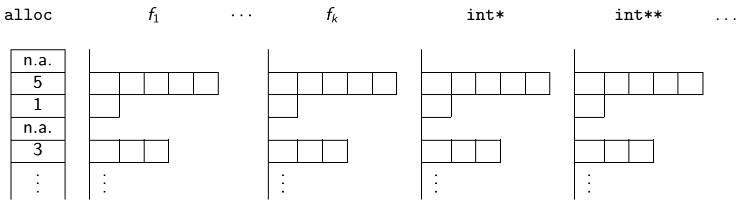
on étend l'idée pour traiter l'arithmétique de pointeurs

- un bloc mémoire est ainsi modélisé



- chaque champ de structure est une table associant des blocs à des pointeurs

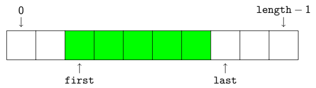
Structure générale de la mémoire



on fait ensuite de la logique de Hoare traditionnelle sur des variables contenant ces différentes tables

une modification de l'une d'entre elles n'a donc pas d'impact sur le contenu des autres

Exemple : file de caractères dans un tableau circulaire




```
struct queue {
  char* contents;
  int length;
  int first, last;
  unsigned int empty, full :1;
} q;
/*@ invariant q_invariant :
  @   \valid_range(q.contents, 0, q.length-1) &&
  @   0 <= q.first < q.length &&
  @   0 <= q.last < q.length
  @*/
```


Spécifier les fonctions

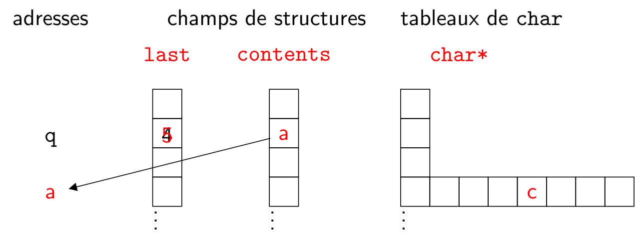
```
/*@ requires !q.full
  @ assigns  q.empty, q.full, q.last, q.contents[q.last]
  @ ensures  !q.empty && q.contents[\old(q.last)] == c
  @*/
void push(char c);

/*@ requires !q.empty
  @ assigns q.empty, q.full, q.first
  @ ensures !q.full && \result == q.contents[\old(q.first)]
  @*/
char pop();
```


Corps de la fonction push

```
/*@ requires !q.full
  @ assigns  q.empty, q.full, q.last, q.contents[q.last]
  @ ensures  !q.empty && q.contents[\old(q.last)] == c
  @*/
void push(char c) {
  q.contents[q.last++] = c;     // insère 'c' dans la file
  if (q.last == q.length)
        q.last = 0;             // boucle si besoin
  q.empty = 0;                  // la file n'est pas vide
  q.full = (q.first == q.last); // la file est pleine si
                                // 'last' atteint 'first'
}
```

Illustration du modèle Burstall-Bornat




```
q.contents[q.last++] = c
    q.last <- 4 + 1
    *(q.contents+4) <- c
```


Modélisation du code C

l'instruction C

```
q.contents[q.last++] = c
```

est modélisée ainsi :

```
assert { valid(!alloc,q) };   // obligation de preuve
let tmp1 = acc(!last,q) in    // tmp1 <- q.last
last := upd(!last,q,tmp1+1);  // q.last <- tmp1+1
let tmp2 = shift(acc(!contents,q),tmp1) in
                              // tmp2 <- q.contents + tmp1
assert { valid(!alloc,tmp2) };// obligation de preuve
intP := upd(!intP,tmp2,c)     // *tmp2 <- c
```


Certification de la fonction push

on obtient 3 obligations de preuve exprimant que

- le code de push ne contient pas de déréférencement illégal de pointeurs (par exemple dans `q.contents[q.last++]`)
- la postcondition et la clause `assigns` de push sont établies
- l'invariant `q_invariant` est préservé par push

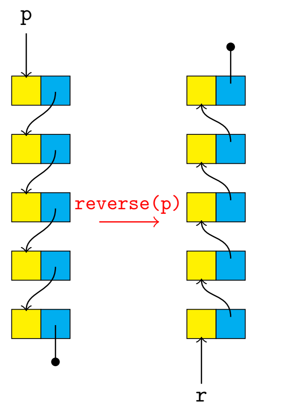
Preuve de ces obligations

- avec Simplify (100%)
- avec Coq (100%), très facile (6 lignes de tactiques)

Autre exemple : retournement en place d'une liste chaînée

```
typedef struct struct_list {
  int hd;
  struct struct_list *tl;
} *list;

list reverse(list p) {
  list r = NULL;
  while (p != NULL) {
    list q = p ;
    p = p->tl;
    q->tl = r;
    r = q;
  }
  return r;
}
```



Déclaration d'éléments logiques

- des prédicats et des fonctions sont déclarés

```
// liste (logique) finie de pointeurs
//@ logic plist nil()
//@ logic plist cons(list p, plist l)
// concaténation et retournement
//@ logic plist app(plist l1, plist l2)
//@ logic plist rev(plist pl)
```

- des axiomes sont énoncés ; par exemple

```
//@ axiom app_nil : \forall plist l; app(nil(),l) == l
```


Spécification de la fonction reverse

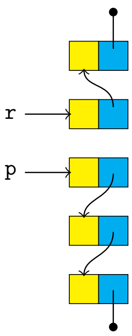
```
/* llist(p,l) spécifie que l est la liste des pointeurs
   depuis p jusqu'à NULL en suivant le champ tl */
//@ predicate llist(list p, plist l) reads p->tl

// is_list(p) spécifie que p est une liste chaînée finie
//@ predicate is_list(list p) { \exists plist l ; llist(p,l) }

/*@ requires is_list(p)
  @ ensures \forall plist l;
  @    \old(llist(p, l)) => llist(\result, rev(l))  */
list reverse(list p);
```


Invariant de boucle pour reverse

```
list reverse(list p) {
  list r = NULL;
 /*@ invariant
    \exists plist lp; \exists plist lr;
      llist(p, lp) && llist(r, lr) &&
      disjoint(lp, lr) &&
      \forall plist l; \old(llist(p, l)) =>
        app(rev(lp), lr) == rev(l)
   @ variant length(p) for length_order */
  while (p != NULL) {
    list q = p;
    p = p->tl; q->tl = r; r = q;
  }
  return r;
}
```



Certification de reverse

- 7 obligations de preuve
- avec Simplify : 71%
- avec Coq : 100%, avec 661 lignes de tactiques