

# Assistants de Preuve

## Preuve de Programmes impératifs

Christine Paulin-Mohring & Jean-Christophe Filliâtre

Université Paris-Sud 11, INRIA

15/01/08

# Preuve de programmes impératifs

Coq est un outil de choix pour la preuve de programmes purement applicatifs

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

# Plan

Monades en Coq

Logique de Hoare

- Principes

- Mise en œuvre dans Coq

Structures de données complexes

- Manipulation de tableaux

- Modèles avancés de la mémoire

# Les monades

Utilisées en Haskell pour simuler les traits impératifs

Une monade est la donnée

- ▶ d'un opérateur de type  $M : \text{Type} \rightarrow \text{Type}$   
 $M A$  représente les calculs de type  $A$
- ▶ d'une opération  $\text{return} : A \rightarrow M A$   
 (aussi notée  $\text{unit}$ ) où  $\text{return } v$  désigne la valeur  $v$  vue comme un calcul
- ▶ d'une opération  $\text{bind} : M A \rightarrow (A \rightarrow M B) \rightarrow M B$   
 $\text{bind}$  séquence deux calculs

Syntaxe Haskell :  $\text{do } p \leftarrow e_1 ; e_2$  dénote  $\text{bind } e_1 \lambda p. e_2$

# Monades : axiomes

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

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

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

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

# 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 :

$$\blacktriangleright M A = S \rightarrow \text{Error} \mid \text{Value of } (S \times A)$$

ou

$$\blacktriangleright M A = S \rightarrow S \times (\text{Error} \mid \text{Value of } A)$$

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: Type) : Type :=  
  | Error: res A  
  | OK: A → state → res A.
```

```
Definition M (A: Type) : Type := state → res A.
```

# La monade d'état et d'erreur dans Coq

## Opérations de la monade

```
Definition ret (A: Type) (x: A) : M A :=
  fun (s: state) => OK x s.
```

```
Definition error (A: Type) : M A :=
  fun (s: state) => Error A.
```

```
Definition bind (A B: Type) (f: M A) (g: A→M B): M 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 : M nat := fun s ⇒ OK s (S s).

**Fixpoint** list\_n (n:nat) { struct n } : M (list nat) :=  
 match n with  
 | 0 ⇒ ret nil  
 | S n' ⇒ do k <- fresh; do l <- list\_n n'; ret (k :: l)  
 end.

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

# Plan

Monades en Coq

Logique de Hoare

Principes

Mise en œuvre dans Coq

Structures de données complexes

Manipulation de tableaux

Modèles avancés de la mémoire

# Logique de Hoare

## petit langage impératif très simplifié

$$e ::= n \mid x \mid e \text{ op } e$$

$$\text{op} ::= + \mid - \mid \times \mid / \mid = \mid \neq \mid < \mid > \mid \leq \mid \geq \mid \text{and} \mid \text{or}$$

$$i ::= \text{skip} \mid x := e \mid i \mid \text{if } e \text{ then } i \text{ else } i \mid \text{while } e \text{ do } i \text{ done}$$

**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

$$E(n) = n$$

$$E(x) = E(x)$$

$$E(e_1 \text{ op } e_2) = E(e_1) \text{ op } E(e_2)$$

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

# 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

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

# Règles de la logique de Hoare

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

$$\frac{}{\{P\} \text{ skip } \{P\}} \quad \frac{\{P \wedge e = \text{true}\} i_1 \{Q\} \quad \{P \wedge e = \text{false}\} i_2 \{Q\}}{\{P\} \text{ if } e \text{ then } i_1 \text{ else } i_2 \{Q\}}$$

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

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

# 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  **$WP(i, Q)$**  cet élément minimal ; il se calcule par récurrence sur  $i$

$$WP(x := e, Q) = Q\{x \leftarrow e\}$$

$$WP(i_1; i_2, Q) = WP(i_1, WP(i_2, Q))$$

$$WP(\text{if } e \text{ then } i_1 \text{ else } i_2, Q) = (e = \text{true} \Rightarrow WP(i_1, Q)) \wedge \\ (e = \text{false} \Rightarrow WP(i_2, Q))$$

$$WP(\text{while } e \text{ do } i, Q) = \text{pas de formule simple !}$$

$WP(i, Q)$  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)

# 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.**

# Outil Why

`http://why.lri.fr`

- ▶ Un langage fonctionnel avec des traits impératifs (références, exceptions)
- ▶ Pas d'alias entre références
- ▶ Analyse d'effets : variables lues et écrites, exceptions levées
- ▶ Description modulaire des programmes et des spécifications
- ▶ Calcul de plus faibles péconditions
- ▶ Génération d'obligations de preuve pour des démonstrateurs interactifs (Coq, Isabelle, PVS) ou automatiques (Ergo, Simplify)

# 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) }
```

# Plan

Monades en Coq

Logique de Hoare

Principes

Mise en œuvre dans Coq

Structures de données complexes

Manipulation de tableaux

Modèles avancés de la mémoire

# Structures de données complexes

la règle d'affectation de la logique de Hoare

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

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)) }
```

Obligations de preuve automatiquement prouvées par Ergo

# 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
  end
done
```

# Spécification de la fonction de tri

```
predicate monochrome(t:color_array,i:int,j:int,c:color) =  
   $\forall k:\text{int}. i \leq k < j \rightarrow \text{acc}(t,k)=c$ 
```

```
let dutch_flag (t:color_array ref) (n:int) =  
  { 0 <= n and length(t) = n and  
     $\forall k:\text{int}. 0 \leq k < n \rightarrow \text{is\_color}(\text{acc}(t,k))$  }  
:  
{  $\exists b:\text{int}. \exists r:\text{int}.$   
  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  
     $\forall k:\text{int}. 0 \leq k < n \rightarrow \text{is\_color}(\text{acc}(t,k))$   
    variant r - i }  
  ⋮  
done
```

# Obligations de preuve

- ▶ l'invariant est vrai initialement
- ▶ l'invariant est préservé et le variant diminue
- ▶ la précondition de `swap`
- ▶ l'accès au tableau dans les bornes
- ▶ la postcondition est satisfaite à l'issue de la fonction

**Toutes automatiquement prouvées par Ergo !**

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

# Preuve de programmes Java

- ▶ Chaque objet est une référence, deux variables différentes peuvent représenter le même emplacement.
- ▶ 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 d'un programme Java est séparée selon les **attributs**.

- ▶ Si  $a$  et  $b$  sont deux attributs distincts alors les deux expressions  $e_1.a$  et  $e_2.b$  ne peuvent jamais représenter le même emplacement mémoire.
- ▶ modèle global : on modélise  $e_1.a$  par  $M[e_1 + \text{offset}(a)]$  avec  $M$  une mémoire globale
- ▶ modèle à la Burstall-Bornat : on introduit une mémoire  $A$  pour l'attribut  $a$  et on modélise  $e_1.a$  par  $A[e_1]$
- ▶ de même les tableaux d'entiers ou les tableaux d'objets peuvent être séparés
- ▶ d'autres séparations peuvent être déduites d'analyses statiques
- ▶ une table `heap` garde trace des adresses allouées, des types dynamiques et tailles de tableau

# Spécification

## Idée de JML (Java Modeling Language)

```
class Purse {  
    //@ public invariant balance >= 0;  
    int balance;  
  
    /*@ public normal_behavior  
       @   requires s >= 0;  
       @   modifiable balance;  
       @   ensures balance == \old(balance)+s;  
    @*/  
    public void credit(int s) {  
        balance += s;  
    }  
}
```

# Comportement exceptionnel

```
/*@ public behavior
@ requires s >= 0;
@ modifiable balance;
@ ensures s<=\old(balance) && balance==\old(balance)-s;
@ signals (NoCreditException)
@     s > balance && balance == \old(balance);
@*/
```

```
public void withdraw(int s) throws NoCreditException
{
    if (balance >= s)
        { balance -= s; }
    else { throw new NoCreditException(); }
}
```

# Traduction vers Why

- ▶ Modélisation du programme et de la spécification
  - ▶ Ensemble de déclarations et d'axiomes validés par un modèle Coq
  - ▶ Ensemble de procédures élémentaires spécifiées pour les constructions de base du langage
- ▶ Les conditions **assigs** correspondent à des contraintes de non modification de portions de la mémoire (ne concerne que les variables Why modifiées)
- ▶ Les exceptions servent à modéliser les structures de contrôle comme **break**, **return** ...

# Modélisation Coq

La théorie Coq sous-jacente :

```
Inductive value : Type :=  
  | Null : value | Ref : adrObject → value.
```

```
Inductive tag : Type :=  
  | Obj : classId → tag  
  | Arr : Z → Arrkind → tag.
```

Definition alloc\_table := pmap.t adrObject tag.

Definition memory A := map.t value A.

Definition mod\_loc := value → Prop.

```
Definition modifiable A (h:store) (m m':memory A)  
  (unchanged:mod_loc) : Prop :=  
  ∀v:value, alive h v → unchanged v → acc m v = acc m' v.
```



# Modélisation Why

## Modélisation logique

```
type value
type tag
type classId
type javaType
type alloc_table
type 'a memory

logic acc : 'a memory, value → 'a

logic Obj : classId → tag
logic typeof : alloc_table, value, javaType → prop

logic Null : → value
```

# Modélisation Why

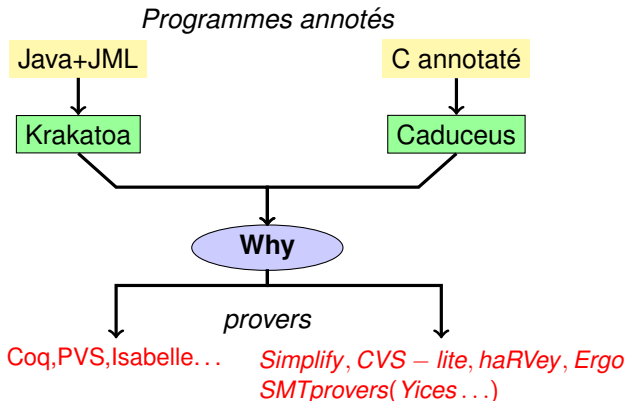
## Modélisation calculatoire

```
exception Exception of value
exception Return_value of value
```

```
parameter alloc : alloc_table ref
parameter intA : int memarray ref
parameter objA : value memarray ref
parameter new_ref : alloc_table → value
parameter allocate
  : alloc_table → value → tag → alloc_table
```

```
parameter alloc_new_obj : c : classId →
  { } value writes alloc
  { result ≠ Null
    and fresh(alloc@, result)
    and typeof(alloc, result, ClassType(c))
    and store_extends(alloc@, alloc)
  }
```

# Architecture de Why



- ▶ Une grande souplesse dans la conception du modèle
- ▶ Adaptation pour modéliser la mémoire des cartes à puces et les arrachages

# Modélisation de code C

## Outil Caduceus, plateforme Frama-C

- ▶ Les champs de structure jouent le rôle des attributs
- ▶ Moins de contraintes de type dans le langage (arithmétique de pointeur, casts ...)
- ▶ Nécessite une modélisation de plus bas niveau ou une analyse statique plus complexe.
- ▶ Langage de spécification à la JML en cours d'élaboration (ACSL)
- ▶ Constructions de programme comme **goto** qui demandent d'associer des invariants à des points de programmes arbitraires
- ▶ Quelques exemples réalisés :
  - ▶ Schorr-Waite (manipulation de pointeurs pour les gc)
  - ▶ Absence de menace dans du code utilisé chez Dassault (70 kloc, 116k obligations, 96% prouvées automatiquement)