Preuves de programmes Java en Coq

Evelyne Contejean
Jean Duprat
Jean-Christophe Filliâtre
Claude Marché
Christine Paulin-Mohring
Xavier Urbain

LRI (CNRS) & INRIA Futurs & ENS Lyon

Plan

1. But recherché — Méthodologie suivie
2. Cas des méthodes statiques
3. Cas des objets simples
4. Cas des tableaux simples
5. Comment ça marche ?
6. Conclusion et perspectives

But recherché

Prouver qu'une méthode Java donnée vérifie sa spécification

- Langage d'entrée: Java, annoté en JML
  - on veut supporter les programmes "simples" (JavaCard, . . .)
  - mais on rejettera une partie de Java (voir plus loin)
- Prouver quoi ?
  - (invariant de classe et pré-condition) implique (invariant de classe et post-condition)
  - invariant de boucle, et variants (correction *totale*)

Exemple: racine carrée entière

```
public final class IntMisc {
    /*@ public behavior
      @   requires
      @     x >= 0;
      @   ensures
      @     \result * \result <= x &&
      @     x < (Math.abs(\result) + 1) * (Math.abs(\result) + 1);
      @*/
    public static int sqrt(int x)
```

Racine carrée entière (suite)

```
public static int sqrt(int x) {
    int count = 0, sum = 1;
     /*@ loop_invariant
       @   count >= 0 &&
       @   x >= count*count &&
       @   sum == (count+1)*(count+1);
       @ decreases x - sum;
       @*/
    while (sum <= x) {
        count++;
        sum = sum + 2*count + 1;
    }
    return count;
}
```

Méthodologie adoptée

Traduction de Java/JML vers *Why* : outil indépendant, produisant des obligations de preuve, pour PVS ou Coq [J.-C. FILLIÂTRE, `http://why.lri.fr`]

- Langage d'entrée de Why : fonctionnel pure typé à la ML
  - avec traits impératifs strictement limités: références, boucle while
  - mais pas de type de données utilisateur : type abstraits
  - avec annotations à la Hoare, +/- en logique du premier ordre
- Philosophie de Why

  Program variable = Logical variable (pas de modèle mémoire)
- Notre "politique de simplicité":

  Philosophie de Why + minimiser le surcoût de la traduction

2. Cas des méthodes statiques

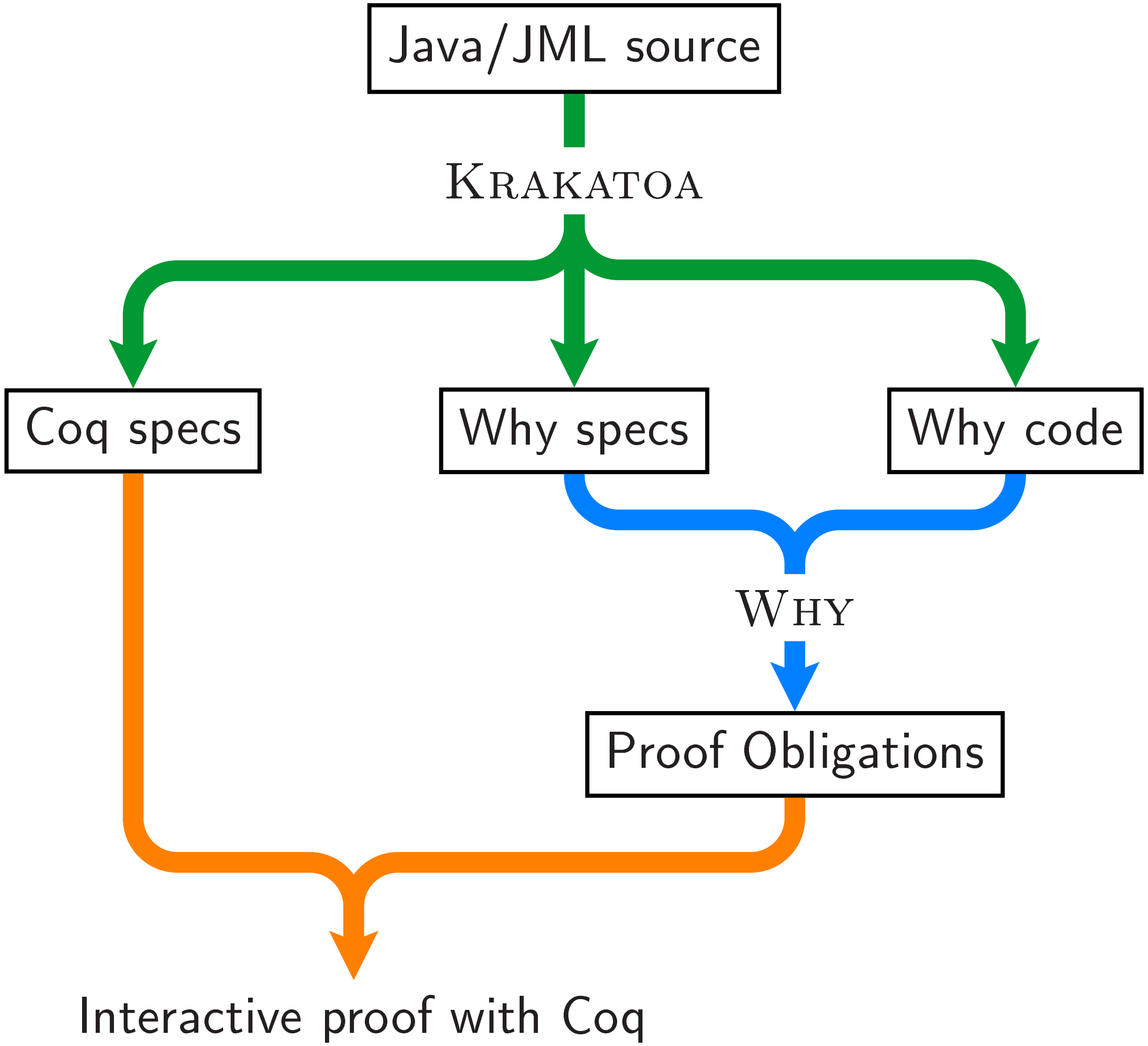
Par exemple: racine carrée entière

Difficulté:

- annotations JML = expression booléenne JAVA
- en particulier: présence d'appel à des méthodes *pures* comme `Math.abs`

Solution adoptée : génération de fichiers de spécifications de ces méthodes pures, en Coq et en Why.

Certification en trois étapes



Racine carrée entière (suite)

- Spécifications Coq :

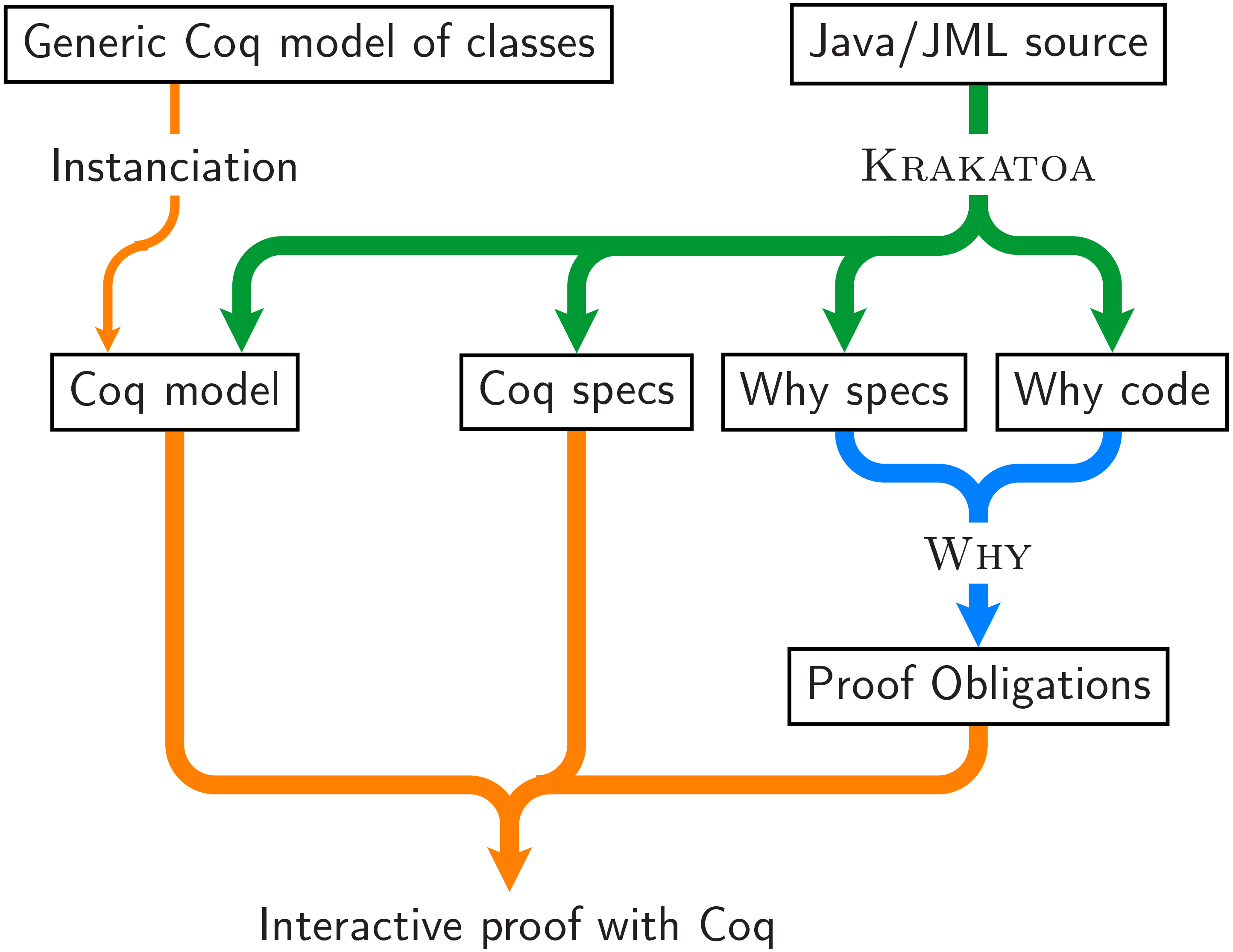
```
Parameter Math_abs : Z -> Z.
Axiom Math_abs_axiom :
 (n:Z)(let result = (Math_abs n) in
  `result >= 0`/\
   (`result = n`\/`result = (-n)`)
```

```
/*@ public behavior
  @ ensures \result>= 0 &&
  @ (\result== n || \result== -n) ;
  @*/
public /*@ pure @*/ static int abs(int n)
```

- Cinq obligations de preuves engendrées

- Affirmation : notre politique promise de simplicité est tenue

3. Cas des objets simples



Le modèle générique Coq

- Type énuméré `classId` des noms de classe, et relation `extends`
- Type énuméré `fieldId` des noms de champ
- Type `value` des valeurs Java : primitives (booléens, entiers), `null` et références (objets et tableaux)
- `instanceof: value -> classId -> bool`, et opération de cast
- fonction d'accès générique aux champs:
  `access: value -> fieldId -> value`
- fonction d'accès générique de mise à jour de champs:
  `update: value -> fieldId -> value -> value`
  ⚠ mise à jour fonctionnelle : pas de modification en place

Le modèle générique (suite)

Lemmes généraux:

- accès à un objet mis à jour :

```
(this:value)(f:fieldId)(v:value)
  (access (update this f v) f)=v

(this:value)(f1,f2:fieldId)(v:value)
  ~(f1=f2)->
    (access (update this f1 v) f2)=(access this f2)
```

- préservation de la class lors d'une mise à jour :

```
(this:value)(c:classId)(f:fieldId)(v:value)
  (instanceof this c)->
    (instanceof (update this f v) c)
```

Exemple d'un porte-monnaie

```
/*@ private invariant 0 <= decPart && decPart < PRECISION ;
  @*/
public class Amount {
    public static final short PRECISION = 100;
    short intPart;
    short decPart;
    /*@ public behavior
      @   requires true;
      @   ensures
      @     this.amount_value() ==
      @         \old(this.amount_value()) + a.amount_value();
      @*/
    public void add(Amount a)
    /*@ private behavior
      @   requires true ;
      @   ensures \result == PRECISION * intPart + decPart ;
      @*/
    private /*@ pure @*/ int amount_value();
```

Exemple (suite)

```
/*@ public behavior
  @   requires true;
  @   ensures
  @     this.amount_value() ==
  @         \old(this.amount_value()) + a.amount_value();
  @*/
public void add(Amount a) {
    short d = (short)(decPart + a.decPart);
    short i = (short)(intPart + a.intPart);
    if (d >= PRECISION) {
        d = d - PRECISION;
        i++;
    }
    intPart = i;
    decPart = d;
}
```

Exemple (suite)

- Trois obligations de preuves engendrées (dont une triviale...)

- Affirmation : la tactique `Krakatoa` cache le surcoût de la traduction, notre politique promise de simplicité est tenue

4. Cas des tableaux

Inclus dans le modèle: longueur, accès et mise à jour (fonctionnelle) génériques.

Lemmes analogues pour accès/mise à jour, et préservation de la longueur.

Exemple: tri par insertion.

5. Comment ça marche ?

- Traduction du code Java vers du code Why
- Difficulté principale : les variables *modifiables* doivent être explicitement déclarées comme *références* en Why
- On ne traite pas la *terminaison abrupte* (sauf cas particuliers)

Règles de Traduction pour les instructions

	Java block b	Why translation $\widetilde{b}$
Local variable decl.	τ *var* $=$ e ; b	`let` *var* = `ref` $\widetilde{e}$ `in` $\widetilde{b}$
	τ *var*; b	`let` *var* = `ref` def_τ `in` $\widetilde{b}$
Assignment	*var* = e ; b	*var*`:=`$\widetilde{e}$; $\widetilde{b}$
Variable access	*var*	`!`*var*
Field access	e`.f`	(`access_f` $\widetilde{e}$)
Field update	*var*`.f` = e	*var* `:=` (`update_f` `!`*var* $\widetilde{e}$)
Method call	*var*`.m`$(e_1, \ldots, e_n)$	`m`(*var*, $\widetilde{e_1}, \ldots, \widetilde{e_n}$)

Traduction des spécifications JML

- Difficulté principale: les méthodes *pures* sont utilisées dans les annotations JML
- Pour chaque méthode : on génère un `external` Why
- Pour les méthodes pures: on génère aussi
  - un `logic` Why
  - un `Parameter` Coq
  - un `Axiom` Coq correspondant à la spécification JML

Correction de la traduction

update fonctionnel → traduction incorrecte en général

Exemple: (contexte: `Amount t[10];`)

Java	Why
`t[2].add(new Amount(1,80));`	`t := !t with 2 <-` `add t[2] new_Amount(1,80);`
Pas de différence observable	

`Amount a = t[2];`
`a.add(new Amount(1,80));`
`t[2]` modifié
	`let a = ref t[2] in`
	`a := add !a new_Amount(1,80);`
	`t[2]` non modifié

Retour sur la philosophie de Why

Exemple:

```
let incr (x : int ref) (y : int ref) =
  begin
    x := !x + 1;
    y := !y + 1;
  end
  { x = x@ + 1 and y = y@ + 1 }
```

alors `incr(x,x)` est interdit

Why rejette les alias de variable

Test de linéarité

- But: définir une vérification statique de *linéarité* inspiré par [Shankar, Efficiently Executing PVS, 1999])
- Principe: dans les mises à jour de champs

Java	Why
`v.f := e`	`v := (update !v f e)`
`t[n] := e`	`t := (array_update !t n e)`

les anciennes valeurs de `v` ou `t` ne doivent plus être accessible.

6. Conclusion et perspectives

Pour l'instant, notre politique de simplicité est tenue

Travaux en cours:

- Preuve de correction de la traduction des programmes linéaires
- Support des exceptions
- Modèle mémoire pour les programmes non-linéaires
- Développement d'exemples significatifs, et comparaison avec autres approches (LOOP, JIVE).

Travaux futurs

- Supporter plus de JML: `model`, etc.
- Supporter les méthodes récursives
- Certifier le non-dépassement de capacité des entiers.
- Programmes concurrents (threads)
- ...