Why : un outil de vérification générique

Jean-Christophe Filliâtre
LRI
CNRS – Université Paris Sud

CEA, 12 février 2004

Plan

1. Why : un outil générique
2. Applications : vérification de programmes C et Java
3. Technique sous-jacente

I

L'outil Why

Concept

$$\text{source} + \text{spécification} \longrightarrow \boxed{\text{VCG}} \longrightarrow \text{obligations de preuve}$$

Généricité

– en entrée : plusieurs langages

– en sortie : plusieurs systèmes de preuve

Un langage intermédiaire

Langage dédié à la preuve de programmes

Langages existants traduits vers ce langage

Gain : on factorise une partie du travail du VCG

(calcul de plus faibles préconditions, d'effets, etc.)

Quel langage intermédiaire

- types purement fonctionnels + références sur ces types
- boucle while
- if-then-else
- séquence
- expressions = instructions
- fonctions récursives
- exceptions

Quelles spécifications

- annotations à la Hoare
  - pré/post-conditions
  - assertions dans le code
  - invariants/variants de boucle
- effets explicités : références lues et modifiées
- déclarations d'éléments logiques :

  types, fonctions, prédicats, axiomes

Annotations écrites dans un syntaxe de prédicats du premier ordre

Exemple

```
let search1 =
  {}
  try
    let i = ref 0 in begin
    while !i < (array_length t) do
      { invariant 0 <= i and forall k:int. 0 <= k < i -> t[k] <> 0
        variant array_length(t) - i }
      if t[!i] = 0 then raise (Found !i);
      i := !i + 1
    done;
    raise Not_found : int
    end
  with Found x ->
    x
  end
  { t[result] = 0
  | Not_found => forall k:int. 0 <= k < array_length(t) -> t[k] <> 0 }
```

Un exemple : le `break`

La construction break peut s'exprimer à l'aide d'exceptions

```
while (b1) {
  /* invariant I */
  if (b2) break;
  s
}
/* Q */
```

```
try
  while b1 do
    { invariant I }
    if b2 then raise Break;
    s
  done
with Break ->
  void
end
{ Q }
```

Obligations

$\vdash I_0$	entrée dans la boucle
$I, b_1, \neg b_2 \vdash \mathsf{wp}(s, I)$	préservation de l'invariant
$I, b_1, b_2 \vdash Q$	sortie par `break`
$I, \neg b_1 \vdash Q$	fin de boucle

L'utilisation d'exceptions est invisible

WP pour les exceptions

$\mathsf{wp}(\mathsf{e}, \mathsf{Q}, \mathsf{R})$ // cas d'une seule exception E

$$\mathsf{wp}(\mathsf{raise}\ \mathsf{E}, \mathsf{Q}, \mathsf{R}) = \mathsf{R}$$

$$\mathsf{wp}(\mathsf{raise}\ (\mathsf{E}\ \mathsf{e}), \mathsf{Q}, \mathsf{R}) = \mathsf{wp}(\mathsf{e}, \mathsf{R}, \mathsf{R})$$

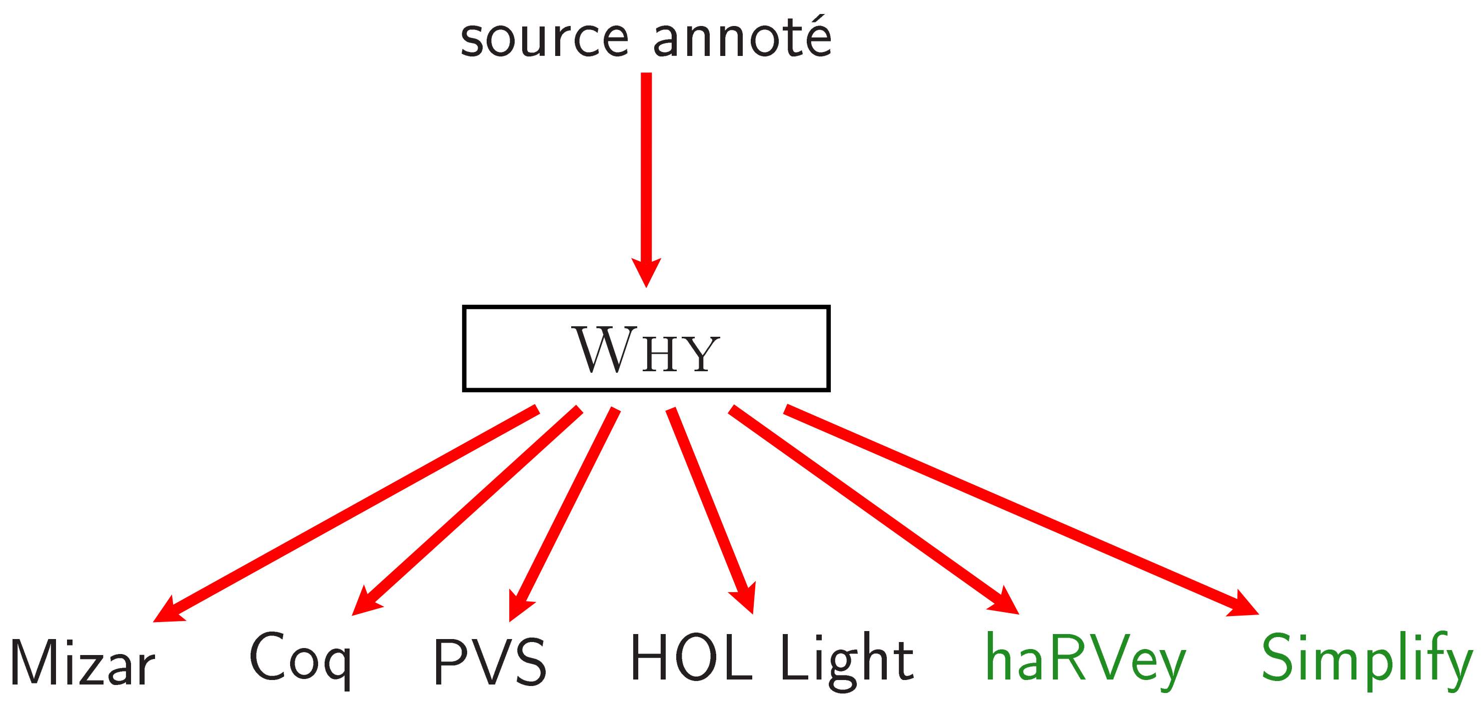
$$\mathsf{wp}(\mathsf{try}\ \mathsf{e}_1\ \mathsf{with}\ \mathsf{E} \rightarrow \mathsf{e}_2, \mathsf{Q}, \mathsf{R}) = \mathsf{wp}(\mathsf{e}_1, \mathsf{Q}, \mathsf{wp}(\mathsf{e}_2, \mathsf{Q}, \mathsf{R}))$$

Autre exemple

Boucle while (e) s où e contient des effets de bord

```
try
  while true do
    if not e then raise Exit;
    s
  done
with Exit ->
  void
end
```

Génération des obligations de preuve



Pourquoi est-ce simple

L'expression des obligations de preuve ne requiert qu'une logique minimale ($\forall \Rightarrow \wedge$)

L'adaptation à un système de preuve nécessite uniquement un pretty-printer pour une logique du premier ordre (moins de 300 lignes de code à chaque fois)

Une partie de la difficulté est en fait cachée dans le modèle

II

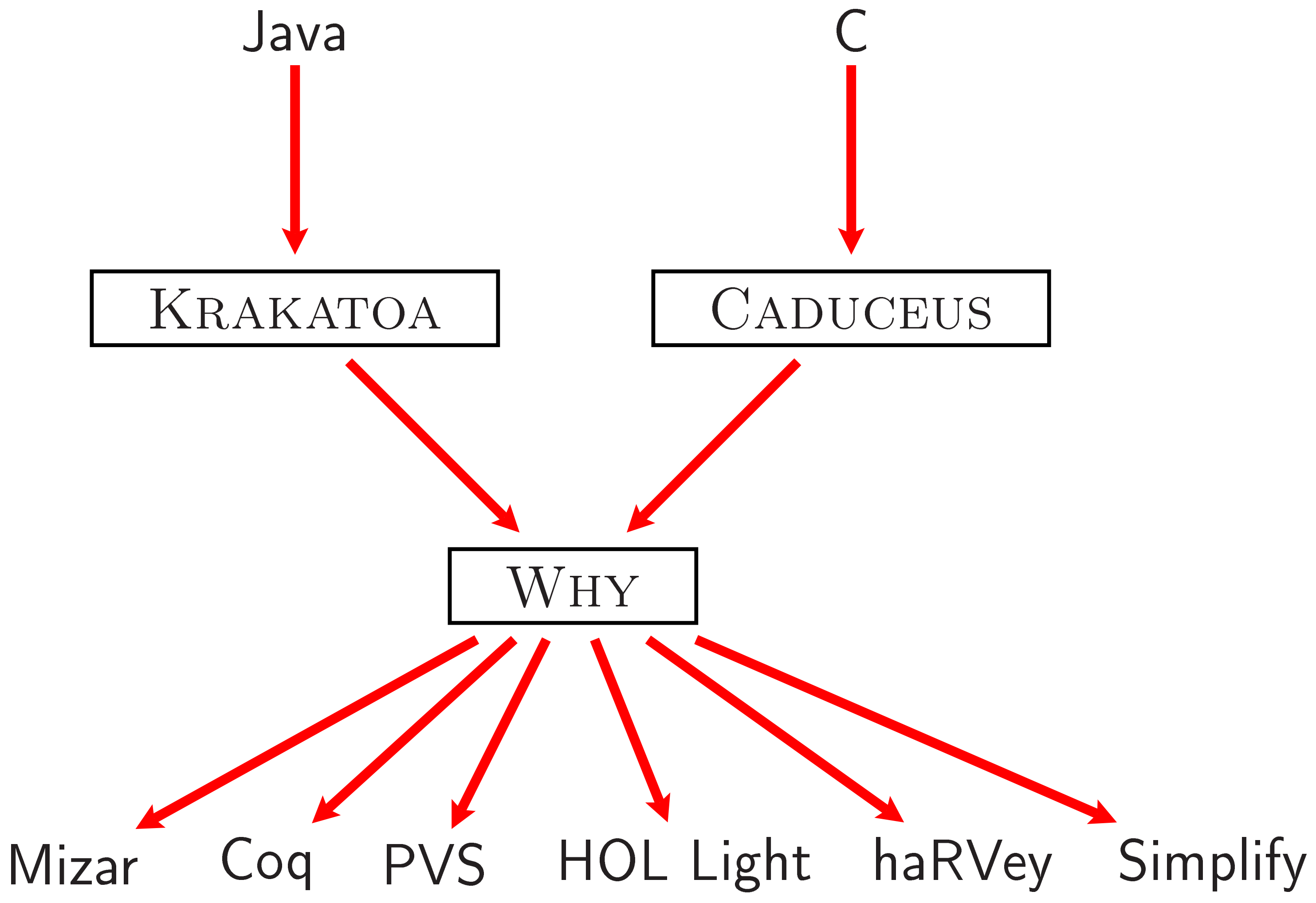
Applications

Programmes C et Java

Deux outils développés au LRI

- Krakatoa : programmes Java annotés avec JML
- Caduceus : programmes C

Programmes C et Java



Krakatoa : programmes Java

Prouver qu'une méthode Java donnée vérifie sa spécification

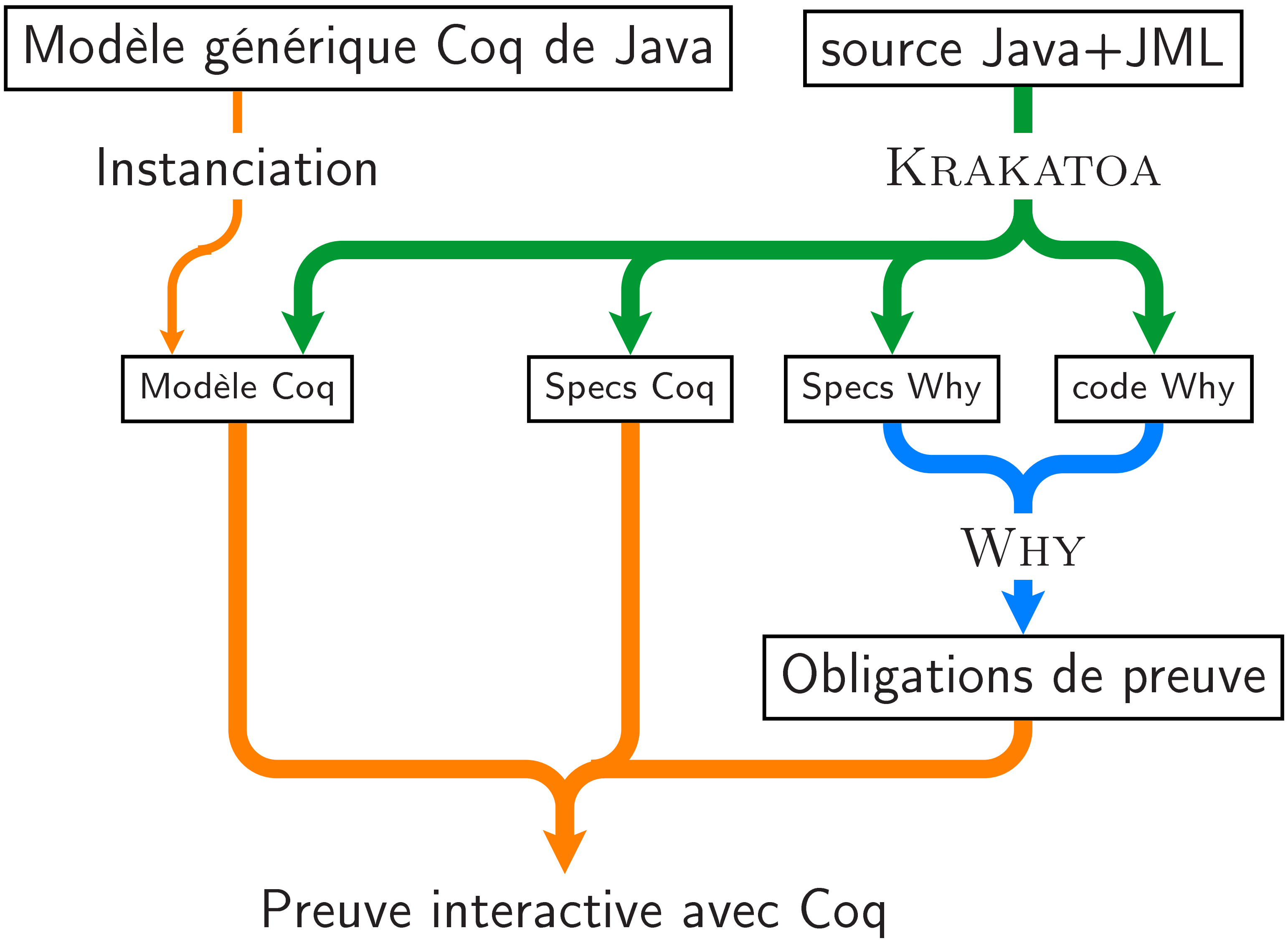
- Langage d'entrée : Java ou JavaCard, annoté en JML
- Prouver quoi ?
  - (invariant de classe et pré-condition) implique (invariant de classe et post-condition)
  - invariant de boucle, et variants (correction totale)

Exemple : porte-monnaie électronique

```
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;
    }
```

Méthodologie adoptée



Programme Why intermédiaire

```
let Purse_credit =
 fun (this : value) (s : int) ->
  { (ge_int(s, 0)
    and ((this<>Null)
        and (well_formed(heap, this, ClassType(Purse))
            and Purse_invariant(heap, this)))) }
  begin
  label init;
  let krak_accu = ((add_int { is_valid_cell(this, Field(Purse_balance)) }
                             ((int_val (((access !heap) this) (Field Purse_balance)))
                             { }) s) in
  { is_valid_update(heap, this, Field(Purse_balance),
    Primitive_int(krak_accu)) }
  (heap := ((((update !heap) this) (Field Purse_balance)) (Primitive_int krak_accu)))
  { }
  end{ ((eq_int(int_val(access(heap, this, Field(Purse_balance))),
         add_int(int_val(access(heap@, this, Field(Purse_balance))), s))
        and Purse_invariant(heap, this))
       and modifiable(heap@, heap, access_loc(this, Field(Purse_balance)))) }
```

Obligations de preuve

- ensemble de lemme Coq, PVS, etc. $\rightarrow$ preuve interactive
- fichier Simpliy $\rightarrow$ Valid / Invalid+contre-exemple

Ici une seule obligation

- une ligne de tactiques Coq
- prouvée automatiquement par Simplify

Cas d'une applet JavaCard

Contexte : Projet VERIFICARD

– applet PSE : étude de cas proposée par Schlumberger

Propriétés demandées :

– confidentialité des informations

– allocation mémoire limitée

– prédiction d'erreurs : seulement `ISOException`

– consistence : propriétés fonctionnelles de l'applet

va démarrer : applet Demoney fournie par Trusted Logic

Programmes C : Caduceus

Programmes C annotés à l'aide de commentaires

Syntaxe inspirée de JML

Modèle similaire à celui de Krakatoa

Fragment supporté : tout le C ANSI sauf

- les `goto` arbitraires
- certains cast de pointeurs

Exemple

```
/* search for a value in an array */

/*@ requires \length(t) == n
    ensures 0 <= result < n => t[result] == v */
int index(int t[], int n, int v)
{
  int i = 0;
  /*@ invariant 0 <= i && \forall int k. 0 <= k < i => t[k] <> v
      variant \length(t) - i */
  while (i < n) {
    if (t[i] == v) break;
    i++;
  }
  return i;
}
```

III

Technique sous-jacente

Illusion de logique de Hoare

En fait, une traduction fonctionnelle des programmes Why

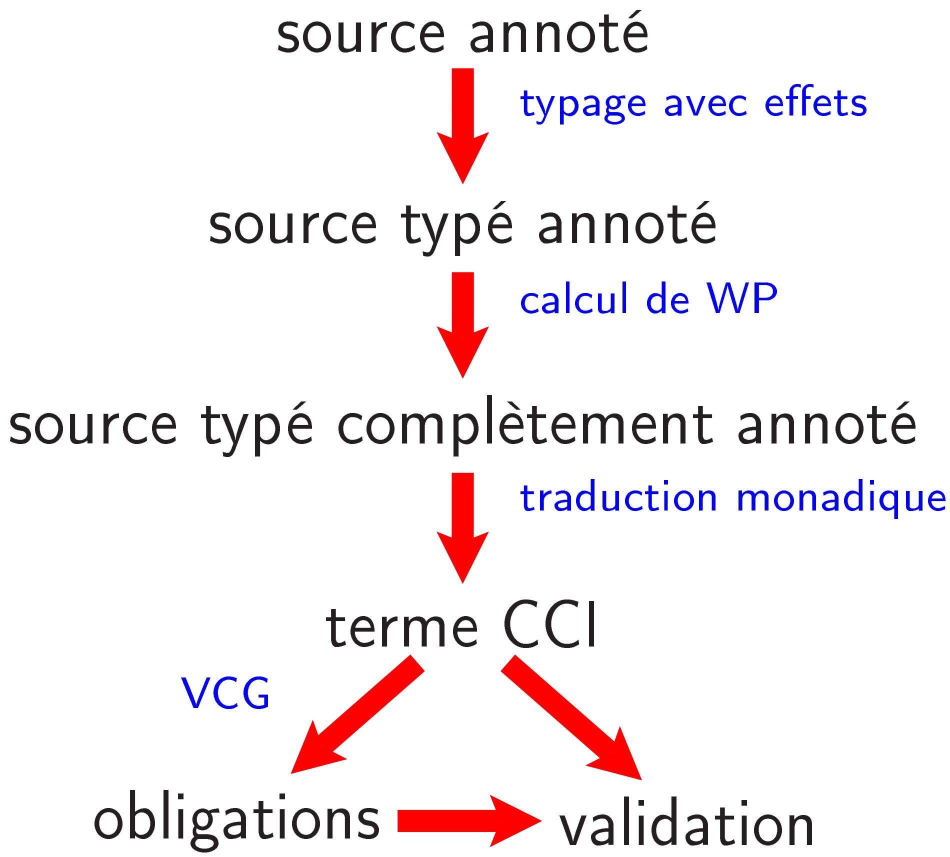
vers le Calcul des Constructions Inductives

à l'aide de monades

$$\{P\}\ p\ \{Q\}$$

$$\widehat{p} : \forall x_1 \ldots x_n.\ P \Rightarrow \exists y_1 \ldots y_m. Q$$

Méthode


source annoté
typage avec effets
source typé annoté
calcul de WP
source typé complètement annoté
traduction monadique
terme CCI
VCG
obligations
validation

Une méthode sûre

La validation exprime la correction du programme,

en supposant les obligations prouvées

La validation peut être typée par Coq...

... que les preuves soient faites avec Coq ou non

Les obligations automatiquement déchargées sont justifiées dans la validation

Disponibilité

`http://why.lri.fr`

- distrib source (12 000 lignes de code) et binaires
- manuel 30 pages (tutoriel + manuel de référence)
- nombreux exemples ($\approx$ 25)

Perspectives

- arithmétique machine : prouver l'absence d'overflow
- mise au point rapide de la spécification
  - dépliage boucles
  - évaluation symbolique sur des valeurs de test
- remontée de WP jusqu'à l'utilisateur
- remontée de contre-exemples