Verifying C and Java programs

Jean-Christophe Filliâtre

CNRS – Université Paris Sud

National Institute of Aerospace, March 9, 2004

Context

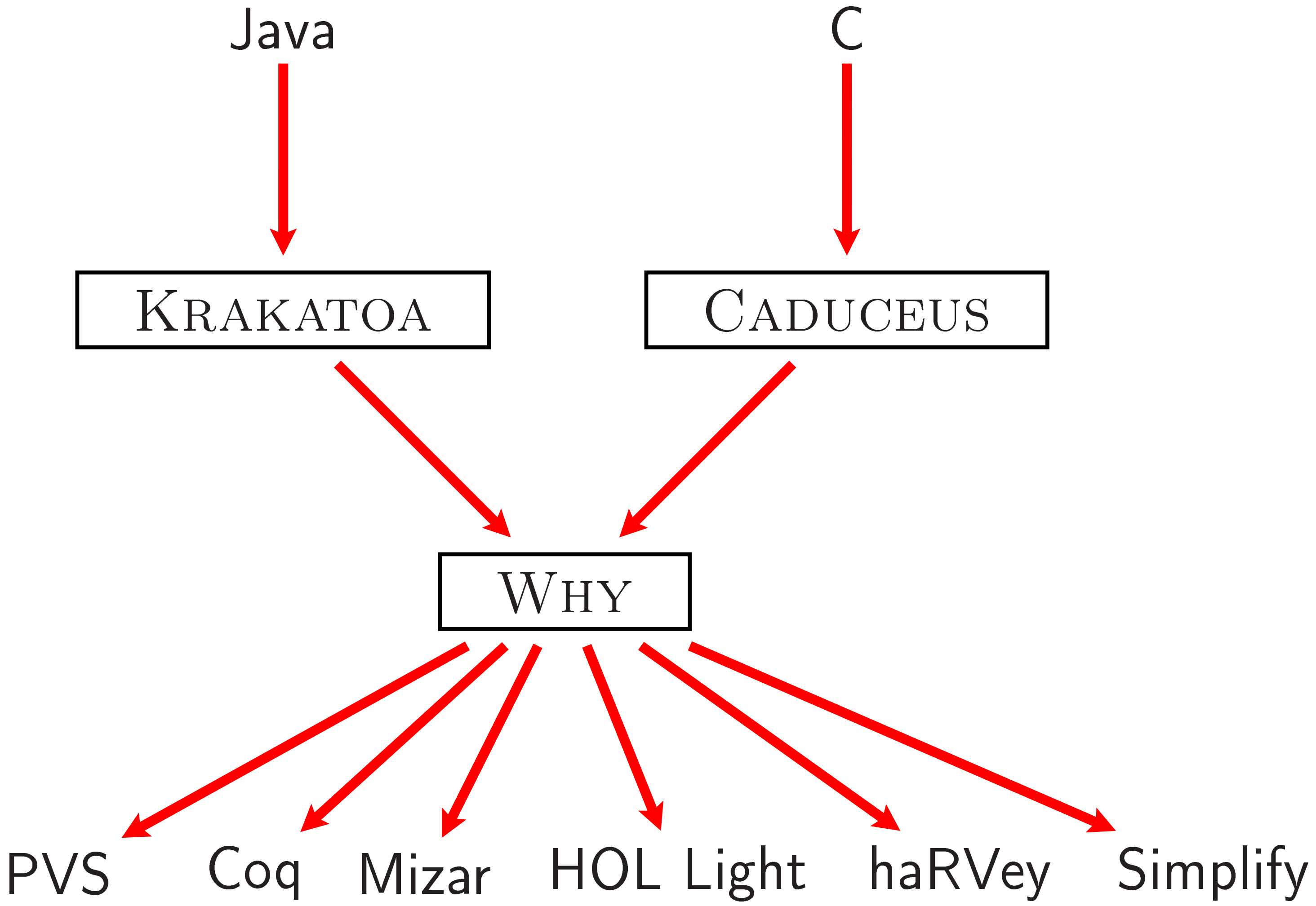
Formal methods at Université Paris Sud

Verification of functional properties of C and Java programs

Applications

- Smart cards (Schlumberger cards, Trusted Logic) Java C
- Avionics (Dassault Aviation) C

Tools developed at Orsay



Outline

1. Why: a generic tool for program verification

2. Verification of C and Java programs

I

The Why tool

Concept

$$\text{source} + \text{specification} \longrightarrow \boxed{\text{VCG}} \longrightarrow \text{proof obligations}$$

Genericity

- input: an adequate intermediate language
- output: several provers

Benefits: most of the VCG implementation is factorized

(weakest preconditions, effects, etc.)

An intermediate language

- purely functional datatypes + variables over these types
- no alias
- while loops
- if-then-else
- sequences
- local variables
- expressions = statements (ML)
- functions (local, recursive)
- exceptions

Specifications

- Hoare-style annotations
  - pre/post-conditions
  - assertions in the code
  - loop invariants/variants
- explicit effects: variables possibly accessed or modified
- logical declarations:

  types, functions, predicates, axioms

Annotations written in first-order predicate syntax

Example

```
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 }
```

Use of exceptions: `break`

The break construct is interpreted using an exception

```
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 }
```

Proof obligations

$\vdash I_0$	entering the loop
$I, b_1, \neg b_2 \vdash \mathsf{wp}(s, I)$	invariant preservation
$I, b_1, b_2 \vdash Q$	exiting with `break`
$I, \neg b_1 \vdash Q$	exiting the loop

The use of exceptions is invisible

Another example

while (e) s where e contains side-effects

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

WP for exceptions

$\mathrm{wp}(e, Q, R)$ // case of a single exception E

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

$\mathrm{wp}(\mathrm{try}\ e_1\ \mathrm{with}\ E \rightarrow e_2, Q, R) = \mathrm{wp}(e_1, Q, \mathrm{wp}(e_2, Q, R))$

Generating proof obligations

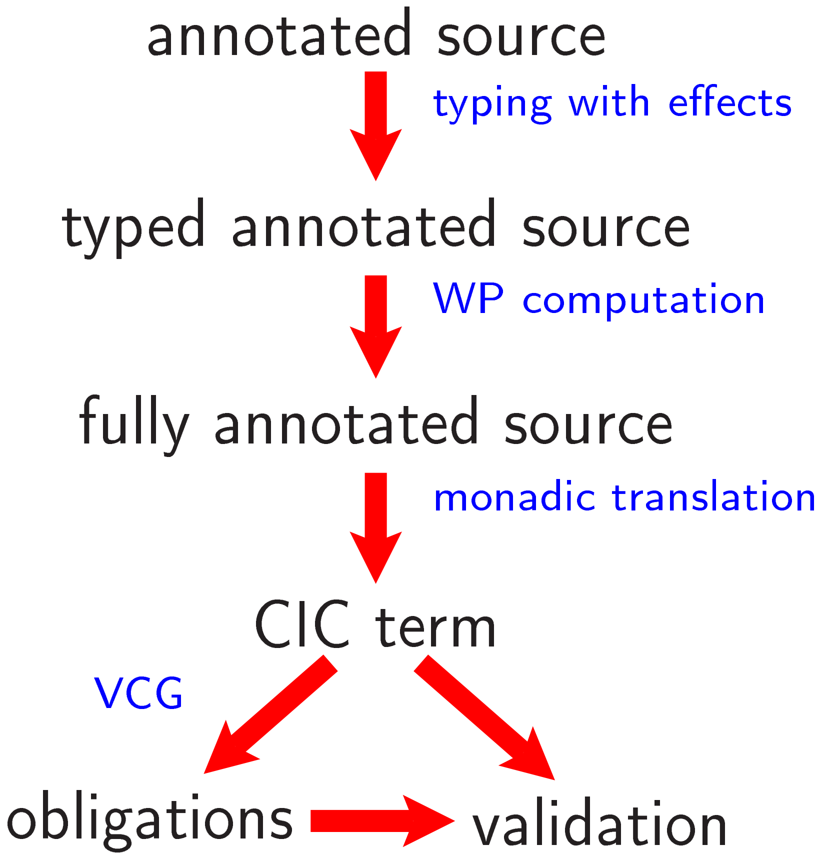
Illusion of Hoare-logic, but ...

actually a translation of Why programs into Type Theory using monads

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

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

Methodology



A safe method

The validation expresses the program correctness,

assuming the validity of obligations

The validation can be type-checked to improve confidence in the tool

Obligations automatically discharged are justified in the validation

Output for several provers

Expressing the obligations only requires a minimal logic ($\forall \Rightarrow \wedge$)

An output for a new prover only requires a 300 lines pretty-printer for a first-order logic

Part of the difficulty is hidden in the model

II

Application to C and Java programs

Recipe

1. choose a language L annotated in S
2. define a model of $L + S$ in prover P
3. interpret $L + S$ in the Why language
4. generate obligations with `why -P`
5. validate them with P

C and Java programs

Two tools developed at Orsay

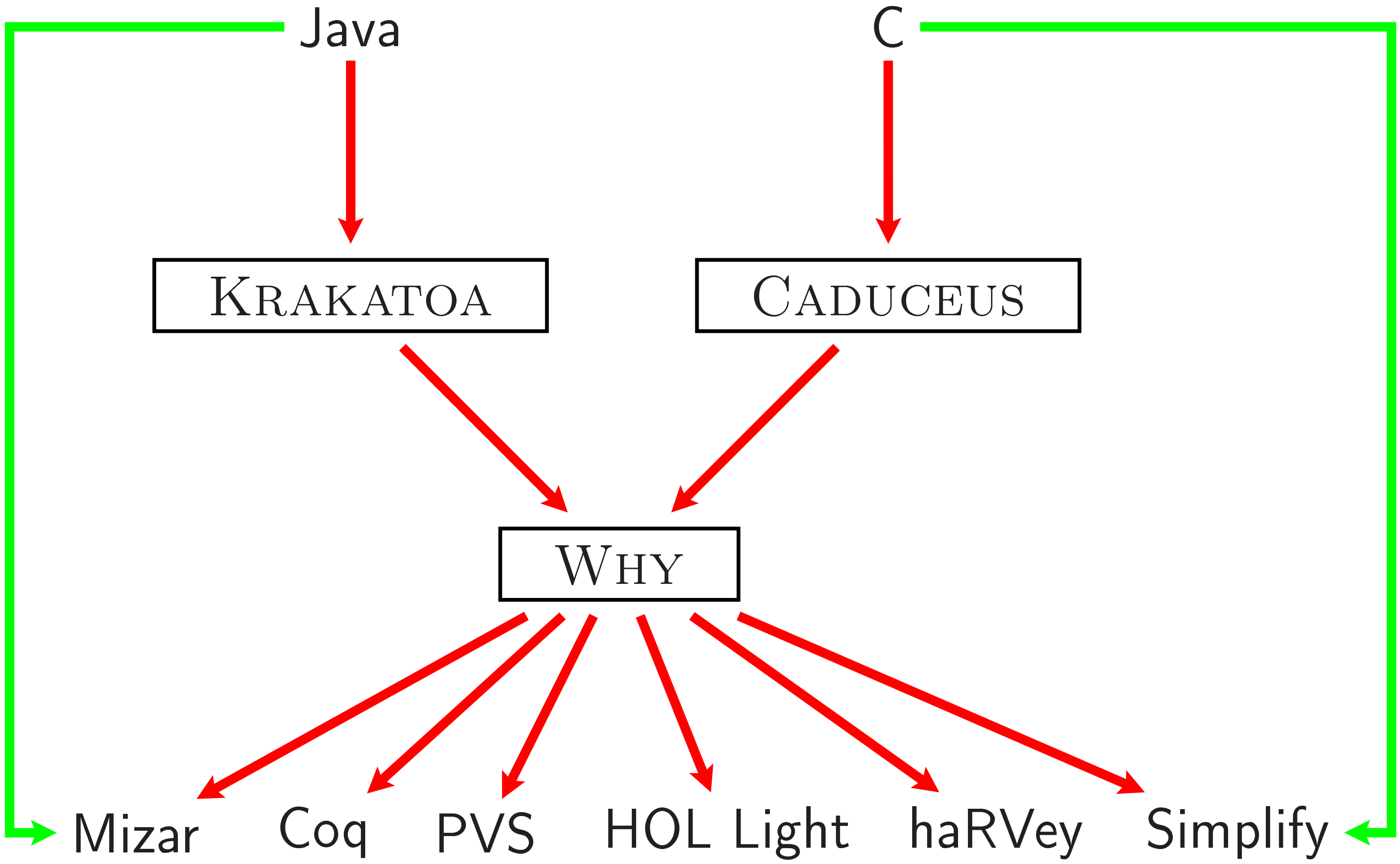
- Krakatoa: Java annotated with JML

  (C. Marché, C. Paulin, X. Urbain)

- Caduceus : C

  (C. Marché, J.-C. Filliâtre)

C and Java programs



Model

R. Burstall 1972

heap-as-array trick

heap-as-several-maps

a structure/object field = a map

R. Bornat

Proving Pointer Programs in Hoare Logic

T. Nipkow and F. Mehta

Proving Pointer Programs in Higher-Order Logic (Isabelle/HOL)

Model

	alloc	x	y	...	int[]	...
a_1	A	3		...		
a_2	B					
a_3	int[3]					
$\vdots$	$\vdots$	$\vdots$	$\vdots$			

```
a1.x = 3
```

Krakatoa: Java programs

- Input: Java or JavaCard,

  annotated with the Java Modeling Language (JML)

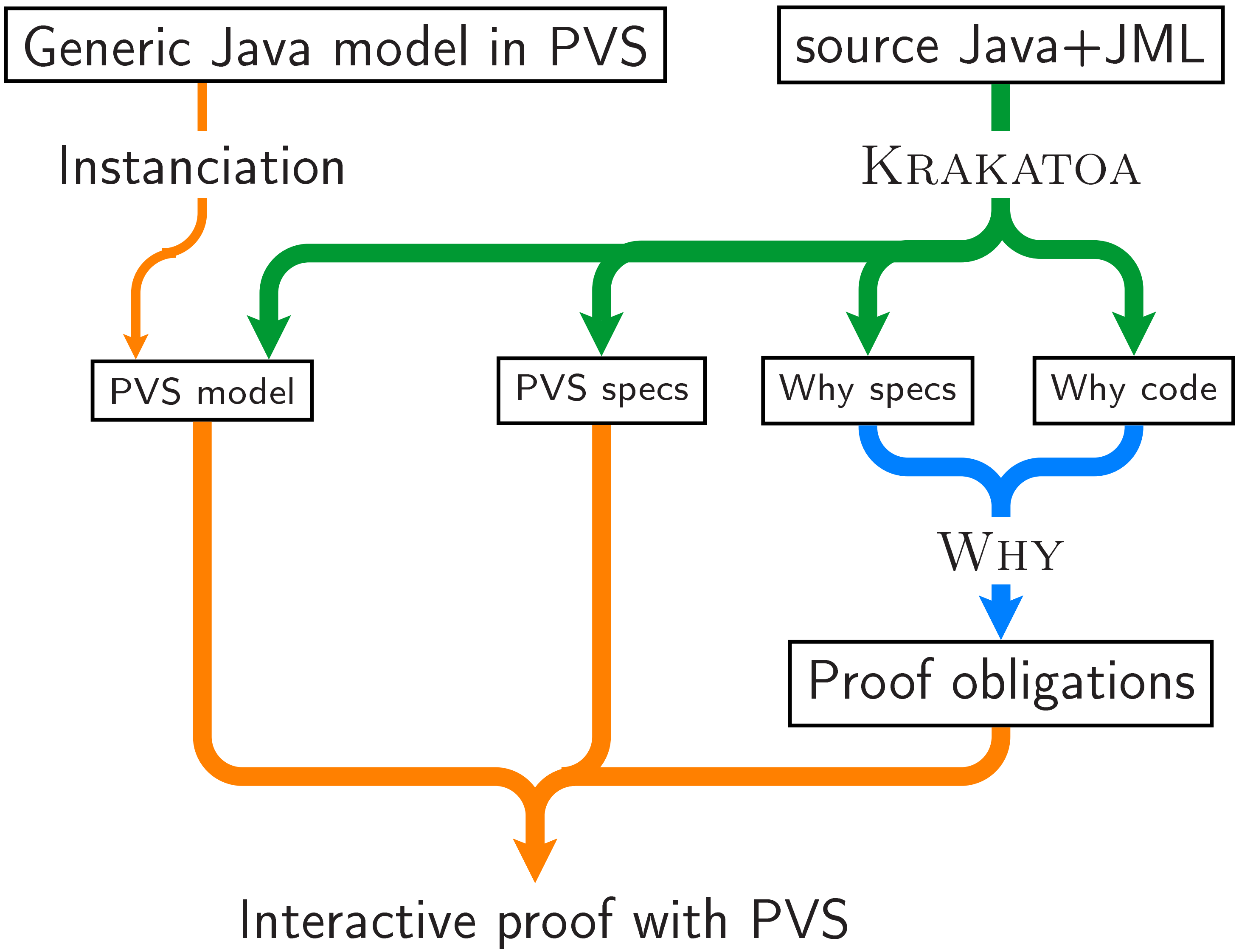
- To be proved:
  - (class invariant and pre-condition) implies (class invariant and post-condition)
  - loop invariant and variant (total correctness)

Example: electronic purse

```
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;
    }
}
```

Methodology



Intermediate Why program

```
let Purse_credit_body =
 fun (this : value) (s : int) ->
  { (ge_int(s, 0)
    and (neqv(this,Null)
        and (instanceof(heap, this, ClassType(Purse))
            and Purse_invariant(Purse_balance, this)))) }
  begin
  label init;
  let krak_acc = ((add_int ((acc !Purse_balance) this)) s) in
  Purse_balance := (((update !Purse_balance) this) krak_acc)
  end{ ((eq_int(acc(Purse_balance, this),
         add_int(acc(Purse_balance@, this), s))
        and Purse_invariant(Purse_balance, this))
       and modifiable(heap@, Purse_balance@, Purse_balance, value_loc(this))) }
```

Proof obligations

- set of PVS lemmas $\longrightarrow$ interactive proof
- Simpliy input file $\longrightarrow$ Valid / Invalid+counterexample

Here a single obligation

- proved with `(grind)`
- validated by Simplify

Case study of a JavaCard applet

Context: VERIFICARD project

- PSE applet: case study proposed by Schlumberger

Properties to be proved:

- confidentiality
- limited memory allocation
- error prediction: only `ISOException` raised
- soundness: functional properties of the applet

just started: Demoney case-study delivered by Trusted Logic

C programs: Caduceus

C programs annotated using a JML-like language

Model similar to the one for Java programs (+ pointer arithmetic)

Supported C fragment : eventually all ANSI C except

- arbitrary `goto`
- some pointers casts

Caduceus is work in progress

Example

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

/*@ requires \valid_range(t,0,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;
}
```

Availability

http://why.lri.fr/

- GPL source code (12 000 lines) and executables
- 30 pages manual (tutorial + reference manual)
- numerous examples (≈ 25)

http://krakatoa.lri.fr/

Caduceus: to be released soon

Future work

- machine arithmetic
  - integer arithmetic without overflow
  - floating point arithmetic
- specification debugging
  - loops unrolling
  - symbolic evaluation on test values
- translating back to the user
  - functions WP
  - decision procedures counterexamples