

Formal Verification of OO Programs: the Krakatoa Approach (part 2)

Claude Marché

INRIA, Orsay, France

FVOOS Winter School, Viinistu, Estonia, January 25-29,
2009



Outline

- 1 Summary of Lecture 1
- 2 Krakatoa to Why
- 3 Separation Analysis
- 4 Other aspects

Outline

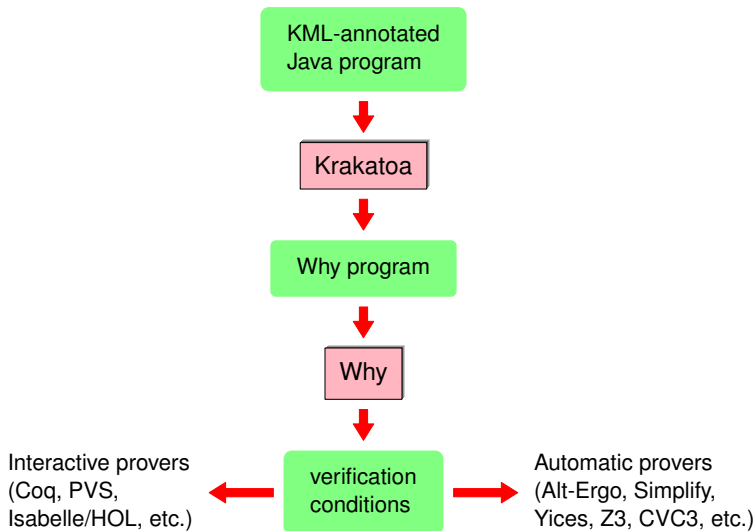
1 Summary of Lecture 1

2 Krakatoa to Why

3 Separation Analysis

4 Other aspects

Krakatoa/Why architecture



The Why VCgen in short

Why language:

- ML-style programming language, few constructs
- Side-effects: alias-free references
- Specifications in first-order logic
- VC generation based on a WP calculus with exceptions

Using Why for Java ?

The Why VCgen:

- deals with control structures
- but do not allow complex data structures: only references to pure types

Workaround:

- Why specification language *allows to specify abstract datatypes by first-order axiomatizations*

Abstract types in Why

- First-order *typed* logic, with ML-polymorphism
- Algebraic-style axiomatization
- Example: generic lists

```
type  $\alpha$  list
logic nil:   $\alpha$  list
logic cons:  $\alpha, \alpha$  list  $\rightarrow \alpha$  list
logic append:  $\alpha$  list,  $\alpha$  list  $\rightarrow \alpha$  list
axiom app_nil:
    forall l: $\alpha$  list.  append(l,nil) = l
...
logic sumlist:  int list  $\rightarrow$  int
axiom sum_single:
    forall n:int,  sumlist(cons(n,nil)) = n
...
```

Outline

- 1 Summary of Lecture 1
- 2 Krakatoa to Why**
- 3 Separation Analysis
- 4 Other aspects

Krakatoa to Why

The translation is based on three components:

- 1 *Representation of heap memory* using alias-free Why references only
- 2 *Computation of read and write effects* on Why references
- 3 *Translation rules* from annotated Java to Why

Modeling of Heap Memory

- “Natural” modeling: heap memory = one big “array”
- Array in Why: a reference to an *applicative array*
- Axiomatization of applicative arrays:

$$\text{select} : \alpha \text{ memory}, \text{pointer} \rightarrow \alpha$$
$$\text{store} : \alpha \text{ memory}, \text{pointer}, \alpha \rightarrow \alpha \text{ memory}$$
$$\forall m p v, \text{select}(\text{store}(m, p, v), p) = v$$
$$\forall m p_1 p_2 v, p_1 \neq p_2 \Rightarrow$$
$$\text{select}(\text{store}(m, p_1, v), p_2) = \text{select}(m, p_2)$$

- Why variable *heap*: reference to a “memory”

Memory as Big Array

The Java code fragment

```
C x, y;
x.f = 0;
y.g = 1;
//@ assert x.f == 0;
```

would be transformed as

```
heap := store(!heap, shift(x, f), 0);
heap := store(!heap, shift(y, g), 1);
assert {select(!heap, shift(x, f)) = 0}
```

the WP is

```
select(store(store(heap, shift(x, f), 0),
           shift(y, g), 1), shift(x, f)) = 0
```

component-as-array modeling

component-as-array modeling:

- old idea due to Burstall
- renewed by Bornat [MPC, 2000]

Principle:

- each structure/object field is mapped to a different array
- relies on the property “*two different fields cannot be aliased*”

component-as-array modeling

```
C x, y;  
x.f = 0;  
y.g = 1;  
//@ assert x.f == 0;
```

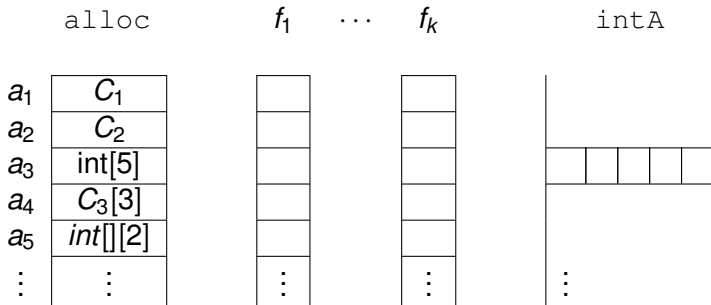
becomes

```
f := store(!f, x, 0);  
g := store(!g, y, 1);  
assert {select(!f, x) = 0}
```

the WP is

```
select(store(f, x, 0), x) = 0
```

Why variables for heap memory



- `alloc`: dynamic type, and size for arrays
- f_1, \dots, f_k : all field names
- `intA`: arrays of `int`
- **also**: `boolA`, `objA`, etc.

types

```
[[int, short, ...]] = int
[[boolean]] = bool
[[float, double]] = real
[[Tref]] = pointer
```

pointer: new Why type to denote Java references

Simple expressions

- constant:

$$\llbracket c \rrbracket = c$$

- binary operator:

$$\llbracket e_1 \text{ op } e_2 \rrbracket = \llbracket e_1 \rrbracket \text{ op } \llbracket e_2 \rrbracket$$

- local variable declaration:

$$\llbracket \tau \ x = e; s \rrbracket = \text{let } x = \text{ref } \llbracket e \rrbracket \text{ in } \llbracket s \rrbracket$$

- local variable access:

$$\llbracket v \rrbracket = !v$$

- local variable assignment:

$$\llbracket v = e \rrbracket = v := \llbracket e \rrbracket$$

Objects fields

- field access:

$$\llbracket e.f \rrbracket = \text{let } tmp = \llbracket e \rrbracket \text{ in}$$
$$\text{assert}\{tmp \neq \text{null}\};$$
$$\text{select}(!f, tmp)$$

- field assignment:

$$\llbracket e_1.f = e_2 \rrbracket = \text{let } tmp_1 = \llbracket e_1 \rrbracket \text{ in}$$
$$\text{let } tmp_2 = \llbracket e_2 \rrbracket \text{ in}$$
$$\text{assert}\{tmp_1 \neq \text{null}\};$$
$$f := \text{store}(!f, tmp_1, tmp_2)$$

NullPointerExceptions are ruled out

Arrays

- WHY logic function

`blocklength : alloc_table, pointer → int`

`blocklength(a, p)`: the size of allocated block of p in table a

- Array access:

```

[[e1[e2]] = let tmp1 = [[e1]] in
               let tmp2 = [[e2]] in
               assert{tmp1 ≠ null ∧
                    0 ≤ tmp2 < blocklength(!alloc, tmp1)};
               select(!intA, shift(tmp1, tmp2))
    
```

IndexOutOfBoundsExceptions are ruled out

Casts and instanceof

On each program:

- **WHY abstract type** `tag_id`
- each class identifier \rightsquigarrow a constant of type `tag_id`
- subclass relation encoded in predicate `subtag`
- **WHY logic function**

`typeof : alloc_table, pointer \rightarrow tag_id`

`typeof(a, p): the dynamic type of a p in table a`

Casts and instanceof

- instanceof:

$$\llbracket e \text{ instanceof } C \rrbracket = \text{subtag}(\text{typeof}(!\text{alloc}, \llbracket e \rrbracket), C)$$

- cast expression:

$$\begin{aligned} \llbracket (C)e \rrbracket = & \\ & \text{let } tmp = \llbracket e \rrbracket \text{ in} \\ & \text{assert } \{\text{subtag}(\text{typeof}(!\text{alloc}, tmp), C)\}; \\ & tmp \end{aligned}$$

ClassCastException are ruled out

Statements

- Java sequence: Why sequence
- `if` statement:

$$\llbracket \text{if } (e) s_1 \text{ else } s_2 \rrbracket =$$
$$\text{if } \llbracket e \rrbracket \text{ then } \llbracket s_1 \rrbracket \text{ else } \llbracket s_2 \rrbracket$$

- Abrupt termination:

$$\llbracket \text{break;} \rrbracket = \text{raise Break;}$$
$$\llbracket \text{continue;} \rrbracket = \text{raise Continue;}$$

While loop

```
[[while (e)s]] =  
try  
  loop  
  if not [[e]] then raise Break;  
  try [[s]]  
  with Continue →void  
with Break →void
```

Exceptions

- Exception throwing:

```
[[throw e]] = raise (Exception [[e]])
```

- Exception catching:

```
try s  
catch (Exc v) s'
```

is translated into:

```
try [[s]]  
with (Exception tmp) ->  
  if (instanceof tmp Exc) then  
    let v = tmp in [[s']]  
  else raise (Exception tmp)
```

Method declarations

Assume in some class C :

```
/*@ requires  $R$  ;
   @ assigns  $A$ ;
   @ ensures  $E$ ;
   @*/  $\tau$   $m(x_1 : \tau_1, \dots, x_n : \tau_n)$  ;
```

leads to a WHY declaration:

```
parameter  $C\_m\_parameter$ :
  this:pointer  $\rightarrow x_1 : \llbracket \tau_1 \rrbracket \rightarrow \dots \rightarrow x_n : \llbracket \tau_n \rrbracket \rightarrow$ 
  {  $this \neq \text{null} \wedge \llbracket R \rrbracket \wedge \text{Inv}(this, x_1, \dots, x_n)$  }
   $\llbracket \tau \rrbracket$  reads  $r$  writes  $w$ 
  {  $\llbracket E \rrbracket \wedge \text{Inv}(this, x_1, \dots, x_n) \wedge \text{Assigns}(A)$  }
```

- effects r and w : *computation of effects*
- $\text{Assigns}(A)$: encoding of A (computed as well)
- $\text{Inv}(\cdot)$: “naive” class invariants

Method calls

$\llbracket e.m(e_1, \dots, e_n) \rrbracket$ is

let $tmp = \llbracket e \rrbracket$ in

let $tmp_1 = \llbracket e_1 \rrbracket$ in

⋮

let $tmp_n = \llbracket e_n \rrbracket$ in

$(C_m_parameter\ tmp\ tmp_1\ \dots\ tmp_n)$

For dynamic calls:

- if $D.m$ overrides $C.m$ then behaviors of $D.m$ are added to $C_m_parameter$ with additional clause
 $/*@$ assumes (*this* instanceof D)
- **Note:** `requires` not allowed for $D.m$

Method bodies

```

let  $C_m$  = fun
  this:pointer  $\rightarrow$   $x_1 : \llbracket \tau_1 \rrbracket \rightarrow \dots \rightarrow x_n : \llbracket \tau_n \rrbracket \rightarrow$ 
  {  $this \neq \text{null} \wedge \llbracket R \rrbracket \wedge \text{Inv}(this, x_1, \dots, x_n)$  }
  let return=ref (any $_{\tau}$  ()) in
  try body with Return  $\rightarrow$  !return
  {  $\llbracket E \rrbracket \wedge \text{Inv}(this, x_1, \dots, x_n) \wedge \text{Assigns}(A)$  }
  
```

where `return` statements are translated as:

$$\llbracket \text{return } e; \rrbracket = \text{return} := \llbracket e \rrbracket ;$$

$$\text{raise Return};$$

Object creation

```
parameter alloc_parameter:  
  s:tag_id -> n:int ->  
  { n >= 0 }  
pointer writes alloc  
  { blocklength(alloc,result) = n and  
    subtag(typeof(alloc,result),s) }
```

new $C(e_1, \dots, e_n)$ is then translated into:

```
let tmp = (alloc_parameter  $C$  1) in  
(cons_ $C$ _parameter tmp  $\llbracket e_1 \rrbracket \cdots \llbracket e_n \rrbracket$ );  
tmp
```

Effects

- reads and write effects are computed “at compile-time”
- example of a rule (see lecture notes):

$$\frac{\Gamma \vdash e_1 : R_1, W_1 \quad \Gamma \vdash e_2 : R_2, W_2}{\Gamma \vdash e_1.f = e_2 : R_1 \cup R_2, W_1 \cup W_2 \cup \{f\}}$$

- effects for all methods are computed by iteration until a fix-point is reached.
- it converges necessarily: number of Why variables of the modeling is fixed.

assigns (A) clause

First step: partition A w.r.t computed write effects W .

- Example:

```
//@ assigns x.f, t[0..5].f, u[3..7];
```

- writes effects: $W = \{f, \text{int}A\}$
- $A_f = \{x, t[0..5]\}$
- $A_{\text{int}A} = \{u[3..7]\}$

General formula:

$$A = \bigcup_{w \in W} \bigcup_{p \in A_w} \text{select}(w, p)$$

Sets of pointers

$$A = \bigcup_{w \in W} \bigcup_{p \in A_w} \text{select}(w, p)$$

- A_w : pset, WHY datatype for sets of pointers:
 - `pset_empty` denotes the empty set
 - `pset_singleton(p)` denotes $\{p\}$
 - `pset_range(s,a,b)` denotes

$$\{\text{shift}(p, i) \mid p \in s, a \leq i \leq b\}$$

- $A_f = \{x, t[0..5]\}$ becomes in Why:

$$\text{pset_union}(\text{pset_singleton}(x), \\ \text{pset_range}(\text{pset_singleton}(t), 0, 5))$$

- $A_{intA} = \{u[3..7]\}$ in Why:

$$\text{pset_range}(\text{pset_singleton}(u), 3, 7)$$

not_assigns predicate

Why predicate $\text{not_assigns}(a, m_1, m_2, s)$:

- means that for any pointer p , allocated in a , which do not belong to s : $\text{select}(m_1, p) = \text{select}(m_2, p)$.

- formula

$$A = \bigcup_{w \in W} \bigcup_{p \in A_w} \text{select}(w, p)$$

translates to Why as

$$\text{Assigns}(A) = \bigwedge_{w \in W} \text{not_assigns}(a, w@, w, A_w)$$

Effects: example

```
//@ assigns x.f, t[0..5].f, u[3..7];
```

- writes effects: $W = \{f, \text{int}A\}$
- $\text{Assigns}(A)$:

```
not_assigns(a, f@, f,  
  pset_union(pset_singleton(x),  
    pset_range(pset_singleton(t), 0, 5)))  
 $\wedge$  not_assigns(a, intA@, intA,  
  pset_range(pset_singleton(u), 3, 7))
```


Hybrid predicates

- Reads effects computed, for each label
- Heap variables are added as parameters

```
/*@ predicate Sorted{L}(int a[], integer l,
    @                                     integer h) =
    @   \forall integer i; l <= i < h ==>
    @       \at(a[i],L) <= \at(a[i+1],L) ;
    @*/
```

becomes

```
predicate Sorted(a:pointer, l:int, h:int,
    intA_at_L: int memory) =
     $\forall i:\text{int}. \quad l \leq i \leq h \rightarrow$ 
    select(intA_at_L, shift(a, i))  $\leq$ 
    select(intA_at_L, shift(a, i + 1))
```

Inductive/Axiomatic block

```

/*@ inductive Permut{L1,L2}(int a[], integer l,
    @                               integer h) {
    @   case Permut_swap{L1,L2}:
    @     \forall int a[], integer l h i j;
    @       l <= i <= h && l <= j <= h &&
    @       \at(a[i],L1) == \at(a[j],L2) &&
    @       \at(a[j],L1) == \at(a[i],L2) &&
    @     ...
    @   }
    @ */

```

becomes

```

inductive Permut(a:pointer, l:int, h:int,
    intA_at_L1: int memory,
    intA_at_L2: int memory) =
...

```

Krakatoa: conclusions

- Why as intermediate language: very suitable, separates concerns:
 - modeling of data handled by Krakatoa
 - control structures handled by Why
- Other front-end: C but also MIX, OCaml
- Krakatoa original features
 - Algebraic-style specifications, not OO specifications
 - hybrid predicates
 - Memory model (including separation, see next section)
- Limitations, e.g: object initialization, concurrency, generics, ...
- Major ongoing work:
 - better handling of class invariants
 - abstract models

Outline

- 1 Summary of Lecture 1
- 2 Krakatoa to Why
- 3 Separation Analysis**
- 4 Other aspects

A toy example

```
/*@ requires x != null && y != null;  
   @ assigns x.f, y.f  
   @ ensures x.f == 1 && y.f == 2  
   @*/  
void m(C x, C y) { x.f = 1; y.f = 2; }  
  
C t1[2], t2[2];  
  
/*@ ensures t1[0].f == 1 && t2[0].f == 2  
void test() { m(t1[0],t2[0]); }
```

- post-condition of `test` is valid at runtime
- Modular static verification: not OK, needs additional precondition `x != y` in `m`

A toy example

```
/*@ requires x != null && y != null;  
   @ assigns x.f, y.f  
   @ ensures x.f == 1 && y.f == 2  
   @*/  
void m(C x, C y) { x.f = 1; y.f = 2; }  
  
C t1[2], t2[2];  
  
/*@ ensures t1[0].f == 1 && t2[0].f == 2  
void test() { m(t1[0],t2[0]); }
```

- post-condition of `test` is valid at runtime
- Modular static verification: not OK, needs additional precondition `x != y` in `m`

A toy example

```
/*@ requires x != null && y != null;  
   @ assigns x.f, y.f  
   @ ensures x.f == 1 && y.f == 2  
   @*/  
void m(C x, C y) { x.f = 1; y.f = 2; }  
  
C t1[2], t2[2];  
  
/*@ ensures t1[0].f == 1 && t2[0].f == 2  
void test() { m(t1[0],t2[0]); }
```

- post-condition of `test` is valid at runtime
- Modular static verification: not OK, needs additional precondition `x != y` in `m`

Why the need of $x \neq y$

Summary of
Lecture 1

Krakatoa to
Why

Separation
Analysis

Other aspects

- Weakest precondition of $x.f == 1$:

$$\textit{select}(\textit{store}(\textit{store}(f, x, 1), y, 2), x) = 1$$

- to apply axioms of arrays:

$$\textit{select}(\textit{store}(m, a, v), a) = v$$

$$\textit{select}(\textit{store}(m, a, v), b) = \textit{select}(m, a) \quad \text{if } a \neq b$$

we need the hypothesis $x \neq y$

Goal: drop this extra precondition

- *static analysis of separation*
on the example: arrays t_1 and t_2 separated
→ x and y separated in the call to m
- *Integrate separation into the component-as-array model:*
on the example: two *regions* for t_1 and t_2
- Why code of m : become
$$f_1 := \text{store}(f_1, x, 1); \quad f_2 := \text{store}(f_2, y, 2)$$
- Weakest precondition of $x.f == 1$: become
$$\text{select}(\text{store}(f_1, x, 1), x) = 1$$

→ provable without need of $x \neq y$

Region parametricity

- Consider a small change on the example:

```
/*@ ensures t1[0].f == 1 && t2[0].f == 2
   @      && t1[1].f == 2 && t2[1].f == 1 ;
   @*/
void test() { m(t1[0],t2[0]); m(t2[1],t1[1])
```

- First call: x in region 1, y in region 2
- Second call: x in region 2, y in region 1
- \rightsquigarrow Pass regions as *parameters* to m :

```
let m( $f_x, f_y, x, y$ ) =
     $f_x := \text{store}(f_x, x, 1); f_y := \text{store}(f_y, y, 2);$ 
```

```
let test() =
     $m(f_1, f_2, t_1[0], t_2[0]);$ 
     $m(f_2, f_1, t_2[1], t_1[1]);$ 
```

Typing with regions

- Separation analysis: type inference for an ML-style polymorphic typing [Tofte, Talpin]
- Types with regions
 - Type of pointers: ρ pointer
 - Type of heap memories: (ρ, α) memory
 - $select$: (ρ, α) memory, ρ pointer $\rightarrow \alpha$
- On the example: type of m is
$$m(f_x : (\rho_1, int)memory, f_y : (\rho_2, int)memory, \\ x : \rho_1 \text{ pointer}, y : \rho_2 \text{ pointer})$$
- Why code generation: $e.f$ interpreted as $select(f_r, e)$ when $e : r$ pointer

Type system

Summary of
Lecture 1

Krakatoa to
Why

Separation
Analysis

Other aspects

- Typing judgements: $\Gamma, \Delta \vdash e : t$
 - Γ maps variables to types
 - Δ maps pairs (region, field name) to types
- Example of typing rules: assignment of references

$$\frac{x : r \text{ pointer} \in \Gamma \quad \Gamma, \Delta \vdash e : r \text{ pointer}}{\Gamma, \Delta \vdash x = e : r \text{ pointer}}$$

field access:

$$\frac{\Gamma, \Delta \vdash e : r \text{ pointer} \quad (r, f) : t \in \Delta}{\Gamma, \Delta \vdash e.f : t}$$

Typing of method calls

Summary of
Lecture 1

Krakatoa to
Why

Separation
Analysis

Other aspects

- Methods type can by polymorphic w.r.t regions
- Method calls:

$$\frac{\Gamma, \Delta \vdash e_1 : t_1 \quad \cdots \quad \Gamma, \Delta \vdash e_n : t_n}{\Gamma, \Delta \vdash id(e_1, \dots, e_n) : t}$$

if $id : (\tau_1, \dots, \tau_n) \rightarrow \tau \in \Gamma$ and there is a region substitution σ such that $t = \tau\sigma$ and for each i , $t_i = \tau_i\sigma$ and $\rho_i\sigma \neq \rho_j\sigma$ when $i \neq j$

- aliasing of regions forbidden

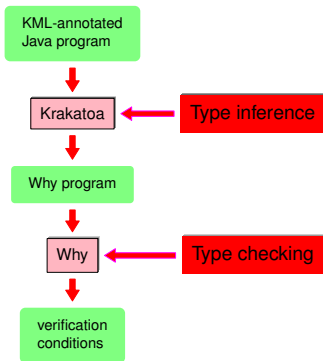
Soundness result

Soundness is relative to soundness of Why code generation without separation:

Theorem *If a program is well-typed in a given environment Γ, Δ , then its Why interpretation with region analysis has the same semantics as its Why interpretation with the component-as-array model*

Type inference

- Region types are not given, they must be inferred
- Type inference: ML-style inference based on unification
- In practice: types inferred are checked by Why:



Example

Inspired from an example of P. Müller (at the Workshop on Challenges in OO Program Verification, Nijmegen, 2006)

count the number n of positive values in an array t , then copy these values to a freshly allocated array of size n

t

2	1	0	4	-1	5	3	0
---	---	---	---	----	---	---	---

$n=5$

u

2	1	4	5	3
---	---	---	---	---

A challenging example

```
int[] m(int t[]) {  
    int i, count = 0;  
    for (i=0; i < t.length; i++)  
        if (t[i] > 0) count++;  
    int[] u = new int[count];  
    count = 0;  
    for (i=0; i < t.length; i++)  
        if (t[i] > 0) u[count++] = t[i];  
    return u;  
}
```

- Challenge: safety of `u[count++] = ..`
- depends on the “behavior” of the first loop
- Need to annotate the code to formalize this behavior

Annotated code

```

int[] m(int t[]) {
    int i, count = 0;
    /*@ loop_invariant
       @   count == num_of_pos(0,i-1,t) ; */
    for (i=0; i < t.length; i++)
        if (t[i] > 0) count++;
    int[] u = new int[count];
    count = 0;
    /*@ loop_invariant
       @   count == num_of_pos(0,i-1,t) ; */
    for (i=0; i < t.length; i++)
        if (t[i] > 0) u[count++] = t[i];
    return u;
}

```

logic function num_of_pos

```
/*@ axiomatic NumOfPos {  
  @ logic integer num_of_pos{L}(integer i,  
    @ integer j,int t[]);  
  @  
  @ axiom num_of_pos_empty{L} :  
  @ \forallall integer i j, int t[];  
  @ i > j ==> num_of_pos(i,j,t) == 0;  
  @  
  @ axiom num_of_pos_true_case{L} :  
  @ \forallall integer i j k, int t[];  
  @ i <= j && t[j] > 0 ==>  
  @ num_of_pos(i,j,t) ==  
  @ num_of_pos(i,j-1,t) + 1;  
  @  
  @ axiom num_of_pos_false_case{L} :  
  @ \forallall integer i j k, int t[];  
  @ i <= j && ! (t[j] > 0) ==>
```

Verification

Second loop:

```
loop_invariant num_of_pos(intA, 0, i - 1, t)
{
...
intA := store(intA, shift(u, count), select(intA, shift(t, i)))
}
```

- assignment to *intA* might change
num_of_pos(*intA*, 0, *i* - 1, *t*)
- we need to know that modification of *u*[*count*] does not affect the *footprint* of num_of_pos(*intA*, 0, *i* - 1, *t*)

Verification

With separation:

```
loop_invariant num_of_pos(intAt, 0, i - 1, t)  
{  
  ...  
  intAu := store(intAu, shift(u, count), select(intAt, shift(t, i)))  
}
```

it comes for free!

- Separation analysis integrated in the memory model achieves (partly) the same goal as
 - separation logic
 - dynamic frames

Experiment on industrial code

- Avionics embedded C code provided by Dassault aviation company
- Many global data structures, structures containing nested arrays of other structures.
- 3 kloc excerpt:
 - Without separation: 1151 VCs, 83.8% solved
 - With separation: 1982 VCs, 100% solved
 - 376 regions inferred for global data
 - 242 region variables added as parameter to functions
- full 70 kloc:
 - Without separation: “out-of-memory”
 - With separation: \sim 100k VCs, 98.5 % solved

Summary

Separation analysis improves WP-based deduction verification:

- Simpler VCs, proofs easier to obtain
- Improves scalability
- Partly provides aspects of separation logic or dynamic frames, although only using standard SMT solvers

Outline

- 1 Summary of Lecture 1
- 2 Krakatoa to Why
- 3 Separation Analysis
- 4 Other aspects**

Java Card support

Summary of
Lecture 1

Krakatoa to
Why

Separation
Analysis

Other aspects

Krakatoa optionally provides Java Card features:

- Java Card API
- Transactions [Rousset, SEFM'06]

Industrial case study (SmartCards at Gemalto):

- Industrial banking applet Payflex (Banking) - 4600 loc
- SIMSave: SIM/Server synchro - 3800 loc
- IAS: government security platform - 20 000 loc

Automatic Generation of annotations

A key feature for scalability: automatic generation of simple annotations:

- loop invariants
- pre-conditions
- `assigns` clauses

Original approach [[Moy, VMCAI'08](#), [PhD 09](#)]: combination of

- Abstract interpretation
- Weakest Preconditions
- Quantifier Elimination

and also [[Rousset, PhD 08](#)]:

- Non-null by default policy

Integer and floating-point Arithmetic

- User can select: unbounded integers/machine integers
- Also: reals/floating-point computation:
 - faithful floating-point computations, with checks for non-overflow [Boldo, Filliâtre, ARITH'07]
 - floating-point computations with special values Infinity/Nan (in progress)

THE END