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

Claude Marché

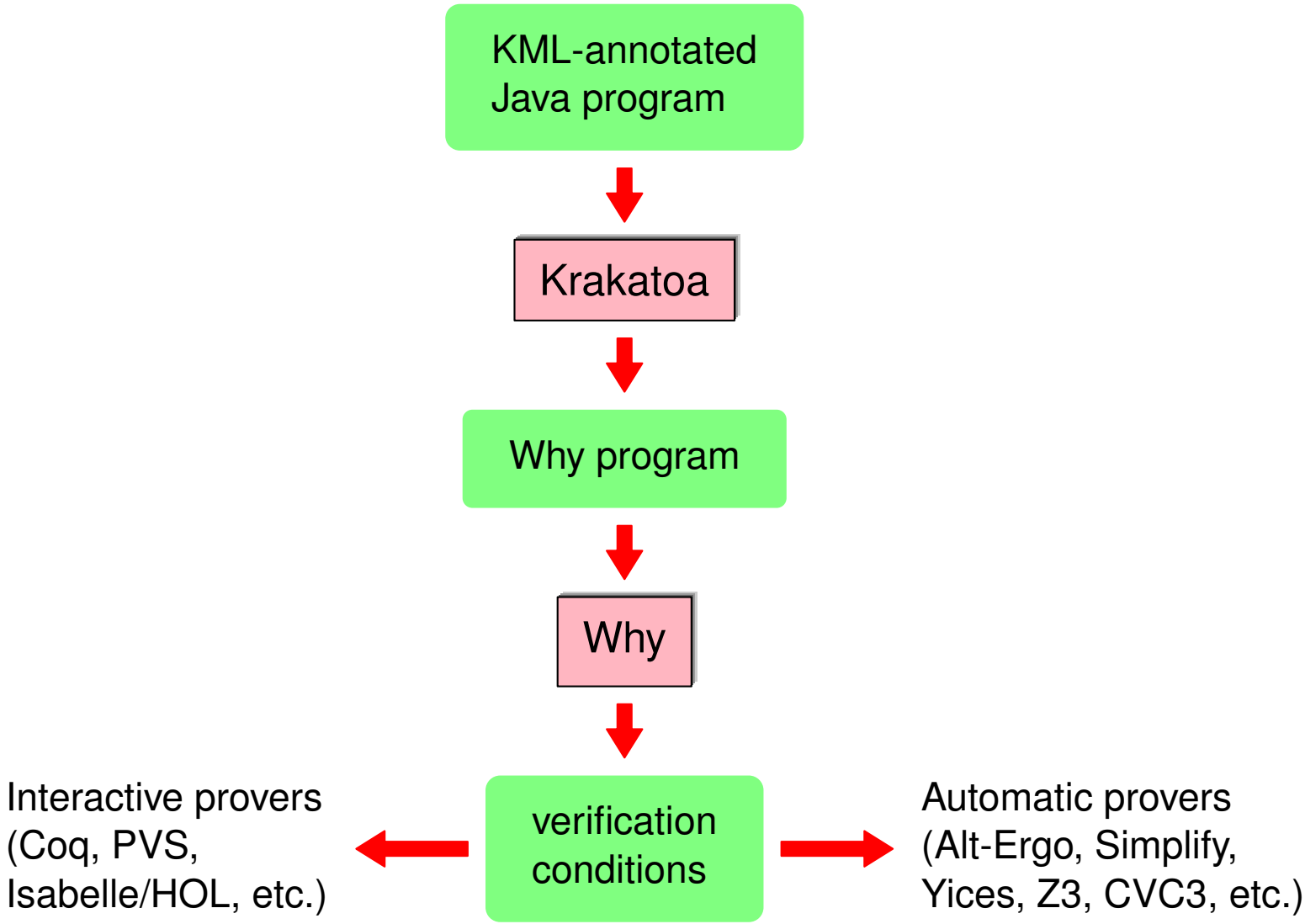
INRIA, Orsay, France

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

Outline

Outline

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 α list
logic nil:  α list
logic cons:  α, α list -> α list
logic append:  α list, α list -> α list
axiom app_nil:
    forall l:α list.  append(l,nil) = l
...
logic sumlist:  int list -> int
axiom sum_single:
    forall n:int, sumlist(cons(n,nil)) = n
...
```

Outline

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:

$$\texttt{select} : \alpha\ \texttt{memory}, \texttt{pointer} \rightarrow \alpha$$
$$\texttt{store} : \alpha\ \texttt{memory}, \texttt{pointer}, \alpha \rightarrow \alpha\ \texttt{memory}$$

$$\forall m\, p\, v,\ \texttt{select}(\texttt{store}(m, p, v), p) = v$$
$$\forall m\, p_1\, p_2\, v,\ p_1 \neq p_2 \Rightarrow \texttt{select}(\texttt{store}(m, p_1, v), p_2) = \texttt{select}(m, p_2)$$

- 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

$$\begin{aligned} &\mathit{heap} := \mathtt{store}(!\mathit{heap}, \mathit{shift}(x, f), 0);\\ &\mathit{heap} := \mathtt{store}(!\mathit{heap}, \mathit{shift}(y, g), 1);\\ &\mathtt{assert}\ \{\mathtt{select}(!\mathit{heap}, \mathit{shift}(x, f)) = 0\} \end{aligned}$$

the WP is

$$\mathtt{select}(\mathtt{store}(\mathtt{store}(\mathit{heap}, \mathit{shift}(x, f), 0), \mathit{shift}(y, g), 1), \mathit{shift}(x, f)) = 0$$

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;
```

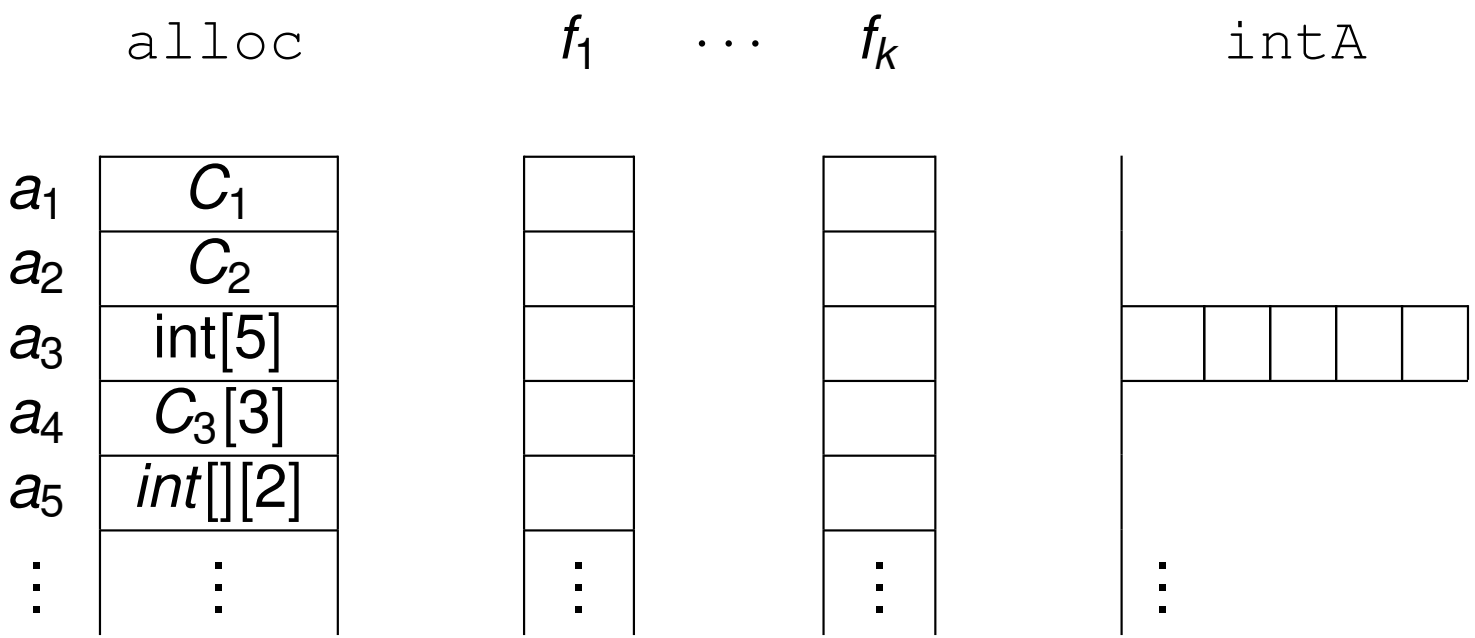
becomes

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

the WP is

$$\texttt{select}(\texttt{store}(f, x, 0), x) = 0$$

Why variables for heap memory



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

types

$$
\begin{aligned}
[\![\texttt{int}, \texttt{short}, \ldots]\!] &= \texttt{int} \\
[\![\texttt{boolean}]\!] &= \texttt{bool} \\
[\![\texttt{float}, \texttt{double}]\!] &= \texttt{real} \\
[\![\tau_{ref}]\!] &= \texttt{pointer}
\end{aligned}
$$

`pointer`: new Why type to denote Java references

Simple expressions

- constant:

$$[\![c]\!] = c$$

- binary operator:

$$[\![e_1 \ op \ e_2]\!] = [\![e_1]\!] \ op \ [\![e2]\!]$$

- local variable declaration:

$$[\![\tau \ x = e; s]\!] = \texttt{let} \ \ x = \texttt{ref} \ \ [\![e]\!] \ \ \texttt{in} \ \ [\![s]\!]$$

- local variable access:

$$[\![v]\!] = !v$$

- local variable assignment:

$$[\![v = e]\!] = v := [\![e]\!]$$

Objects fields

- field access:

$$\begin{aligned}[\![e.f]\!] \quad = \quad & \texttt{let}\ \ tmp = [\![e]\!]\ \texttt{in} \\ & \texttt{assert}\{tmp \neq \texttt{null}\}; \\ & \texttt{select}(!f, tmp)\end{aligned}$$

- field assignment:

$$\begin{aligned}[\![e_1.f = e_2]\!] \quad = \quad & \texttt{let}\ \ tmp_1 = [\![e_1]\!]\ \texttt{in} \\ & \texttt{let}\ \ tmp_2 = [\![e_2]\!]\ \texttt{in} \\ & \texttt{assert}\{tmp_1 \neq \texttt{null}\}; \\ & f := \texttt{store}(!f, tmp_1, tmp_2)\end{aligned}$$

`NullPointerException`s are ruled out

Arrays

- WHY logic function

$$\texttt{blocklength} : \texttt{alloc\_table}, \texttt{pointer} \rightarrow \texttt{int}$$

$\texttt{blocklength}(a, p)$: the size of allocated block of p in table a

- Array access:

$$\begin{aligned}
[\![e_1[e_2]]\!] \quad = \quad & \texttt{let}\ \ tmp_1 = [\![e_1]\!]\ \texttt{in} \\
& \texttt{let}\ \ tmp_2 = [\![e_2]\!]\ \texttt{in} \\
& \texttt{assert}\{tmp_1 \neq \texttt{null} \wedge \\
& \quad 0 \leq tmp_2 < \texttt{blocklength}(!alloc, tmp_1)\}; \\
& \texttt{select}(!intA, \texttt{shift}(tmp_1, tmp_2))
\end{aligned}$$

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

$$\texttt{typeof} : \texttt{alloc\_table}, \texttt{pointer} \rightarrow \texttt{tag\_id}$$

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

Casts and `instanceof`

- instanceof:

$$[\![e \texttt{ instanceof } C]\!] = \texttt{subtag}(\texttt{typeof}(!\mathit{alloc}, [\![e]\!]), C)$$

- cast expression:

$$[\![(C)e]\!] = \texttt{let } \mathit{tmp} = [\![e]\!] \texttt{ in } \texttt{assert } \{\texttt{subtag}(\texttt{typeof}(!\mathit{alloc}, \mathit{tmp}), C)\}; \mathit{tmp}$$

`ClassCastException` are ruled out

Statements

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

$$[\![\texttt{if } (e) s_1 \texttt{ else } s_2]\!] = \texttt{if } [\![e]\!] \texttt{ then } [\![s_1]\!] \texttt{ else } [\![s_2]\!]$$

- Abrupt termination:

$$[\![\texttt{break;}]\!] = \texttt{raise Break;}$$
$$[\![\texttt{continue;}]\!] = \texttt{raise Continue;}$$

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;
  @*/ τ m(x_1 : τ_1, ..., x_n : τ_n) ;
```

leads to a WHY declaration:

```
parameter C_m_parameter:
  this:pointer-> x_1 : [[τ_1]] -> ··· -> x_n : [[τ_n]] ->
  { this ≠ null ∧ [[R]] ∧ Inv(this, x_1, ..., x_n) }
  [[τ]] reads r writes w
  { [[E]] ∧ Inv(this, x_1, ..., x_n) ∧ Assigns(A) }
```

- effects r and w: *computation of effects*
- *Assigns*(A): encoding of A (computed as well)
- *Inv*(..): "naive" class invariants

Method calls

$[\![e.m(e_1, \ldots, e_n)]\!]$ is

```
let tmp = [[e]] in
let tmp_1 = [[e_1]] in
⋮
let tmp_n = [[e_n]] in
(C_m_parameter tmp tmp_1 ··· 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-> x_1 : [[τ_1]] -> ··· -> x_n : [[τ_n]] ->
  { this ≠ null ∧ [[R]] ∧ Inv(this, x_1, ..., x_n) }
  let return=ref (any_τ ()) in
  try body with Return -> !return
  { [[E]] ∧ Inv(this, x_1, ..., x_n) ∧ Assigns(A) }
```

where `return` statements are translated as:

$$[\![\texttt{return}\ e;]\!] \quad = \quad \texttt{return} := [\![e]\!]\ ; \quad \texttt{raise Return;}$$

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` $C(e_1, \ldots, e_n)$ is then translated into:

`let` $tmp = ($`alloc_parameter` $C\ 1)$ `in`
$($`cons_`C`_parameter` $tmp\ [\![e_1]\!] \cdots [\![e_n]\!])$;
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: $W = \{f, intA\}$
- $A_f = \{x, t[0..5]\}$
- $A_{intA} = \{u[3..7]\}$

General formula:

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

Sets of pointers

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

- 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

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

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

$$\texttt{pset\_union}(\texttt{pset\_singleton}(x), \texttt{pset\_range}(\texttt{pset\_singleton}(t), 0, 5))$$

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

$$\texttt{pset\_range}(\texttt{pset\_singleton}(u), 3, 7)$$

`not_assigns` predicate

Why predicate $\texttt{not\_assigns}(a, m_1, m_2, s)$:

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

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

translates to Why as

$$\mathit{Assigns}(A) = \bigwedge_{w \in W} \texttt{not\_assigns}(a, w@, w, A_w)$$

Effects: example

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

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

$$\begin{aligned}&\texttt{not\_assigns}(a, f@, f,\\ &\quad \texttt{pset\_union}(\texttt{pset\_singleton}(x),\\ &\quad\quad \texttt{pset\_range}(\texttt{pset\_singleton}(t), 0, 5)))\\ &\wedge\ \texttt{not\_assigns}(a, intA@, intA,\\ &\quad\quad \texttt{pset\_range}(\texttt{pset\_singleton}(u), 3, 7))\end{aligned}$$

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) =
  ∀ i:int.  l ≤ i ≤ h ->
    select(intA_at_L, shift(a, i)) ≤
    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

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 != y`

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

$$select(store(store(f, x, 1), y, 2), x) = 1$$

- to apply axioms of arrays:

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

we need the hypothesis `x != y`

Goal: drop this extra precondition

- *static analysis of separation*
  on the example: arrays `t1` and `t2` separated
  $\longrightarrow$ `x` and `y` separated in the call to `m`
- *Integrate separation into the component-as-array model*:
  on the example: two *regions* for `t1` and `t2`
- Why code of `m`: become

$$f_1 := \mathit{store}(f_1, x, 1); \qquad f_2 := \mathit{store}(f_2, y, 2)$$

- Weakest precondition of `x.f == 1`: become

$$\mathit{select}(\mathit{store}(f_1, x, 1), x) = 1$$

  $\longrightarrow$ provable without need of `x != 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:

$$\begin{array}{l}\text{let } m(f_x, f_y, x, y) = \\ \qquad f_x := \texttt{store}(f_x, x, 1); \quad f_y := \texttt{store}(f_y, y, 2);\end{array}$$

$$\begin{array}{l}\text{let } test() = \\ \qquad m(f_1, f_2, t_1[0], t_2[0]); \\ \qquad m(f_2, f_1, t_2[1], t_1[1]);\end{array}$$

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

$$m(f_x : (\rho_1, int)\texttt{memory}, f_y : (\rho_2, int)\texttt{memory}, x : \rho_1\ \texttt{pointer}, y : \rho_2\ \texttt{pointer})$$

- Why code generation: `e.f` interpreted as $select(f_r, e)$ when $e : r$ `pointer`

Type system

- 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

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

field access:

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

Typing of method calls

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

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

  if $id : (\tau_1, \cdots, \tau_n) \rightarrow \tau \in \Gamma$ 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

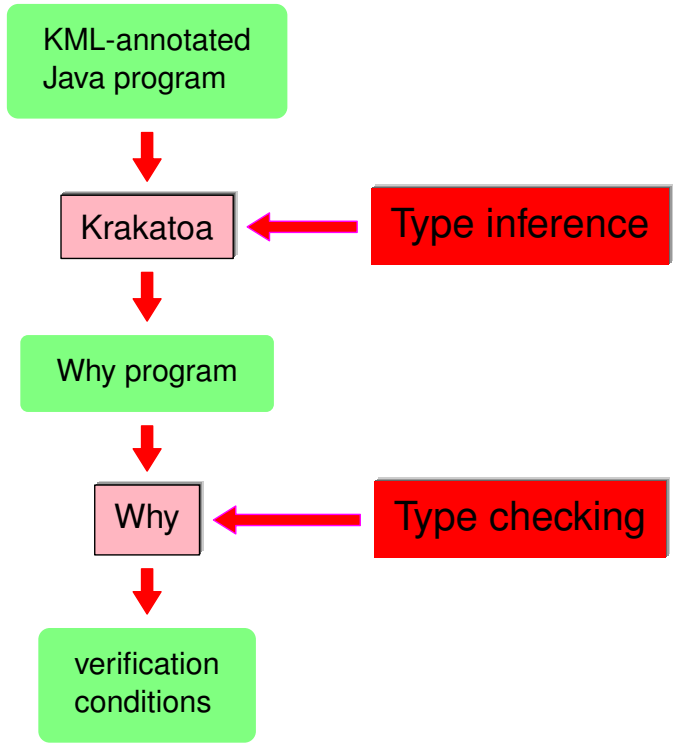
Soundness result

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

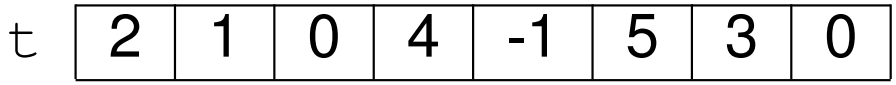
- Region types are not given, they must be inferred
- Type inference: ML-style inference based on unification
- In practice: types inferred are checked by Why:



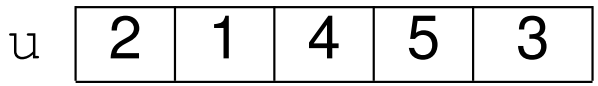
Example

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*



n=5



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} :
  @   \forall integer i j, int t[];
  @    i > j ==> num_of_pos(i,j,t) == 0;
  @
  @  axiom num_of_pos_true_case{L} :
  @   \forall 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} :
  @   \forall 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`$(intA_t, 0, i-1, t)$
{
...
$intA_u := store(intA_u, shift(u, count), select(intA_t, shift(t, i)))$
}

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

Java Card support

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