KRAKATOA

Reasoning on Java Programs

Christine Paulin-Mohring (with Claude Marché)
INRIA Futurs & Université Paris Sud, Orsay, France

Proofs of Programs and Formalisation of Mathematics
TYPES Summer School 2005

Warning

- KRAKATOA is based on the WHY tool and uses a model in COQ.
- WHY and COQ will be presented next week . . .

These lectures mainly focus on (an example of) applying type theory to programming language modeling and program verification

---

Outline

- Introduction
- Modeling JAVA
- KRAKATOA
- Conclusion
- Demo on Saturday

Lecture 1

Introduction

Motivations

Tools & methods which improve the quality of software development

Programs are :

- manipulated (compiled, executed) by a computer
- written and read by a human

We need :

- Less runtime errors
- Explicit link between documentation and code

Possible solutions

- Type-checking at compile time detects a certain class of errors and reduce the number of dynamic checks
- Many common errors are undecidable :
  - non-termination, division by zero . . .

  Abstract interpretation can help detecting certain errors
- Many more properties can be interesting for the programmer
  - an array is sorted, a linked structure does not contain cycles . . .

  Logical assertions to be proved.

How to prove programs ?

- Proving programs requires to analyse a mathematical model of the program and its specification.
- Find an apropriate model (many different semantics)
  - Denotational: mathematical functions on domains
  - Operational: execution steps
  - Axiomatic: relation between programs and properties of states
  - Monads: pure functional terms on complex data
- Proofs can be informal on paper or formal on computer

Formal proofs on computers

- Language for specifications
  - Understandable by both computers and humans
- A formal mathematical model for the specification language
- A formal correctness relation between programs and specifications
- Support for building the mathematical model of both program and specification and checking correctness

Which programming and specification language ?

- Most programming languages have complex syntax and semantics
- Semantics is not always abstractly defined but can be compiler dependent (requires a low level model of execution)
- Specification languages should be used during development and consequentely well accepted by the programmer

What about Type Theory ?

Type theory is definitely one solution:

- Programs are purely functional terms, with a natural mathematical model (strong termination)
- Dependent types are a natural specification language (can express directly properties of objects and programs)
- Curry-Howard : correctness is type-checking (of course with additional proof information)

More on this during Summer School !

The world is not yet ready to use Type Theory for programming!

What about JAVA ?

A high-level language designed for secure applications
(mobile code executed on different platforms)

- garbage collection
- strong typing at compile time
- static checking of byte-code
- dynamic checking
  - security policies (sandbox, firewall)

JAVACARD

- A subset of JAVA designed for smartcards (sequential, no dynamic loading ...)
- Additional features for smartcards : (atomic transactions, persistent data, API ...)
- JAVACARD is a good target for verification
  - simple applets ...
  - evidence of security required (Common Criteria)
  - many smartcards based on JAVACARD or similar technologies

Lecture 1

Modeling Java (JavaCard)

Outline

- More on strong typing
- Different approaches (deep versus shallow embedding)
- Our model of Java

Modeling Java

Strong typing

About strong typing

Type soundness :

ML a terminating program of type `list` evaluates to `nil` or `cons`

Java access to a field or a method of a non-null object always succeeds

Other dynamic errors may occur :

- access to fields or methods of a null object (raises NullPointerException)
- incorrect instantiation of arrays (raises ArrayStoreException)

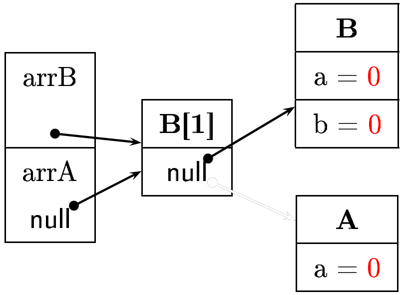
Instantiation of arrays : static view

Typing rule for arrays : $B \preceq A$ implies $B[] \preceq A[]$

```
class A { int a; }
class B extends A { int b; }
public static void main (String args[]) {
  A arrA[] ; B arrB[] = new B[1];
  arrB[0]=new B();
  arrA=arrB;
  arrA[0]=new A();
  System.out.println(arrB[0].b);
}
```

Instantiation of arrays : dynamic view

arrA[0]=new A(); raises ArrayStoreException



Modeling Java

Different approaches

Studying the Java or JavaCard platforms

Type theory is a good framework to formally study the underlying definitions, algorithms and properties.

- Type soundness
- Operational and axiomatic semantics
- Java & JavaCard virtual machines
- Byte-code verifiers
- Sandbox or Firewall mechanisms

References

Models of plaform components using proof assistants:

- Bali Project (T. Nipkow, Munich) using Isabelle/HOL
  `http://isabelle.in.tum.de/Bali/`
- Formavie project (Trusted Logic, Axalto) using Coq
  - certification at level EAL7
  - non-interference properties
- Certicartes (G. Barthe, Sophia-Antipolis) using Coq
  `http://www-sop.inria.fr/lemme/verificard/`
  Functional definition of semantics (Jakarta)

Applications

- Better understanding of semantics
- Useful for program verification
  - correct model of programs
  - identify properties valid from type-checking and properties which need logical verification
- Compilers, verifiers are programs that are likely to be written in a functional way

---

Proving a specific Java program

- Deep embedding : formalisation of the programming language (can reuse the work on platforms)
  - Abstract syntax tree formalised in the proof assistant
  - Translation from syntax to semantics done by an internal function
- Shallow embedding : direct representation of the program as a logical object
  - Programs constructions interpreted as notations
  - Translation from syntax to semantics done at the meta-level

---

Example

Concrete Syntax

$$expr ::= \texttt{var} \mid \texttt{cte} \mid expr.\texttt{field} \mid expr\ \texttt{op}\ expr$$

Semantics

Values are integers, null object or references in the heap

Example : deep embedding

Abstract syntax trees

```
type expr = Var of var | Cte of int |
            Acc of expr * field |
            Bin of  expr * op * expr
```

Values

```
type value = Int of int | Null | Ref of addr
```

Stack and heap

```
type env = var → value
type store = addr → (field → value)
```

Relational semantics

sem(s:env,h:store,e:expr,v:value) inductively defined

$$\frac{}{\mathtt{sem}(s,h,\mathtt{Var}(v),s(v))} \qquad \frac{}{\mathtt{sem}(s,h,\mathtt{Cte}(n),\mathtt{Int}(n))}$$

$$\frac{\mathtt{sem}(s,h,e,\mathtt{Ref}(a))}{\mathtt{sem}(s,h,\mathtt{Acc}(e,f),h(a,f))}$$

$$\frac{\mathtt{sem}(s,h,e1,\mathtt{Int}(n1)) \quad \mathtt{sem}(s,h,e2,\mathtt{Int}(n2))}{\mathtt{sem}(s,h,\mathtt{Bin}(e1,\mathtt{op},e2),\mathtt{Int}(\mathtt{semop}(n1,n2)))}$$

Functional semantics

sem(s:env,h:store,e:expr) :value option recursively defined

```
sem(s,h,Var(v)) = Some(s(v))
sem(s,h,Cte(n)) = Some(Int(n))
sem(s,h,Acc(e,f)) =  match sem(s,h,e) with
           None           ⇒ None
         | Some(Null)     ⇒ None
         | Some(Ref(a))   ⇒ Some(h(a,f))
         | Some(Int(n))   ⇒ None %Should not happen
sem(s,h,Bin(e1,op,e2)) =
   match sem(s,h,e1),sem(s,h,e2) with
    Some(Int(n1)),Some(Int(n2)) ⇒ Some(Int(semop(n1,n2)))
     | _                        ⇒ None
```

Shallow embedding

Can use static analysis for a more direct functional interpretation

- Expressions of static type *integer* are interpreted as logical integers
- *Objects* are interpreted as reference values

  ```
  type value = Null | Ref of addr
  ```

- Stack and heap are splitted in two parts

  ```
  type envo = var → value
  type envi = var → int
  type store = addr → (field→value) * (field→int)
  ```

Functional interpretation

$[e]^i_{si,so,h} : \mathtt{int\ option} \qquad [e]^o_{si,so,h} : \mathtt{value\ option}$

$$\begin{aligned}
&[\mathtt{n}]^i_{si,so,h} = \mathtt{Some}(\mathtt{n})\\
&[e_1\ \mathtt{op}\ e_2]^i_{si,so,h} = \mathsf{match}\ ([e_1]^i_{si,so,h}, [e_2]^i_{si,so,h})\ \mathsf{with}\\
&\quad (\mathtt{Some}(n_1), \mathtt{Some}(n_2)) \Rightarrow \mathtt{Some}(\mathtt{semop}(n_1, n_2))\ |\ \_ \Rightarrow \mathtt{None}\\
&[\mathtt{v}]^i_{si,so,h} = \mathtt{Some}(si(\mathtt{v})) \qquad [\mathtt{v}]^o_{si,so,h} = \mathtt{Some}(so(\mathtt{v}))\\
&[\mathtt{e.f}]^i_{si,so,h} = \mathsf{match}\ ([e]^o_{si,so,h})\ \mathsf{with}\\
&\quad \mathtt{Some}(\mathtt{Ref}(a)) \Rightarrow \mathsf{let}\ (\_, hi) = h(a)\ \mathsf{in}\ hi(f)\ |\ \_ \Rightarrow \mathtt{None}
\end{aligned}$$

Remarks

- Shallow embedding takes advantage of static analyses; it avoids syntactic encodings
- Dependent types allows to attach static types to expression and avoid the `value` disjoint union in deep embedding

References

- A shallow embedding of JAVA in PVS has been done in the Loop project (B. Jacobs, Nijmegen) `http://www.sos.cs.ru.nl/research/loop/`

Modeling JAVA

Formalising JAVA programs

Basic model : types and values

Classes `classId`, `Object`:`classId`

simple inheritance : `super`:`classId` →`classId option`

Types primitive types : `int`, `bool`, `float` ...

reference types : arrays indexed by types, classes.

Primitive values represented by logical values of type boolean, integer, reals ...

Reference values represented by an address (type `addr`) in the heap or the null value (type `value`)

State

An implicit set of locations containing values :

Stack Local variables, parameters

Global variables corresponding to static fields

Heap One cell for an address of an object and a field, or for the address of an array and an index

Each allocated address is associated to a tag which gives dynamic type information: object (class) or array (size, type of elements).
A table of allocations (type `store`) contains a finite set of allocated addresses with corresponding tags.

Computation

- reads and writes state, returns a value
- possible exceptional behavior
  (still returns the exceptional value and a state)
  exceptions are also useful to model control flow
  (break, continue . . .)

Idea

Java programs can be translated in a (CAML-like) language with functional values, references and exceptions.

This is what Why provides and what is used in Krakatoa.

Logical functions

Corresponding to primitive Java operations

- `arraylength` : `value` →`int`
  get information from the tag in the allocation table,
  0 as a default value
- `instanceof` : `value` →`javaType` →`bool`
  assume `super` does not generate infinite chains, uses the allocation table to look at the dynamic type of value
- `new_ref` : `value`
  `allocate` : `value` →`tag` →`unit`
  update the store

Examples with exceptions

try{ . . . throw new Exci () . . . } catch(Exc1 e){ . . . } catch(Exc2 e){ . . . }	exception JavaExc of value try{ . . . raise (JavaExc (Exci ())) . . . } with JavaExc e → if instanceof e Exc1 then . . . else if instanceof e Exc2 then . . . else raise (JavaExc e)
while (*test*) {. . . break; . . . } *code*	try while *test* do . . . raise Break . . . done with Break →(); *code*

More on the state

Functional interpretation of modifiable variables $x : \alpha$

$$x := a \;\Big|\; \lambda(x : \alpha) \mapsto a$$

Proving $P(x)$ holds after executing program p

$$\forall x. P(\tilde{p}(x))$$

Alias problem

With different variables :

$$(x, y) := (a, b) \;\Big|\; \lambda(x : \alpha)(y : \beta) \mapsto (a, b)$$

Correct when different variables correspond to different locations.
Proving $x \neq y$ after $(x, y) := (0, 1)$ is not just $0 \neq 1$

Possible solution

$$\lambda(s : \mathtt{state}) \mapsto s\{x := a[s(\xi)/\xi], y := b[s(\xi)/\xi]\}$$

Reasoning on a variable z requires analysing $s\{\xi_i := e_i\}(z)$

Memory model in Java

- Different left-values (x, $e.f$, $e[i]$) can refer to the same location
- Variables are separate locations (call by value)
- No possible conversion between basic types and references
- Different fields correspond to different locations $a.f \neq b.g$
- $a.f$ only expression for the location corresponding to a field f

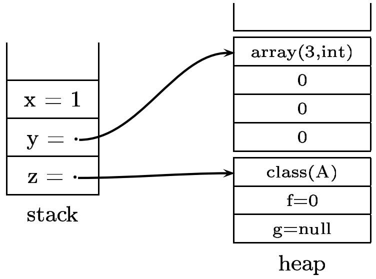
$$a.f \text{ interpreted as } \mathtt{f}[\tilde{a}]$$

with $\mathtt{f}$ a new global state variable for each field f.
Following Burstall (see also Bornat, Nipkow. . .)

Example

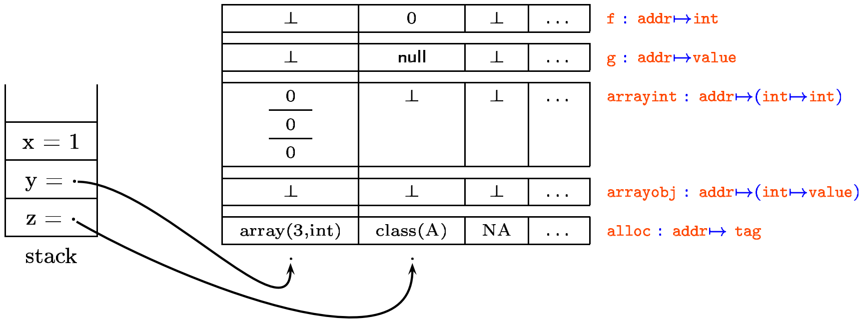
Standard Java memory model

```
class A = { int f; A g }
int x = 1;
int y[] = new int[3];
A z = new A;
```



Example: Krakatoa memory model

The heap is structured in separate maps indexed by addresses, containing primitive values or references or arrays.



⊥	0	⊥	...	`f : addr↦int`
⊥	null	⊥	...	`g : addr↦value`
0 0 0	⊥	⊥	...	`arrayint : addr↦(int↦int)`
⊥	⊥	⊥	...	`arrayobj : addr↦(int↦value)`
array(3,int)	class(A)	NA	...	`alloc : addr↦ tag`

Lecture 2

Krakatoa

Outline

How to do proofs of Java programs ?

- JML presentation
- Krakatoa architecture based on Why
- Interpreting Java/JML programs in Why
- Solving proof obligations

Krakatoa

JML presentation

JML : JAVA Modeling Language

http://www.jmlspecs.org

- Strongly related to the programming language: includes JAVA boolean expression without side effects
- Integrated to the source code : special comments, ignored by the JAVA compiler
- Different classes of specifications: pre and post conditions, class invariants, frame conditions, ghost variables …
- Special additional operators (\forall, \old, \result …)

JML example : an 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;
    }
```

Exceptional behavior

```
/*@ public behavior
      @  requires s >= 0;
      @  modifiable balance;
      @  ensures s <= \old(balance) && balance == \old(balance)-s;
      @  signals (NoCreditException)
      @           s > balance && balance == \old(balance);
      @*/
public void withdraw(int s) throws NoCreditException {
        if (balance >= s) { balance -= s; }
        else { throw new NoCreditException(); }
    }
```

Loops

```
    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;
    }
```

Tools using JML

Reference: *An overview of JML tools and applications*
Lilian Burdy, Yoonsik Cheon, David Cok, Michael Ernst, Joe Kiniry,
Gary T. Leavens, K. Rustan M. Leino, and Erik Poll. (STTT, 2005).

- Documentation (jmldoc), test (jmlunit)
- Dynamic checking (defensive code) (jmlc, jass)
- Partial automatic verification (ESC/Java(2), Chase)
- Total interactive verification (Loop, JIVE, Jack, Krakatoa)

Also JML specification of JAVACARD API (E. Poll, Nijmegen)

KRAKATOA

Architecture based on WHY

The WHY tool

A generic language for proving annotated programs
J.-C. Filliâtre, `http://why.lri.fr`

- Specification : multi-sorted predicate logic
- Body of programs : functions, references, exceptions, labels, assertions . . .
- Signature of programs : extended with pre & post-conditions, + effects (read & written variables, exceptions)

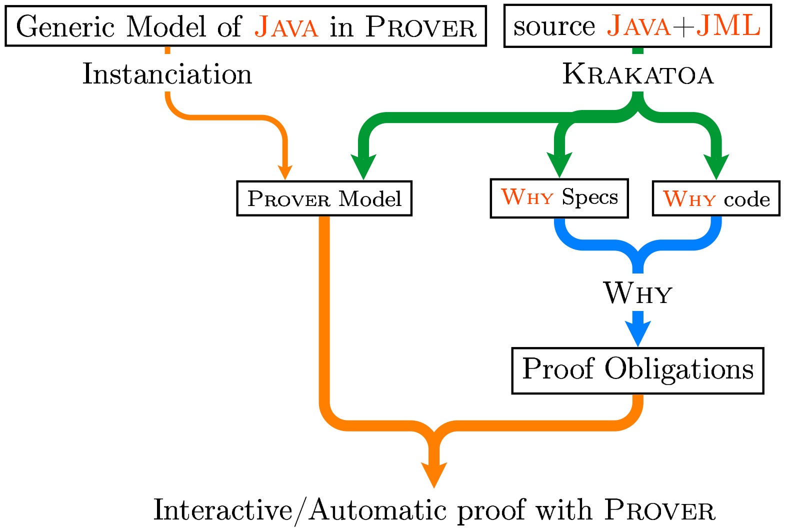
WHY advantages

- A modular view of programs and specifications
- Generates sufficient proof obligations (pre, post, assertions)
- Proof obligations generated for interactive or automatic theorem provers : PVS, Coq, HOL, Mizar, Simplify, haRVey. . .

KRAKATOA approach

- Model the JAVA program (see before)
- Model the JML specification
- Translate JAVA/JML programs into WHY annotated programs (preserving semantics)
- Proof that the program meets its specification by generating proof obligations in WHY

KRAKATOA general architecture


Generic Model of JAVA in PROVER

source JAVA+JML

Instanciation

KRAKATOA

PROVER Model

WHY Specs

WHY code

WHY

Proof Obligations

Interactive/Automatic proof with PROVER

KRAKATOA

WHY model of programs

WHY parametric theory

```
parameter alloc : store ref
parameter alloc_new_obj : (c:classId) → { }
   value reads alloc writes alloc
   { result≠ Null and fresh(alloc@, result)
     and typeof(alloc, result, ClassType(c))
     and store_extends(alloc@,alloc)}

external logic fresh : store, value → prop
external logic store_extends : store, store → prop
external logic Null : → value
external logic ClassType : classId → javaType
```

Body of programs

```
external parameter new_ref : store  → value
external parameter allocate : store → value → tag → store
external parameter Obj : classId → tag

let alloc_new_obj = fun (c:classId) →  { }
   let this = new_ref !alloc in
   begin alloc := allocate !alloc this (Obj c); this end
   { result≠ Null
     and fresh(alloc@, result)
     and typeof(alloc, result, ClassType(c))
     and store_extends(alloc@,alloc)
   }
```

Translation of expressions

Conditions to protect access and avoid runtime exceptions

e.f	{e≠Null} (acc !f e)
e.f=v	{e≠Null} f:=(update !f e v)
e[i]	{e≠Null ∧ 0≤i<(arraylength alloc e)} (array_acc !arrayint e i)
e[i]=v	{e≠Null ∧ 0≤i<(arraylength alloc e) ∧ instanceof alloc v (arrayelemtype alloc e)} arrayobj:=(array_update !arrayobj e i v)

Handling methods

- Find a WHY specification for each JAVA method
  - Computes which variables are read or written (field variables, array variables, alloc . . .)
  - Transforms the JML specification into pre/post conditions
- Keep a local and modular approach
- Handle partial correctness of recursive methods

WHY specification for methods

```
parameter Purse_credit_parameter :
 this:value → s:int →
   { s ≥ 0 ∧ this ≠ Null
     ∧ instanceof(alloc,this,ClassType(Purse))
     ∧ Purse_invariant(Purse_balance,this)}
   unit reads Purse_balance,alloc  writes Purse_balance
   { acc(Purse_balance,this)=acc(Purse_balance@,this)+s
      ∧ Purse_invariant(Purse_balance,this)
      ∧ modifiable(alloc@,Purse_balance@,Purse_balance,
                           value_loc(this))}
```

KRAKATOA

Solving proof obligations

The corresponding Coq theory

```
Inductive tag:Set := Obj: classId→ tag | Arr: N→ kind→ tag.
Definition store := (fmap.t tag).

Definition alive (h:store) (v:value) :=
  match v with  Null => True | Ref a => find h a ≠ None end.

Definition store_extends (h h':store) :=
  ∀ v:value, alive h v → tag_of h v = tag_of h' v.

Lemma typeof_extends_stable :
∀ (h h':store) (t:javaType) (v:value),
   typeof h v t → store_extends h h' → typeof h' v t.
```

Frame condition modeling

```
Variable A : Set
Definition memory := map.t value A.

Definition mod_loc := value → Prop.
Definition unchanged (ml:mod_loc) (v:value) := ml v.

Definition modifiable (h:store) (m m':memory) (ml:mod_loc) :=
  ∀v:value,alive h v → unchanged ml v → acc m v = acc m' v.

Definition value_loc (v: value) : mod_loc := fun w ⇒ v ≠ w.
Lemma value_loc_intro :
   ∀v1 v:value, v1 ≠ v -> unchanged (value_loc v1) v.
```

Coq theory generated for a particular program

```
Inductive classId : Set :=
  Object : classId | Math : classId | Purse : classId ...

Definition super (i:classId) : option classId :=
  match i with
  | Object => None  | Math => Some Object
  | Purse  => Some Object | ...
  end

Definition Purse_invariant (Purse_balance:memory Z) (this:value)
:= (acc Purse_balance this) >= 0.
```

Automatic proofs

- Extract an axiomatic first-order theory from the Coq model
- Use an automatic prover (mainly Simplify) in order to validate proof obligations

Good results on small programs (sorting, sets, purse . . .)

Lecture 2

Conlusion

Related work

Tools with similar goals

- ESC/Java (Compaq) : only partial correctness, errors
- KeY (Chalmers, Karlsruhe) : UML specification, dynamic logic
- LOOP (Nijmegen) : shallow embedding in PVS
- JIVE (Hagen): ad-hoc axiomatic semantics, global memory, interface
- Jack (Gemplus, INRIA) : obligations originally for the B prover, nice interface

---

Remarks on Krakatoa

A good combination of known techniques

- A rigorous approach
- Specification and proofs are integrated in real programs
- Proofs are partly automated
- Experimented on two JavaCard applets

A very preliminary tool under development

- Many important features of Java are not (yet) covered
- The interface is not really user-friendly

Choice of architecture

- An open-source system
- Each step of translation is readable
- WHY language (functions, references and exceptions) is a powerful language for representing operational semantics
- The same architecture can be used for other input programming languages:
  CADUCEUS for C, J.-C. Filliâtre & C. Marché
- The best of each theorem provers can be used (even combined)

More on specifications

Writing apropriate specifications can be as hard as writting programs and proofs . . .

The tool should help you in this process

How convenient are JML specifications ?

- Some relations are not easily defined by pure JAVA programs but would be naturally specified inductively.
  Example:A linked structure does not contains loops
- Global security properties :
  - Security automata : control the correct sequences of method calls
  - Non interference properties : we cannot infer secret information from looking at public variables

Can be checked using JAVA/JML technology
(Everest project, Sophia-Antipolis)

That is the end . . .

See the demo on Saturday!