Winter School on Object-Oriented Verification
Viinistu, Estonia
25-29 January 2009

The Krakatoa tool for Deductive Verification of Java Programs

Claude Marché
INRIA Saclay – F-91893 Orsay cedex

The chapter on the Why VC generator is contributed by
Jean-Christophe Filliâtre
CNRS & Université Paris Sud – F-91405 Orsay cedex

Contents

Introduction

The Krakatoa tool is part of the Why platform for deductive program verification [11], developed by the ProVal research group (`http://proval.lri.fr`). The platform is available as open source software at `http://why.lri.fr/`. Krakatoa is the front-end dedicated to Java source code. There is another front-end called Caduceus dedicated to C source code.

The general approach is to annotate source by formal specifications (given in a special kind of comments) and then to generate *Verification Conditions* (VCs for short): logical formulas whose validity implies the soundness of the code with respect to the given specifications. This includes automatically generated VCs to guarantee the absence of run-time errors: null pointer dereferencing, out-of-bounds array access, etc. Then the VCs can be discharged using one or several theorem provers. The main originality of this platform is that a large part is common to C and Java. In particular there is a unique, stand-alone, VCs generator called Why, which is able to output VCs in the native syntax of many provers, either automatic or interactive ones. The overall architecture is presented on Figure 1.

Frama-C is a general framework for static analysis of C code (`http://frama-c.cea.fr`) developed jointly by CEA List and the ProVal team. Jessie is another intermediate language, aiming at factorizing various analyses and transformations common to C and Java.

These lecture notes summarize the underlying theory of

1. The *Verification Condition Generator* (Chapter 1). This chapter is a contribution of Jean-Christophe Filliâtre, written for the Types Summer School in 2007 (`http://www.lri.fr/~filliatr/types-summer-school-2007/`)

2. The translation from Java to the Why dedicated input language, performed by Krakatoa (Chapter 2).

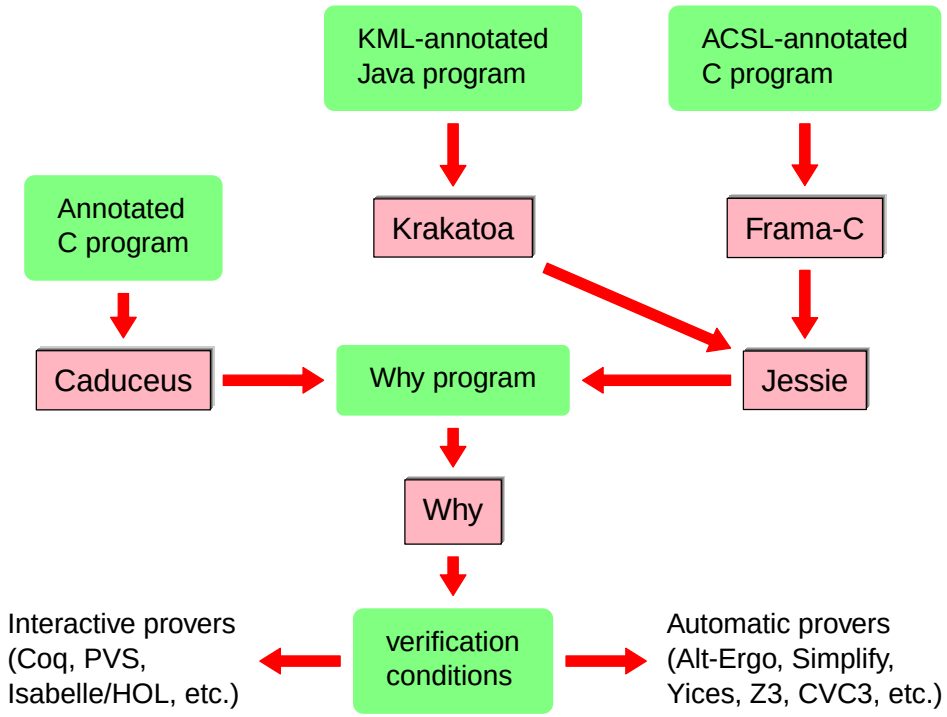

Figure 1: Why platform Architecture

Chapter 1

Why: an Intermediate Language for Program Verification

This chapter is an introduction to the Why tool. This tool implements a programming language designed for the verification of sequential programs. This is an intermediate language to which existing programming languages can be compiled and from which verification conditions can be computed.

These notes aim at introducing the theory behind the Why tool: syntax, typing, semantics and weakest preconditions for its language.

Implementing a verification condition generator (VCG) for a realistic programming language such as C is a lot of work. Each construct requires a specific treatment and there are many of them. Though, almost all rules will end up to be instances of the five historical Hoare Logic rules [13]. Reducing the VCG to a core language thus seems a good approach. Similarly, if one has written a VCG for C and has to write another one for Java, there are clearly enough similarities to hope for this core language to be reused. Last, if one has to experiment with several logics, models and/or proof tools, this core language should ideally remain the same.

The Why tool implements such an intermediate language for VCGs, that we call HL in the following (for *Hoare Language*). Syntax, typing, semantics and weakest preconditions calculus are given below, but we first start with a tour of HL features.

Genericity. HL annotations are written in a first-order predicate syntax but are not interpreted at all. This means that HL is independent of the underlying logic in which the annotations are interpreted. The WP calculus only requires the logic to be minimal i.e. to include universal quantification, conjunction and implication.

ML syntax. HL has an ML-like syntax where there is no distinction between expressions and statements. This greatly simplifies the language—not only the syntax but also the typing and semantics. However HL has few in common with the ML family languages (functions are not first-class values, there is no polymorphism, no type inference, etc.)

Aliases. HL is an alias-free language. This is ensured by the type checking rules. Being alias free is crucial for reasoning about programs, since the rule for assignment

$$\{P[x \leftarrow E]\}\, x := E\, \{P\}$$

implicitly assumes that any variable other than x is left unmodified. Note however that the absence of alias in HL does not prevent the interpretation of programs with possible aliases: such programs can be interpreted using a more or less complex memory model made of several unaliased variables.

Exceptions. Beside conditional and loop, HL only has a third kind of control statement, namely exceptions. Exceptions can be thrown from any program point and caught anywhere upper in the control-flow. Arbitrary many exceptions can be declared and they may carry values. Exceptions can be used to model exceptions from the source language (e.g. Java's exceptions) but also to model all kinds of abrupt statements (e.g. C and Java's `return`, `break` or `continue`).

Typing with effects. HL has a typing with effects: each expression is given a type together with the sets of possibly accessed and possibly modified variables and the set of possibly raised exceptions. Beside its use for the alias check, this is the key to modularity: one can declare and use a function without implementing it, since its type mentions its side-effects. In particular, the WP rule for function call is absolutely trivial.

Auxiliary variables. The usual way to relate the values of variables at several program points is to used the so-called *auxiliary variables*. These are variables only appearing in annotations and implicitly universally quantified over the whole Hoare triple. Though auxiliary variables can be given a formal meaning [24] their use is cumbersome in practice:

they pollute the annotations and introduce unnecessary equality reasoning on the prover side. Instead we propose the use of program *labels*—similar to those used for `gotos`—to refer to the values of variables at specific program points. This appears to be a great improvement over auxiliary variables, without loss of expressivity.

1.1 Syntax

1.1.1 Types and Specifications

Program annotations are written using the following minimal first-order logic:

$$\begin{array}{lcl} t & ::= & c \mid x \mid \,!\,x \mid \phi(t,\ldots,t) \mid \texttt{old}(t) \mid \texttt{at}(t,L) \\ p & ::= & P(t,\ldots,t) \mid \forall x:\beta.p \mid p \Rightarrow p \mid p \wedge p \mid \ldots \end{array}$$

A term t can be a constant c, a variable x, the contents of a reference x (written $!\,x$) or the application of a function symbol ϕ. It is important to notice that ϕ is a function symbol belonging to the logic: it is not defined in the program. The construct $\texttt{old}(t)$ denotes the value of term t in the precondition state (only meaningful within the corresponding postcondition) and the construct $\texttt{at}(t, L)$ denotes the value of the term t at the program point L (only meaningful within the scope of a label L).

We assume the existence of a set of *pure types* (β) in the logical world, containing at least a type `unit` with a single value `void` and a type `bool` for booleans with two values `true` and `false`.

Predicates necessarily include conjunction, implication and universal quantification as they are involved in the weakest precondition calculus. In practice, one is likely to add at least disjunction, existential quantification, negation and true and false predicates. An atomic predicate is the application of a predicate symbol P and is not interpreted. For the forthcoming WP calculus, it is also convenient to introduce an `if-then-else` predicate:

$$\begin{array}{l} \texttt{if}\ t\ \texttt{then}\ p_1\ \texttt{else}\ p_2 \ \equiv \\ \quad (t = \texttt{true} \Rightarrow p_1) \wedge (t = \texttt{false} \Rightarrow p_2) \end{array}$$

Program types and specifications are classified as follows:

$$\begin{array}{lcl} \tau & ::= & \beta \mid \beta\ \mathsf{ref} \mid (x:\tau) \rightarrow \kappa \\ \kappa & ::= & \{p\}\,\tau\,\epsilon\,\{q\} \\ q & ::= & p; E \Rightarrow p; \ldots; E \Rightarrow p \\ \epsilon & ::= & \texttt{reads}\ x,\ldots,x\ \texttt{writes}\ x,\ldots,x \\ & & \texttt{raises}\ E,\ldots,E \end{array}$$

A value of type τ is either an immutable variable of a pure type (β), a reference containing a value of a pure type (β ref) or a function of type $(x:\tau) \rightarrow \{p\}\,\beta\,\epsilon\,\{q\}$ mapping the formal parameter x to the specification of its body, that is a precondition p, the type τ for the returned value, an effect ϵ and a postcondition q. An effect is made of tree lists of variables: the references possibly accessed (`reads`), the references possibly modified (`writes`) and the exceptions possibly raised (`raises`). A postcondition q is made of several parts: one for the normal termination and one for each possibly raised exception (E stands for an exception name).

When a function specification $\{p\}\,\beta\,\epsilon\,\{q\}$ has no precondition and no postcondition (both being `true`) and no effect (ϵ is made of three empty lists) it can be shortened to τ. In particular, $(x_1:\tau_1) \rightarrow \cdots \rightarrow (x_n:\tau_n) \rightarrow \kappa$ denotes the type of a function with n arguments that has no effect as long as it not applied to n arguments. Note that functions can be partially applied.

1.1.2 Expressions

The syntax for program expressions is the following:

$$\begin{array}{lcl} t & ::= & c \mid x \mid \,!\,x \mid \phi\,(t,\ldots,t) \\ e & ::= & t \\ & \mid & x := e \\ & \mid & \texttt{let}\ x = e\ \texttt{in}\ e \\ & \mid & \texttt{let}\ x = \texttt{ref}\ e\ \texttt{in}\ e \\ & \mid & \texttt{if}\ e\ \texttt{then}\ e\ \texttt{else}\ e \\ & \mid & \texttt{loop}\ e\ \{\texttt{invariant}\ p\ \texttt{variant}\ t\} \\ & \mid & L\texttt{:}e \\ & \mid & \texttt{raise}\ (E\ e) : \tau \\ & \mid & \texttt{try}\ e\ \texttt{with}\ E\ x \rightarrow e\ \texttt{end} \\ & \mid & \texttt{assert}\ \{p\};\ e \\ & \mid & e\ \{q\} \\ & \mid & \texttt{fun}\ (x:\tau) \rightarrow \{p\}\ e \\ & \mid & \texttt{rec}\ x\ (x:\tau)\ldots(x:\tau) : \beta \\ & & \ \{\texttt{variant}\ t\} = \{p\}\ e \\ & \mid & e\ e \end{array}$$

In particular, programs contain *pure terms* (t) made of constants, variables, dereferences (written $!\,x$) and application of function symbols from the logic to pure terms. The syntax mostly follows ML's one. `ref` e introduces a new reference initialized with e. `loop` e $\{$`invariant` p `variant` $t\}$ is an infinite loop of body e, invariant p and which termination is ensured by the variant t. The `raise` construct is annotated with a type τ since there is no polymorphism in HL. There are two ways to insert proof obligations in programs: `assert` $\{p\};\ e$ places an assertion p to be checked right before e and $e\ \{q\}$ places a postcondition q to be checked right after e.

The traditional sequence construct is only syntactic sugar for a `let-in` binder where the variable does not occur in e_2:

$$e_1\texttt{;}\ e_2 \ \equiv\ \texttt{let}\ \_ = e_1\ \texttt{in}\ e_2$$

We also simplify the `raise` construct whenever both the exception contents and the whole `raise` expression

have type `unit`:

$$\texttt{raise}\ E\ \equiv\ \texttt{raise}\ (E\ \texttt{void}) : \texttt{unit}$$

The traditional `while` loop is also syntactic sugar for a combination of an infinite loop and the use of an exception *Exit* to exit the loop:

$$\begin{array}{l}\texttt{while}\ e_1\ \texttt{do}\ e_2\ \{\texttt{invariant}\ p\ \texttt{variant}\ t\}\ \equiv\\ \quad\texttt{try}\\ \quad\quad\texttt{loop if}\ e_1\ \texttt{then}\ e_2\ \texttt{else raise}\ \mathit{Exit}\\ \quad\quad\{\texttt{invariant}\ p\ \texttt{variant}\ t\}\\ \quad\texttt{with}\ \mathit{Exit}\ \texttt{\_ -> void end}\end{array}$$

1.1.3 Functions and Programs

A program (p) is a list of declarations. A declaration (d) is either a definition introduced with `let` or a declaration introduced with `val`, or an exception declaration.

$$\begin{array}{lcl} p & ::= & \emptyset \mid d\ p \\ d & ::= & \texttt{let}\ x = e \\ & \mid & \texttt{val}\ x : \tau \\ & \mid & \texttt{exception}\ E\ \texttt{of}\ \beta \end{array}$$

1.2 Typing

This section introduces typing and semantics for HL.

Typing environments contain bindings from variables to types of values, exceptions declarations and labels:

$$\Gamma \quad ::= \quad \emptyset \mid x : \tau, \Gamma \mid \texttt{exception}\ E\ \texttt{of}\ \beta, \Gamma \mid \texttt{label}\ L, \Gamma$$

The type of a constant or a function symbol is given by the operator *Typeof*. A type τ is said to be *pure*, and we write τ `pure`, if it is not a reference type. We write $x \in \tau$ whenever the reference x appears in type τ i.e. in any annotation or effect within τ.

An effect is composed of three sets of identifiers. When there is no ambiguity we write (r, w, e) for the effect `reads` r `writes` w `raises` e. Effects compose a natural semi-lattice of bottom element $\bot = (\emptyset, \emptyset, \emptyset)$ and supremum $(r_1, w_1, e_1) \sqcup (r_2, w_2, e_2) = (r_1 \cup r_2, w_1 \cup w_2, e_1 \cup e_2)$. We also define the erasing of the identifier x in effect $\epsilon = (r, w, e)$ as $\epsilon \backslash x = (r \backslash \{x\}, w \backslash \{x\}, e \backslash \{x\})$.

We introduce the typing judgment $\Gamma \vdash e : (\tau, \epsilon)$ with the following meaning: in environment Γ the expression e has type τ and effect ϵ. Typing rules are given in Figure 1.1. They assume the definitions of the following extra judgments:

- $\Gamma \vdash \kappa$ `wf` : the specification κ is well formed in environment Γ,
- $\Gamma \vdash p$ `wf` : the precondition p is well formed in environment Γ,
- $\Gamma \vdash q$ `wf` : the postcondition q is well formed in environment Γ,
- $\Gamma \vdash t : \beta$: the logical term t has type β in environment Γ.

The purpose of this typing with effects is two-fold. First, it rejects aliases: it is not possible to bind one reference variable to another reference, neither using a `let in` construct, nor a function application. Second, it will be used when interpreting programs in Type Theory (in Section 1.5 below).

1.3 Semantics

We give a big-step operational semantics to HL. The notions of values and states are the following:

$$\begin{array}{lcl} v & ::= & c \mid E\ c \mid \texttt{rec}\ f\ x = e \\ s & ::= & \{(x, c), \ldots, (x, c)\} \end{array}$$

A value v is either a constant value (integer, boolean, etc.), an exception E carrying a value c or a closure `rec` $f\ x = e$ representing a possibly recursive function f binding x to e. For the purpose of the semantic rules, it is convenient to add the notion of closure to the set of expressions:

$$e\ ::=\ \ldots \mid \texttt{rec}\ f\ x = e$$

In order to factor out all semantic rules dealing with uncaught exceptions, we introduce the following set of contexts R:

$$\begin{array}{lcl} R & ::= & [] \mid x\ \texttt{:=}\ R \mid \texttt{let}\ x = R\ \texttt{in}\ e \\ & \mid & \texttt{let}\ x = \texttt{ref}\ R\ \texttt{in}\ e \\ & \mid & \texttt{if}\ R\ \texttt{then}\ e\ \texttt{else}\ e \\ & \mid & \texttt{loop}\ R\ \{\texttt{invariant}\ p\ \texttt{variant}\ t\} \\ & \mid & \texttt{raise}\ (E\ R) : \tau \\ & \mid & R\ e \end{array}$$

The semantics rules are given Figure 1.2.

1.4 Weakest Preconditions

Programs correctness is defined using a calculus of weakest preconditions. We note $wp(e, q; r)$ the weakest precondition for a program expression e and a postcondition $q; r$ where q is the property to hold when terminating normally and $r = E_1 \Rightarrow q_1; \ldots; E_n \Rightarrow q_n$ is the set of properties to hold for each possibly uncaught exception. Expressing the correctness of a program e is simply a matter of computing $wp(e, \mathsf{True})$.

The rules for the basic constructs are given on the first part of Figure 1.3. On the traditional constructs of Hoare logic, these rules simplify to the well known identities. For instance, the case of the assignment of a side-effect free expression gives

$$wp(x\ \texttt{:=}\ t, q) = q[x \leftarrow t]$$

$$\frac{\mathit{Typeof}(c) = \beta}{\Gamma \vdash c : (\beta, \bot)} \qquad \frac{x : \tau \in \Gamma \qquad \tau \ \texttt{pure}}{\Gamma \vdash x : (\tau, \bot)} \qquad \frac{x : \beta \ \mathsf{ref} \in \Gamma}{\Gamma \vdash !x : (\beta, \texttt{reads}\ x)}$$

$$\Gamma \vdash t_i : (\beta_i, \epsilon_i$$

$$\frac{}{s, c \longrightarrow s, c} \qquad \frac{}{s, \,!\,x \longrightarrow s, s(x)} \qquad \frac{s, t_i \longrightarrow s, c_i}{s, \phi\,(t_1, \ldots, t_n) \longrightarrow s, \phi\,(c_1, \ldots, c_n)}$$

$$\frac{s, e \longrightarrow s', E\ c}{s, R[e] \longrightarrow s', E\ c} \qquad \frac{s, e \longrightarrow s', c}{s, x := e \longrightarrow s' \oplus \{x \mapsto c\}, \texttt{void}}$$

$$\frac{s, e_1 \longrightarrow s_1, v_1 \quad v_1 \text{ not exc.} \quad s_1, e_2[x \leftarrow v_1] \longrightarrow s_2, v_2}{s, \texttt{let}\ x = e_1\ \texttt{in}\ e_2 \longrightarrow s_2, v_2}$$

$$\frac{s, e_1 \longrightarrow s_1, c_1 \quad s_1 \oplus \{x \mapsto c_1\}, e_2 \longrightarrow s_2, v_2}{s, \texttt{let}\ x = \texttt{ref}\ e_1\ \texttt{in}\ e_2 \longrightarrow s_2, v_2}$$

$$\frac{s, e_1 \longrightarrow s_1, \texttt{true} \quad s_1, e_2 \longrightarrow s_2, v_2}{s, \texttt{if}\ e_1\ \texttt{then}\ e_2\ \texttt{else}\ e_3 \longrightarrow s_2, v_2} \qquad \frac{s, e_1 \longrightarrow s_1, \texttt{false} \quad s_1, e_3 \longrightarrow s_3, v_3}{s, \texttt{if}\ e_1\ \texttt{then}\ e_2\ \texttt{else}\ e_3 \longrightarrow s_3, v_3}$$

$$\frac{s, e \longrightarrow s', \texttt{void} \quad s', \texttt{loop}\ e\ \{\texttt{invariant}\ p\ \texttt{variant}\ t\} \longrightarrow s'', v}{s, \texttt{loop}\ e\ \{\texttt{invariant}\ p\ \texttt{variant}\ t\} \longrightarrow s'', v}$$

$$\frac{s, e \longrightarrow s', v}{s, L\texttt{:}e \longrightarrow s', v} \qquad \frac{s, e \longrightarrow s', c}{s, \texttt{raise}\ (E\ e) : \tau \longrightarrow s', E\ c}$$

$$\frac{s, e_1 \longrightarrow s_1, E'\ c \quad E' \neq E}{s, \texttt{try}\ e_1\ \texttt{with}\ E\ x \rightarrow e_2\ \texttt{end} \longrightarrow s_1, E'\ c}$$

$$\frac{s, e_1 \longrightarrow s_1, E\ c \quad s_1, e_2[x \leftarrow c] \longrightarrow s_2, v_2}{s, \texttt{try}\ e_1\ \texttt{with}\ E\ x \rightarrow e_2\ \texttt{end} \longrightarrow s_2, v_2} \qquad \frac{s, e_1 \longrightarrow s_1, v_1 \quad v_1 \text{ not exc.}}{s, \texttt{try}\ e_1\ \texttt{with}\ E\ x \rightarrow e_2\ \texttt{end} \longrightarrow s_1, v_1}$$

$$\frac{s, e \longrightarrow s', v}{s, \{p\}\ e \longrightarrow s', v} \qquad \frac{s, e \longrightarrow s', v}{s, e\ \{q\} \longrightarrow s', v}$$

$$\frac{}{s, \texttt{fun}\ (x : \tau) \rightarrow \{p\}\ e \longrightarrow s, \texttt{rec}\ \_\ x = e}$$

$$\frac{}{\begin{array}{c} s, \texttt{rec}\ f\ (x_1 : \tau_1) \ldots (x_n : \tau_n) : \tau\ \{\texttt{variant}\ t\} = \{p\}\ e \longrightarrow \\ s, \texttt{rec}\ f\ x_1 = \texttt{rec}\ \_\ x2 = \ldots \texttt{rec}\ \_\ x_n = e \end{array}}$$

$$\frac{s, e_1 \longrightarrow s_1, \texttt{rec}\ f\ x = e \quad s_1, e_2 \longrightarrow s_2, v_2 \quad s_2, e[f \leftarrow \texttt{rec}\ f\ x = e, x \leftarrow v_2] \longrightarrow s_3, v}{e_1\ e_2 \longrightarrow s_3, v}$$

Figure 1.2: Semantics

$$
\begin{aligned}
wp(t, q; r) &= q[result \leftarrow t] \\
wp(x := e, q; r) &= wp(e, q[result \leftarrow \texttt{void}; x \leftarrow result]; r) \\
wp(\texttt{let } x = e_1 \texttt{ in } e_2, q; r) &= wp(e_1, wp(e_2, q; r)[x \leftarrow result]; r) \\
wp(\texttt{let } x = \texttt{ref } e_1 \texttt{ in } e_2, q; r) &= wp(e_1, wp(e_2, q; r)[x \leftarrow result]; r) \\
wp(\texttt{if } e_1 \texttt{ then } e_2 \texttt{ else } e_3, q; r) &= wp(e_1, \texttt{if } result \texttt{ then } wp(e_2, q; r) \texttt{ else } wp(e_3, q; r); r) \\
wp(L\texttt{:}e, q; r) &= wp(e, q; r)[\texttt{at}(x, L) \leftarrow x]
\end{aligned}
$$

$$
\begin{aligned}
wp(\texttt{raise } (E\ e) : \tau, q; r) &= wp(e, r(E); r) \\
wp(\texttt{try } e_1 \texttt{ with } E\ x \rightarrow e_2 \texttt{ end}, q; r) &= wp(e_1, q; E \Rightarrow wp(e_2, q; r)[x \leftarrow result]; r) \\
wp(\texttt{assert } \{p\};\ e, q; r) &= p \wedge wp(e, q; r) \\
wp(e\ \{q'; r'\}, q; r) &= wp(e, q' \wedge q; r' \wedge r)
\end{aligned}
$$

Figure 1.3: WP computation rules

and the case of a (exception free) sequence gives

$$wp(e_1\texttt{;}\ e_2, q) = wp(e_1, wp(e_2, q))$$

The cases of exceptions and annotations are also straightforward, given on second part of Figure 1.3. The case of an infinite loop is more subtle:

$$
\begin{aligned}
&wp(\texttt{loop } e\ \{\texttt{invariant } p\ \texttt{variant } t\}, q; r) = \\
&\quad p\ \wedge\ \forall\omega.\, p \Rightarrow wp(L\texttt{:}e, p \wedge t < \texttt{at}(t, L); r)
\end{aligned}
$$

where ω stands for the set of references possibly modified by the loop body (the `writes` part of e's effect). Here the weakest precondition expresses that the invariant must hold initially and that for each turn in the loop (represented by ω), either p is preserved by e and e decreases the value of t (to ensure termination), or e raises an exception and thus must establish r directly.

By combining this rule and the rule for the conditional, we can retrieve the rule for the usual while loop:

$$
\begin{aligned}
&wp(\texttt{while } e_1 \texttt{ do } e_2\ \{\texttt{invariant } p\ \texttt{variant } t\}, q; r) \\
&= p\ \wedge\ \forall\omega.\, p \Rightarrow \\
&\quad wp(L\texttt{:if } e_1 \texttt{ then } e_2 \texttt{ else raise } E, \\
&\qquad p \wedge t < \texttt{at}(t, L), E \Rightarrow q; r) \\
&= p\ \wedge\ \forall\omega.\, p \Rightarrow \\
&\quad wp(e_1, \\
&\qquad \texttt{if } result \texttt{ then } wp(e_2, p \wedge t < \texttt{at}(t, L)) \texttt{ else } q, \\
&\qquad r)[\texttt{at}(x, L) \leftarrow x]
\end{aligned}
$$

Finally, we give the rules for functions and function calls. Since a function cannot be mentioned within the postcondition, the weakest preconditions for function constructs `fun` and `rec` are only expressing the correctness of the function body:

$$
\begin{aligned}
&wp(\texttt{fun } (x : \tau) \rightarrow \{p\}\ e, q; r) = \\
&\quad q\ \wedge\ \forall x.\forall\rho.p \Rightarrow wp(e, \mathsf{True})
\end{aligned}
$$

$$
\begin{aligned}
&wp(\texttt{rec } f\ (x_1 : \tau_1) \ldots (x_n : \tau_n) : \tau \\
&\qquad \{\texttt{variant } t\} = \{p\}\ e, q; r) = \\
&\quad q\ \wedge\ \forall x_1. \ldots \forall x_n.\forall\rho.p \Rightarrow wp(L\texttt{:}e, \mathsf{True})
\end{aligned}
$$

where ρ stands for the set of references possibly accessed by the loop body (the `reads` part of e's effect). In the case of a recursive function, $wp(L\texttt{:}e, \mathsf{True})$ must be computed within an environment where f is assumed to have type $(x_1 : \tau_1) \rightarrow \cdots \rightarrow (x_n : \tau_n) \rightarrow \{p \wedge t < \texttt{at}(t, L)\}\ \tau\ \epsilon\ \{q\}$ i.e. where the decreasing of the variant t has been added to the precondition of f.

The case of a function call $e1\ e2$ can be simplified to the case of an application $x_1\ x_2$ of one variable to another, using the following transformation if needed:

$$e_1\ e_2 \equiv \texttt{let } x_1 = e_1 \texttt{ in let } x_2 = e_2 \texttt{ in } x_1\ x_2$$

Then assuming that x_1 has type $(x : \tau) \rightarrow \{p'\}\ \tau'\ \epsilon\ \{q'\}$, we define

$$wp(x_1\ x_2, q) = f^{\rho.p}_{(q.1(7)]TJ/F47\ 9.9626\ Tf\ 6.642\ 0\ Td}$$

Chapter 2

Krakatoa

Krakatoa expects as input a Java source file, annotated with the *Krakatoa Modeling Language* (KML for short). KML is largely inspired from the *Java Modeling Language* [16, 5]. As JML, KML specifications are given as annotations in the source code, in a special style of comments. KML also shares many features with the *ANSI/ISO C Specification Language* [2], which is the specification language for the Frama-C platform.

This chapter is organized as follows. Section 2.1 gives an overview of the KML specification language. Section 2.2 describes the translation from annotated Java to the Why input language, which amounts first to axiomatize the Java memory heap, and then provide translation rules. Section 2.3 explores more advanced features.

2.1 KML Specification Language

We present the basic features of KML. For more details please look the reference manual (`http://krakatoa.lri.fr/manual/`) or at the ACSL design document [2].

2.1.1 Behavioral Specifications

Behavioral properties are attached to programs using the following kinds of annotations.

Method contracts

Method *contracts* are made of a *precondition* and a set of *behaviors*. The precondition is a proposition introduced by keyword `requires` which is supposed to hold in the pre-state of the method, i.e. when it is called. It must be checked valid by the caller.

A *normal* behavior has the form:

```
/*@ ...
  @ behavior b:
  @   assumes A;
  @   assigns L;
  @   ensures E;
  @*/
```

The semantics is as follows:

- In the post-state, i.e when the method returns normally, the property `\old`(A)`==>`E holds.
- if A holds in the pre-state then each memory location (already allocated in the pre-state) that does not belong to the set L remains allocated and is left unchanged in the post-state.

In E, the notation `\result` denotes the returned value, and `\old`(e) denotes the value of e in the pre-state.

An *exceptional* behavior as the form

```
/*@ ...
  @ behavior b:
  @   assumes A;
  @   assigns L;
  @   signals (Exc x) E;
  @*/
```

The semantics is similar to normal behaviors, but where properties must hold when the method returns abruptly with exception Exc.

Code annotations

Annotations can be attached to statements. A local assertion, introduced by keyword `assert`, must hold at the corresponding program point.

Loop annotations can be given just in front of loop constructs (`while`, `for`, etc.). They have the form

```
/*@ loop_invariant I
  @ for b:  loop_invariant I_b;
  @ loop_variant V;
  @*/
```

It states that I is an *inductive invariant*: it must hold at loop entry and be preserved by any iteration of

Ghost code

The purpose of ghost code is to compute some additional information in order to monitor the program execution. Ghost variables are additional, specification only variables that can be added to method bodies. They are declared with

`/*@ ghost` τ x `=` e `;`

and can be assigned with

`/*@ ghost` x `=` e `;`

Class invariants

Class invariants are declared at the level of class members, they have the form

`/*@ invariant` id`:` e`;`

It states that property e must "always" hold for the current object. The always is quoted here because indeed an invariant may be temporarily invalid inside execution of a method. Also, it need not to be true until the object constructor has returned.

Giving a precise semantics to class invariants and provide ways to check them is a fundamental issue [17], see a discussion further in Section 2.2.4.

2.1.2 Logic definitions

Unlike in JML, KML does not allow *pure* methods to be used in annotations. But it permits to declare new logic functions and predicates. They must be placed at the global level i.e. outside any class declaration, and respectively have the form

`//@ logic` τ $id(\tau_1\ x_1, \ldots, \tau_n\ x_n) = e$ `;`
`//@ predicate` $id(\tau_1\ x_1, \ldots, \tau_n\ x_n) = p$ `;`

where e must have type τ, and p must be a proposition. The types τ and τ_i can be either Java types or purely logic types: `integer`, `real`, or user-introduced as shown in next section.

Lemmas are additional properties that can be added, usually to give hints to provers. A lemma is declared as

`//@ lemma id:` p`;`

Inductive predicates

It is sometimes handy to define predicates *inductively*, by a set of clauses, like it is done in Prolog, or in proof assistants like PVS or Coq. An inductive definition has the form

`/*@ inductive` $P(\tau_1\ x_1, \ldots, \tau_n\ x_n)$ `{`
`@ case` c_1 `:` p_1`;`
`...`
`@ case` c_k `:` p_k`;`
`@ }`
`@*/`

where each c_i is an identifier and each p_i is a proposition. The semantics of such a definition is that P is the least fixpoint of the cases, i.e. is the smallest predicate (in the sense that it is false the most often) satisfying the propositions $p_1, \ldots, p_k$. To ensure existence of a least fixpoint, it is required that each proposition p_i has the form

`\forall` $y_1, \ldots, y_m$`,`
h_1 `==>` $\cdots$ `==>` h_l `==>` $P(t_1, \ldots, t_n)$

where P occurs only positively in hypotheses $h_1, \ldots, h_l$.

Hybrid predicates

A proposition may contain expressions which depend on the memory heap: $o.f$, $t[i]$. In such a case, the proposition depends on some memory state. More generally, it is possible to define functions or predicates depending on several memory states, via the use of labels:

`//@ logic` τ $id\{L_1, \ldots, L_k\}(\tau_1\ x_1, \ldots) = e$`;`
`//@ predicate` $id\{L_1, \ldots\}(\tau_1\ x_1, \ldots) = p$`;`

in such a case, the state where memory is accessed can be made precise using the forms `\at`$(o.f, L_i)$, `\at`$(t[i], L_j)$, etc.

Labels are those defined in the code, and also the following predefined labels:

- `Here` is visible in contracts and all statement annotations, where it refers to the current state where the annotation appears.
- `Old` is visible in contracts and refers to the pre-state of this contract.
- `Pre` is visible in all statement annotations, and refers to the pre-state of the method it occurs in.
- `Post` is visible in contracts and refers to the post-state.

2.1.3 Algebraic Data types

To go further than explicit definitions of functions and predicates, one may introduce *axiomatic* blocks where are declared a set of type names, and a set of axiomatic declarations of predicates and logic functions, which amounts to declare their expected profiles and a set of axioms they satisfy. A representative example is:

```
/*@ axiomatic IntLists {
  @  type ilist;
  @  ilist nil();
  @  ilist cons(integer n, ilist l);
  @  ilist append(ilist l1, ilist l2);
  @  axiom append_nil:
  @    \forall ilist l;
  @    append(nil(),l) == l;
  @  axiom append_cons:
```

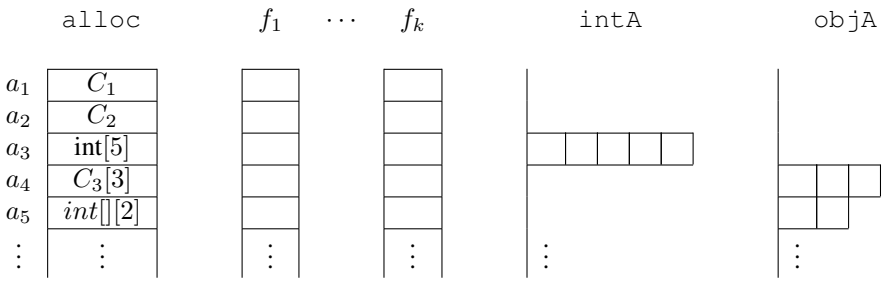

Figure 2.1: component-as-array style modeling of Java memory heap

```
@     \forall integer n;
@     \forall ilist l1 l2;
@     append(cons(n,l1),l2) ==
@          cons(n,append(l1,l2));
@ }
@*/
```

It must be emphasized that such axiomatic block cannot be guaranteed to be consistent.

2.2 Translation to Why

A key idea for the Krakatoa/Why toolchain is that Why is used as a programming language to program the semantics of Java. This happens in two parts: first, the algebraic specification language of Why is used to provide a modeling for Java memory heap. Second, a set of translation rules from Java to Why code is provided [19, 18].

For clarity of the presentation, we interpret machine integers as mathematical integers, and floating-point numbers as reals. See Section 2.3.3 for a more faithful interpretation.

2.2.1 Axiomatization of the heap

A natural modeling of memory heap would be to represent heap by a single large array of data indexed by object addresses (as in the Spec#/Boogie toolchain). Our modeling is based on the so-called *component-as-array* modeling due to Burstall and emphasized by Bornat [4]: to each non-static field is associated a Why reference. This separation of heap into independant pieces is valid because with Java semantics guarantees that modification of a field f of an object cannot modify any field g of another (or the same) object (unlike in C, where pointer casts falsify this property).

In Why, a variable representing a field named f for all object will have the logic type α `memory`, where α is the type of f. Type `memory` indeed acts as so-called functional arrays, via two functions

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

satisfying the theory of arrays:

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

For Java such a modeling for object fields must be extended by similar variables for arrays: one such variable respectively for arrays of ints, arrays of booleans, arrays of reals and arrays of object references. Finally, for dynamically representing the dynamic types of objects and the sizes of arrays, a additional Why variable called *allocation table* is added (of some new type `alloc_table`). The resulting modeling is schematized on Figure 2.1.

2.2.2 Translation rules

Types

The translation rules for types are

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

where $\tau_{int} \in \{\texttt{byte}, \texttt{char}, \texttt{short}, \texttt{int}, \texttt{long}\}$, $\tau_{float} \in \{\texttt{float}, \texttt{double}\}$ and τ_{ref} any reference type, that is objects or arrays.

Expressions

The translation of simple expressions are:

$$\begin{aligned} [\![c]\!] &= c && \text{constants} \\ [\![v]\!] &= !v && \text{local variables} \\ [\![e_1\ op\ e_2]\!] &= [\![e_1]\!]\ op\ [\![e2]\!] && \text{bin operator} \\ [\![e.f]\!] &= \texttt{let}\ tmp = [\![e]\!]\ \texttt{in} \\ &\quad \texttt{assert}\{tmp \neq \texttt{null}\}; \\ &\quad \texttt{select}(!f, tmp) && \text{field access} \end{aligned}$$

Null pointer dereferencing is ruled out by the assert guard.

Array accesses are translated similarly as

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

where A is intA, boolA, realA or objA depending on the type of e_1. `shift` is another logic function in the model. Similarly to field access, array access out of bounds are ruled out by the assert guard.

Other expressions like method calls and object creation are delayed to section 2.2.2

`instanceof` and casts

During translation, a new Why type `tag_id` is generated, where each class identifier becomes a constant of type `tag_id`. The subclass relation is represented by a `subtag` binary predicate over `tag_id`.

A Why logic function

```
typeof: alloc_table, pointer-> tag_id
```

is used to retrieve the dynamic type of a reference, from a given allocation table

instanceof:

$$
\begin{array}{l}
[\![e \texttt{ instanceof } C]\!] = \\
\quad \texttt{subtag}(\texttt{typeof}(alloc, [\![e]\!]), [\![C]\!])
\end{array}
$$

cast expression:

$$
\begin{array}{l}
[\![(C)e]\!] = \\
\quad \texttt{let } tmp = [\![e]\!] \texttt{ in} \\
\quad \texttt{assert } \texttt{subtag}(\texttt{typeof}(alloc, tmp, [\![C]\!]); \\
\quad tmp
\end{array}
$$

Notice that it is mandatory to show the absence of ClassCastException.

Assignments

local variable assignment:

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

field assignment:

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

array assignment:

$$
\begin{array}{l}
[\![e_1[e_2] = e_3]\!] = \\
\quad \texttt{let } tmp_1 = [\![e_1]\!] \texttt{ in} \\
\quad \texttt{let } tmp_2 = [\![e_2]\!] \texttt{ in} \\
\quad \texttt{let } tmp_3 = [\![e_3]\!] \texttt{ in} \\
\quad \texttt{assert}\{tmp_1 \neq \texttt{null} \wedge \\
\qquad 0 \leq tmp_2 < \texttt{blocklength}(!alloc, tmp_1) \\
\qquad \wedge (\text{cf remark})\}; \\
\quad A := \texttt{store}(!A, \texttt{shift}(tmp_1, tmp_2), tmp_3)
\end{array}
$$

Remark: for completeness, one should add an additional assertion to ensure the absence of ArrayStoreException [12].

Statements

Declarations of local variables are translated into declarations of references:

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

Remark: when no initializer is given, $[\![e]\!]$ is replaced by a call to a "any" Why function of appropriate return type.

Java sequences are naturally translated into Why sequences.

Translation of the if statement is straightforward:

$$
\begin{array}{l}
[\![\texttt{if } (e) s_1 \texttt{ else } s_2]\!] = \\
\quad \texttt{if } [\![e]\!] \texttt{ then } [\![s_1]\!] \texttt{ else } [\![s_2]\!]
\end{array}
$$

Translation of the while loop is more complex because of possible break and continue statements:

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

where abrupt statements are translated by Why exceptions declared once and for all:

$$
\begin{array}{rcl}
[\![\texttt{break;}]\!] & = & \texttt{raise Break;} \\
[\![\texttt{continue;}]\!] & = & \texttt{raise Continue;} \\
[\![\texttt{return } e\texttt{;}]\!] & = & \texttt{return := } [\![e]\!] \texttt{ ;} \\
 & & \texttt{raise Return;}
\end{array}
$$

see translation of method bodies for details about the `return` variable.

Exception throwing is as follows:

$$[\![\texttt{throw } e]\!] = \texttt{raise (Exception } [\![e]\!]\texttt{)}$$

Exception catching

$$
\begin{array}{l}
\texttt{try } s \\
\texttt{catch } (Exc_1\ v_1)\ s_1 \\
\vdots \\
\texttt{catch } (Exc_n\ v_n)\ s_n
\end{array}
$$

is translated into:

$$
\begin{array}{l}
\texttt{try } [\![s]\!] \\
\texttt{with (Exception } tmp\texttt{) ->} \\
\quad \texttt{if (instanceof } tmp\ Exc_1\texttt{) then} \\
\qquad \texttt{let } v_1 = tmp \texttt{ in } [\![s_1]\!] \\
\quad \texttt{else} \\
\quad \vdots \\
\quad \texttt{else if (instanceof tmp } Exc_n\texttt{) then} \\
\qquad \texttt{let } v_n = tmp \texttt{ in } [\![s_n]\!] \\
\quad \texttt{else raise (Exception } tmp\texttt{)}
\end{array}
$$

$$\frac{\Gamma \vdash e : R, W}{\Gamma \vdash e.f : R \cup \{f\}, W}$$

$$\frac{e_1 \text{ has type } \texttt{int[]} \qquad \Gamma \vdash e_1 : R_1, W_1 \qquad \Gamma \vdash e_2 : R_2, W_2 \qquad \Gamma \vdash e_3 : R_3, W_3}{\Gamma \vdash e_1[e_2] = e_3 : R_1 \cup R_2 \cup R_3 \cup \{\texttt{intA}, \texttt{alloc}\}, W_1 \cup W_2 \cup W_3 \cup \{\texttt{intA}\}}$$

$$\frac{m : R, W \text{ in } \Gamma \qquad \Gamma \vdash e_1 : R_1, W_1 \qquad \cdots \qquad \Gamma \vdash e_k : R_k, W_k}{\Gamma \vdash m(e_1, \ldots, e_k) : \bigcup_i R_i \cup R, \bigcup_i W_i \cup W}$$

$$\frac{C : R, W \text{ in } \Gamma \qquad \Gamma \vdash e_1 : R_1, W_1 \qquad \cdots \qquad \Gamma \vdash e_k : R_k, W_k}{\Gamma \vdash \texttt{new } C(e_1, \ldots, e_n) : \bigcup_i R_i \cup R \cup \{\texttt{alloc}\}, \bigcup_i W_i \cup W \cup \{\texttt{alloc}\}}$$

Figure 2.2: Effects computation

Method Declarations and Method Calls

For each method in a class C annotated with KML specification:

```
/*@ requires R ;
  @ assigns A;
  @ ensures E;
  @*/ τ m(x1 : τ1, ..., xn : τn) ;
```

a Why *parameter* is generated:

```
parameter C_m_parameter:
  this:pointer ->
    x1 : [|τ1|] -> ... -> xn : [|τn|] ->
  { [|R|] ∧ this ≠ null
    ∧ Inv(this, x1, ..., xn) }
  [|τ|]
  reads r writes w
  { [|E|] ∧ Inv(this, x1, ..., xn)
    ∧ Assigns(A) }
```

The read effects r and write effects w are computed as shown in next section, together with the Assigns(A) formula. The role of the Inv formula is to handle class invariants, see Section 2.2.4 for discussion.

The method body is translated into

```
let return=ref (any_τ ()) in
try body with Return e -> e
```

A method call $e.m(e_1, \ldots, e_n)$ is then translated into

```
let tmp = [|e|] in
let tmp1 = [|e1|] in
⋮
let tmpn = [|en|] in
(C_m_parameter tmp tmp1 ... tmpn)
```

Notice that there is no such thing as "dynamic call" in Why: a dynamic method call in Java is translated as above. The main point here is that if the method m is overridden in some subclass D of C, then the Why contract of m_parameter should incorporate the behaviors of $D.m$ with an additional assumes clause:

```
/*@ behavior some behavior of D.m ;
  @ assumes (this instanceof D);
  @ ...
  @*/
```

Notice that when a method overrides another, it is not allowed to add any `requires` clause.

Effects and Assigns clauses

Because Java methods can be mutually recursive, the effect analysis uses an iterative process in order to compute the maximal effects of each method, starting from an empty effect. We introduce an environment Γ which associates to each method its (currently known) effects, we write $\Gamma \vdash e : R, W$ to mean that expression (or statement) e reads variables R and writes variables W assuming the methods have effects as given in Γ. We give on Figure 2.2 a few rules for computing $\Gamma \vdash e : R, W$.

From a given Γ, and a given method m with body e, we compute R, W such that $\Gamma \vdash e : R, W$ and update consequentely the information associated to method m in Γ until a fixpoint is reached, which happens in finite time because set of effects are bounded: the number of variables is fixed.

The set of assigned locations A in a given method is then partitioned with respect to the set W: to each Why variable w in W we associate the set of corresponding locations A_w so that

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

The A_w are Why datatypes of a new abstract type pset corresponding to sets of pointers. There is an axiomatic theory of those sets, including the following functions:

- `pset_empty: -> pset`
  denotes the empty set
- `pset_singleton: pointer -> pset`
  `pset_singleton`(p) denotes the singleton set $\{p\}$

- `pset_union: pset,pset -> pset` denotes the union
- `pset_deref: pointer memory, pset -> pset` `pset_deref`(m,s) denotes the set

$$\{\texttt{select}(m,p) \mid p \in s\}$$

- `pset_range: pset, int, int -> pset` `pset_range`(s,a,b) denotes the set

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

(the complete axiomatization is in the Why file `lib/why/jessie.why` of the Why platform)

Finally, a new predicate is introduced:

$$\texttt{not\_assigns} : \texttt{alloc\_table}, \alpha\,\texttt{memory}, \alpha\,\texttt{memory}, \texttt{pset} -> \mathit{prop}$$

$\texttt{not\_assigns}(a, m_1, m_2, s)$ means that for any pointer p allocated in allocation table a and which do not belong to s: $\texttt{select}(m_1, p) = \texttt{select}(m_2, p)$. The complete formula $\mathit{Assigns}(A)$ translating the `assigns` clause A is then

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

See example in Section 2.2.3 below

Object allocation

Object creation involves both an allocation on the heap, and a call to a constructor. For allocation, we introduce a new Why parameter:

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

Object creation expression `new` $C(e_1, \ldots, e_n)$ is then translated into:

$$\texttt{let}\ \mathit{tmp} = (\texttt{alloc\_parameter}\ C\ 1)\ \texttt{in}\ (\texttt{cons\_}C\texttt{\_parameter}\ \mathit{tmp}\ [\![e_1]\!] \cdots [\![e_n]\!])\ ;\ \mathit{tmp}$$

where `cons_`C`_parameter` is the Why translation of the corresponding constructor of class C.

Array creation is translated in a similar way.

The body of constructors are translated in a way very similar to methods, except that

- some statements are added for initializing fields of that class to their default value.
- then there is a special statement for calling the appropriate constructor of the super class (or of the same class if Java body starts with a statement `this(...)`)
- the invariant formula Inv is not the same in the pre- and in the post-condition: the currently created object need not to satisfy its class invariant in the pre-state.

2.2.3 Example

Let's consider the example of a toy electronic purse:

```
class NoCreditException
  extends Exception {
    public NoCreditException() { }
}

public class Purse {

 private int balance;
 /*@ invariant balance_positive:
   @   balance > 0;
   @*/

 /*@ requires s >= 0;
   @ assigns balance;
   @ ensures s < \old(balance) &&
   @  balance == \old(balance) - s;
   @ behavior amount_too_large:
   @  assigns \nothing;
   @  signals (NoCreditException)
   @   s >= \old(balance) ;
   @*/
 public void withdraw(int s)
  throws NoCreditException {
   if (s < balance)
    balance = balance - s;
   else
    throw new NoCreditException();
 }

}
```

An excerpt of translated Why code is

```
type tag_id
logic Exception : tag_id
logic NoCreditException : tag_id
logic Purse : tag_id
logic sub_tag: tag_id, tag_id -> prop
axiom sub_tag(NoCreditException, Exception)

predicate
 balance_positive(balance:int memory,
                  this:pointer) =
```

```
  select(balance,this) > 0

parameter balance : int memory ref

parameter Purse_withdraw_parameter :
 this:pointer -> s:int ->
  { s >= 0 and this <> null and
    balance_positive(balance, this) }
  unit
  reads alloc
  writes balance
  raises Exception
  { s < select(balance, this)
    and select(balance, this) =
        select(balance@,heap, this) - s
    and balance_positive(balance, this)
    and not_assigns(alloc,
           balance@, balance,
           pset_singleton(this))
  | Exception =>
    instanceof(result, NoCreditException)
    and s >= select(balance@, this)
    and balance_positive(balance, this)
    and not_assigns(alloc,
           balance@, balance,
           pset_empty) }
```

and also, for withdraw method implementation, with *pre* and *post* the same conditions as in the parameter declaration above:

```
let Purse_withdraw =
 fun (this : pointer) (s : int) -> { pre }
  begin
  if s < (assert { this != null } ;
          select(balance,this))
  then
    let tmp = (assert { this != null } ;
               select(balance,this)) - s
    in
    assert { this != null } ;
    balance := store(balance,this,tmp)
  else
   raise (Exception
    (let tmp =
      (alloc_parameter NoCreditException 1)
     in
     cons_NoCreditException_parameter tmp;
     tmp)
  end { post }
```

2.2.4 Limitations

There are some limitations in the current state of Krakatoa.

class invariants The support for class invariants is very naive: there are treated just as macros for automatically inserting pre- and post-conditions to methods. There is no universally known best solutions [17] and research is very active on this topics: ownership a la Spec#/Boogie [1], universe type systems [8], etc. Krakatoa currently has an experimental implementation of ownership using the pragma `//@+ InvariantPolicy = Ownership`.

class initialization Krakatoa does not take into account the class initialization.

generics Krakatoa does not support Java 1.5 generics

annotated API Programs that make heavy use of the standard API are difficult to handle, because there is no "universal" specifications for the API. Krakatoa is shipped with its own annotated API, both for Java and Java Card, but it is very incomplete.

2.3 Additional features

2.3.1 Java Card transactions

Krakatoa has a specific support for Java Card [6] applets. We proposed a support for the so-called *transactions* by adapting the memory model: Why variables which encode memory heap are duplicated, so that the Why encoding is able to roll back modifications in case of aborted transactions [20, 23]

2.3.2 Separation Analysis

The idea of separation of memory in the component-as-array modeling can be pushed further: a fine-grain partition of the heap can be computed on each program, and the number of Why variables in memory modeling can be specific to each program [15, 14, 22]. This has shown to improve a lot the automation of the VC verification. In Krakatoa, this separation analysis is turned on by the pragma
`//@+ SeparationPolicy = Regions`.

2.3.3 Integer and Floating-Point Arithmetic

Instead of interpreting machine integers as mathematical unbounded integers, it is possible to interpret them faithfully, by requiring to prove that no overflow occur. This is indeed achieved at the level of the Jessie intermediate language, which allows to declare any *range type*, e.g:

```
type byte = -128..127
```

With such a subtype of integers, a few additional Why declarations are generated:

```
logic integer_of_byte: byte -> int
parameter byte_of_integer :
 x:int ->
  { -128 <= x <= 127}
```

```
byte
{ integer_of_byte(result) = x }
```

and all operations such as addition are guarded by an overflow check, e.g

```
x = y + z;
```

for variables of type `byte` is translated into

```
x := byte_of_integer
  (integer_of_byte !y +
   integer_of_byte !z);
```

A similar approach exists for floating-point computations, where one is required to show that no overflow occur [3].

2.3.4 Automatic generation of annotations

A technique for automatically generating loop invariants, post-condition and pre-conditions has been proposed by Moy and Rousset [21, 23, 22]. It is based on a combination of abstract interpretation techniques [7], weakest precondition calculi and quantifier elimination.

These techniques are implemented in the Why platform, and has been successfully experimented for automatically generating necessary annotations for memory safety of Java Card applets[23] and C libraries on strings [22].

These techniques are also used in combination with an original technique for implementing a "non-null by default" policy [23]: In Krakatoa this policy is activated by the pragma

```
//@+ NonNullByDefault = all
```

References

[1] Mike Barnett, Robert DeLine, Manuel Fähndrich, K. Rustan M. Leino, and Wolfram Schulte. Verification of object-oriented programs with invariants. *Journal of Object Technology*, 3(6):27–56, June 2004.

[2] Patrick Baudin, Jean-Christophe Filliâtre, Claude Marché, Benjamin Monate, Yannick Moy, and Virgile Prevosto. *ACSL: ANSI/ISO C Specification Language*, 2008. `http://frama-c.cea.fr/acsl.html`.

[3] Sylvie Boldo and Jean-Christophe Filliâtre. Formal Verification of Floating-Point Programs. In *18th IEEE International Symposium on Computer Arithmetic*, pages 187–194, Montpellier, France, June 2007.

[4] Richard Bornat. Proving pointer programs in Hoare logic. In *Mathematics of Program Construction*, pages 102–126, 2000.

[5] Lilian Burdy, Yoonsik Cheon, David Cok, Michael Ernst, Joe Kiniry, Gary T. Leavens, K. Rustan M. Leino, and Erik Poll. An overview of JML tools and applications. *International Journal on Software Tools for Technology Transfer*, 2004.

[6] Zhiqun Chen. *JavaCard™ Technology for Smart Cards*. The Java Series. Addison-Wesley, 2000.

[7] Patrick Cousot. Methods and logics for proving programs. In J. van Leeuwen, editor, *Handbook of Theoretical Computer Science*, volume B, pages 841–993. North-Holland, 1990.

PhD o Inof5[8] Werner Dietl and Pener ,MflSva55 -16.598 -18-690(MflSv)2i7J

[13] C. A. R. Hoare. An axiomatic basis for computer programming. *Communications of the ACM*, 12(10):576–580 and 583, October 1969.

[14] Thierry Hubert. *Analyse Statique et preuve de Programmes Industriels Critiques*. Thèse de doctorat, Université Paris-Sud, June 2008.

[15] Thierry Hubert and Claude Marché. Separation analysis for deductive verification. In *Heap Analysis and Verification (HAV'07)*, pages 81–93, Braga, Portugal, March 2007. `http://www.lri.fr/~marche/hubert07hav.pdf`.

[16] Jml-java modeling language. `www.jmlspecs.org`.

[17] Gary T. Leavens, K. Rustan M. Leino, and Peter Müller. Specification and verification challenges for sequential object-oriented programs. *Formal Aspects of Computing*, 2007.

[18] Claude Marché and Christine Paulin-Mohring. Reasoning about Java programs with aliasing and frame conditions. In J. Hurd and T. Melham, editors, *18th International Conference on Theorem Proving in Higher Order Logics*, volume 3603 of *Lecture Notes in Computer Science*, pages 179–194. Springer, August 2005.

[19] Claude Marché, Christine Paulin-Mohring, and Xavier Urbain. The KRAKATOA tool for certification of JAVA/JAVACARD programs annotated in JML. *Journal of Logic and Algebraic Programming*, 58(1–2):89–106, 2004. `http://krakatoa.lri.fr`.

[20] Claude Marché and Nicolas Rousset. Verification of Java Card applets behavior with respect to transactions and card tears. In Dang Van Hung and Paritosh Pandya, editors, *4th IEEE International Conference on Software Engineering and Formal Methods (SEFM'06)*, Pune, India, September 2006. IEEE Comp. Soc. Press.

[21] Yannick Moy. Sufficient preconditions for modular assertion checking. In Francesco Logozzo, Doron Peled, and Lenore Zuck, editors, *9th International Conference on Verification, Model Checking, and Abstract Interpretation*, volume 4905 of *Lecture Notes in Computer Science*, pages 188–202, San Francisco, California, USA, January 2008. Springer.

[22] Yannick Moy. *Automatic Modular Static Safety Checking for C Programs*. PhD thesis, Université Paris-Sud, January 2009.

[23] Nicolas Rousset. *Automatisation de la Spécification et de la Vérification d'applications Java Card*. Thèse de doctorat, Université Paris-Sud, June 2008.

[24] T. Schreiber. Auxiliary Variables and Recursive Procedures. In *TAPSOFT'97: Theory and Practice of Software Development*, volume 1214 of *Lecture Notes in Computer Science*, pages 697–711. Springer-Verlag, April 1997.