The Java Verification Tool KeY

An FM 2024 Tutorial

Bernhard Beckert   Richard Bubel   Daniel Drodt
Reiner Hähnle   Florian Lanzinger   Wolfram Pfeifer
Mattias Ulbrich   Alexander Weigl

Karlsruhe Institute of Technology
Technical University of Darmstadt

May 9, 2025

Funded by
DFG Deutsche Forschungsgemeinschaft
German Research Foundation

Outline

- Introduction to Specification and Verification of Java Programs
- Demo I
- Java Features: Heap, Exceptions, Loops, Integer Types

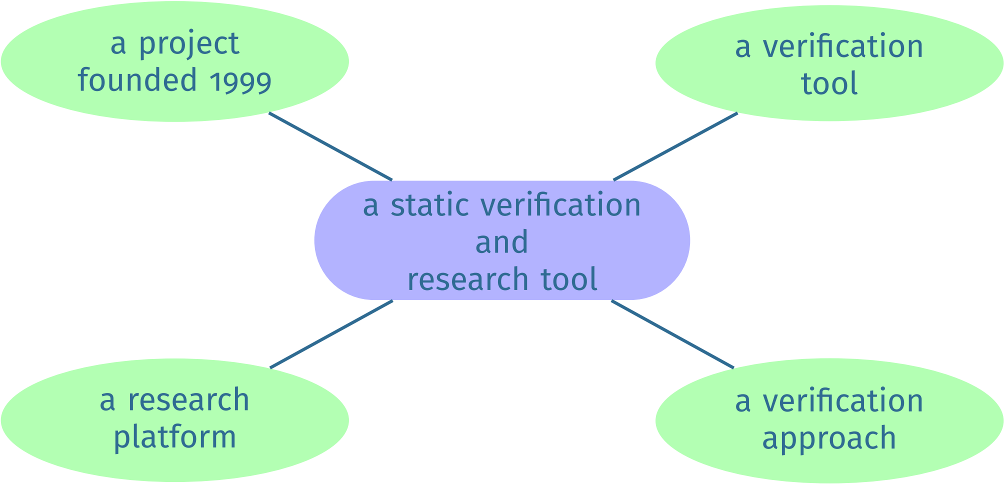
∞

- Handling Framing with KeY
- Demo II
- Taclets (Extending KeY)
- Hands-On Exercise

Part I

Introduction

What is KeY?


a project founded 1999
a verification tool
a static verification and research tool
a research platform
a verification approach

Tutorial Objectives

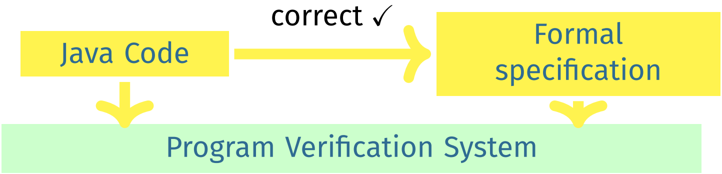
After this tutorial you know the basic concepts of

- formal specification of object-oriented programs
  - functional behavior
  - method contracts
  - framing of memory access
- the design of a deductive verification system based on
  - a logic calculus and
  - symbolic execution

After this tutorial you are able to

- write a formal specification in the Java Modeling Language (JML)
- verify that a Java program satisfies its JML specification using the KeY tool

Deductive Verification



Proof rules establish relation "implementation conforms to specification"

Computer support essential for verification of real programming languages

boolean ArrayList:contains(Object o)

- Typical small Java library method implementation

Behavioral Proof	▶ ca. 1,750 proof steps, ca. 0.6 secs with KeY
	▶ 15 case distinctions, fully automatic
Framing	▶ ca. 6,700 proof steps, ca. 2.4 secs with KeY
	▶ 50 case distinctions, fully automatic

One Main Use Case of KeY

Verification of JDK Library Source Code Implementations

- Fully Verified Java Card API Reference Implementation (2007)
- OpenJDK's Sort Method for Generic Collections (2015)
- JDK's Dual Pivot Quicksort (2017)
- JDK's Identity Hash Map (2022)
- OpenJDK's LinkedList (2022)
- OpenJDK's BitSet (2023)
- State-of-art sorter ips^{4}o (2024)

Buggy!

Buggy!

Buggy!

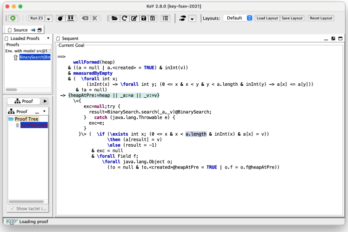
Buggy!

Part II

Verification Approach

Proof Obligation Generator

$$pre \rightarrow \langle p \rangle post$$



?

Specification and Verification Target

In Object-Oriented Setting:

Units to be specified are interfaces, classes, and their methods

Focus on methods

Method specifications must include the following aspects:

- Initial value of formal parameters
- Expected result value and any changes to field values
- Accessible part of pre-/post-state

In this tutorial we focus on sequential Java programs

Useful analogy to stress the different roles/obligations/responsibilities:

Method specification as a contract
(between method implementor/callee and user/caller)

"Design by Contract" methodology (Meyer, 1992, EIFFEL)

Callee guarantees certain outcome provided caller guarantees prerequisites

Contract describes effect of a method execution in terms of logical formulas

Advantages of Contracts

- Correctness proof follows call graph, is procedure modular
- Instead of inlining method implementation, apply contract
- Replace program execution by substitution and deduction
- Avoid state explosion due to non-linear call structure
- Handle unbounded recursion

First used in (Hoare, 1971, LNM 188, pp. 102–116)

Method Contract: Definition

Let m be a method; a contract for m has the form:

$$\mathrm{Contract}(m) \;:=\; (pre, post[, mod]\,[, acc]\,[, trm])$$

- Formulas pre and $post$ are called pre- and postcondition
- Optional modifiers mod and acc are sets of memory locations
- Optional termination witness trm is a term equipped with a well-order $\prec$

Meaning of a Contract (for Total Correctness)

If the caller of m ensures that pre holds at call time, method m guarantees:

1. $post$ holds in the reached final state;
2. at most locations in mod where modified (default: all visible);
3. the result of m only depends on locations in acc (default: all visible);
4. m terminates: trm stays non-negative and strictly decreases at recursive calls

Part III

Specification with JML

Java Modeling Language (JML)

JML is a specification language tailored to Java,
a behavioral interface specification language (BISL)

General JML Philosophy

Integrate

- specification and
- implementation

in one single language ("single-tier approach")

$\Rightarrow$ JML is not external to Java, but an extension of Java

JML
is
Java + First-Order Logic + Contracts + Invariants + more ...

Running Example

```
private int binSearch(int[] a, int v, int low, int up) {
  if (low < up) {
    int mid = low + ((up - low) / 2);
    if (v == a[mid])       { return mid; }
    else if (v < a[mid])  { return binSearch(a, v, low, mid); }
    else                   { return binSearch(a, v, mid + 1, up); }
  }
  return -1;
}
```

Observations

- **Internal method** for binary search in contiguous part [low, up) of array a (for search in complete array call binSearch(a, v) = binSearch(a, v, 0, a.length))
- **Recursive** implementation

Natural Language Specification

If the caller guarantees that

(i) low is less-or-equal than up, both are in the bounds of a (incl. a.length),
(ii) a is not null and **(iii)** a is sorted,

then the method guarantees that

(iv) the result is -1 or in [low, up)
(v) if v is in a then an index of v in a is returned, else -1 is returned
(vi) it terminates w/o an exception
(vii) the heap is not modified, and
(viii) up - low is a termination witness

```
/*@ private normal_behavior
  @ requires  0 <= low <= up <= a.length;
  @ requires (\forall int x, y;
  @              0 <= x < y < a.length; a[x] <= a[y]);
  @                                                  < up;
  @
  @                                                  ?
  @      \result >= low && a[\result] == v
  @    : \result == -1;
  @ assignable \nothing;
  @ measured_by up - low;
  @*/
 private int binSearch(int[] a, int v, int low, int up
```

Default in JML
Implicit precondition:
requires a!=null;

Part IV

Deductive Verification

Only static properties expressible in (typed) first-order logic (FOL), for example:

Value of a field is in a certain range at a given time in a computation

Talks about a single program state

Required:
Express behavior of a program in terms of state changes, for example:

If method `setAge(int newAge)` is called on an object *o* of type `Person`
and the method argument `newAge` is positive

then afterwards *o*'s field `age` has the same value as `newAge`
and all other fields are unchanged

Requirements on a logic to reason about programs

- Can relate different program states, i.e., before and after execution, within a single formula
- First-order (quantified) variables evaluated in same state to help automation
- $\Rightarrow$ Program variables represented by constant symbols whose value depends on interpretation in a given program state

First-order dynamic Logic is a program logic that meets these requirements

Dynamic Logic (Pratt, 1976), (Harel, Meyer & Pratt, 1977)

KIV Dynamic Logic (Heisel, Reif & Stephan, 1987), Java Dynamic Logic (Beckert, 2000)

First-Order Logic (FOL) with Java type hierarchy

- + Java programs p
- + behavioral modalities $\langle p \rangle \phi$, $[p]\phi$ (p program, ϕ DL formula)
- + symbolic state updates $v := e$

An Example

$$\mathtt{i} > 5 \rightarrow [\mathtt{i = i + 10;}]\mathtt{i} > 15$$

Meaning?

If program variable i is greater than 5 in current state, then after executing the Java statement "i = i + 10;", i is greater than 15 (unprovable in Java)

Program variable i evaluated in differing state outside and under modality

Program Variables

Dynamic Logic = Typed FOL + …

$$\mathtt{i} > 5 \rightarrow [\mathtt{i = i + 10;}]\mathtt{i} > 15$$

Program variable i evaluated in different states before / after execution

Consequences

- Program variables cannot be first-order variables
  - Quantified FO variable has value fixed by variable assignment
- Program variables such as `i` are state-dependent constant symbols
- Value of state-dependent symbol can be changed by a program

Three words one meaning: $\boxed{\text{state-dependent, non-rigid, flexible}}$

Dynamic Logic = Typed FOL + programs + ...

Programs here: any legal sequence of Java statements

(can be incomplete, no need for surrounding method or class or return)

Example

Program variables: **int** r, i, n;

Then a permitted program fragment appearing in a DL formula is:

```
i = 0;
r = 0;
while (i<n) {
  i = i+1;
  r = r+i;
}
r = r+r-n;
```

Relating Program States: Modalities

Dynamic Logic extends FOL with two additional (mix-fix) operators:

$\langle p \rangle \phi$ "diamond" $[p]\phi$ "box"

where p is a program, ϕ again DL formula

ϕ is in scope of p, can see its program variables

Intuitive Meaning

- $\langle p \rangle \phi$: p terminates and formula ϕ holds in final state — (total correctness)
- $[p]\phi$: If p terminates then formula ϕ holds in final state — (partial correctness)

Sequential Java programs are deterministic:
If a Java program terminates normally
then exactly one final state is reached from a given initial state

Dynamic Logic: Examples

Let i, old_i denote program variables of type **int**

Give the meaning in natural language:

1. $\mathtt{i} \doteq \mathtt{old\_i} \rightarrow \langle \mathtt{i{+}{+};} \rangle \mathtt{i} > \mathtt{old\_i}$
   "If i++; is executed in a state where i and old_i have the same value, then the program terminates and in its final state the value of i is greater than the value of old_i" (not provable, final state not precise)
2. $\mathtt{i} \doteq 0 \rightarrow [\mathtt{while\ (true)\ \{i{+}{+};\}}]\mathtt{i} \doteq 42$
   "If the program is executed in a state where i is equal to 0 and if the program terminates then in its final state the value of i is equal to 42" (provable)

Dynamic Logic Formulas

Definition (Dynamic Logic (DL) Formulas, inductive definition)

- Each first-order logic (FOL) formula is a **DL formula**
- If p is a program and ϕ a DL formula then $\left\{\begin{matrix}\langle \mathrm{p} \rangle \phi \\ [\mathrm{p}]\phi\end{matrix}\right\}$ is a **DL formula**
- DL formulas are **closed** under FOL quantifiers and connectives

Recap

- Program variables are **flexible constants**: never bound in quantifiers
- Java Programs contain **no FOL variables**
- Modal DL formulas can appear **nested** inside each other

```
1 a = a - b;
2 if (a < 0) {
3     a = -a;
4 }
5 r = b / a;
```

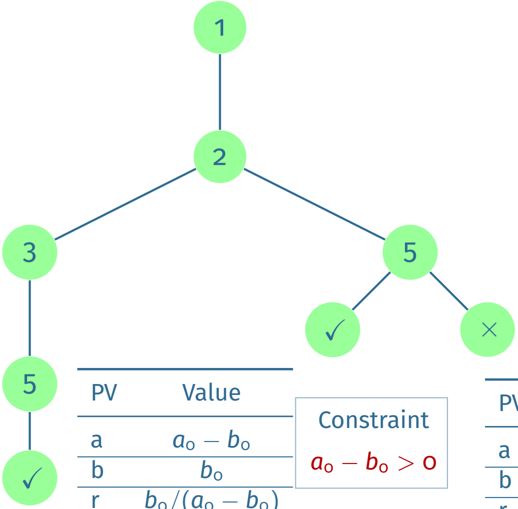
Program counter
$\hat{=}$
Line number executed next

PV	Value
a	3
b	5
r	0

PV	Value
a	−2
b	5
r	0

PV	Value
a	2
b	5
r	2

```
1 a = a - b;
2 if (a < 0) {
3     a = -a;
4 }
5 r = b / a;
```



Constraint

$-a_0 + b_0 < 0$

PV	Value
a	$-a_0 + b_0$
b	b_0
r	$b_0/(-a_0 + b_0)$

PV	Value
a	$a_0 - b_0$
b	b_0
r	$b_0/(a_0 - b_0)$

Constraint

$a_0 - b_0 > 0$

PV	Value
a	$a_0 - b_0$
b	b_0
r	r_0

Constraint

$a_0 - b_0 = 0$

Proving Validity of Dynamic Logic (DL) Formulas

Syntactic, rule-based formula transformation to realize symbolic execution in DL

A sequent

$$\overbrace{\phi_1, \ldots, \phi_n}^{\text{antecedent}} \Longrightarrow \overbrace{\psi_1, \ldots, \psi_m}^{\text{succedent}}$$

has the same meaning as

$$(\phi_1 \wedge \cdots \wedge \phi_n) \;\rightarrow\; (\psi_1 \vee \cdots \vee \psi_m)$$

Schematic sequent rules describe transformation (read from bottom to top)

$$\text{ruleName}\; \frac{\overbrace{\Gamma_1 \Longrightarrow \Delta_1 \quad \cdots \quad \Gamma_k \Longrightarrow \Delta_k}^{\text{premises}}}{\underbrace{\Gamma \Longrightarrow \Delta}_{\text{conclusion}}}$$

where $\Gamma, \Delta, \Gamma_i, \Delta_i$ match sets of DL formulas

Symbolic Execution in a DL Sequent Calculus

Symbolic Execution of Conditional with Simple Guard

$$\text{if}\ \frac{\Gamma, \mathsf{b} \doteq \mathsf{true} \Longrightarrow \langle \mathsf{p};\ \mathsf{r}\rangle\phi, \Delta \qquad \Gamma, \mathsf{b} \doteq \mathsf{false} \Longrightarrow \langle \mathsf{q};\ \mathsf{r}\rangle\phi, \Delta}{\Gamma \Longrightarrow \langle \mathsf{if}\ (\mathsf{b})\ \{\ \mathsf{p}\ \}\ \mathsf{else}\ \{\ \mathsf{q}\ \};\ \mathsf{r}\rangle\phi, \Delta}$$

- Calculus rules for symbolic execution work on first active statement
- Symbolic execution must consider all possible execution branches

Symbolic Execution of Loops: Unwind

$$\text{unwindLoop}\ \frac{\Gamma \Longrightarrow \langle \mathsf{if}\ (\mathsf{b})\ \{\ \mathsf{p};\ \mathsf{while}\ (\mathsf{b})\ \{\mathsf{p}\}\ \};\ \mathsf{r}\rangle\phi, \Delta}{\Gamma \Longrightarrow \langle \mathsf{while}\ (\mathsf{b})\ \{\mathsf{p}\};\ \mathsf{r}\rangle\phi, \Delta}$$

Need to model control flow and state changes

Requirements of Explicit Notation for Symbolic State Changes

- Symbolic execution interprets program in forward direction: Avoid ghost variables
- Simplify effects of state change eagerly
  $\Rightarrow$ Succinct representation of state changes effected by incremental SE step
- Apply state changes lazily (to post condition)

A dedicated notation for symbolic state changes: Symbolic updates

Symbolic State Updates

Definition (Syntax of Updates, Updated Terms/Formulas)

Let v be a program variable of type T, e a term of type T,
e' any term, ϕ any formula, then

- $\mathsf{v} := e$ is an **elementary update** (of v to e)
- $\{\mathsf{v} := e\}e'$ is a DL **term** and $\{\mathsf{v} := e\}\phi$ is a DL **formula**

Definition (Informal Semantics of Updates)

- $\mathsf{v} := e$ modifies current state into a state, where v has value of e
  (and all other program variables have same value as in current state)
- $\{\mathsf{v} := e\}e'$ is the value of e' in the state, where all v's in e' have value of e
- $\{\mathsf{v} := e\}\phi$ is true, if ϕ is true in the state, where all v's in ϕ have value of e

The formal semantics of updates is characterized by a set of rewrite rules

Explicit State Updates: Observations

Facts about updates $v := t$

- Update semantics almost identical to that of assignment statement
- Updates are not assignments:
  - right-hand side is a term or formula, not a program expression;
  - $\langle$x = i++;$\rangle\phi$ cannot be turned into update (has side effect)
- Updates are not equations: they change value of v
- Application of updates is similar to lazy, explicit substitution

Purpose of updates is to represent the effect of assignments
in terms of simple, symbolic state changes

Assignment Rule Formulated with Updates

Symbolic execution of assignment with updates

$$\text{assign}\ \frac{\Gamma \Longrightarrow \{\mathtt{x} := \mathtt{e}\}\langle p \rangle \phi, \Delta}{\Gamma \Longrightarrow \langle \mathtt{x} = \mathtt{e};\ p \rangle \phi, \Delta}$$

- Simple! No variable renaming, no ghost variables
- Dedicated rules needed for x = $\mathtt{e}_1 + \mathtt{e}_2$, etc.
- Works for scalar variable x and as long as e has no side effects
  $\Rightarrow$ need to come back to these issues

How to apply updates on updates?

Example

Symbolic execution of

```
x = x + y;
y = x - y;
x = x - y;
```

yields:

```
{x := x + y}{y := x - y}{x := x - y}
```

Need to compose three sequential state changes into a single one!

Parallel Updates

Compose several elementary updates into one parallel update:

Definition (Parallel Update)

A parallel update is an expression of the form $\{\mathrm{v}_1 := r_1 \,||\, \cdots \,||\, \mathrm{v}_n := r_n\}$

- All r_i computed in old state before update is applied
- Updates of all program variables v_i executed simultaneously
- Upon conflict $\mathrm{v}_i = \mathrm{v}_j, r_i \neq r_j$ later update ($\max\{i, j\}$) wins

Update composition achieved by rewrite rules such as:

$$\{\mathrm{v}_1 := r_1\}\{\mathrm{v}_2 := r_2\} \rightsquigarrow \{\mathrm{v}_1 := r_1 \,||\, \mathrm{v}_2 := \{\mathrm{v}_1 := r_1\}r_2\}$$

Parallel Updates: Example

Example

- $\{x := x + 1\}\{y := 2 * x\} \quad \rightsquigarrow \quad \{x := x + 1 \parallel y := 2 * (x + 1)\}$
  - Outer update also applied on right side of inner update
  - Sequential application replaced by simultaneous application
- $\{x := y \parallel y := x\}$
  - Describes swap of values of program variables x, y
  - Elementary updates within a parallel update independent of each other
- $\{x := 5 \parallel x := y + 1\} \quad \rightsquigarrow \quad \{x := y + 1\}$
  - Last variable assignment wins

Parallel updates store intermediate state of symbolic execution

Relation of DL Calculus to Symbolic Execution

```
a = a - b;
if (a < 0) {
    a = -a;
}
r = b / a;
```

1

Constraint
true

PV	Value
a	a_0
b	b_0
r	r_0

$$\mathit{true} \implies \{a := a_0 \| b := b_0 \| r := r_0\} \langle a = a - b; \text{if } (a < 0) \ldots \rangle \mathit{post}$$

```
a = a - b;
if (a < 0) {
    a = -a;
}
r = b / a;
```

1

Constraint
pre

PV	Value
a	a_0
b	b_0
r	r_0

$$pre \Longrightarrow \{a := a_0 \| b := b_0 \| r := r_0\} \langle \texttt{a = a - b; if (a < 0) ...} \rangle post$$

Unfolding complex expressions (here on the left side)

$$\frac{\Gamma \Longrightarrow \langle T_{nse}\ \mathsf{v};\ \mathsf{v} = nse;\ \mathsf{v}[\mathsf{e}] = \mathsf{e}';\ \mathsf{r}\rangle\phi, \Delta}{\Gamma \Longrightarrow \langle nse[\mathsf{e}] = \mathsf{e}';\ \mathsf{r}\rangle\phi, \Delta}$$

- Complex expressions may have side effects
- Unfold complex expressions in Java evaluation order (left-to-right)

Consequence: guards can assumed to be simple and side effect-free:

$$\text{if}\ \frac{\Gamma, \mathsf{b} \doteq \mathsf{true} \Longrightarrow \langle \mathsf{p};\ \mathsf{r}\rangle\phi, \Delta \qquad \Gamma, \mathsf{b} \doteq \mathsf{false} \Longrightarrow \langle \mathsf{q};\ \mathsf{r}\rangle\phi, \Delta}{\Gamma \Longrightarrow \langle \mathsf{if}\ (\mathsf{b})\ \{\ \mathsf{p}\ \}\ \mathsf{else}\ \{\ \mathsf{q}\ \};\ \mathsf{r}\rangle\phi, \Delta}$$

Array Assignment

$$\frac{\begin{array}{c}\Gamma, \mathsf{a} \not\doteq \mathsf{null}, \mathsf{o} \leq e < \mathsf{a.length} \Longrightarrow \{\mathsf{v} := \mathsf{a}[e]\}\ \langle \mathsf{r} \rangle \phi, \Delta \\ \Gamma, \mathsf{a} \doteq \mathsf{null} \Longrightarrow \langle \mathsf{throw\ new\ NullPointerException();}\ \mathsf{r} \rangle \phi, \Delta \\ \Gamma, \mathsf{a} \not\doteq \mathsf{null}, \mathsf{o} > e \vee e \geq \mathsf{a.length} \Longrightarrow \langle \mathsf{throw\ new\ AIOB();}\ \mathsf{r} \rangle \phi, \Delta\end{array}}{\Gamma \Longrightarrow \langle \mathsf{v = a[e];\ r} \rangle \phi, \Delta}$$

- Use symbolic **array update** $\mathsf{v} := \mathsf{a}[e]$ with dedicated set of rewrite rules
- All outcomes of array assignment must be considered
  (AIOB = ArrayIndexOutOfBoundException)

Method Invocation

$$\frac{?}{\Gamma \Longrightarrow \langle \mathsf{v = m(se); r} \rangle \phi, \Delta}$$

Option 1: Inline body of method m

- \+ Follows symbolic execution paradigm
- \+ Easy to implement

- \- Change to invoked method m requires re-verification of all callers breaks modularity
- \- Non-linear calls expensive & unbound recursion impossible

$$\mathrm{Contract}(m) \ := \ (pre, post[, mod]\,[, acc]\,[, trm])$$

Prerequisite: partial correctness, $mod = \emptyset$ (also no new objects)
(assumption can be removed, but beyond scope of tutorial; see later 'loop rule')

$$\frac{\Gamma \Longrightarrow \{u\}\,\{\mathrm{arg} := \mathrm{se}\}\,pre, \Delta \qquad \Gamma \Longrightarrow \{u\}\,\{\mathrm{arg} := \mathrm{se} \parallel \mathrm{res} := c\}\,(post \rightarrow \{\mathrm{v} := \mathrm{res}\}\,[r]\phi), \Delta}{\Gamma \Longrightarrow \{u\}\,[\mathrm{v} = m(se);\ r]\phi, \Delta}$$

- Program variables arg, res refer to method parameter, return value in *pre*, *post*
- c is Skolem constant

Correctness of contract application depends on proven contract for m:

$$pre \rightarrow [res = m(arg);]post \qquad \text{(where } m \text{ inlined!)}$$

Part V

Demo: Binary Search (Recursive)

Part VI

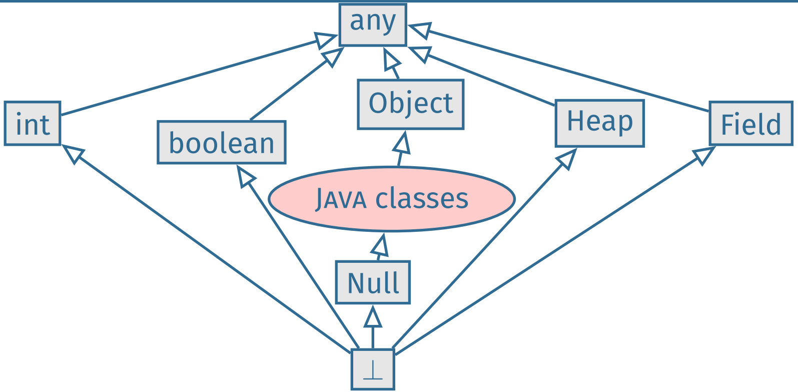
Towards Real Java

No escape from reality

Rules and updates work fine for scalar values, but in the real world…

- Java is object-oriented
  - Inheritance
  - Values on stack and heap
  - Complex object creation
  - …
- Aliasing
- Exceptions are thrown
- Loops have unknown bounds

Fixing a Java-based Type Hierarchy



Each interface and class in API and in target program becomes type
with appropriate subtype relation

Modeling the Heap in FOL

The Java Heap

Values of reference types (objects) live on the heap

- Heap values dynamically change during symbolic execution
- Each program state (model) relates objects to fields and values

The Java Heap Model of KeY

Data type Heap models content of heap in a given state (model)
Rigid functions model read and write access to fields in a given heap:

Write `Heap store (Heap, Object, Field, any);`
Modifies value of field of object to the value in the last argument

Read `any select (Heap, Object, Field);`
Selects value of field of object

Modeling Fields in FOL

Modeling instance fields

Person
int age **int** id
int setAge(**int** p_age) **int** getId()

- For each Java reference type C there is a signature type $C \in \mathrm{TSym}$, for example, Person
- For each Java field `f` there is a *unique* constant $\mathtt{f} \in \mathrm{FSym}$ of type Field, e.g., `Person::$age`

  When obvious, write `age` instead of `Person::$age`
- Domain of all Person objects: $\mathcal{D}^{\text{Person}}$
- Heap relates objects and fields to values (as seen)

Reading Fields (Simplified)

Signature FSym: `any select (Heap, Object, Field);`

Java expression p.age >= 0

Typed FOL `select(heap, p, age)` $>=$ `0`

heap is reserved program variable for "current" heap

Modeling Fields in FOL
The Full Story

Reading Fields

Signature FSym: `any select (Heap, Object, Field);`

`select(heap, p, age) >= 0` *well-formed?*

- Return type is "any"—need to cast to `int`
- There can be many fields with name `age`

Use function `int::select(heap, p, Person::$age)`

(`int::select` has same meaning as `(int)select`)

Writing to Fields

Signature FSym: `Heap store (Heap, Object, Field, any);`

Use function `store(heap, p, Person::$age, 42)`

Field Updates

The Global Program Variable heap

JavaDL has reserved program variable Heap heap

Heap stored in heap is used by Java program under verification for read / write field access

Changing the value of fields

How to translate assignment to field, for example, p.age=17;?

$$\frac{\Gamma \Longrightarrow \{\mathtt{heap} := \mathtt{store}(\mathtt{heap}, \mathtt{p}, \mathtt{age}, 17)\}\langle \mathtt{r}\rangle\phi, \Delta}{\Gamma \Longrightarrow \langle \mathtt{p.age = 17;\ r}\rangle\phi, \Delta}$$

Reasoning about Heaps: Symbolic Execution of Field Access

Reading a Field Value

Symbolic execution of accessing value of field f of object o with type T in heap h :
Rewrite rule performs lookup in h using pair (o, f) as key / index

selectOfStore

$$
\begin{array}{l}
select_T(store(h, u, g, v), o, f) \rightsquigarrow \\
\quad \text{if } (u \doteq o \wedge g \doteq f \wedge \neg(g \doteq \texttt{java.lang.Object.<created>})) \\
\quad \text{then } (v) \ \text{ else } (select_T(h, o, f))
\end{array}
$$

where

- h is a schema variable matching terms of type *Heap*
- u, o and v are schema variables matching terms of type *Any*
- f, g are schema variables matching terms of type *Field*

selectOfStore never changes value of field `<created>` used for object creation

Symbolic Execution of Field Access: Example

Example

f, g are fields of type **int** declared in class C; o, u program variables of type C

$$\mathbf{int}::\mathtt{select}(\mathtt{store}(\mathtt{heap}, \mathtt{o}, \mathtt{f}, 15), \mathtt{o}, \mathtt{f}) \leadsto 15$$
$$\mathbf{int}::\mathtt{select}(\mathtt{store}(\mathtt{heap}, \mathtt{o}, \mathtt{f}, 15), \mathtt{o}, \mathtt{g}) \leadsto \mathtt{int}::\mathtt{select}(\mathtt{heap}, \mathtt{o}, \mathtt{g})$$
$$\mathbf{int}::\mathtt{select}(\mathtt{store}(\mathtt{heap}, \mathtt{o}, \mathtt{f}, 15), \mathtt{u}, \mathtt{f}) \leadsto$$
$$\mathrm{if}\,((\mathtt{o} \doteq \mathtt{u}) \land \mathtt{f} \doteq \mathtt{f} \land \neg(\mathtt{f} \doteq \mathtt{<created>}))\,\mathrm{then}\,(15)\,\mathrm{else}\,(\mathbf{int}::\mathtt{select}(\mathtt{heap}, \mathtt{u}, \mathtt{f}))$$
$$\leadsto \mathrm{if}\,(\mathtt{o} \doteq \mathtt{u})\,\mathrm{then}\,(15)\,\mathrm{else}\,(\mathbf{int}::\mathtt{select}(\mathtt{heap}, \mathtt{u}, \mathtt{f}))$$

Pretty Printing

$T :: \mathtt{select}(\mathtt{heap}, \mathtt{o}, \mathtt{f})$	is shown as	$\mathtt{o.f}$
$\mathtt{select}(\mathtt{store}(\mathtt{heap}, \mathtt{o}, \mathtt{f}, 17), \mathtt{u}, \mathtt{f})$	is shown as	$\mathtt{u.f@heap[o.f} := 17]$

Symbolic Execution of Field Access: Example

Example

f, g are fields of type **int** declared in class C; o, u program variables of type C

$\mathbf{int}::\mathtt{select}(\mathtt{store}(\mathtt{hea}$ …

In the following we often use the pretty-printed version and omit the T :: prefix

… $(\mathtt{f} = \mathtt{<created>}))$ then (15) else $(\mathbf{int}::\mathtt{select}(\mathtt{heap}, \mathtt{u}, \mathtt{f}))$

$\leadsto$ if $(\mathtt{o} \doteq \mathtt{u})$ then (15) else $(\mathbf{int}::\mathtt{select}(\mathtt{heap}, \mathtt{u}, \mathtt{f}))$

Pretty Printing

$T :: \mathtt{select}(\mathtt{heap}, \mathtt{o}, \mathtt{f})$	is shown as	$\mathtt{o.f}$
$\mathtt{select}(\mathtt{store}(\mathtt{heap}, \mathtt{o}, \mathtt{f}, 17), \mathtt{u}, \mathtt{f})$	is shown as	$\mathtt{u.f@heap}[\mathtt{o.f} := 17]$

Recall method contract rule:

$$\frac{\Gamma \Longrightarrow \{u\}\,\{\mathrm{arg} := \mathrm{se}\}\, \mathit{pre}, \Delta \qquad \Gamma \Longrightarrow \{u\}\,\{\mathrm{arg} := \mathrm{se} \,||\, \mathit{res} := c\}\,\left(\mathit{post} \rightarrow \{\mathrm{v} := \mathrm{res}\}\,[r]\phi\right), \Delta}{\Gamma \Longrightarrow \{u\}\,[v = m(se);\; r]\phi, \Delta}$$

Assumed $\mathit{mod} = \emptyset$. To weaken this restriction:

1. Introduce **fresh** constant of type Heap, e.g., heap$'$
2. Anonymize current heap with location set *mod*:

$$\mathrm{anon}(\mathrm{heap}, \mathit{mod}, \mathrm{heap}')$$

3. Reassign current heap in **anonymizing update**:

$$\mathcal{V}_{\mathtt{mod}} = \{\mathrm{heap} := \mathrm{anon}(\mathrm{heap}, \mathit{mod}, \mathrm{heap}')\}$$

$\mathrm{anon}(\mathrm{h}, \mathit{locs}, \mathrm{h}')$ coincides with h on all locations except those in locs. These have the value in h'

With

$$\mathcal{V}_{\mathtt{mod}} = \{\mathrm{heap} := \mathrm{anon}(\mathrm{heap}, \mathit{mod}, \mathrm{heap}')\}$$

we have

$$\frac{\begin{array}{l}\Gamma \Longrightarrow \{u\}\,\{\mathrm{arg} := \mathrm{se}\}\,\mathit{pre}, \Delta \\ \Gamma \Longrightarrow \{u\}\,\{\mathcal{V}_{\mathtt{mod}} \,||\, \mathit{arg} := \mathrm{se} \,||\, \mathit{res} := c\}\,\big(\mathit{post} \rightarrow \{\mathrm{v} := \mathrm{res}\}\,[r]\phi\big), \Delta\end{array}}{\Gamma \Longrightarrow \{u\}\,[v = m(se);\ r]\phi, \Delta}$$

Still simplified. E.g., exceptions!

Idea behind loop invariants

- Formula *inv* whose validity is preserved by loop guard and body
- If, *inv* was valid at start of loop, it still holds after arbitrarily many loop iterations
- If the loop terminates at all, then *inv* must hold afterwards
- Like for contracts, anonymize heap after at least one iteration ($\mathcal{V}_{\mathtt{mod}}$)

$$\frac{\begin{array}{ll}\Gamma \Longrightarrow \{u\}\, inv, \Delta & \text{(Initially valid)}\\ \Gamma \Longrightarrow \{u\}\,\{\mathcal{V}_{\mathtt{mod}}\}\,((inv \wedge \mathrm{b} \doteq \mathit{TRUE}) \rightarrow [\mathit{body}](inv \wedge \mathit{frame})), \Delta & \text{(Preserved)}\\ \Gamma \Longrightarrow \{u\}\,\{\mathcal{V}_{\mathtt{mod}}\}\,((inv \wedge \mathrm{b} \doteq \mathit{FALSE}) \rightarrow [\mathrm{r}]\phi), \Delta & \text{(Use case)}\end{array}}{\Gamma \Longrightarrow \{u\}\,[\mathrm{while}\ (\mathrm{b})\ \{\mathit{body}\};\ \mathrm{r}]\phi, \Delta}$$

Limits of Simple Invariants

Limitations

The basic loop invariant rule:

1. Does not work for abrupt termination (break, return, exception), and
2. Does not allow guards with side effects

But KeY can deal with these as well!

Semantics of Abrupt Termination

Does abrupt termination count as normal termination?
No! Need to distinguish *normal* and *exceptional* termination

- $\langle p \rangle \phi$: p terminates normally and formula ϕ holds in final state (total correctness)
- $[p]\phi$: if p terminates normally then formula ϕ holds in final state (partial correctness)

Abrupt termination counts as non-termination! (More later)

Null Pointers

Null Pointer Exceptions

There are no "exceptions" in FOL: $\mathcal{I}(f)$ is a total function for $f \in \mathrm{FSym}$

Need to model possibility that o $\doteq$ null when symbolically executing o.a

- KeY branches over o != null upon each field access

JavaDL Assignment Rule for Fields

$$\text{assignmentToField}\quad \frac{\begin{array}{l}\Gamma, \neg(o \doteq \mathtt{null}) \Longrightarrow \{\mathtt{heap} := \mathtt{store}(\mathtt{heap}, o, f, v)\}\langle \mathrm{r}\rangle\phi, \Delta \\ \Gamma, (o \doteq \mathtt{null}) \Longrightarrow \langle \mathbf{throw\ new}\ \mathrm{NullPointerException}();\ \mathrm{r}\rangle\phi, \Delta\end{array}}{\Gamma \Longrightarrow \langle o.f = v;\ \mathrm{r}\rangle\phi, \Delta}$$

o, v schema variables matching program variables
f schema variable matching fields

`exceptional_behavior` specification case

Assume precondition (**`requires`** clause) *P* fulfilled

- Requires method to throw exception when pre-state satisfies *P*
- Keyword **`signals`** specifies post-state, depending on type of thrown exception
- Keyword **`signals_only`** specifies permitted type of thrown exception

JML specifications must separate normal and exceptional specification cases by *logically disjoint* preconditions

Real-world Integers

$$i \geq 0 \rightarrow \langle i = i + 1 \rangle (i > 0)$$

Is this formula valid for the Java type **int**?

- Obviously, not true in Java, for example, i == Integer.MAX_VALUE
- But we can currently prove it!
- Java integers on $(+, -, /, \%, \ldots)$ *do not* have the same meaning as in $\mathbb{Z}$

Comparison of Different Integer Semantics

Semantics	Sound	Complete	Remarks
$\mathtt{Java}_{\mathtt{math}}$	no	no	Good automation; Used for: teaching, prototyping proofs
$\mathtt{Java}_{\mathtt{javaSemantics}}$	yes	yes	Renders proofs complex, automation less powerful Use when correctness depends on overflow
$\mathtt{Java}_{\mathtt{checkedOverflow}}$	yes	no	Detects over-/underflow Usually, automation as good as in $\mathtt{Java}_{\mathtt{math}}$ Use when no overflow must happen

- “math” is called “arithmeticSemanticsIgnoringOF” in the actual KeY GUI
- sound and complete: relative to Java semantics as described in JLS

Part VII

Advanced Features for Object Orientation

What do we need to specify and verify complex (object-oriented) data structures?

Important Concepts

- Data Abstraction: State of a data structure can be represented using mathematical values.
- Data Encapsulation: Allows local reasoning.

Class/Object Invariants

How to encode properties about the valid states of the data structure?

```
class EvenIntArray {
    int[] a;                    // fields are non_null in JML by default
    //@ instance invariant (\forall int i; 0 <= i < a.length; a[i] % 2 == 0);
}
```

Invariant Semantics in KeY

- Invariant of `this` has to hold before and after each method call on `this`
- Invariant of `this` has to hold after termination of each constructor
- Exception: methods/constructors annotated with `helper`
- All other invariants need to be added explicitly: `\invariant_for(o)`

Model Fields

Model fields are specification-only[1] fields that

- can have a specification-only type (**\bigint**, **\seq**, ...)
- are observers (heap dependent functions), cannot be updated explicitly
- are computed from Java fields (i.e., do not add to the state space)
- must not be inconsistent (e.g. **represents** `x = x + 1;`)

Example:

```
//@ model \bigint absVal;
//@ represents absVal = f*c + g;   // c, f, g are "normal" Java fields
```

[1] no influence on the Java program, cannot be accessed in Java

Model methods are a generalization of model fields that

- consist of only a single return statement
- can be recursive
- can have contracts
- are often used for custom predicates/functions or lemmas

Example:

```
/*@ model_behavior
  @  ensures (\sum int i; 0 <= i < a.length; a[i]) == a.length * c;
  @ model boolean isConst(int[] a, int c) {
  @   return (\forall int i; 0 <= i < a.length; a[i] == c);
  @ } @*/
```

Ghost Fields/Variables

Ghost fields are specification-only fields that

- are treated like normal Java fields during verification
- are stored on the heap, accessed via select/store (i.e., add to the state space)
- need to be updated explicitly (via JML `set` statement)
- are usually coupled to the Java fields via object invariants

Example:

```
//@ ghost \bigint absVal;
//@ invariant absVal == c*f + g;    // f, g are "normal" Java fields

// in the constructor/method when updating the Java fields:
//@ set absVal = c*f + g;
```

Besides fields, also local ghost variables can be used (e.g. for intermediate results).

Model vs. Ghost

Model Fields/Methods

- do not add to the state space (more “beautiful” concept)
- provide an abstraction of the state
- proofs tend to get difficult, often need more interaction

Ghost Fields

- add to the state space
- need explicit set statements
- constructive nature often facilitates proofs

ADTs in KeY

What type can do we use for the specification-only fields that hold the abstract value of our data structure?

Algebraic Data Types (ADTs)

- built-in: **\seq** (with functions `seqGet`, `seqLength`, `seqUpdate`, ...)
- built-in: **\map** (with functions `mapGet`, `mapUpdate`, `mapRemove`, ...)
- user-defined ADTs (in .key file)

```
\datatypes {
    List = Nil | Cons(any head, List tail);
}
```

From this, some rules are generated (for manual application).

Inheritance of Specifications

Inheritance is an important OO concept, so what about specifications?

Behavioral Subtyping/Liskov Substitution Principle

Objects of subtype behave as specified in the superclass, i.e., they can be used wherever an object of the superclass is expected.

In KeY, behavioral subtyping is ensured:

- Contracts of superclasses are conjoined to those of subclasses.
- Object invariants are inherited.
- Model and ghost fields are inherited.
- Model methods are inherited and can be overwritten.

The Framing Problem

Encapsulation: We want to reason locally/modularly!

```
class Client {
  int x;
  int y;


  void m() {
    y = 5;
    resetX();
    assert y == 5;
  }
```

```
  /*@ ensures x == 0;
    @ assignable x;
    @
    @*/
  void resetX() {

    ...x = 0;
    y = 42;
  }
}
```

Does the assertion hold?

The Frame Problem: Concrete Aliasing

```java
class Client {
  IntList x;
  IntList y;

  //@ requires x != y;
  void m() {
    y.add(5);
    resetX();
    assert y.contains(5);
  }
```

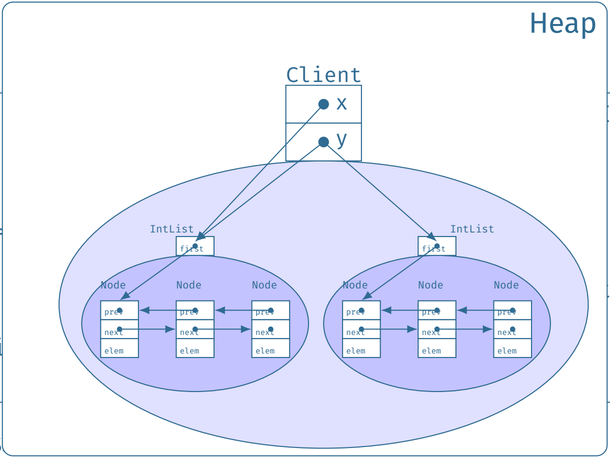
```java
  /*@ ensures x.isEmpty();
    @ assignable x;
    @
    @*/
  void resetX() {

    x = new IntList();

  }
}
```

What about this assertion?

```
class Client {
  IntList x;
  IntList y;

  //@ requires x !=
  void m() {
    y.add(5);
    resetX();
    assert y.contai
  }
```

```
mpty();



();
```

What about this as



The Frame Problem: Abstract Aliasing

```
class Client {
  IntList x;
  IntList y;

  /*@ requires x != y;
    @ requires \disjoint(x.footprint,
    @                    y.footprint);   @*/
  void m() {
    y.add(5);
    resetX();
    assert y.contains(5);
  }

  //@ assignable x.footprint;
  void resetX() {
    x.setElementsToZero();
  }
}
```

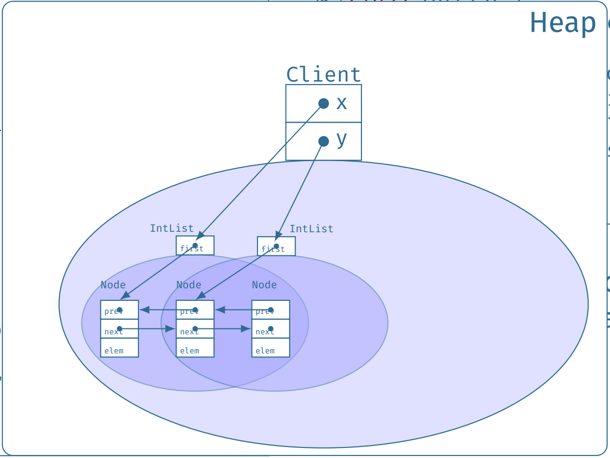
What about this assertion?

```
class IntList {
  /*@ nullable @*/ Node first;

  //@ ghost \locset footprint;
  /*@ invariant footprint == (this.* ∪
        first == null ? \empty
                 : first.footprint); @*/
    ...
}
```

Dealing with abstract aliasing is very challenging, especially for modular reasoning!

```
class Client {
  IntList x;
  IntList y;

  /*@ requires x != y;
    @ requires \disjoi
    @
  void m() {
    y.add(5);
    resetX();
    assert y.contains(
  }

  //@ assignable x.foo
  void resetX() {
    x.setElementsToZer
  }
}
```

```
class IntList {
e first;

otprint;
int == (this.* ∪
? \empty
st.footprint); @*/
```



ct aliasing is very
ally for modular

What about this assertion?

Dynamic Frame

Heap region that belongs to a data structure ("memory footprint").

- Described via ghost/model field or model method (usually "footprint" or "fp")
- JML type **`\locset`**: set of (object, field) pairs
- "Dynamic": Can grow over time, e.g. when nodes are added to a list.

Other Approaches for the Framing Problem

Separation Logic, Ownership Types, Implicit Dynamic Frames, ...

Further Framing Concepts

Read and Write Effects

- **assignable** `ls`: Write effect.
- **accessible** `ls`: Read effect (for non-void methods).

Important Syntax

- **assignable** `\nothing`: Only creation of new objects allowed.
- **assignable** `\strictly_nothing`: Nothing at all changed on the heap.
- `\fresh(ls)`: All locations in ls not allocated in the prestate.
- `\new_elems_fresh(ls)`: Only freshly allocated locations added to ls.
- `a[i..j]`: Location set containing the array elements a[i] to a[j].
- `o.*`: Location set containing all fields of o.

Dependency Specification Example

```
1  class Client {
2    IntList x, y;
3
4    /*@ requires x != y;
5      @ requires \disjoint(x.footprint,
6      @                    y.footprint); @*/
7    void m() {
8      assume y.get(0) == 5;
9      resetX();
10     assert y.get(0) == 5;
11   }
12
13   //@ assignable x.footprint;
14   void resetX() {
15     x.setElementsToZero();
16   }
17 }
```

```
18 class IntList {
19
20     //@ ghost \locset footprint;
21     //@ invariant footprint == ...
22
23     //@ accessible footprint;
24     int get(int idx) { ... }
25 }
```

We can deduce that the assertion holds (with lines 13 and 23 and disjointness of footprints)!

Method Calls in Specifications

Methods that are pure (i.e., change nothing on the heap and terminate) are allowed to be “called" in specifications.

```
interface IntList {
  /*@ pure @*/ int get(int idx);

  //@ ensures get(idx) == v;
  void set(int idx, int v);
}
```

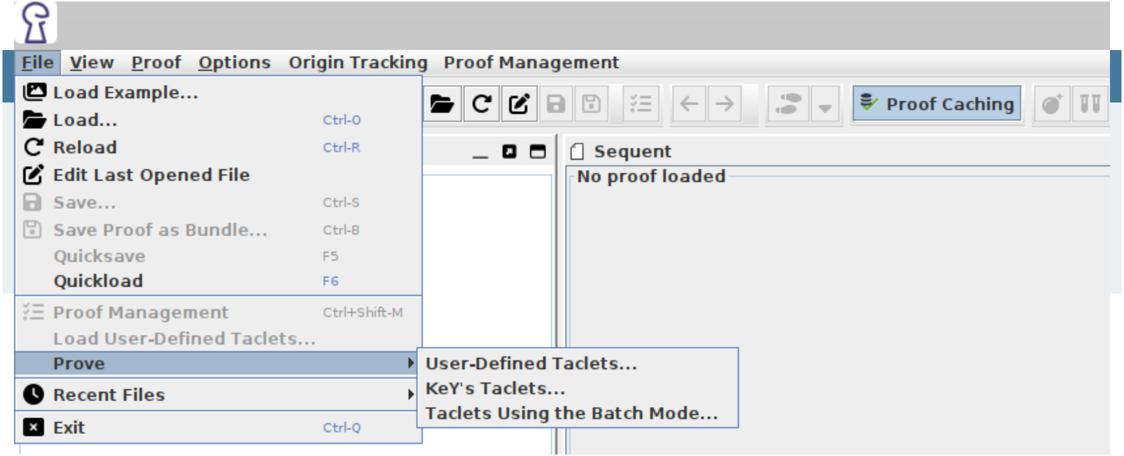
Note: Often, proofs are easier when abstraction and model/ghost fields are used instead (avoids additional modalities)!

Part VIII

Demo: ArrayList (with Ghost Fields)

Part IX

Inside KeY's core


File
View
Proof
Options
Origin Tracking
Proof Management
Load Example...
Load... Ctrl-O
Reload Ctrl-R
Edit Last Opened File
Save... Ctrl-S
Save Proof as Bundle... Ctrl-B
Quicksave F5
Quickload F6
Proof Management Ctrl+Shift-M
Load User-Defined Taclets...
Prove
Recent Files
Exit Ctrl-Q
User-Defined Taclets...
KeY's Taclets...
Taclets Using the Batch Mode...
Proof Caching
Sequent
No proof loaded

Adding a simple custom rule with Taclets

Implication to disjunction

$$a \rightarrow b \rightsquigarrow \neg a \vee b$$

Taclet

```
\schemaVariables { \formula a, b; }
\rules { impToOr {
      \find(a → b)
      \replacewith(¬ a ∨ b)
      \heuristics(simplify) };}
```

Adding a simple custom rule with Taclets

Implication to disjunction

```
\schemaVariables { \formula a, b; }
\rules { impToOr {
      \find(a → b)
      \replacewith(¬ a ∨ b)
      \heuristics(simplify) };}
```

Elements of taclets

Schema variables match against terms, formulas, or variables, according to their type

Implication to disjunction

```
\schemaVariables { \formula a, b; }
\rules { impToOr {
      \find(a → b)
      \replacewith(¬ a ∨ b)
      \heuristics(simplify) };}
```

Elements of taclets

find clause defines the *focus* formula to which the rule is applied

- Match only against formulas on antecedent or succedent by **\find**($\Longrightarrow$ formula) or **\find**(formula $\Longrightarrow$)

Adding a simple custom rule with Taclets

Implication to disjunction

```
\schemaVariables { \formula a, b; }
\rules { impToOr {
      \find(a → b)
      \replacewith(¬ a ∨ b)
      \heuristics(simplify) };}
```

Elements of taclets

replace clause replaces the focus

- Also *add clause* which adds a new formula: **add**($\Longrightarrow$ formula) or **add**(formula $\Longrightarrow$)

Implication to disjunction

```
\schemaVariables { \formula a, b; }
\rules { impToOr {
      \find(a → b)
      \replacewith(¬ a ∨ b)
      \heuristics(simplify) };}
```

Elements of taclets

heuristics clause adds the rule to the automatic proof search strategy

Taclet

```
null_can_always_be_stored_in_a_reference_type_array {
  \find(arrayStoreValid(array, null))
  \replacewith(true)
  \assumes( ==> array = null)
  \sameUpdateLevel
  \varcond(\isReferenceArray(array))
  \heuristics(simplify)
};
```

Elements of taclets

Assume clause for additional conditions

More complex taclets

Taclet

```
null_can_always_be_stored_in_a_reference_type_array {
   \find(arrayStoreValid(array, null))
   \replacewith(true)
   \assumes( ==> array = null)
   \sameUpdateLevel
   \varcond(\isReferenceArray(array))
   \heuristics(simplify)
  };
```

Elements of taclets

Taclet only applies if focus and assumption *under same update*

More complex taclets

Taclet

```
null_can_always_be_stored_in_a_reference_type_array {
  \find(arrayStoreValid(array, null))
  \replacewith(true)
  \assumes( ==> array = null)
  \sameUpdateLevel
  \varcond(\isReferenceArray(array))
  \heuristics(simplify)
};
```

Elements of taclets

Variable conditions state side conditions not expressible as formulas

Part X

Hands-On

Goal: Get the hands dirty with KeY.

Roadmap

Setup KeY → Selection Sort → Linked List

Exercise 1

Verification of **Selection Sort**

- Same level as Binary Search
- Algorithmic w/o Object-orientation

Exercise 2

Verification of **Linked List**

- Same level as Array List
- OO dealing with ghost

Setup KeY

1

Download the Hands-On Kit

- Download from key-project.org/tutorial-fm-2024/handson.zip
- Includes KeY, and the exercise files.
- `java -jar key-2.13.3-exe.jar`
- Test installation with built-in example: `SumAndMax`.

2

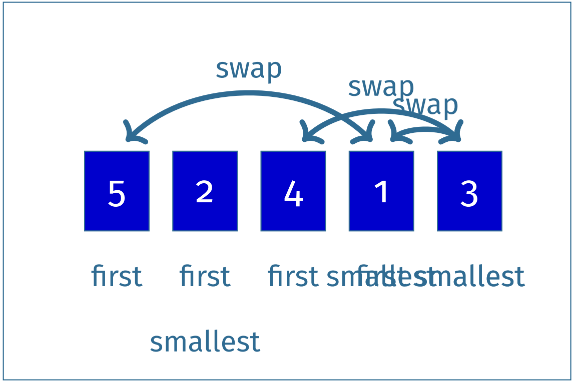
Editing

You can use any text editor you like.

3

Profit

A give-away for every KeY installation.



1. Divide list into sorted and unsorted sub-lists.
2. Search for the smallest element in the unsorted sub-list.
3. Swap first element of unsorted sub-list with smallest element.
4. Increase sorted sub-list.
5. *Repeat from (2)* until unsorted sub-list is larger than 1 element.

Selection Sort: Exercise I

1

Get into the `SelectionSort.java`

Start with **specification and verification** of `swap(array,i,j)`.

2

Start with loop invariant

Specification and verification of `min(array)`.

3

Proof sortedness of array

Find **post-conditions and loop invariant** to show:

$$a_1 \leq a_2 \leq \ldots \leq a_{n-1} \leq a_n$$

4

Permutation

Permutation is part of KeY theories:

- `\dl_array2seq(a)` – translates Java array *a* into a *Seq* (KeY sort)
- `\dl_seqPerm(a, \old(a))` – *a* is a permutation of $\backslash old(a)$

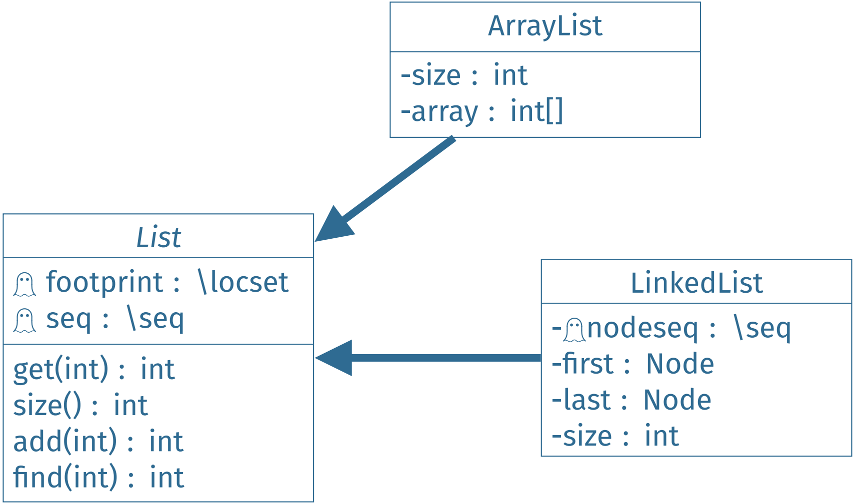
Selection Sort

Roadmap for specification & verification:

1. `swap(a, i, j)`
2. `min(a, start)`
3. `sort(a)` – sortedness $a_i \leq a_{i+1}$
4. `sort(a)` – permutation `\dl_seqPerm(a, \old(a))`

25 minutes

```
public class SelectionSort{
/*@ public normal_behaviour
    ensures (\forall int i; 0 <= i && i < a.length - 1; a[i] <= a[i+1]);
    ensures \dl_seqPerm(\dl_array2seq(a), \old(\dl_array2seq(a)));
    assignable a[*];
@*/
public void sort(int[] a) {
   /*@
   loop_invariant 0 <= i <= a.length;
   loop_invariant (\forall int j; 0 <= j && j < i; (\forall int k; j < k && k < a.length; a[j] <= a[k]));
   loop_invariant \dl_seqPerm(\dl_array2seq(a), \old(\dl_array2seq(a)));
   decreases a.length - i;
   assignable a[*];
   @*/
   for(int i = 0; i < a.length; i++){ int m = min(a,i); swap(a,m,i); }
}
/*@ public normal_behaviour
    requires 0 <= i < a.length && 0 <= j < a.length;
    ensures \old(a[i]) == a[j] && \old(a[j]) == a[i];
    ensures \dl_seqPerm(\dl_array2seq(a), \old(\dl_array2seq(a)));
    assignable a[i], a[j];
@*/
public void swap(int[] a, int i, int j){ int temp = a[i]; a[i] = a[j]; a[j] = temp; }
/*@ public normal_behaviour
    requires 0 <= start && start < a.length;
    ensures (\forall int i; start <= i && i < a.length; a[\result] <= a[i]);
    ensures start <= \result < a.length;
    assignable \strictly_nothing;
@*/
```


ArrayList
-size : int
-array : int[]
List
footprint : \locset
seq : \seq
get(int) : int
size() : int
add(int) : int
find(int) : int
LinkedList
-nodeseq : \seq
-first : Node
-last : Node
-size : int

Linked List: Excercise

1

Load

Try to inspect the proof obligation for `LinkedList` class.

2

Find the invariant to couple `\nodeseq` with

- `first`, `last` – the last and first node
- `size` – number of values
- `seq` – the sequence of values to the sequence of nodes
- `footprint` – the heap location of your `LinkedLists`

```
/*@ private invariant first == (size == 0 ? null : (Node)nodeseq[0]);
  @ private invariant last == (size == 0 ? null : (Node)nodeseq[size-1]);
  @
  @ private invariant size == seq.length && size == nodeseq.length;
  @
  @ private invariant (\forall int i; 0<=i && i<size;
  @           ((Node)nodeseq[i]) != null
  @        && ((Node)nodeseq[i]).data == (\bigint)seq[i]
  @        && (\forall int j; 0<=j && j<size;
  @              (Node)nodeseq[i] == (Node)nodeseq[j] ==> i == j)
  @        && ((Node)nodeseq[i]).next == (i==size-1 ? null : (Node)nodeseq[i+1]));
  @
  @ private invariant footprint == \set_union(this.*,
  @        (\infinite_union int i; 0<=i && i<size; ((Node)nodeseq[i]).*));
  @*/
```

Closing

Down the rabbit hole ...

`key-project.org/thebook2`

`key-project.org/tutorial-fm-2024`

KeYProject/key

`support@key-project.org`

Case Studies

- ips4o (TACAS'24)
- IdentityHashMap (iFM'22)
- DualPivotQuickSort (VSTTE'17)
- TimSort (CAV'15)

KeY is a **tool, library, and a platform** for/of your research.

Thank you for joining the tutorial!

Have a lot of fun at FM 2024!