Formal Methods for Java

Lecture 15: Jahob

Jochen Hoenicke

Software Engineering
Albert-Ludwigs-University Freiburg

December 11, 2012

Internals of a Static Checker

- Topic of the next lectures:
  How does a Static Checker work?
- We will look into Jahob.

The Jahob system

Focus of Jahob: verifying properties of data structures.

Developed at

- EPFL, Lausanne, Switzerland (Viktor Kuncak)
- MIT, Cambridge, USA (Martin Rinard)
- Freiburg, Germany (Thomas Wies)

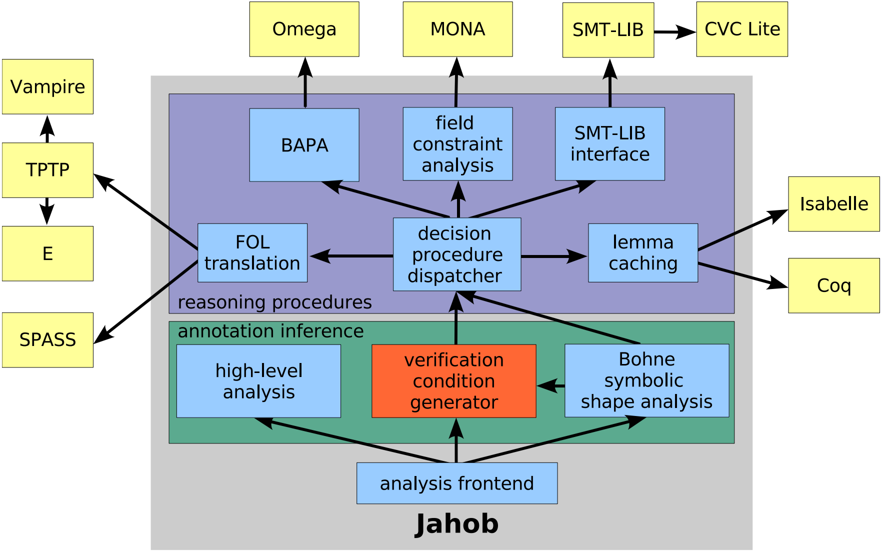
References

- Jahob webpage: `http://lara.epfl.ch/w/jahob_system`
- Viktor Kuncak's PhD thesis

Comparison of ESC/Java and Jahob

	ESC/Java	Jahob
Goal	find bugs	prove correctness
Spec. language	JML	based on Isabelle/HOL
Java support	aims at full Java	subset of Java (no exceptions, no concurrency, no generics, no dyn. dispatch, ...)
Loop invariants	optional	provided by user or automatically derived
Completeness	only linear arithmetic with free function symbols	general purpose theorem provers and decision procedures for specialized theories

Jahob system architecture



Isabelle/HOL

Jahob's assertion language is a subset of the interactive theorem prover Isabelle/HOL which is built on the simply typed lambda calculus.

Why Isabelle/HOL and not e.g. JML?

- ➜ natural syntax
- ➜ unifying semantic foundation for all specification constructs
- ➜ no artificial limitations regarding expressiveness
- ➜ decision procedures can be used to automate reasoning
- ➜ interactive theorem provers can be used for
  - debugging the system
  - proving the most difficult theorems interactively

Core syntax of HOL

Terms and Formulas:

f	$::=$	$\lambda x :: t.\, f$	lambda abstraction (λ is also written %)
	$\mid$	$f_1\ f_2$	function application
	$\mid$	x	variable or constant
	$\mid$	$f :: t$	typed formula

Types:

t	$::=$	bool	truth values
	$\mid$	int	integers
	$\mid$	obj	uninterpreted objects
	$\mid$	$t_1 \Rightarrow t_2$	total functions
	$\mid$	t set	sets
	$\mid$	$t_1 * t_2$	*pairs*

Predefined constants in HOL

Core syntax is enriched with predefined constants:

- Boolean connectives: `~ F, F & G, F | G, F --> G, F <-> G`
- (dis)equality: `f = g, f ~= g`
- sets and set operations:
  `{f_1, ..., f_n}, {x. F}, f : S, S Un T, S Inter T, S - T`
- quantification: `ALL x. F, EX x. F`
- reflexive transitive closure of predicates: `rtrancl_pt P a b`
- the null object: `null`
- ...

Example formula:

```
rtrancl_pt = % (P :: obj => obj => bool) (a :: obj) (b :: obj).
         ALL S. a : S & (ALL x y. x : S & P x y --> y : S) -->
                b : S
```

Verification conditions

Goal: reduce correctness of a program to the validity of logical formulae.

Consider program fragment (verification condition):

$$assume(F); c; assert(G);$$

Idea for proving correctness:

- start from G and symbolically execute c backwards
- prove that F implies the resulting formula

Backwards execution is done by computing weakest preconditions.

Weakest precondition $\mathrm{wp}(c, G)$ is the weakest formula such that

$$\forall q_0, q_1 .\, q_0 \models \mathrm{wp}(c, G) \wedge q_0 \xrightarrow{c} q_1 \text{ implies } q_1 \models G$$

Loop-free guarded commands

Internally, Jahob uses a simplified language to represent programs.

$$
\begin{array}{rll}
c ::= & x := \mathit{formula} & \text{(side-effect free assignment statement)} \\
| & \text{havoc}(x) & \text{(non-deterministic assignment to } x\text{)} \\
| & \text{assume}(\mathit{formula}) & \text{(assume statement)} \\
| & \text{assert}(\mathit{formula}) & \text{(assert statement)} \\
| & c_1 \,;\, c_2 & \text{(sequential composition)} \\
| & c_1 \,\Box\, c_2 & \text{(non-deterministic choice)}
\end{array}
$$

Semantics of guarded commands

Weakest precondition semantics of guarded commands:

$$
\begin{aligned}
\mathrm{wp}(x := e, G) &\equiv \forall x'.\, x' = e \rightarrow G[x'/x] && x' \text{ fresh}\\
\mathrm{wp}(\mathit{havoc}(x), G) &\equiv \forall x.\, G\\
\mathrm{wp}(\mathit{assert}(F), G) &\equiv F \wedge G\\
\mathrm{wp}(\mathit{assume}(F), G) &\equiv F \rightarrow G\\
\mathrm{wp}(c_1 \,;\, c_2, G) &\equiv \mathrm{wp}(c_1, \mathrm{wp}(c_2, G))\\
\mathrm{wp}(c_1 \,\Box\, c_2, G) &\equiv \mathrm{wp}(c_1, G) \;\wedge\; \mathrm{wp}(c_2, G)
\end{aligned}
$$

Generated formulas are linear in the size of the program.

Translating Java to Guarded Commands (1)

Jahob does not support Java statements with side effects such as

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

Instead one can transform this to side-effect free code beforehand:

```
x = y;
y = y+1;
```

Translating Java to Guarded Commands (2)

Conditions are translated to choice and assume:

```
if (x > 0) { z = x } else { z = -x }
```

is translated to

$$(assume(x > 0); z := x) \,\Box\, (assume(\neg(x > 0)); z := -x)$$

Desugaring loops with invariants

$$\texttt{while}\ [\texttt{inv}\ I]\ (F)\ c$$

Combine previous cases to one guarded command:

$assert(I);$
$havoc(x_1, ..., x_n);$
$assume(I);$
$\quad (assume(\neg F) \,\square$
$\quad assume(F);$
$\quad c;$
$\quad assert(I);$
$\quad assume(false))$

Desugaring method calls

Call of a method p: $z := p(v)$

where $p(u)$ has specification:

$\quad$ *requires* $pre(x, y, u)$
$\quad$ *modifies* x
$\quad$ *ensures* $post(old(x), x, y, u, result)$

call is desugared to:

$\quad assert(pre(x, y, v));$
$\quad x_0 := x;$
$\quad havoc(x);$
$\quad havoc($"private representation"$);$
$\quad havoc(z);$
$\quad assume(post(x_0, x, y, v, z))$

Notice: Before any reentrant call to an object of the same class the class invariants must be reestablished.

References and fields (1)

Fields are total functions on objects:

$$Node.next :: obj \Rightarrow obj$$

we have by definition $Node.next\ null = null$.

Field access is just function application:

$$\texttt{y = x.next} \quad \text{becomes} \quad y := Node.next\ x$$

References and fields (2)

Fields are total functions on objects:

$$Node.next :: obj \Rightarrow obj$$

we have by definition $Node.next\ null = null$.

Field update is function update:

$$\texttt{x.next = y} \quad \text{becomes} \quad Node.next := Node.next[x := y]$$

where $f[x := y](z) = f(z)$ for $z \neq x$ and $f[x := y](x) = y$.

Updates on fields can be eliminated:

$$\begin{aligned} & \mathrm{wp}(Node.next := Node.next[x := y], Node.next\ z = t) \\ \equiv\ & Node.next[x := y]\ z = t \\ \equiv\ & (z = x \land y = t)\ \lor\ (z \neq x \land Node.next\ z = t) \end{aligned}$$

Allocation of objects

Introduce a new set valued variable $Object.alloc$ :: obj set to denote all allocated objects

```
x = new T();
```

becomes:

$havoc(x)$;
$assume(x \notin Object.alloc)$;
$assume(x \in T)$;
$Object.alloc := Object.alloc \cup \{x\}$;
Translation of call of constructor $x.T()$

Demo