Java+ITP: A Verification Tool Based on Hoare Logic and Algebraic Semantics

Ralf Sasse, José Meseguer

Department of Computer Science
University of Illinois at Urbana-Champaign

6th International Workshop on Rewriting Logic and its Applications, 2006
1 Motivation
   - Long-Term Goals
   - General Idea
   - Previous Work

2 Our Work
   - Preliminaries
   - Main Results
   - An Example
Investigate modularity and extensibility of
- programming languages
- Hoare logics

Source-code level reasoning.

Generic and modular program logics wanted.

Develop theorem proving technology on top of the logics.

This case study is a first step in this direction.
Problem Considered

- Sequential Java subset:
  - arithmetic expressions, assignments, sequential composition, loops
- Hoare logic for this programming language with side-effects, mathematically justified.
- Standard Hoare rules break down!
- Mechanization available, supporting:
  - Compositional reasoning to decompose Hoare triples
  - Generation of first-order verification conditions
  - Discharging verification conditions with Maude’s inductive theorem prover (ITP)
We have benefitted from previous experience at UCM on ASIP+ITP, a tool that works on Hoare logics for a simple toy language.

However, in ASIP+ITP:

- no side-effects in conditions
- no variable declarations and shadowing
- simple state infrastructure: direct mapping of variables to values
- no compositional decomposition of Hoare triples supported
Java Semantics

- Algebraic semantics (only equations used)
- Continuation passing style approach
- Extracted from a larger rewriting logic semantics for Java (JavaFAN)
Hoare Triples

- Consist of:
  - pre condition \( A \)
  - post condition \( B \)
  - program \( p \)

- Written: \{A\} \( p \) \{B\}

- Implicit state \( S \) used, made explicit when necessary.

- Meaning: \( A(S) \Rightarrow B(S/(S|p)) \)
Our Hoare Rules

- Standard sequential composition rule:

\[
\{ A \} \ p \ \{ B \} \quad \{ B \} \ q \ \{ C \}
\]

\[
\{ A \} \ p \ q \ \{ C \}
\]

- Standard conditional rule, breaks down here:

\[
\{ A \land t \} \ p \ \{ B \} \quad \{ A \land \neg t \} \ q \ \{ B \}
\]

\[
\{ A \} \text{ if } t \ p \ \text{ else } q \ \{ B \}
\]

- Correct conditional rule, for side-effects in condition:

\[
\{ A \land t \} \ t \mid p \ \{ B \} \quad \{ A \land \neg t \} \ t \mid q \ \{ B \}
\]

\[
\{ A \} \text{ if } t \ p \ \text{ else } q \ \{ B \}
\]
Our Hoare Rules

- **Standard sequential composition rule:**

\[
\frac{\{A\} p \{B\} \quad \{B\} q \{C\}}{\{A\} p \ q \ \{C\}}
\]

- **Standard conditional rule,** breaks down here:

\[
\frac{\{A \land t\} p \{B\} \quad \{A \land \neg t\} q \{B\}}{\{A\} \text{ if } t \ p \ \text{ else } q \ \{B\}}
\]

- **Correct conditional rule,** for side-effects in condition:

\[
\frac{\{A \land t\} t \ | \ p \ \{B\} \quad \{A \land \neg t\} t \ | \ q \ \{B\}}{\{A\} \text{ if } t \ p \ \text{ else } q \ \{B\}}
\]
Our Hoare Rules

- Standard sequential composition rule:

$$\{A\} p \{B\} \quad \{B\} q \{C\}$$

$$\{A\} p \; q \; \{C\}$$

- Standard conditional rule, breaks down here:

$$\{A \land t\} p \{B\} \quad \{A \land \neg t\} q \{B\}$$

$$\{A\} \text{if } t \; p \; \text{else } q \{B\}$$

- Correct conditional rule, for side-effects in condition:

$$\{A \land t\} t \mid p \{B\} \quad \{A \land \neg t\} t \mid q \{B\}$$

$$\{A\} \text{if } t \; p \; \text{else } q \{B\}$$
Our Hoare Rules

- Standard while loop rule, **breaks down here**:

\[
\{A \land t\} p \{A\}
\]

\[
\{A\} \text{ while } t \ p \ \{A \land \neg t\}
\]

- Correct while loop rule, for side-effects in condition:

\[
\{A \land t\} t \mid p \{A\} \quad \{A \land \neg t\} \ t \ \{A \land \neg t\}
\]

\[
\{A\} \text{ while } t \ p \ \{A \land \neg t\}
\]
Our Hoare Rules

- Standard while loop rule, breaks down here:

\[
\{A \land t\} \quad p \quad \{A\}
\]
\[
\{A\} \quad \text{while } t \quad p \quad \{A \land \neg t\}
\]

- Correct while loop rule, for side-effects in condition:

\[
\{A \land t\} \quad t \mid p \quad \{A\} \\
\{A \land \neg t\} \quad t \quad \{A \land \neg t\}
\]
\[
\{A\} \quad \text{while } t \quad p \quad \{A \land \neg t\}
\]
Java+ITP tool: javax-inv

- The `javax-inv` command applies multiple rules and creates first-order proof obligations.
- Target Hoare triple is `{P} init loop {Q}`.

\[
\begin{align*}
\{P\} \text{ init } \{I\} & \quad \frac{I \Rightarrow I}{\{I\} \text{ loop } \{I \land \neg t\}} \quad \frac{(I \land \neg t) \Rightarrow Q}{\{Q\}} \\
\{P\} \text{ init } \text{ loop } \{Q\}
\end{align*}
\]

- `loop = while t p`:

\[
\begin{align*}
\{I \land t\} \text{ t } \mid p \{I\} & \quad \{I \land \neg t\} \text{ t } \{I \land \neg t\} \\
\{I\} \text{ while t p } \{I \land \neg t\}
\end{align*}
\]
Java+ITP tool: javax-inv

- The \texttt{javax-inv} command applies multiple rules and creates first-order proof obligations.
- Target Hoare triple is $\{P\} \text{ init } \text{ loop } \{Q\}$.

\[
\begin{array}{c}
\{P\} \text{ init } \{I\} \\
I \Rightarrow I \quad \{I\} \text{ loop } \{I \land \neg t\} \quad (I \land \neg t) \Rightarrow Q \\
\{I\} \text{ loop } \{Q\} \\
\{P\} \text{ init } \text{ loop } \{Q\}
\end{array}
\]

- loop = while $t$ $p$:

\[
\begin{array}{c}
\{I \land t\} \quad t \mid p \quad \{I\} \quad \{I \land \neg t\} \quad t \quad \{I \land \neg t\} \\
\{I\} \quad \text{while } t = p \quad \{I \land \neg t\}
\end{array}
\]
Java+ITP tool: Compositionality

- Hoare triples can be added as proof goals without translating them to first-order goals right away.
- Java+ITP supports decomposition of such Hoare triples.
- Simpler Hoare triples can be translated to first-order goals on user command.
Example Program

Binomial coefficient program for $\binom{n}{k}$, with inputs ‘N and ‘K.

```java
int 'N ; int 'Nfac ; int 'K ; int 'Kfac ;
int 'N-Kfac ; int 'BC ; int 'I ;

'I = #i(0) ; 'Nfac = #i(1) ;
while ('I < 'N)
    { 'I = 'I + #i(1) ; 'Nfac = 'Nfac * 'I ; }
'I = #i(0) ; 'Kfac = #i(1) ;
while ('I < 'K)
    { 'I = 'I + #i(1) ; 'Kfac = 'Kfac * 'I ; }
'I = #i(0) ; 'N-Kfac = #i(1) ;
while ('I < ('N - 'K))
    { 'I = 'I + #i(1) ; 'N-Kfac = 'N-Kfac * 'I ; }
'BC = 'Nfac / ('Kfac * 'N-Kfac) ;
```

Ralf Sasse, José Meseguer
Verification Property

Property to be verified, with $S$ the state and $N$ and $K$ the inputs:

\[
\{ S[\text{'N}] = N \land S[\text{'K}] = K \\
\land 0 \leq N \land 0 \leq K \land 0 \leq N - K \}
\]

**binomial-coefficient-program**

\[
\{ S[\text{'Nfac}] = N! \\
\land S[\text{'Kfac}] = K! \\
\land S[\text{'N-Kfac}] = (N - K)! \\
\land S[\text{'BC}] = bc(N, K) \}
\]
Example Roadmap

- Load all necessary parts into Maude:
  - our modified ITP
  - our Java semantics
  - the module defining the binomial coefficient code
- Enter the property we want to show, as seen above.
- Decompose the proof obligation.
- Transform proof obligations to first-order proof obligations, done for one proof obligation here.
- Use ITP to discharge the first-order proof obligations.
select ITP-TOOL .
loop init-itp .

(add-hoare-triple CHOOSE-JAVAX :
--- specification variables
(N:Int ; K:Int)
--- precondition
   ((int-val(S:WrappedState[‘N])) = (N:Int)
& (int-val(S:WrappedState[‘K])) = (K:Int)
& (0 <= N:Int) = (true)
& (0 <= K:Int) = (true)
& (0 <= N:Int - K:Int) = (true))
--- program
choose
--- postcondition
((int-val(S:WrappedState[‘Nfac]))
 = ((N:Int)!) & (int-val(S:WrappedState[‘Kfac]))
 = ((K:Int)!) & (int-val(S:WrappedState[‘N-Kfac]))
 = ((N:Int - K:Int)!) & (int-val(S:WrappedState[‘BC]))
 = (choose(N:Int, K:Int)))
--- initialization code
choose-init.

Ralf Sasse, José Meseguer
Decompose Proof Obligation

(decomp:
---- name of the HT
choose@0
---- prefix code
facN
---- midcondition
   ((int-val(S:WrappedState['N])) = (N:Int)
& (int-val(S:WrappedState['K])) = (K:Int)
& (0 <= N:Int) = (true)
& (0 <= K:Int) = (true)
& (0 <= N:Int - K:Int) = (true)
& (int-val(S:WrappedState['Nfac]))
   = ((N:Int)!))
---- rest of code
choose/facN .)
First Decomposed Proof Obligation

hoare-label: choose@1.0

\{ \text{int-val}(S:\text{WrappedState}[\text{'N'}]) = N:\text{Int})
& \text{int-val}(S:\text{WrappedState}[\text{'K'}]) = K:\text{Int})
& (0 \leq N:\text{Int} - K:\text{Int} = \text{true})
& (0 \leq N:\text{Int} = \text{true})
& (0 \leq K:\text{Int} = \text{true}) \} \text{facN}

\{ \text{int-val}(S:\text{WrappedState}[\text{'Nfac'}]) = N:\text{Int} !)
& \text{int-val}(S:\text{WrappedState}[\text{'N'}]) = N:\text{Int})
& \text{int-val}(S:\text{WrappedState}[\text{'K'}]) = K:\text{Int})
& (0 \leq N:\text{Int} - K:\text{Int} = \text{true})
& (0 \leq N:\text{Int} = \text{true})
& (0 \leq K:\text{Int} = \text{true}) \}
Create First-Order Proof Obligation for `choose@1.0`

```
(create-FO-goal-hoare-inv:
  ---- name of the HT
  choose@1.0
  ----</invariant
  ((int-val(S:WrappedState['Nfac]))
  = ((int-val(S:WrappedState['I']))!)
  & (0 <= int-val(S:WrappedState['I'])) = (true)
  & (int-val(S:WrappedState['I'])
  <= int-val(S:WrappedState['N'])) = (true)
  & (int-val(S:WrappedState['N'])) = (N:Int)
  & (int-val(S:WrappedState['K'])) = (K:Int)
  & (0 <= N:Int) = (true)
  & (0 <= K:Int) = (true)
  & (0 <= N:Int - K:Int) = (true) ) .)
```
Proof of concept for Hoare logics for a non-trivial language.
Compositionality on the source-code level.
Useful in teaching about algebraic semantics and Hoare logics.

Outlook
Extend this to a larger Java fragment.
Generalize it, independent of the actual language, each rule only dependent on a single language feature.