Testing, Debugging, Program Verification

Formal Verification, Part I

Wolfgang Ahrendt & Vladimir Klebanov & Moa Johansson

27 November 2012

Block 4: Formal Verification

- Three lectures (today + Mon, Wed next week)
- One exercise session (next Wed)
- One assignment to hand in.

Todays main topics:

- Symbolic Execution
- Hoare Logic
- A first look at the KeY-Hoare system (if we have time).

Formal Software Verification: Motivation

Limitations of Testing

- Testing ALL inputs is usually impossible.
- Even strongest coverage criteria cannot guarantee abcence of further defects.

Goal of Formal Verification

Given a formal specification S of the behaviour of a program P:
Give a mathematically rigorous proof that each run of P conforms to S

P is correct with respect to S

The Main Steps towards Formal Verification

1. Write a specification of a given program that can be proven
2. Devic a correctness proof method without exhaustive case analysis
3. Design mathematically rigorous proof rules: "calculus"

Formal Software Verification: Limitations

- No absolute notion of program correctness!
  - Correctness always relative to a given specification
  - Example: forgot to specify permutation property for `sort()`
- Hard and expensive to develop provable formal specifications
  - In practice, no attempt to specify full functionality.
  - Safety properties e.g.
    - Well-formed data
    - Exception freeness, ...
- Some properties may be difficult or impossible to specify. e.g.
  - Time and memory (possible, but not done here)
  - User behaviour, the environment in general

Formal Software Verification: Limitations cont.

- Requires lots of expertise and expenses
- Even fully specified & verified programs can have runtime failures
  - Defects in the compiler
  - Defects in the runtime environment
  - Defects in the hardware

Possible & desirable: Exclude defects in source code wrt a given spec

What is Symbolic Execution?

Concrete Execution

- State: a concrete valuation of all variables (stack) and fields (heap)
- (Execution) Path: finite OR infinite sequence of states that a program passes as it executes
- Program Counter: States along the path are annotated with next (sub-)statement to be executed.
- Initial state given explicitly

What is Symbolic Execution?

Symbolic Execution

- State: a "symbolic" valuation of all variables and fields.
  - New symbols to denote initial value of variables etc.
  - Each term represents a set of possible concrete values
- Execution Tree: finite OR infinite tree of states
- Program Counter: States in the tree are annotated with next (sub-)statement to be executed
- Path Condition: Annotations on branching state transitions.
- Each concrete execution path is an instance of some symbolic path through the tree
- Initial state given explicitly or by a symbolic precondition

What is Symbolic Execution?

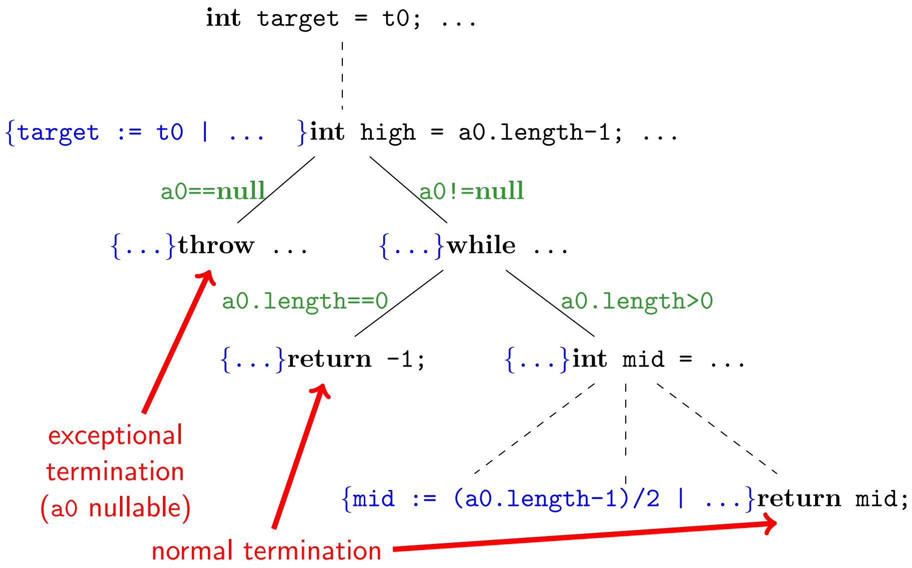
Execute a program with symbolic (abstract) initial values

Assume we could write a Java program such as this:

```
int target  = t0;
int[] array = a0;
return search(array, target);
```

where t_0 and a_0 are arbitrary start values.

Symbolic Execution Tree



Symbolic Execution by Example

```
int target  = t0;   ⟵ Execute this statement
int[] array = a0;
int low = 0;
int high = array.length-1;

while ( low <= high ) {
  int mid = (low + high) / 2 ;
  if ( target < array[ mid ] ) {
     high = mid - 1;
  } else if ( target > array[ mid ] ) {
     low = mid + 1;
  } else {
     return mid;
  }
}
return -1;
```

Symbolic Execution by Example

{target := t_0} ← Symbolic Program State

```
int[] array = a0;          ← First Active Statement (Program Counter)
int low = 0;
int high = array.length-1;

while ( low <= high ) {
  int mid = (low + high) / 2 ;
  if ( target < array[ mid ] ) {
     high = mid - 1;
  } else if ( target > array[ mid ] ) {
     low = mid + 1;
  } else {
     return mid;
  }
}
return -1;
```

Symbolic Execution by Example

{target := t_0 | array := a_0}

```
int low = 0;
int high = array.length-1;

while ( low <= high ) {
  int mid = (low + high) / 2 ;
  if ( target < array[ mid ] ) {
     high = mid - 1;
  } else if ( target > array[ mid ] ) {
     low = mid + 1;
  } else {
     return mid;
  }
}
return -1;
```

Symbolic Execution by Example

{target := t_0 | array := a_0 | low := 0}

```
int high = array.length-1;   <-- What next?

while ( low <= high ) {
  int mid = (low + high) / 2 ;
  if ( target < array[ mid ] ) {
     high = mid - 1;
  } else if ( target > array[ mid ] ) {
     low = mid + 1;
  } else {
     return mid;
  }
}
return -1;
```

Symbolic Execution by Example

{target := t_0 | array := a_0 | low := 0}

```
int high = a0.length-1;
```

← Execution depends on a_0 != null

```
while ( low <= high ) {
  int mid = (low + high) / 2 ;
  if ( target < array[ mid ] ) {
     high = mid - 1;
  } else if ( target > array[ mid ] ) {
     low = mid + 1;
  } else {
     return mid;
  }
}
return -1;
```

Symbolic Execution by Example

a_0`!=null` ← Path Condition (fulfilled in JML by default)

{`target :=` t_0 `| array :=` a_0 `| low := 0 | high :=` a_0`.length-1`}

```
while ( low <= high ) {
  int mid = (low + high) / 2 ;
  if ( target < array[ mid ] ) {
     high = mid - 1;
  } else if ( target > array[ mid ] ) {
     low = mid + 1;
  } else {
     return mid;
  }
}
return -1;
```

Symbolic Execution by Example

a_0!=null

{target := t_0 | array := a_0 | low := 0 | high := a_0.length-1}

```
while ( low <= high ) {      <-- depends on  a0.length>0
  int mid = (low + high) / 2 ;
  if ( target < array[ mid ] ) {
     high = mid - 1;
  } else if ( target > array[ mid ] ) {
     low = mid + 1;
  } else {
     return mid;
  }
}
return -1;
```

Symbolic Execution by Example

a_0`!=null && `a_0`.length > 0`

`{target := `t_0` | array := `a_0` | low := 0 | high := `a_0`.length-1}`

```
int mid = (low + high) / 2 ;                  ⟵ unwind loop
if ( target < array[ mid ] ) {
   high = mid - 1;
} else if ( target > array[ mid ] ) {
   low = mid + 1;
} else {
   return mid;
}                                             ⟵ end of loop body
while ( low <= high ) {
 ...
}
return -1;
```

Symbolic Execution by Example

```
a0!=null && a0.length > 0
{target := t0 | array := a0 | low := 0 | high := a0.length-1 |
 mid := (a0.length-1)/2}

if ( target < array[ mid ] ) {
   high = mid - 1;
} else if ( target > array[ mid ] ) {
   low = mid + 1;
} else {
   return mid;
}
while ( low <= high ) {
 ...
}
return -1;
```

Symbolic Execution by Example

```
a₀!=null && a₀.length > 0
{target := t₀ | array := a₀ | low := 0 | high := a₀.length-1 |
 mid := (a₀.length-1)/2}

if ( t₀ < a₀[ (a₀.length-1)/2 ] ) {   Why no exception thrown?
   high = mid - 1;
} else if ( target > array[ mid ] ) {
   low = mid + 1;
} else {
   return mid;
}
while ( low <= high ) {
 ...
}
return -1;
```

Symbolic Execution by Example

```
a0!=null && a0.length > 0
{target := t0 | array := a0 | low := 0 | high := a0.length-1 |
 mid := (a0.length-1)/2}

if ( t0 < a0[ (a0.length-1)/2 ] ) { let t0==a0[(a0.length-1)/2]
   high = mid - 1;
} else if ( target > array[ mid ] ) {
   low = mid + 1;
} else {
   return mid;
}
while ( low <= high ) {
 ...
}
return -1;
```

Symbolic Execution by Example

a_0!=null && a_0.length > 0 && t_0==a_0[(a_0.length-1)/2]
{target := t_0 | array := a_0 | low := 0 | high := a_0.length-1 | mid := (a_0.length-1)/2}

```
if ( target > array[ mid ] ) {
   low = mid + 1;
} else {
   return mid;
}
while ( low <= high ) {
 ...
}
return -1;
```

Symbolic Execution by Example

a_0`!=null && `a_0`.length > 0 && `t_0`==`a_0`[ (`a_0`.length-1)/2 ]`

{`target := `t_0` | array := `a_0` | low := 0 | high := `a_0`.length-1 |`
` mid := (`a_0`.length-1)/2`}

```
if ( t0 > a0[ (a0.length-1)/2 ] ) {   ← false!
    low = mid + 1;
} else {
    return mid;
}
while ( low <= high ) {
  ...
}
return -1;
```

Symbolic Execution by Example

a_0`!=null && `a_0`.length > 0 && `t_0`==`a_0`[ (`a_0`.length-1)/2 ]`
`{target := `t_0` | array := `a_0` | low := 0 | high := `a_0`.length-1 |`
` mid := (`a_0`.length-1)/2}`

```
return mid;
while ( low <= high ) {
  ...
}
return -1;
```

Symbolic Execution by Example

```
a0!=null && a0.length > 0 && t0==a0[ (a0.length-1)/2 ]
{target := t0 | array := a0 | low := 0 | high := a0.length-1 |
 mid := (a0.length-1)/2}

return (a0.length-1)/2;
```

Symbolic Execution by Example

```
a0!=null && a0.length > 0 && t0==a0[ (a0.length-1)/2 ]
{target := t0 | array := a0 | low := 0 | high := a0.length-1 |
 mid := (a0.length-1)/2}

return (a0.length-1)/2;
```

One conclusion from this symbolic execution example:

All executions following path condition:

`array!=null && array.length>0 && target==array[(array.length-1)/2]`

return the result

`(array.length-1)/2`

Properties of Symbolic Execution

Important Conclusions

- One symbolic execution path corresponds to ∞ many test runs
- Only one symbolic execution path shown in example — need to explore all others as well!
- Programs with loops or recursion usually have ∞ many symbolic execution paths

Main Properties of Symbolic Execution

1. Even symbolic execution cannot cover all execution paths
2. But symbolic execution covers all execution paths up to finite depth

Quiz: Symbolic Execution

Symbolically execute the following program. Assume the initial values are x_0 and y_0 for x andy.

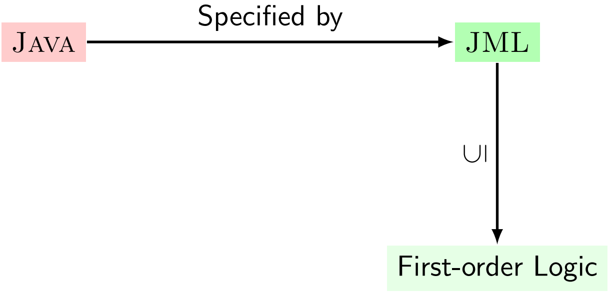
- What is the possible final states (with path conditions)?

```
if (x > y) {
   y = y + 1;
}
else {
  x = x + 1;
}
```

A: `x > y` $\Rightarrow$ {`x:=` x_0 `|` `y:=` y_0 `|` `y:= y + 1`}
B: `!(x > y)` $\Rightarrow$ {`x:=` x_0 `|` `y:=` y_0 `|` `x:= x + 1`}

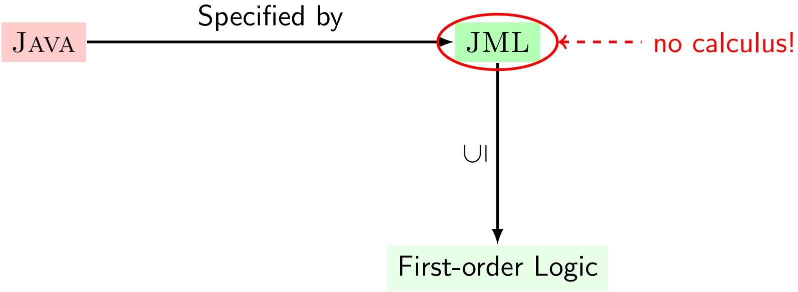
- Are the following test-cases covered by any of the symbolic execution paths leading to the final states? Which one?
  - {`x = -1, y = 1`} (B) {`x = 1, y = -1`} (A)
  - {`x = 0, y = 0`} (B) {`x = 43, y = 6`} (A)

Formal Software Verification: Our Setup



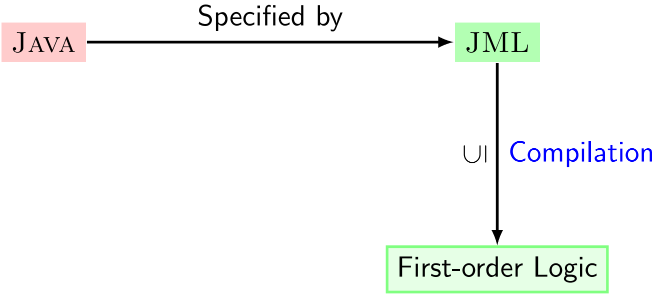
As introduced in lectures on Formal Specification

Formal Software Verification: Our Setup



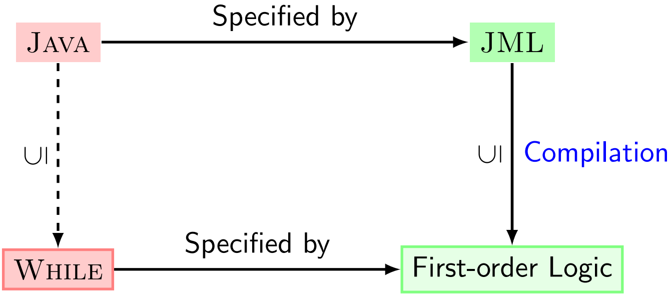
JML too complex and too unstable to provide calculus directly

Formal Software Verification: Our Setup



Automatic, but (as usual in compilation) result not very readable

Formal Software Verification: Our Setup



We specify a subset of Java in first-order logic

Java and JML in their full glory treated in SEuFM TDA292

While Language is Simpler than Java

- JAVA-like syntax (see handout for full grammar)
- Program variables refered to as locations.
- Two types: **int** and **boolean**
  - **int** type is the mathematical integers, not 32 bits
- Arrays `a`, `b`
  - Different name, different array. E.g. `a` and `b` not considered equal.
  - Array lookups `a[i]` as locations.
- While-loops, if-statements
- Boolean operators (`&`, `|`, `==`, `!`)
- Integer operators (`*`, `/`, `%`, `+`, `-`)
- Comparison operators (`<`, `>`, `<=`, `>=`)

While Language is Simpler than Java

- No objects, no reference types, no aliasing
- No method calls, no exceptions
- Expressions have no side effects

While Language: Example

Integer Division

For two non-negative integers `x` and `y` compute the quotient `q = x / y` and the remainder `r = x % y`

```
int x,y,q,r;

q = 0;
r = x;
while (r >= y) {
 r = r - y;
 q = q + 1;
}
```

How to specify the functionality of this program?

Signature: Different Kinds of Symbols

Built-in functions and predicates

Examples `TRUE`, `+`, `==`, `42`, ...

Appear in programs and specs

Meaning can change? No. Fixed once and everywhere (=rigid)

Program locations

Examples `a[i]`, **boolean** `b;`, ...

Appear in programs and specs

Meaning can change? Yes, from state to state (=non-rigid)

User-defined functions and predicates

Examples **int** *mySpecialHelperFunction*(**int**, **int**), ...

Appear in specs only

Meaning can change? Yes, from one program to another but same in all states of one. (=rigid)

First-Order Logic Signature

First-Order Signature for a While program

Types the types of WHILE: **int**, **boolean**

Functions

- built-in **int** operators/constants of WHILE:
  IntOp | $z \in \mathbb{Z}$
- program locations: **int** variables, array lookups
- user-defined functions with result type **int**
  - Example: **int** *c* ;
  - Example: **int** *f* (**int** , **boolean**) ;

Predicates

- built-in Boolean operators/constants of WHILE:
  BooleanOp | *ComparisonOp* | `TRUE` | `FALSE`
- program locations: **boolean** variables
- user-defined functions with result type **boolean**
  - Example: **boolean** *myFlag* ;
  - Example: **boolean** *notEq*(**int** , **int**) ;

Models and Program States

For a given program π

Definition (Program State)

A state s of program π is a first-order model giving meaning to symbols of the signature for π.

Definition (Kripke Structure)

A Kripke Structure $\mathcal{K}$ of program π is the set of all states with a given meaning of rigid symbols.

We write $l^s = v$ when location l has value v in s; $l^s \in \mathbb{B} \cup \mathbb{Z}$
We write $c^{\mathcal{K}} = v$ when a rigid constant c has value v in $\mathcal{K}$; $c^{\mathcal{K}} \in \mathbb{B} \cup \mathbb{Z}$

Specification: First-Order Logic Formula Syntax

First-Order Logic Formula Syntax

First-order terms and formulas over signature WHILE

Use JML/WHILE-like syntax

BooleanOp ::= `&` | `|` | `!` | `->` | `<->`

FOLFormula ::= *Atom* |
FOLFormula BooleanOp FOLFormula |
Quantifier Type LogicalVariable; *FOLFormula*

Quantifier ::= **`\forall`** | **`\exists`**

Example

```
\forall int i; i >= 0
```

First-Order Formulas and Program States

First-order formulas define sets of program states

Given a Kripke structure $\mathcal{K}$ and a formula F with program locations.

Then F is true in some states and not true in others.

Example

- `(i>j & j>=0)` is true in exactly those states s where $\mathtt{i}^s > \mathtt{j}^s$ and $\mathtt{j}^s$ is non-negative
- **`\exists int`** `i; l = i`

  is true in any state s, because the value of `i` can be chosen to be $\mathtt{l}^s$

Hoare Logic

Hoare logic is a language for reasoning about imperative programs

- Programs are WHILE programs
- Contracts are pre-/postcondition (simplified requires/ensures) pairs
  - expressed in first-order logic
  - used to define intended sets of program states (see previous slide)
  - no `\assignable`, no `\old`, no exceptional cases

Hoare Logic: Historical Remarks

- First suggested in 1969 by C. A. R. Hoare
- Numerous extensions (arrays, procedures, concurrency, objects)
- Our formulation from paper by R. Bubel & R. Hähnle
  A Hoare-Style Calculus with Explicit State Updates, FORMED'08

Hoare Logic, Cont'd

Definition (Hoare Triple)

A Hoare triple is an expression of the form

$$\{P\}\,\pi\,\{Q\}$$

where P and Q are first-order formulas over locations appearing in the WHILE program π.

(P, Q) is a "first-order logic contract" for π

Hoare Logic, Truth, and Validity

Definition (Truth in a state)

A Hoare triple $\{P\}\,\pi\,\{Q\}$ is true in state s of some Kripke structure $\mathcal{K}$, exactly when:
If P is true in s, and π terminates when started in s, then Q is true in the final state reached by π.

Definition (Truth in a Kripke structure)

A Hoare triple $\{P\}\,\pi\,\{Q\}$ is true in a Kripke structure $\mathcal{K}$, if it is true in all states of $\mathcal{K}$.

Definition (Validity of a Hoare triple)

A Hoare triple $\{P\}\,\pi\,\{Q\}$ is valid, if it is true in all Kripke structures.

Specifications with "Old" Values

Example (Swapping the value of two integer variables)

```
int x,y,d;

d = x;
x = y;
y = d;
```

What should a suitable contract express?

Specifications with “Old” Values

Example (Swapping the value of two integer variables)

```
int x,y,d;

d = x;
x = y;
y = d;
```

precondition `x` and `y` have certain initial values x_0 and y_0

postcondition `x` has now value y_0 and `y` has value x_0

Specifications with "Old" Values

Example (Swapping the value of two integer variables)

```
int x,y,d;

d = x;
x = y;
y = d;
```

precondition {x=x_0 & y=y_0}

postcondition {x=y_0 & y=x_0}

Remarks

- Precondition is true exactly in those $\mathcal{K}$ and states $s \in \mathcal{K}$ where $\mathtt{x}^s = x_0^{\mathcal{K}}$, $\mathtt{y}^s = y_0^{\mathcal{K}}$, and $\mathtt{d}^s$ is arbitrary: we don't care!
- x_0 and y_0 are rigid constants, cannot be changed by π

Contracts: the Good, the Bad, and the Ugly

Example (`swap`)

```
int x,y,d;

d = x;
x = y;
y = d;
```

precondition P : {x=x_0 & y=y_0}

postcondition Q : {x=y_0 & y=x_0}

$\{P\}$ `swap` $\{Q\}$ is a valid Hoare triple with suitable contract

Contracts: the Good, the Bad, and the Ugly

Example (`swap`)

```
int x,y,d;

d = x;
x = y;
y = d;
```

precondition P : {x=17 & y=42}

postcondition Q : {x=42 & y=17}

$\{P\}$ swap $\{Q\}$ is a valid Hoare triple, but contract can be more general

Contracts: the Good, the Bad, and the Ugly

Example (swap)

```
int x,y,d;

d = x;
x = y;
y = d;
```

precondition P : $\{$x=x_0 & y=y_0 & d=17$\}$

postcondition Q : $\{$x=y_0 & y=$x_0\}$

$\{P\}$ swap $\{Q\}$ is a valid Hoare triple, but precondition stronger than needed

Contracts: the Good, the Bad, and the Ugly

Example (`swap`)

```
int x,y,d;

d = x;
x = y;
y = d;
```

precondition P : {x=x_0 & y=y_0}

postcondition Q : {x=y_0}

$\{P\}$ `swap` $\{Q\}$ is a valid Hoare triple, but postcondition weaker than possible

Contracts: the Good, the Bad, and the Ugly

Example (swap)

```
int x,y,d;

d = x;
x = y;
y = d;
```

precondition P : $\{$x=x_0 & y=$y_0\}$

postcondition Q : $\{$x=x_0 & y=$x_0\}$

$\{P\}$ swap $\{Q\}$ is not a valid Hoare triple:
counter example $x_0^{\mathcal{K}} = 1$, $y_0^{\mathcal{K}} = 2$

Proving Validity of Hoare Triples

To prove Hoare triples we need a calculus.

A calculus is a set of (schematic) rules.

The rules of our calculus will perform symbolic execution of programs

Elements of Symbolic Execution

Components of a State during Symbolic Execution

Path condition — when is this execution path taken?

Program counter — next executable source code statement

Symbolic program state — the values currently assigned to variables

Symbolic Execution in Hoare Logic

Path condition	Can be part of precondition of Hoare triple
Program counter	Remaining program inside Hoare triple
Symbolic state	???

To mimick symbolic execution in Hoare Logic we must add a representation of intermediate symbolic program states

Symbolic Program State Updates

Definition (Atomic Update)

If `l` is a program location, t a FOL term of same type as `l`,
then $\texttt{l} := t$ is an atomic update.

Updates help specify program state(s) in which a formula is evaluated

Definition (Semantics of Atomic Updates)

Updates describe state changes:
Given a program state s, the result of applying an update $\mathcal{U} = (\texttt{l} := t)$ to s is the state $\mathcal{U}^s$, such that:

$$\texttt{loc}^{\mathcal{U}^s} = \begin{cases} t^s & \text{if } \texttt{loc} = \texttt{l} \\ \texttt{loc}^s & \text{otherwise} \end{cases}$$

Symbolic Program State Updates Cont'd

Definition (Complex Update)

If $\mathcal{U}$ and $\mathcal{V}$ are updates, then $\mathcal{U}, \mathcal{V}$ is a sequential update and $\mathcal{U}||\mathcal{V}$ is a parallel update.

Definition (Semantics of Complex Updates)

Sequential Update For $\mathcal{U}, \mathcal{V}$, first apply $\mathcal{U}$ to s to obtain state $\mathcal{U}^s$, then apply $\mathcal{V}$ to $\mathcal{U}^s$.

Parallel Update For $\mathcal{U} = (\mathtt{l}_1 := t_1 || \cdots || \mathtt{l}_m := t_m)$ in parallel set the value of each $\mathtt{l}_i$ to the value of t_i in s.

Conflict: if $\mathtt{l}_j = \mathtt{l}_k$, then $t_{\max(j,k)}$ "wins"

Example

The effect of $(\mathtt{l} := c || \mathtt{l} := 17)$ is identical to that of $\mathtt{l} := 17$.

Hoare Triples with State Updates

Definition (Hoare Triple with Update)

A Hoare Triple with Update is an expression of the form

$$\{P\}\,[\mathcal{U}]\,\pi\,\{Q\}$$

where P and Q are first-order formulas over locations appearing in the WHILE program π and $\mathcal{U}$ is an update.

Definition (Truth of a Hoare Triple with Update in a State)

A Hoare triple $\{P\}\,[\mathcal{U}]\,\pi\,\{Q\}$ is true in state s of some Kripke structure $\mathcal{K}$, exactly when:
If P is true in s, and π terminates when started in $\mathcal{U}^s$, then Q is true in the final state reached by π.

Truth in $\mathcal{K}$ and validity defined just as for triples without updates.

A Calculus for Hoare Logic

General Form of Rule

$$\frac{\{P_1\}\,[\mathcal{U}_1]\,\rho\,\{Q\} \qquad \cdots \qquad \{P_n\}\,[\mathcal{U}_n]\,\rho\,\{Q\}}{\{P\}\,[\mathcal{U}]\,\pi\,\rho\,\{Q\}}$$

- Symbolically execute first statement π of program in conclusion
- May be necessary to branch (unknown guard in conditional)
  - This effect we know already from symbolic execution
- Update $\mathcal{U}$ is symbolic state computed so far
- New path conditions $P_1, \ldots, P_n$ and updates $\mathcal{U}_1, \ldots, \mathcal{U}_n$ in premisses
- Rules applied until program completely executed
  - Resulting premisses are FOL formulas $\Rightarrow$ automated theorem prover

Rules of Calculus for Hoare Logic

Assignment

$$\text{assignment}\ \frac{\{P\}\,[\mathcal{U},\ \mathtt{x} := \mathsf{e}]\,\pi\,\{Q\}}{\{P\}\,[\mathcal{U}]\,\mathtt{x\ =\ e;}\pi\,\{Q\}}$$

- Turn assignment into update and append sequentially
  - Important that e has no side effects
  - e can be evaluated as FOL term
- Schematic rule: match against concrete update, program, etc.

Example

$$\frac{\{P\}\,[\mathtt{x} := x_0,\ \mathtt{x} := \mathtt{x\ +\ 17}]\,\pi\,\{Q\}}{\{P\}\,[\mathtt{x} := x_0]\,\mathtt{x\ =\ x\ +\ 17;}\pi\,\{Q\}}$$

Rules of Calculus for Hoare Logic

Assignment

$$\text{assignment}\ \frac{\{P\}\,[\mathcal{U}, \mathtt{x} := \mathtt{e}]\,\pi\,\{Q\}}{\{P\}\,[\mathcal{U}]\,\mathtt{x}\ \mathtt{=}\ \mathtt{e;}\pi\,\{Q\}}$$

- Turn assignment into update and append sequentially
  - Important that e has no side effects
  - e can be evaluated as FOL term
- Schematic rule: match against concrete update, program, etc.
- Turn sequential into parallel update

Example

$$\frac{\{P\}\,[\mathtt{x} := x_0 \,||\, \mathtt{x} := x_0\ \mathtt{+}\ \mathtt{17}]\,\pi\,\{Q\}}{\{P\}\,[\mathtt{x} := x_0]\,\mathtt{x}\ \mathtt{=}\ \mathtt{x}\ \mathtt{+}\ \mathtt{17;}\pi\,\{Q\}}$$

Rules of Calculus for Hoare Logic

Assignment

$$\text{assignment}\ \frac{\{P\}\,[\mathcal{U},\ \mathtt{x} := \mathsf{e}]\,\pi\,\{Q\}}{\{P\}\,[\mathcal{U}]\,\mathtt{x\ =\ e;}\pi\,\{Q\}}$$

- Turn assignment into update and append sequentially
  - Important that e has no side effects
  - e can be evaluated as FOL term
- Schematic rule: match against concrete update, program, etc.
- Turn sequential into parallel update, then simplify

Example

$$\frac{\{P\}\,[\mathtt{x} := x_0\ \mathtt{+}\ 17]\,\pi\,\{Q\}}{\{P\}\,[\mathtt{x} := x_0]\,\mathtt{x\ =\ x\ +\ 17;}\pi\,\{Q\}}$$

Rules of Calculus for Hoare Logic, Cont'd

Exit

$$\text{exit}\ \frac{\vdash\ P \mathrel{-\!\!>} \mathcal{U}(Q)}{\{P\}\,[\mathcal{U}]\ \{Q\}}$$

- Applied when original program is fully symbolically executed
- "Precondition implies postcondition in final state of the original program, which is now summarized by $\mathcal{U}$"
- The meaning of $\mathcal{U}(Q)$ is to apply $\mathcal{U}$ to Q:
  - If $\mathtt{x} := t$ is atomic update in $\mathcal{U}$ then replace each occurrence of $\mathtt{x}$ in Q with t
  - Assume that $\mathcal{U}$ is a parallel update
- Premiss is FOL formula, handed over to automated theorem prover $\vdash$

Example Proof of Validity of Hoare Triple

Rules assignment and exit suffice to do earlier example:

Example (`swap`)

{x = x_0 & y = y_0}
[]
d = x; x = y; y = d;
{x = y_0 & y = x_0}

Start with empty update []

Rules Used

$$\text{assignment}\ \frac{\{P\}\,[\mathcal{U},\ \texttt{x} := \texttt{e}]\,\pi\,\{Q\}}{\{P\}\,[\mathcal{U}]\,\texttt{x = e};\pi\,\{Q\}} \qquad \text{exit}\ \frac{\vdash\ P \texttt{->} \mathcal{U}(Q)}{\{P\}\,[\mathcal{U}]\ \{Q\}}$$

Example Proof of Validity of Hoare Triple

Rules assignment and exit suffice to do earlier example:

Example (swap)

{x = x_0 & y = y_0}
[d := x]
x = y; y = d;
{x = y_0 & y = x_0}

Rules Used

$$\text{assignment}\ \frac{\{P\}\,[\mathcal{U},\, \mathtt{x} := \mathtt{e}]\,\pi\,\{Q\}}{\{P\}\,[\mathcal{U}]\,\mathtt{x\ =\ e;}\pi\,\{Q\}} \qquad \text{exit}\ \frac{\vdash\ P \mathbin{{-}{>}} \mathcal{U}(Q)}{\{P\}\,[\mathcal{U}]\ \{Q\}}$$

Example Proof of Validity of Hoare Triple

Rules assignment and exit suffice to do earlier example:

Example (swap)

$\{$x = x_0 & y = $y_0\}$
[d := x || x := y]
y = d;
$\{$x = y_0 & y = $x_0\}$

Rules Used

$$\text{assignment}\ \frac{\{P\}\,[\mathcal{U},\ \mathtt{x} := \mathtt{e}]\,\pi\,\{Q\}}{\{P\}\,[\mathcal{U}]\,\mathtt{x\ =\ e;}\pi\,\{Q\}} \qquad \text{exit}\ \frac{\vdash\ P \mathbin{{-}{>}} \mathcal{U}(Q)}{\{P\}\,[\mathcal{U}]\ \{Q\}}$$

Example Proof of Validity of Hoare Triple

Rules assignment and exit suffice to do earlier example:

Example (`swap`)

{x = x_0 & y = y_0}
[(d := x || x := y), y := d]

{x = y_0 & y = x_0}
Empty program

Rules Used

assignment $\dfrac{\{P\}\,[\mathcal{U},\,\mathtt{x} := \mathtt{e}]\,\pi\,\{Q\}}{\{P\}\,[\mathcal{U}]\,\mathtt{x\ =\ e;}\pi\,\{Q\}}$ exit $\dfrac{\vdash\ P \mathrel{-\!\!>} \mathcal{U}(Q)}{\{P\}\,[\mathcal{U}]\ \{Q\}}$

Example Proof of Validity of Hoare Triple

Rules assignment and exit suffice to do earlier example:

Example (`swap`)

{x = x_0 & y = y_0}
[d := x || x := y || y := x]

{x = y_0 & y = x_0}

Parallel update: use previous value of `d`!

Rules Used

$$\text{assignment}\ \frac{\{P\}\,[\mathcal{U},\, \mathtt{x} := \mathtt{e}]\,\pi\,\{Q\}}{\{P\}\,[\mathcal{U}]\,\mathtt{x\ =\ e;}\pi\,\{Q\}} \qquad \text{exit}\ \frac{\vdash\ P \mathbin{{-}{>}} \mathcal{U}(Q)}{\{P\}\,[\mathcal{U}]\ \{Q\}}$$

Example Proof of Validity of Hoare Triple

Rules assignment and exit suffice to do earlier example:

Example (swap)

$\vdash$ `(x =` x_0 `& y =` y_0`) ->`
`[d := x || x := y || y := x](x =` y_0 `& y =` x_0`)`

Exit

Rules Used

$$\text{assignment}\ \frac{\{P\}\,[\mathcal{U},\ \mathtt{x} := \mathtt{e}]\,\pi\,\{Q\}}{\{P\}\,[\mathcal{U}]\,\mathtt{x\ =\ e;}\pi\,\{Q\}} \qquad \text{exit}\ \frac{\vdash\ P \mathrel{-\!\!>} \mathcal{U}(Q)}{\{P\}\,[\mathcal{U}]\ \{Q\}}$$

Example Proof of Validity of Hoare Triple

Rules assignment and exit suffice to do earlier example:

Example (`swap`)

$\vdash$ `(x =` x_0 `& y =` y_0`) -> (y =` y_0 `& x =` x_0`)`

Apply update to postcondition — valid FOL formula!

Rules Used

$$\text{assignment}\ \frac{\{P\}\,[\mathcal{U},\, \mathtt{x} := \mathtt{e}]\,\pi\,\{Q\}}{\{P\}\,[\mathcal{U}]\,\mathtt{x\ =\ e;}\pi\,\{Q\}} \qquad \text{exit}\ \frac{\vdash\ P \mathbin{->} \mathcal{U}(Q)}{\{P\}\,[\mathcal{U}]\ \{Q\}}$$

Summary

- Testing cannot replace verification
- Formal verification can prove properties for all runs,
  ... but has inherent limitations, too
- We verify WHILE programs specified with first-order logic
- Hoare logic is tailored for verification in this setup
- To avoid exhaustive simulation one uses symbolic reasoning
- Symbolic execution useful paradigm for formal verification
- Use updates to represent intermediate symbolic states
- Calculus for symbolic execution of WHILE programs
  Rule application driven by next executable statement