Testing, Debugging, Program Verification

Formal Verification, Part II

Wolfgang Ahrendt & Vladimir Klebanov & Moa Johansson

3 December 2012

Recap: Symbolic Execution

- Symbolic initial values.
  E.g. set variable `x := ` x_0 for new symbol x_0.
- Arbitrary but fixed values.
- Represents a set of possible concrete values.
- Execution Tree with branches annotated by Path Conditions
- Symbolic run says something about a set of concrete runs.
  E.g. all runs where array `a` is non-null and has length >0.

Recap: State, State Update

- The symbolic execution state, denoted $\mathcal{U}$, records values of variables.
- Updated as execution progresses.

Let $\mathcal{U}$ and $\mathcal{V}$ be updates, with $\mathcal{U} = (\texttt{x:=}\ x_0)$, $\mathcal{V} = (\texttt{x:=1})$.

- Sequential update: $\mathcal{U}$, $\mathcal{V}$
  - First apply $\mathcal{U}$ to state s, obtaining state $\mathcal{U}^s$.
  - Then apply $\mathcal{V}$ to $\mathcal{U}^s$.
  - What is the final value of `x`?
- Parallel update: $\mathcal{U} \parallel \mathcal{V} = (\texttt{x:=}\ x_0 \parallel \texttt{x:=1})$
  - If same location updated, rightmost update wins.
  - Apply $\mathcal{U} \parallel \mathcal{V}$ to an intial state s.
  - What is the value of `x` afterwards?

We can turn sequential updates into parallel ones.

Recap: Hoare Triples (with 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 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.

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

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

- 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

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

Rules of Calculus for Hoare Logic

Assignment

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

- 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

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

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

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

Example Proof of Validity of Hoare Triple

Rules assignment and exit suffice to do earlier example:

Example (`swap`)

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

Rules Used

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

Example Proof of Validity of Hoare Triple

Rules assignment and exit suffice to do earlier example:

Example (`swap`)

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

Rules Used

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

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 := d]

{x = y_0 & y = x_0}
Empty program

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`)

$\{$`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

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

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) ->
    [d := x || x := y || y := x](x = y_0 & y = x_0)

Exit

Rules Used

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

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

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

KeY-Hoare

What is KeY-Hoare?

- Interactive software verification system for WHILE programs
- Uses Hoare Calculus with Updates
- Derived from KeY system for (almost) full JAVA and JML
- Symbolic execution rules must be applied "by hand"
- Display, navigate, and pretty-print formulas, proof trees
- Validity of FOL formulas, update simplification/application: automatic!
- System takes care of correctness: can't prove invalid Hoare triple!

KeY-Hoare: Input File Syntax

```
\functions {
  FirstOrderFunctionDeclaration*
    // initial values, user-defined functions
}
\programVariables {
  ProgramLocationDeclaration*
    // all locations appearing in Update, Program below
}
\hoare {
  { Precondition }
  [ Update ]
  \[{ // funny brackets needed for KeY-compatibility
      Program
  }\]
  { Postcondition }
}
```

KeY-Hoare: Demo

```
\functions {
  int x0; // one per line
  int y0;
}

\programVariables {
  int x, y, d;
}

\hoare {
  { x = x0 & y = y0 } // can skip empty initial update
  \[{
      d = x; x = y; y = d;
  }\]
  { x = y0 & y = x0 }
}
```

Rules of Calculus for Hoare Logic, Cont'd

Conditional

$$\frac{\{P \,\&\, \mathcal{U}(\texttt{b=TRUE})\}\,[\mathcal{U}]\,\pi_1\,\rho\,\{Q\} \qquad \{P \,\&\, \mathcal{U}(\texttt{b=FALSE})\}\,[\mathcal{U}]\,\pi_2\,\rho\,\{Q\}}{\{P\}\,[\mathcal{U}]\,\textbf{if}\,\texttt{(b)}\,\{\pi_1\}\textbf{else}\{\pi_2\}\,\rho\,\{Q\}}$$

- Case distinction necessary, because value of `b` symbolic
  - In general, Hoare calculus proofs are trees
- Important that `b` has no side effects
  - Can treat `b` as FOL Boolean term
- In premisses `b` must be evaluated in state $\mathcal{U}$
- $\mathcal{U}$(`b=TRUE`) and $\mathcal{U}$(`b=FALSE`) extend existing path condition P
  - Can simplify `b=TRUE` to `b` and `b=FALSE` to `!b`

Example Proof with Conditional

Example (`max`)

```
{true}
[]
if (x > y) { res = x; } else { res = y; }
{???}
```

Postcondition?

Example Proof with Conditional

Example (max)

```
{true}
[]
if (x > y) { res = x; } else { res = y; }
{(res = x | res = y) & res >= x & res >= y}
```

Next Rule Used

$$\frac{\{P \;\&\; \mathcal{U}(\texttt{b=TRUE})\}\,[\mathcal{U}]\,\pi_1\,\rho\,\{Q\} \qquad \{P \;\&\; \mathcal{U}(\texttt{b=FALSE})\}\,[\mathcal{U}]\,\pi_2\,\rho\,\{Q\}}{\{P\}\,[\mathcal{U}]\;\textbf{if}\,(\texttt{b})\,\{\pi_1\}\text{else}\{\pi_2\}\rho\,\{Q\}}$$

Example Proof with Conditional

Example (max)

```
{x > y}
[]
res = x;
{(res = x | res = y) & res >= x & res >= y}
```

Left premiss

Next 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 with Conditional

Example (max)

```
{x > y}
[res := x]

{(res = x | res = y) & res >= x & res >= y}
```

Next Rules Used

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

Example Proof with Conditional

Example (max)

```
x > y ->
  [res := x]((res = x | res = y) & res >= x & res >= y)
```

Example Proof with Conditional

Example (`max`)

```
x > y ->
  ((x = x | x = y) & x >= x & x >= y)
```

Example Proof with Conditional

Example (max)

```
x > y ->
  (true & true & x >= y)
```

Valid FOL formula!

Example Proof with Conditional

Example (`max`)

```
{!(x > y)}
[]
res = y;
{(res = x | res = y) & res >= x & res >= y}
```

Right premiss, similar as before

Example Proof with Conditional

Demo: `max.key`

Verifying Programs with Loops

Difficulties of While Loops

- Need to “unwind” loop body one by one
- In general, no fixed loop bound known (depends on input)
- New mathematical principle needed! Can you guess which?

A Program With Loops

Example (Loop with fixed bound)

```
{true}
[]
i = 0;
n = 2;
while (i < n) {
  i = i + 1;
}
{i = n}
```

A Program With Loops

Example (Loop with fixed bound)

```
{true}
[i := 0 || n := 2]
while (i < n) {
  i = i + 1;
}
{i = n}
```

A Program With Loops

Example (Loop with fixed bound)

```
{true}
[i := 0 || n := 2]
while (i < n) {
  i = i + 1;
}
{i = n}
```

Possible Rule: Unwind

$$\frac{\{P\}\,[\mathcal{U}]\,\textbf{if}\ \ \texttt{(b)}\ \{\pi\,\textbf{while}\ \ \texttt{(b)}\ \{\pi\}\}\,\rho\,\{Q\}}{\{P\}\,[\mathcal{U}]\,\textbf{while}\ \ \texttt{(b)}\ \{\pi\}\,\rho\,\{Q\}}$$

A Program With Loops

Example (Loop with fixed bound)

```
{true}
[i := 0 || n := 2]
if (i < n) {
  i = i + 1;
  while (i < n) {
    i = i + 1;
}}
{i = n}
```

Possible Rule: Unwind

$$\frac{\{P\}\,[\mathcal{U}]\,\mathbf{if}\ \ (\mathtt{b})\ \{\pi\,\mathbf{while}\ \ (\mathtt{b})\ \{\pi\}\}\,\rho\,\{Q\}}{\{P\}\,[\mathcal{U}]\,\mathbf{while}\ \ (\mathtt{b})\ \{\pi\}\,\rho\,\{Q\}}$$

A Program With Loops

Example (Loop with fixed bound)

```
{true}
[i := 0 || n := 2]
if (i < n) {
  i = i + 1;
  while (i < n) {
    i = i + 1;
}}
{i = n}
```

Symbolic execution of conditional and loop body
(slightly simplified, use that values of `i` and `n` are known)

A Program With Loops

Example (Loop with fixed bound)

```
{true}
[i := 0 || n := 2]
i = i + 1;
while (i < n) {
  i = i + 1;
}
{i = n}
```

Symbolic execution of conditional and loop body
(slightly simplified, use that values of `i` and `n` are known)

A Program With Loops

Example (Loop with fixed bound)

```
{true}
[i := 1 || n := 2]
while (i < n) {
  i = i + 1;
}
{i = n}
```

Symbolic execution of conditional and loop body
(slightly simplified, use that values of `i` and `n` are known)

A Program With Loops

Example (Loop with fixed bound)

```
{true}
[i := 1 || n := 2]
if (i < n) {
  i = i + 1;
  while (i < n) {
    i = i + 1;
}}
{i = n}
```

Unwind second time

$$\frac{\{P\}\,[\mathcal{U}]\,\mathbf{if}\ \ (\mathtt{b})\ \{\pi\,\mathbf{while}\ \ (\mathtt{b})\ \{\pi\}\}\,\rho\,\{Q\}}{\{P\}\,[\mathcal{U}]\,\mathbf{while}\ \ (\mathtt{b})\ \{\pi\}\,\rho\,\{Q\}}$$

A Program With Loops

Example (Loop with fixed bound)

```
{true}
[i := 1 || n := 2]
if (i < n) {
  i = i + 1;
  while (i < n) {
    i = i + 1;
}}
{i = n}
```

Symbolic execution of conditional and loop body
(slightly simplified, use that values of `i` and `n` are known)

A Program With Loops

Example (Loop with fixed bound)

```
{true}
[i := 1 || n := 2]
i = i + 1;
while (i < n) {
  i = i + 1;
}
{i = n}
```

Symbolic execution of conditional and loop body
(slightly simplified, use that values of `i` and `n` are known)

A Program With Loops

Example (Loop with fixed bound)

```
{true}
[i := 2 || n := 2]
while (i < n) {
  i = i + 1;
}
{i = n}
```

Symbolic execution of conditional and loop body
(slightly simplified, use that values of `i` and `n` are known)

A Program With Loops

Example (Loop with fixed bound)

```
{true}
[i := 2 || n := 2]
if (i < n) {
  i = i + 1;
  while (i < n) {
    i = i + 1;
}}
{i = n}
```

Unwind third time

$$\frac{\{P\}\,[\mathcal{U}]\,\textbf{if}\ (\texttt{b})\ \{\pi\,\textbf{while}\ (\texttt{b})\ \{\pi\}\}\,\rho\,\{Q\}}{\{P\}\,[\mathcal{U}]\,\textbf{while}\ (\texttt{b})\ \{\pi\}\,\rho\,\{Q\}}$$

A Program With Loops

Example (Loop with fixed bound)

```
{true}
[i := 2 || n := 2]

{i = n}
```

Guard of conditional is **false**, else branch is empty

The Problem with Loops

How to handle a loop with...

- 0 iterations? Unwind $1\times$
- 10 iterations? Unwind $11\times$
- 10000 iterations? Unwind $10001\times$
  (and don't make any plans for the rest of the day)
- an unknown number of iterations?

We need an invariant rule (or some other form of induction)

Loop Invariants

Idea behind loop invariants

- A formula *Inv* whose validity is preserved by loop guard and body
- Consequence: if *Inv* was true at start state of the loop, then it still holds after arbitrarily many loop iterations
- If the loop terminates at all, then *Inv* holds afterwards
- Make sure to include the desired postcondition after loop into *Inv*

Invariant Rule

$$\text{loopInvariant}\ \frac{\begin{array}{ll} \vdash\ P \texttt{ -> } \mathcal{U}(\mathit{Inv}) & \text{(initially valid)} \\ \{\mathit{Inv}\ \&\ \ \texttt{b}\}\,[]\,\pi\,\{\mathit{Inv}\} & \text{(preserved)} \\ \{\mathit{Inv}\ \&\ \texttt{!b}\}\,[]\,\rho\,\{Q\} & \text{(use case)} \end{array}}{\{P\}\,[\mathcal{U}]\,\mathbf{while}\ \texttt{(b)}\ \{\pi\}\,\rho\,\{Q\}}$$

Loop Invariant: Example

Example (Unbounded Loop)

```
{n >= 0}
[]
i = 0;
while (i < n) {
  i = i + 1;
}
{i = n}
```

Loop Invariant: Example

Example (Unbounded Loop)

```
{n >= 0}
[i := 0]
while (i < n) {
  i = i + 1;
}
{i = n}
```

Invariant Rule

$$\text{loopInvariant}\ \frac{\begin{array}{ll}\vdash\ P \mathrel{-\!\!>} \mathcal{U}(\mathit{Inv}) & \text{(initially valid)}\\ \{\mathit{Inv}\ \&\ \mathtt{b}\}\,[]\,\pi\,\{\mathit{Inv}\} & \text{(preserved)}\\ \{\mathit{Inv}\ \&\ \mathtt{!b}\}\,[]\,\rho\,\{Q\} & \text{(use case)}\end{array}}{\{P\}\,[\mathcal{U}]\,\mathbf{while}\ \mathtt{(b)}\ \{\pi\}\,\rho\,\{Q\}}$$

Loop Invariant: Example

Example (Unbounded Loop)

```
{n >= 0}
[i := 0]
while (i < n) {
  i = i + 1;
}
{i = n}
```

What is a suitable invariant?

Invariant Rule

$$\text{loopInvariant}\ \frac{\begin{array}{ll} \vdash\ P \texttt{->} \mathcal{U}(\mathit{Inv}) & \text{(initially valid)} \\ \{\mathit{Inv}\ \&\ \texttt{b}\}\,[]\,\pi\,\{\mathit{Inv}\} & \text{(preserved)} \\ \{\mathit{Inv}\ \&\ \texttt{!b}\}\,[]\,\rho\,\{Q\} & \text{(use case)} \end{array}}{\{P\}\,[\mathcal{U}]\,\textbf{while}\ \texttt{(b)}\ \{\pi\}\,\rho\,\{Q\}}$$

Loop Invariant: Example

Example (Unbounded Loop)

`|- n >= 0 -> [i := 0](i <= n)`	`{i <= n & i < n} [] i = i + 1; {i <= n}`	`{i <= n & i >= n} [] {i = n}`

Try `i <= n`

Invariant Rule

$$\text{loopInvariant}\ \frac{\begin{array}{ll} \vdash\ P \texttt{ -> } \mathcal{U}(\mathit{Inv}) & \text{(initially valid)} \\ \{\mathit{Inv}\ \&\ \texttt{b}\}\,[]\,\pi\,\{\mathit{Inv}\} & \text{(preserved)} \\ \{\mathit{Inv}\ \&\ \texttt{!b}\}\,[]\,\rho\,\{Q\} & \text{(use case)} \end{array}}{\{P\}\,[\mathcal{U}]\,\mathbf{while}\ \texttt{(b)}\ \{\pi\}\,\rho\,\{Q\}}$$

Loop Invariant: Example

Example (Unbounded Loop)

`|- n >= 0 ->`
`[i := 0](i <= n)`
	`{i <= n & i < n}`
	`[]`
	`i = i + 1;`
	`{i <= n}`
		`{i <= n & i >= n}`
		`[]`
		
		`{i = n}`

Try `i <= n`

Note that precondition P and $\mathcal{U}$ are missing in preserves/use case! Why?

Invariant Rule

$$\text{loopInvariant}\ \frac{\begin{array}{ll} \vdash\ P \mathrel{-\!\!>} \mathcal{U}(\mathit{Inv}) & \text{(initially valid)} \\ \{\mathit{Inv}\ \&\ \mathtt{b}\}\,[]\,\pi\,\{\mathit{Inv}\} & \text{(preserved)} \\ \{\mathit{Inv}\ \&\ \mathtt{!b}\}\,[]\,\rho\,\{Q\} & \text{(use case)} \end{array}}{\{P\}\,[\mathcal{U}]\,\mathbf{while}\ \mathtt{(b)}\ \{\pi\}\,\rho\,\{Q\}}$$

How to Derive Loop Invariants without Magic?

Example (when Symbolic Execution at start of loop)

```
{n >= 0}
[i := 0]
while (i < n) { i = i + 1; }
{i = n}
```

Look at desired postcondition (`i = n`)

What, in addition to negated guard (`i >= n`), is needed? (`i <= n`)

Is (`i <= n`) established at beginning and preserved?

Yes! We have found a suitable loop invariant!

Obtaining Invariants by Strengthening

Example (Slightly changed spec—Demo `count.key`)

```
{n >= 0 & n = m}
[i := 0]
while (i < n) { i = i + 1; }
{i = m}
```

Look at desired postcondition (`i = m`)

What, in addition to negated guard (`i >= n`), is needed? (`i = m`)

Is (`i = m`) established at beginning and preserved? No!

(`i = m`) is neither preserved nor true at the start!
Can we use something from the precondition or $\mathcal{U}$?

- If we know that (`n = m`) then (`i <= n`) suffices
- Strengthen the invariant candidate to: (`i <= n & n = m`)

Why Does the Invariant Rule Work?

Induction Argument

We prove by induction over the number n of loop iterations that *Inv* holds in all loop iterations (used in third premiss)

Hypothesis *Inv* holds in the first n loop iterations

Base Case *Inv* holds in the first 0 loop iterations
iff *Inv* holds in the state at the start of the loop
iff the first premiss of the invariant rule holds

Step Case If *Inv* holds in the first n loop iterations, then *Inv* holds even in the first $n+1$ loop iterations
follows from: in any[a] state where *Inv* holds and the guard is true *Inv* holds after one more iteration
iff the second premiss of the invariant rule holds

---

[a] For this reason we cannot use P or $\mathcal{U}$ in (preserved) and (use case)

Generalization

Example (Addition)

```
{ x = x0 & y = y0 & y0 >= 0 }
[]
while (y > 0) {
  x = x + 1;
  y = y - 1;
}
{ x = x0 + y0 }
```

Finding the invariant

First attempt: use postcondition `x = x0 + y0`

- Not true at start whenever `y0 <> 0`
- Not preserved by loop, because `x` is increased

Generalization

Example (Addition)

```
{ x = x0 & y = y0 & y0 >= 0 }
[]
while (y > 0) {
  x = x + 1;
  y = y - 1;
}
{ x = x0 + y0 }
```

Finding the invariant

What stays invariant?

- The sum of `x` and `y`: `x + y = x0 + y0` "Generalization"
- Can help to think of partial result: "δ" between `x` and `x0 + y0`

Generalization

Example (Addition)

```
{ x = x0 & y = y0 & y0 >= 0 }
[]
while (y > 0) {
  x = x + 1;
  y = y - 1;
}
{ x = x0 + y0 }
```

Checking the invariant

Is `x + y = x0 + y0` a good invariant?

- Holds in the beginning and is preserved by loop
- But postcondition not achieved by `x + y = x0 + y0 & y <= 0`

Generalization

Example (Addition)

```
{ x = x0 & y = y0 & y0 >= 0 }
[]
while (y > 0) {
  x = x + 1;
  y = y - 1;
}
{ x = x0 + y0 }
```

Strenghtening the invariant

Postcondition holds if `y = 0`

- Sufficient to add `y >= 0` to `x + y = x0 + y0 & y <= 0`

Generalization

Example (Addition)

```
{ x = x0 & y = y0 & y0 >= 0 }
[]
while (y > 0) {
  x = x + 1;
  y = y - 1;
}
{ x = x0 + y0 }
```

Demo `addition3.key`

Example: Fibonacci

Example (Fibonacci)

```
{n = n0 & n > 0}

[]
x1 = 1;
x2 = 1;
while (n > 2) {
  x2 = x1 + x2;
  x1 = x2 - x1;
  n = n - 1;
}
{ x2 = ?? }
```

How do we specify the result?

Example: Fibonacci

Example (Fibonacci)

```
{n = n0 & n > 0 & fib(1) = 1 & fib(2) = 1 &
 \forall int m; (m > 2 -> fib(m) = fib(m-1) + fib(m-2))}
[]
x1 = 1;
x2 = 1;
while (n > 2) {
  x2 = x1 + x2;
  x1 = x2 - x1;
  n = n - 1;
}
{ x2 = fib(n0) }
```

Introduce `\function int fib(int);`

Example: Fibonacci

Example (Fibonacci)

```
{n = n0 & n > 0 & fib(1) = 1 & fib(2) = 1 &
 \forall int m; (m > 2 -> fib(m) = fib(m-1) + fib(m-2))}
[]
x1 = 1;
x2 = 1;
while (n > 2) {
  x2 = x1 + x2;
  x1 = x2 - x1;
  n = n - 1;
}
{ x2 = fib(n0) }
```

Loop invariant must express complex relation between loop and `fib()`!

Inductive Reasoning (Patterns)

Example (Fibonacci)

```
x1 = fib(1);
x2 = fib(2);
while (n > 2) {
  x2 = x1 + x2;
  x1 = x2 - x1;
  n = n - 1;
}
```

Simulate loop to discover pattern

#	x1	x2	n
0	fib(1)	fib(2)	n0
1	fib(2)	fib(3)	n0 - 1
2	fib(3)	fib(4)	n0 - 2

Partial result:
express argument of `fib()` as relation between `n` and `n0`

Conjecture: `x1 = fib(n0-n+1)`
`x2 = fib(n0-n+2)`

Invariants and Definitions

Example (Fibonacci)

```
{n = n0 & n > 0 & fib(1) = 1 & fib(2) = 1 &
 \forall int m; (m > 2 -> fib(m) = fib(m-1) + fib(m-2))}
[x1 := 1 || x2 := 1]
while (n > 2) {
  x2 = x1 + x2;
  x1 = x2 - x1;
  n = n - 1;
}
{ x2 = fib(n0) }
```

Definition of `fib()` not available in preserves case!

$$\frac{\vdash\ P \texttt{->} \mathcal{U}(Inv) \qquad \{Inv\ \&\ \texttt{b}\}\,[]\,\pi\,\{Inv\} \qquad \{Inv\ \&\ \texttt{!b}\}\,[]\,\rho\,\{Q\}}{\{P\}\,[\mathcal{U}]\,\textbf{while}\ \texttt{(b)}\ \{\pi\}\,\rho\,\{Q\}}$$

Invariants and Definitions

Example (Fibonacci)

```
{n = n0 & n > 0 & F}

[x1 := 1 || x2 := 1]
while (n > 2) {
  x2 = x1 + x2;
  x1 = x2 - x1;
  n = n - 1;
}
{ x2 = fib(n0) }
```

Add definition `F` of `fib()` to invariant

Inv = (`x1 = fib(n0-n+1) & x2 = fib(n0-n+2) & F`)

Invariants and Definitions

Example (Fibonacci)

```
{n = n0 & n > 0 & F}

[x1 := 1 || x2 := 1]
while (n > 2) {
  x2 = x1 + x2;
  x1 = x2 - x1;
  n = n - 1;
}
{ x2 = fib(n0) }
```

Does postcondition follow from *Inv* `& n <= 2` **?**

Inv = (`x1 = fib(n0-n+1) & x2 = fib(n0-n+2) & F`)

Invariants and Definitions

Example (Fibonacci)

```
{n = n0 & n > 0 & F}

[x1 := 1 || x2 := 1]
while (n > 2) {
  x2 = x1 + x2;
  x1 = x2 - x1;
  n = n - 1;
}
{ x2 = fib(n0) }
```

Does postcondition follow from *Inv* `& n <= 2` ?

Inv = (`x1 = fib(n0-n+1) & x2 = fib(n0-n+2) & F`)

Yes, provided that `n >= 2`! Add this to *Inv* — now use case ok!

Strengthening, Again

Example (Fibonacci, the "preserves" case)

```
{x1 = fib(n0-n+1) & x2 = fib(n0-n+2) & F & n >= 2 & n > 2}
[]
x2 = x1 + x2;
x1 = x2 - x1;
n = n - 1;
{x1 = fib(n0-n+1) & x2 = fib(n0-n+2) & F & n >= 2}
```

Perform symbolic execution, exit, and update simplification . . .
Five times `andRight`, all but 2nd case close
After several `andLeft`, equations become applicable

Demo: `fib.key`

Strengthening, Again

Example (Fibonacci, the "preserves" case)

```
{x1 = fib(n0-n+1) & x2 = fib(n0-n+2) & F & n >= 2 & n > 2}
[]
x2 = x1 + x2;
x1 = x2 - x1;
n = n - 1;
{x1 = fib(n0-n+1) & x2 = fib(n0-n+2) & F & n >= 2}
```

Perform symbolic execution, exit, and update simplification . . .
Five times `andRight`, all but 2nd case close
After several `andLeft`, equations become applicable

Demo: `fib.key`

Strengthening, Again

Example (Fibonacci, the "preserves" case, open subgoal)

```
{x1 = fib(n0-n+1) & x2 = fib(n0-n+2) & F & n > 2 ->
  fib(n0-n+3) = fib(n0-n+1) + fib(n0-n+2) }
```

Look into definition of `F`

```
\forall int m; (m > 2 -> fib(m) = fib(m-1) + fib(m-2))
```

Strengthening, Again

Example (Fibonacci, the "preserves" case, open subgoal)

```
{x1 = fib(n0-n+1) & x2 = fib(n0-n+2) & F & n > 2 ->
  fib(n0-n+3) = fib(n0-n+1) + fib(n0-n+2) }
```

Look into definition of `F` — this looks ok, after all!

```
\forall int m; (m > 2 -> fib(m) = fib(m-1) + fib(m-2))
```

Instantiate `m` with `n0-n+3`

Strengthening, Again

Example (Fibonacci, the "preserves" case, open subgoal)

```
{x1 = fib(n0-n+1) & x2 = fib(n0-n+2) & F & n > 2 ->
  fib(n0-n+3) = fib(n0-n+1) + fib(n0-n+2) }
```

But need to prove that `n0-n+3 > 2`, so we add `n0 >= n`

```
\forall int m; (m > 2 -> fib(m) = fib(m-1) + fib(m-2))
```

Instantiate `m` with `n0-n+3`

Strengthening, Again

Example (Fibonacci, the "preserves" case, open subgoal)

```
{x1 = fib(n0-n+1) & x2 = fib(n0-n+2) & F & n > 2 ->
  fib(n0-n+3) = fib(n0-n+1) + fib(n0-n+2) }
```

(Almost) final loop invariant

```
x1 = fib(n0-n+1) & x2 = fib(n0-n+2) & F & n >= 2 & n0 >= n
```

Is preserved, but also initially valid?

A Final Problem: Weakening

Example (Fibonacci, the "initially valid" case)

```
n = n0 & n > 0 & F ->
  [x1 := 1 || x2 := 1]
  {x1 = fib(n0-n+1) & x2 = fib(n0-n+2) & F &
   n >= 2 & n0 >= n}
```

A Final Problem: Weakening

Example (Fibonacci, the "initially valid" case)

```
n = n0 & n > 0 & F ->
  (1 = fib(1) & 1 = fib(2) & F & n >= 2 & n0 >= n}
```

After update and other simplification . . .

A Final Problem: Weakening

Example (Fibonacci, the "initially valid" case, open subgoal)

```
|-  n = n0
  & n >  0
  & ...
    -> n >= 2
```

Cannot be shown! `n >= 2` **is too strong**

We get `n > 0` from precondition and `n <= 2` from negated guard.

The critical case seems to be `n = 1`

A Final Problem: Weakening

Example (Fibonacci, the "use" case, again)

```
|- x1 = fib(n0-n+1) & x2 = fib(n0-n+2) & F & n >=2 &
                              n0 >= n & n <= 2
   -> x2 = fib(n0)
```

Original invariant with `n >= 2`

A Final Problem: Weakening

Example (Fibonacci, the "use" case, again)

```
|- x1 = fib(n0-n+1) & x2 = fib(n0-n+2) & F & n > 0 &
                              n0 >= n & n <= 2
   -> x2 = fib(n0)
```

Weaken to `n > 0`. Too weak!

A Final Problem: Weakening

Example (Fibonacci, the "use" case, again)

```
|- x1 = fib(n0-n+1) & x2 = fib(n0-n+2) & F & n > 0 &
   (n = 1 -> n0 = 1) & n0 >= n & n <= 2
   -> x2 = fib(n0)
```

Works if we know that `n = 1` can only occur if `n0 = 1`

A Final Problem: Weakening

Example (Fibonacci, the “use” case, again)

```
|- x1 = fib(n0-n+1) & x2 = fib(n0-n+2) & F & n > 0 &
   (n = 1 -> n0 = 1) & n0 >= n & n <= 2
   -> x2 = fib(n0)
```

Final Invariant

```
x1 = fib(n0-n+1) & x2 = fib(n0-n+2) & F & n > 0
(n = 1 -> n0 = 1) & n0 >= n
```

Some Tips On Finding Invariants

General Advice

- Invariants must be developed, they don't come out of thin air!
- Be as systematic in deriving invariants as when debugging a program
- Don't forget: the program or contract (more likely) can be buggy
  - In this case, you won't find an invariant!

Some Tips On Finding Invariants, Cont'd

Technical Tips

- The desired postcondition is a good starting point
  - What, in addition to negated loop guard, is needed for it to hold?
- If the invariant candidate is not preserved by the loop body:
  - Does it need strengthening?
  - Can you add stuff from the precondition?
  - Try to express the relation between partial and final result
- Simulate a few loop body executions to discover invariant patterns
- If the invariant is not initially valid:
  - Can it be weakened such that the postcondition still follows?
  - Did you forget an assumption in the precondition?
- Several "rounds" of weakening/strengthening might be required
- Use the KeY-Hoare tool
  - Symbolic execution (of body), exit, update simplification, `andRight`
  - Look at open first-order goals: what is needed to make them closed?
  - After each change of the invariant make sure all cases are ok
  - Use the "pruning" mechanism to supply a new invariant

Summary

- Symbolic execution of loops by unwinding
  can only deal with fixed loop bounds
- In general some variant of induction is required
  to prove properties of programs with loops
- Invariant rule encodes induction over # of executed loop bodies
- Invariant rule has three parts:
  - The invariant must hold at the beginning of the loop
  - The invariant must be preserved by an arbitrary execution of the loop body provided that the guard is true
  - The negated guard plus the invariant imply the desired postcondition
- Loop invariants can be developed systematically
  - Start with the desired postcondition
  - Discover patterns through execution of a few loop bodies
  - Use strengthening, generalization, weakening
  - Use guidance by open first-order goals
- If you can't find a proof your program or contract might be wrong!

What is Left to Do?

1. Proving termination of programs
2. Proving correctness of programs with arrays