Incorrectness Logic

COL731 Course Presentation
Based on Peter W. O'Hearn's Paper & Talk @ POPL '20

Ramneet Singh

IIT Delhi

October 2023

Section 1

Introduction

Motivation

- Disconnect between Industrial Tools and Academic Theory
  - Sound program logics for reasoning about **correctness**. But code is seldom correct!
  - Industrial automated reasoning tools often **find bugs**
- Q: *Can reasoning about the presence of bugs be underpinned by sound techniques in a principled logical system?*
  - "Reimagine" static-analysis tools
  - Provide symbolic bug-catchers a principled basis
- A: *Underapproximate Reasoning*! (What is that?)

Underapproximation

- Hoare Logic Specification:

  $$\{\texttt{pre-condition}\}\ \texttt{code}\ \{\texttt{post-condition}\}$$

  $$\texttt{post-condition} \supseteq \text{strongest-post}_{\texttt{code}}\ (\texttt{pre-condition})$$

- Incorrectness Logic Specification:

  $$[\texttt{presumption}]\ \texttt{code}\ [\texttt{result}]$$

  $$\texttt{result} \subseteq \text{strongest-post}_{\texttt{code}}\ (\texttt{presumption})$$

- Have separate post-assertions for errors, normal termination
  - Assertions describe erroneous states that *can be* reached by actual program executions

Underapproximation (but picture)

- We obtain a logic which can be used to prove *the presence of bugs, but not their absence*.

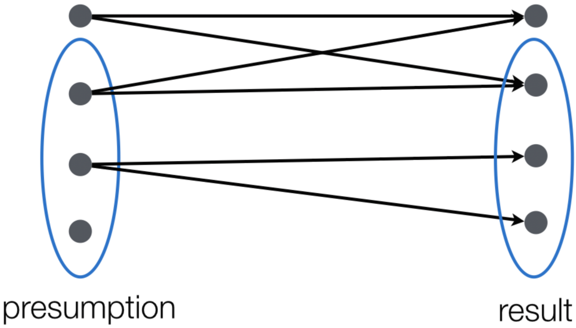

Figure 1: Source: Incorrectness Logic Paper

'*Hoare triples speak the whole truth, where the under-approximate triples speak nothing but the truth.*'

Section 2

A Unified Picture (Of Correctness and Incorrectness)

Category-Theoretic Notion

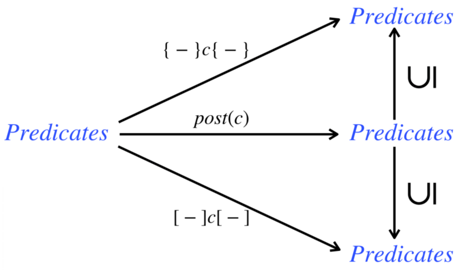

Figure 2: Commuting Diagram (Source : Incorrectness Logic Paper)

- *Predicates* $\approx 2^{\text{Program States}}$, arrows $\approx$ binary relations on *Predicates*
- $post(c)$ is a function, the other two are non-functional
- $[-]c[-] = post(c);\supseteq$ and $\{-\}c\{-\} = post(c);\subseteq$
- $post(c)p$ = strongest post of p = weakest under-approximating post of p

Reasoning Principles - I

$$\wedge\vee\ \textit{Symmetry:}\quad [p]c[q_1] \wedge [p]c[q_2] \iff [p]c[q_1 \vee q_2]$$

$$\{p\}c\{q_1\} \wedge \{p\}c\{q_2\} \iff \{p\}c\{q_1 \wedge q_2\}$$

$$\Uparrow\Downarrow\ \textit{Symmetry:}\quad p' \Leftarrow p \wedge [p]c[q] \wedge q \Leftarrow q' \implies [p']c[q']$$

$$p' \Rightarrow p \wedge \{p\}c\{q\} \wedge q \Rightarrow q' \implies \{p'\}c\{q'\}$$

Figure 3: Correctness & Incorrectness Principles (Source : Incorrectness Logic Paper)

- $[p]c[q \vee r] \implies [p]c[q]$ allows you to *drop paths* going forward.
  - Not possible in overapproximate logics - but can *forget information* along each path
- Rules of consequence allow specifications to be adapted to broader contexts

Reasoning Principles - II

$$\textit{Principle of Agreement:} \quad [u]c[u'] \wedge u \Rightarrow o \wedge \{o\}c\{o'\} \implies u' \Rightarrow o'$$

$$\textit{Principle of Denial:} \quad [u]c[u'] \wedge u \Rightarrow o \wedge \neg(u' \Rightarrow o') \implies \neg(\{o\}c\{o'\})$$

Figure 4: Correctness & Incorrectness Principles (Source : Incorrectness Logic Paper)

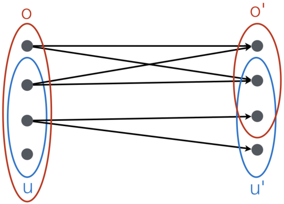

Figure 5: Analogy with Testing (Source : Incorrectness Logic Paper)

- Program testing works on the principle of denial (traditionally, $|u| = |u'| = 1$, a test run)

Isn’t Incorrectness Just Not Correctness?

- Yes, but we aren’t powerful enough to precisely compute either!
- *’The inability to prove an over-approximate spec (whether found by a tool or specified by a human) does not imply an error in a program, and neither does not having found a bug imply that there are none: thus, the need for dedicated techniques for each.’*

Section 3

Build Your Muscle

Under-Approximating Triples - I

$$[z = 11]$$

```
if (x is even) {
    if (y is odd) {
        z = 42;
    }
}
```

$$[z = 42]$$

Under-Approximating Triples - I

$$[z = 11]$$

```
if (x is even) {
    if (y is odd) {
        z = 42;
    }
}
```

$$[z = 42]$$

This triple *does not hold*! The state `[z : 42, x : 1, y : 3]` has no predecessor!

Under-Approximating Triples - II

$[true]$

```
if (x is even) {
    if (y is odd) {
        z = 42;
    }
}
```

$[z = 42]$

Under-Approximating Triples - II

$[true]$

```
if (x is even) {
    if (y is odd) {
        z = 42;
    }
}
```

$[z = 42]$

This triple *holds*!

Under-Approximating Triples - III

$$[z = 11]$$

```
if (x is even) {
    if (y is odd) {
        z = 42;
    }
}
```

$$[z = 42 \wedge (x \text{ is even }) \wedge (y \text{ is odd })]$$

Under-Approximating Triples - III

$$[z = 11]$$

```
if (x is even) {
    if (y is odd) {
        z = 42;
    }
}
```

$$[z = 42 \land (x \text{ is even }) \land (y \text{ is odd })]$$

This triple *holds*!

Under-Approximating Triples - IV

$$[\text{true}]$$

```
if (x is even) {
    if (y is odd) {
        z = 42;
    }
}
```

$$[z = 42 \land (x \text{ is even }) \land (y \text{ is odd })]$$

Under-Approximating Triples - IV

$$[\text{true}]$$

```
if (x is even) {
    if (y is odd) {
        z = 42;
    }
}
```

$$[z = 42 \land (x \text{ is even }) \land (y \text{ is odd })]$$

This triple *holds*!

Specifying Incorrectness

- Reasoning about errors?

Specifying Incorrectness

- Reasoning about errors?
- Have separate result-assertion forms for normal and (erroneous or abnormal) termination.

```
void foo(char * str)
/* presumes: [ *str[]==s ]
   achieves: [ er: *str[]==s && length(s) > 16 ] */
{
    char buf[16];
    strcpy(buf,str);
}

int main(int argc, char *argv[])
{ foo(argv[1]); }
```

- Spec: if the length of the input string is greater than 16 then we can get an error (in this case a buffer overflow).

Under-approximate Success

- Why not over-approximate for successful and under-approximate for erroneous termination?
  - Under-approximate result assertions describing successful computations can help us **soundly discover bugs that come after a procedure is called**.

Under-approximate Success

- Why not over-approximate for successful and under-approximate for erroneous termination?
  - Under-approximate result assertions describing successful computations can help us **soundly discover bugs that come after a procedure is called**.

```
void mkeven()
/* presumes: [true], wrong achieves: [ok: x==2 || x==4] */
{ x=2; }

void usemkeven()
{ mkeven(); if (x==4) {error();} }
```

- We don't want false positives!

Section 4

Proof System

Setup

- Simple imperative language. `error()` halts execution and raises an error signal, `er`.
- Abnormal control flows impact reasoning about sequential composition
  - Solution: associate assertions with a set of exit conditions ϵ
  - ϵ includes (at least) ok for normal termination and er causes by `error()`
- $[p]C[\epsilon : q] = q$ under-approximates the states when C exits via ϵ starting from states in p.
- x is **not** free in p iff $\sigma \in p \iff (\forall v \,.\, (\sigma | x \mapsto v) \in p)$. [BUG]
- Treat p, q semantically (i.e., any $\subseteq \Sigma$, the set of program states) – don't fix a language.
  - By treating assertions semantically, we are essentially appealing to mathematics (or set theory) as an oracle in our proof theory when we use $\implies$ in proof rules.
- $[p]C[\mathrm{ok} : q][\mathrm{er} : r]$ as shorthand for $[p]C[\mathrm{ok} : q]$ and $[p]C[\mathrm{er} : r]$.

Generic Proof Rules - I

Empty under-approximates

$$[p]C[\epsilon: \mathit{false}]$$

Consequence

$$\frac{p' \Leftarrow p \quad [p]C[\epsilon: q] \quad q \Leftarrow q'}{[p']C[\epsilon: q']}$$

Disjunction

$$\frac{[p_1]C[\epsilon: q_1] \quad [p_2]C[\epsilon: q_2]}{[p_1 \vee p_2]C[\epsilon: q_1 \vee q_2]}$$

Unit

$$[p]\texttt{skip}[ok: p][er: \mathit{false}]$$

Sequencing (short-circuit)

$$\frac{[p]C_1[er: r]}{[p]C_1; C_2[er: r]}$$

Sequencing (normal)

$$\frac{[p]C_1[ok: q] \quad [q]C_2[\epsilon: r]}{[p]C_1; C_2[\epsilon: r]}$$

Iterate zero

$$[p]C^{\star}[ok: p]$$

Iterate non-zero

$$\frac{[p]C^{\star}; C[\epsilon: q]}{[p]C^{\star}[\epsilon: q]}$$

Backwards Variant (where n fresh)

$$\frac{[p(n) \wedge nat(n)]C[ok: p(n+1) \wedge nat(n)]}{[p(0)]C^{\star}[ok: \exists n. p(n) \wedge nat(n)]}$$

Choice (where $i = 1$ or 2)

$$\frac{[p]C_i[\epsilon: q]}{[p]C_1 + C_2[\epsilon: q]}$$

Error

$$[p]\texttt{error()}[ok: \mathit{false}][er: p]$$

Assume

$$[p]\texttt{assume } B[ok: p \wedge B][er: \mathit{false}]$$

$$\texttt{while } B \texttt{ do } C \quad =_{def} \quad (\texttt{assume}(B); C)^{\star}; \texttt{assume}(\neg B)$$

$$\texttt{if } B \texttt{ then } C \texttt{ else } C' \quad =_{def} \quad (\texttt{assume}(B); C) + (\texttt{assume}(\neg B); C')$$

$$\texttt{assert}(B) \quad =_{def} \quad \texttt{assume}(B) + (\texttt{assume}(\neg B); \texttt{error()})$$

Figure 6: Generic Proof Rules of Incorrectness Logic (Source: Incorrectness Logic Paper)

Generic Proof Rules - Axioms

- Valid across different models of states and commands
  - Usual: States = Variables $\rightarrow$ Values and Commands = Binary Relations on States
  - Others based on traces, separation logic etc.

$$\text{Assume}\ \frac{}{[p]\texttt{assume}B[\text{ok} : p \wedge B][\text{er} : \text{false}]} \qquad \text{Skip}\ \frac{}{[p]\texttt{skip}[\text{ok} : p][\text{er} : \text{false}]}$$

$$\text{Empty under-approximates}\ \frac{}{[p]C[\epsilon : \text{false}]}$$

- `assume(B)` statement : `B` is a Boolean expression, can be from an otherwise-unspecified first-order logic signature.
- Axioms for `assume` and `skip` : give the expected assertions for normal termination, but specify `false` (the empty set of states) for abnormal.

Generic Proof Rules - Consequence, Disjunction & Choice

$$\text{Consequence}\ \frac{p' \Longleftarrow p \quad [p]C[\epsilon : q] \quad q \Longleftarrow q'}{[p']C[\epsilon : q']}$$

$$\text{Disjunction}\ \frac{[p_1]\ C\ [\epsilon : q_1] \quad [p_2]\ C\ [\epsilon : q_2]}{[p_1 \vee p_2]\ C\ [\epsilon : q_1 \vee q_2]}$$

$$\text{Choice (where } i = 1, 2)\ \frac{[p]\ C_i\ [\epsilon : q]}{[p]\ C_1 + C_2\ [\epsilon : q]}$$

- The rule of consequence lets us *enlarge (weaken) the pre* and *shrink (strengthen) the post-assertion*.
  - Allows us to *drop disjuncts in the post* and *drop conjuncts in the pre*.
- *'Enlarging the pre was used in the Abductor tool ([Calcagno et al. 2011], which led to Facebook Infer), when guessing pre-conditions in programs with loops.'*
  - Was unsound in the over-approximating logic used there, required a re-execution step which filtered out unsound pre-conditions

Generic Proof Rules - Sequencing and Iteration

Sequencing(short-circuit)

$$\frac{[p]\ C_1\ [\mathrm{er}: r]}{[p]\ C_1; C_2\ [\mathrm{er}: r]}$$

Sequencing(normal)

$$\frac{[p]\ C_1\ [\mathrm{ok}: q] \quad [q]\ C_2\ [\epsilon: r]}{[p]\ C_1; C_2\ [\epsilon: r]}$$

Iterate zero

$$\frac{}{[p]\ C^*\ [\mathrm{ok}: p]}$$

Iterate non-zero

$$\frac{[p]\ C^*; C\ [\epsilon: q]}{[p]\ C^*\ [\epsilon: q]}$$

- The *Iterate zero* rule shows that **any assertion is a valid under-approximate invariant for Kleene iteration**.
  - Loop invariants don't play a central role in under-approximate reasoning. Notion of *subvariants* mentioned in POPL'23 tutorial.
- The *Iterate non-zero* rule uses $C^*; C$ rather than $C; C^*$ to help reasoning about cases where an error is thrown inside an iteration. Will see an example later.

Generic Proof Rules - Derived Choice and Iteration, Backwards Variant

Derived Unrolling Rule

$$\frac{[p]\ C^i\ [\epsilon : q_i]\ ,\ \ \text{all } i \leq \ \text{bound}}{[p]\ C^*\ [\epsilon : \bigvee_{i \leq \text{bound}} q_i]}$$

Derived Rule of Choice

$$\frac{[p]\ C_1\ [\epsilon : q_1] \quad [p]\ C_2\ [\epsilon : q_2]}{[p]\ C_1 + C_2\ [\epsilon : q_1 \vee q_2]}$$

- One of the things that iteration can do is execute its body i times.
- The *Unrolling* rule gives a similar capability symbolic bounded model checking (but we need the *Backwards Variant* rule too in general).

$$\text{Backwards Variant (where } n \text{ fresh)}\ \frac{[p(n) \wedge \mathrm{nat}(n)]\ C\ [\epsilon : p(n+1) \wedge \mathrm{nat}(n)]}{[p(0)]\ C^*\ [\epsilon : \exists n\, .\, p(n) \wedge \mathrm{nat}(n)]}$$

- $p(.)$ = a parameterized predicate (a function from expressions to predicates).

Backwards Variant relation with Program Termination

- [presumption] c [ϵ : result] expresses a reachability property that involves termination.
  - *Every state in the result is reachable from some state in the presumption.*
- But this does not imply that a loop must terminate on all executions!
  - Enough paths terminate to cover all the states in result, while other paths may diverge.
- *Backward variant* rule is similar to proof rules for proving program termination (typically use a "variant" that decreases on each loop iteration)
  - But reflects the *backward* nature of this property. p goes down when executing backwards.
- What about the forward variant? $[\exists n \,.\, p(n) \wedge \mathrm{nat}(n)]$ C^* $[\mathrm{ok} : p(0)]$.

Backwards Variant relation with Program Termination

- [presumption] c [ϵ : result] expresses a reachability property that involves termination.
  - *Every state in the result is reachable from some state in the presumption.*
- But this does not imply that a loop must terminate on all executions!
  - Enough paths terminate to cover all the states in result, while other paths may diverge.
- *Backward variant* rule is similar to proof rules for proving program termination (typically use a "variant" that decreases on each loop iteration)
  - But reflects the *backward* nature of this property. p goes down when executing backwards.
- What about the forward variant? $[\exists n \,.\, p(n) \wedge \mathrm{nat}(n)]\ C^*\ [\mathrm{ok} : p(0)]$.
- It is always true :)

Reachability and Liveness

- Liveness : "something (good) will eventually happen".
- Our reachability property:
  - Backwards: For every state in the result, it is possible to eventually reach a state in the pre by executing backwards.
  - Forwards: If we *explore (enumerate pre-states, backtrack, dovetail)* executions from all pre-states, then eventually any given state in the result will be encountered.
- The "eventually" in our forwards does not concern all paths, rather it is an "existential liveness property".
- The over-approximating triple $\{\text{pre}\}C\{\text{post}\}$ describes a safety property, that "nothing bad (= not post) will happen".

Specific Proof Rules - Variables and Mutation

Assignment

$[p]x = e[ok{:}\ \exists x'.p[x'/x] \wedge x = e[x'/x]][er{:}\ false]$

Nondet Assignment

$[p]x = \texttt{nondet()}[ok{:}\ \exists x' p][er{:}\ false]$

Constancy

$$\frac{[p]C[\epsilon{:}\ q]}{[p \wedge f]C[\epsilon{:}\ q \wedge f]}\ Mod(C) \cap Free(f) = \emptyset$$

Local Variable

$$\frac{[p]C(y/x)[\epsilon{:}\ q]}{[p]\texttt{local}\ x.C[\epsilon{:}\ \exists y.q]}\ y \notin Free(p, C)$$

Substitution I

$$\frac{[p]C[\epsilon{:}\ q]}{([p]C[\epsilon{:}\ q])(e/x)}\ \big(Free(e) \cup \{x\}\big) \cap Free(C) = \emptyset$$

Substitution II

$$\frac{[p]C[\epsilon{:}\ q]}{([p]C[\epsilon{:}\ q])(y/x)}\ y \notin Free(p, C, q)$$

Figure 7: Rules for Variables and Mutation (Source: Incorrectness Logic Paper)

- Sound when states are functions of type $\text{Variables} \to \text{Values}$.
- *Mod*(C) is the set of variables modified by assignment statements in C, and *Free*(r) is the set of free variables in an assertion r.
- e and `nondet()` are syntactically distinct.
  - e is an expression built up from a first-order logic signature, can appear within assertions, and is side-effect free.
  - `nondet()` does not appear in assertions.

Specific Proof Rules - Variables and Mutation

Assignment

$$[p]x = e[ok: \exists x'.p[x'/x] \wedge x = e[x'/x]][er: false]$$

Nondet Assignment

$$[p]x = \texttt{nondet}()[ok: \exists x' p][er: false]$$

Constancy

$$\frac{[p]C[\epsilon: q]}{[p \wedge f]C[\epsilon: q \wedge f]} \; Mod(C) \cap Free(f) = \emptyset$$

Local Variable

$$\frac{[p]C(y/x)[\epsilon: q]}{[p]\texttt{local}\ x.C[\epsilon: \exists y.q]} \; y \notin Free(p, C)$$

Substitution I

$$\frac{[p]C[\epsilon: q]}{([p]C[\epsilon: q])(e/x)} \; \left(Free(e) \cup \{x\}\right) \cap Free(C) = \emptyset$$

Substitution II

$$\frac{[p]C[\epsilon: q]}{([p]C[\epsilon: q])(y/x)} \; y \notin Free(p, C, q)$$

Figure 8: Rules for Variables and Mutation (Source: Incorrectness Logic Paper)

- Sound when states are functions of type $\mathrm{Variables} \to \mathrm{Values}$.
- $Mod(C)$ is the set of variables modified by assignment statements in C, and $Free(r)$ is the set of free variables in an assertion r.
- e and `nondet()` are syntactically distinct.
  - e is an expression built up from a first-order logic signature, can appear within assertions, and is side-effect free.
  - `nondet()` does not appear in assertions. [BUG] in *Nondet Assignment* rule

Specific Proof Rules - Assignment

- Incorrectness logic uses Floyd's forward-running assignment axiom rather than Hoare's backwards-running one.

$$\text{Assignment}\ \frac{}{[p]\ x = e\ [\mathrm{ok} : \exists x' .\, p[x'/x] \land x = e[x'/x]]\ [\mathrm{er} : \mathrm{false}]}$$

- Would the below rule be correct?

$$\text{Assignment'}\ \frac{}{[p[e/x]]\ x = e\ [\mathrm{ok} : p]\ [\mathrm{er} : \mathrm{false}]}$$

Specific Proof Rules - Assignment

- Incorrectness logic uses Floyd's forward-running assignment axiom rather than Hoare's backwards-running one.

$$\text{Assignment}\ \frac{}{[p]\ x = e\ [\mathrm{ok} : \exists x'\,.\,p[x'/x] \land x = e[x'/x]]\ [\mathrm{er} : \mathrm{false}]}$$

- Would the below rule be correct?

$$\text{Assignment'}\ \frac{}{[p[e/x]]\ x = e\ [\mathrm{ok} : p]\ [\mathrm{er} : \mathrm{false}]}$$

- No! For example, $[y == 42]\ x = 42\ [\mathrm{ok} : x == y]$ is not valid (take the post-state `[x : 3, y : 3]`).

Specific Proof Rules - Substitution, Constancy, & Local Variable Rule

Substitution I

$$\frac{(\mathrm{Free}(e) \cup \{x\}) \cap \mathrm{Free}(C) = \emptyset \qquad [p]\ C\ [\epsilon : q]}{([p]\ C\ [\epsilon : q])(e/x)}$$

Constancy

$$\frac{\mathrm{Mod}(C) \cap \mathrm{Free}(f) = \emptyset \qquad [p]\ C\ [\epsilon : q]}{[p \wedge f]\ C\ [\epsilon : q \wedge f]}$$

Substitution II

$$\frac{y \notin \mathrm{Free}(p, C, q) \qquad [p]\ C\ [\epsilon : q]}{([p]\ C\ [\epsilon : q])(y/x)}$$

Local Variable

$$\frac{y \notin \mathrm{Free}(p, C) \qquad [p]\ C(y/x)\ [\epsilon : q]}{[p]\ \texttt{local}\ x\,.\,C\ [\epsilon : \exists y\,.\,q]}$$

- The rules of *Substitution*, *Constancy* & *Consequence* are important for adapting specifications for use in different contexts.

Exercise: Derive rules for `assert`

- Recall `assert(B) = assume(B) + (assume(!B) ; error())`

$$[p \wedge B]\ \texttt{assert}(B)\ [\mathrm{ok} : (p \wedge B)]\ [\mathrm{er} : \mathrm{false}]$$

$$[p \wedge \neg B]\ \texttt{assert}(B)\ [\mathrm{ok} : \mathrm{false}]\ [\mathrm{er} : (p \wedge \neg B)]$$

$$[p]\ \texttt{assert}(B)\ [\mathrm{ok} : (p \wedge B)]\ [\mathrm{er} : (p \wedge \neg B)]$$

Section 5

Reasoning with the Logic

Setup

- Examples motivated by existing tools, *but* "we are not claiming at this time that incorrectness logic leads to better practical results than these mature tools"
- *'A basic test of a potential foundational formalism is how it expresses a variety of patterns that have arisen naturally.'*
- No formal treatment of procedures. Assume summary-like hypotheses for reasoning.

$$[p]\ \texttt{foo()}\ [\mathrm{ok} : q]\ [\mathrm{er} : r] \vdash [p']\ C\ [\mathrm{ok} : q']\ [\mathrm{er} : r']$$

- *Principle of reuse*: Reason about `foo()`'s body once, don't revisit at call sites (aka summary-based analysis - COL729 throwback)

loop0 - I

```
void loop0() {
    /* (default presumes is "true" when not specified)
     * achieves: [ok: x>=0 ] */
    int n = nondet();
    x=0;
    while (n > 0) {
        x = x + n;
        n = nondet();
    }}

void client0() { /* achieves: [er: x==200000] */
    loop0();
    if (x == 200000) error(); }
```

- Assuming `loop0` summary, can prove `client0` spec using below followed by sequencing rule.

$$\frac{[\mathrm{true}]\ \texttt{loop0()}\ [ok : x \geq 0] \quad x \geq 0 \Longleftarrow x == 200000}{[\mathrm{true}]\ \texttt{loop0()}\ [ok : x == 200000]}$$

loop0 - II

- How to prove `loop0()` spec?

loop0 - II

- How to prove `loop0()` spec?
- Just unroll once! Then apply *Local Variable* rule + *Unrolling* rule + *Rule of Consequence*.

```
[ x==0 ]
    if (n>0) {
        [ x==0 && n>0 ]
        x = x+n; n = nondet(); [ x>0 ]
    } else
    { [ x==0 && n<=0 ] skip;
    }
    [ x>0 || (x==0 && n<=0) ]
        assume (n<=0);
    [ (x>0 && n<=0) || (x==0 && n<=0) ]
[ ok: x>=0 && n<=0 ]
```

loop1 - I

```
void loop1()
/*  achieves1: [ok: x==0 || x==1 || x==2]
    achieves2: [ok: x>=0] */
{   x = 0;
    Kleene-star {
        x = x + 1;
}   }

void client1()
/* achieves: [er: x==200000] */
{   loop1();
    if ( x==200000 ) error();
}
```

loop1 - II

- Infinitely many paths through `loop1()`, and the loop is not guaranteed to terminate.
- *Unrolling* rule: post-conditions for any finite-depth unrollings of the loop. `achieves1`==2 unrollings.
- Not enough to trigger the error in `client1()`. (Unroll 200000 times?)
- Need the backwards variant rule!

$$n \text{ fresh} \frac{[x == n \wedge \text{nat}(n)]\ x = x + 1\ [\text{ok} : x == n + 1 \wedge \text{nat}(n)]}{[x == 0]\ (x = x + 1)^*\ [\text{ok} : \exists n\,.\, x == n \wedge \text{nat}(n)]}$$

loop2 - I

- Error inside iteration: This is why we need $C^*; C$, not $C; C^*$!

```
void loop2()
/* achieves: [er: x==200000] */
{   x = 0;
    Kleene-star{
        if (x==200000) error();
        x = x + 1;
}   }
```

- How can we show this?

loop2 - II

- Use *Backwards Variant* rule ($p(n) = 0 \leq x \leq 200000 \wedge x == n$).

$$[x == 0] \ (\mathrm{Body})^* \ [\mathrm{ok} : 0 \leq x \leq 200000]$$

$$[x == 0] \ (\mathrm{Body})^* \ [\mathrm{ok} : x == 200000]$$

loop2 - II

- Use *Backwards Variant* rule ($p(n) = 0 \leq x \leq 200000 \wedge x == n$).

$$[x == 0]\ (\text{Body})^*\ [\text{ok} : 0 \leq x \leq 200000]$$

$$[x == 0]\ (\text{Body})^*\ [\text{ok} : x == 200000]$$

- *Assume* + *Error* + *Sequencing* + *Short-Circuit* gives us

$$[x == 200000]\ \text{Body}\ [\text{er} : x == 200000]$$

loop2 - II

- Use *Backwards Variant* rule ($p(n) = 0 \leq x \leq 200000 \land x == n$).

$$[x == 0]\ (\mathrm{Body})^*\ [\mathrm{ok} : 0 \leq x \leq 200000]$$

$$[x == 0]\ (\mathrm{Body})^*\ [\mathrm{ok} : x == 200000]$$

- *Assume* + *Error* + *Sequencing* + *Short-Circuit* gives us

$$[x == 200000]\ \mathrm{Body}\ [\mathrm{er} : x == 200000]$$

- *Sequencing*

$$[x == 0]\ (\mathrm{Body})^*; \mathrm{Body}\ [\mathrm{er} : x == 200000]$$

`loop2` - II

- Use *Backwards Variant* rule ($p(n) = 0 \leq x \leq 200000 \wedge x == n$).

$$[x == 0]\ (\mathrm{Body})^*\ [\mathrm{ok} : 0 \leq x \leq 200000]$$

$$[x == 0]\ (\mathrm{Body})^*\ [\mathrm{ok} : x == 200000]$$

- *Assume* + *Error* + *Sequencing* + *Short-Circuit* gives us

$$[x == 200000]\ \mathrm{Body}\ [\mathrm{er} : x == 200000]$$

- *Sequencing*

$$[x == 0]\ (\mathrm{Body})^*; \mathrm{Body}\ [\mathrm{er} : x == 200000]$$

- *Iterate non-zero*

$$[x == 0]\ (\mathrm{Body})^*\ [\mathrm{er} : x == 200000]$$

loop3

- What if we used $C; C^*$? The proof for `loop2()` spec would have 200000 applications of *Sequencing*.

```
void loop3()
/* achieves: [er: \exists n (x==n /\ n <= 200000)] */
{ y = nondet();
  x = 0;
  Kleene-star {
    if (y == 200000) error();
    x = x + 1;
    y = y + 1;
} }
```

loop3

- What if we used $C; C^*$? The proof for `loop2()` spec would have 200000 applications of *Sequencing*.

```
void loop3()
/* achieves: [er: \exists n (x==n /\ n <= 200000)] */
{ y = nondet();
  x = 0;
  Kleene-star {
    if (y == 200000) error();
    x = x + 1;
    y = y + 1;
} }
```

- We don't know the number of iterations it'll take to get an error, and cannot prove the `er` assertion with finitely many unrollings.

loop3

- What if we used $C; C^*$? The proof for `loop2()` spec would have 200000 applications of *Sequencing*.

```
void loop3()
/* achieves: [er: \exists n (x==n /\ n <= 200000)] */
{ y = nondet();
  x = 0;
  Kleene-star {
    if (y == 200000) error();
    x = x + 1;
    y = y + 1;
} }
```

- We don't know the number of iterations it'll take to get an error, and cannot prove the `er` assertion with finitely many unrollings.
- But we can be `cool` and use *Backwards Variant* to derive more general under-approximate assertions than unrolling, and use the original *Iterate non-zero* to derive an error from the general assertion (with just one C statement).

Conditionals

- Use of Boolean conditions that are difficult for current theorem provers to deal with causes expressiveness issues.
  - E.g. multiplication goes beyond the decidable subsets of arithmetic encoded in automatic theorem provers.
- How do tools deal with this? And how can Incorrectness Logic deal with this?

Conditionals - Approach I

```
int difficult(int y)
{   return (y*y); /* or hash(y) or obfuscated code */
}

void client()
/* achieves1 : [ok: y==49 && x==1] */
{   int z = nondet();
    if (y == difficult(z))
        x=1;
    else
        x=2;
}
```

- Pragmatic Approach from Dynamic Symbolic Execution: *Concretize symbolic variables*. (replace z with 7).
- Do this in incorrectness logic by *shrinking the post*. Have `[y==z*z] assume(y==difficult(z)) [ok: y==z*z]` and $y == z * z \Longleftarrow y == z * z \wedge z == 7$.

Conditionals - Approach II

```
void client()
/* achieves2 : [ok: exists z . (y==difficult(z) && x==1)
    || (y!=difficult(z) && x==2)] */
{   int z = nondet();
    if (y == difficult(z))
        x=1;
    else
        x=2;
}

void test1()
/* achieves: [er: exists z .
    (y==difficult(z) && x==1)
    || (y != difficult(z) && x==2)] */
{   client(); if (x==1 || x==2) error();
}
```

- Record information lazily (hoping difficulty won't matter, like in `test1()`).

Conditionals - Approach III

```
void client()
/* achieves3 : [ok: x==1 || x==2] */
{   int z = nondet();
    if (y == difficult(z))
        x=1;
    else
        x=2;
}

void test2()
{   client(); if (x==2) error(); }
```

- Record disjuncts for both branches, but discard the difficult bits. Unsound! (e.g. [x:1, y:3] not reachable).
- Used for pragmatic reasons in tools like SMART, Infer.RacerD.
- RacerD: it is *an under-approximation of an over-approximation*, where the over-approximation arises by replacing Booleans it doesn't understand with nondeterministic choice.

Tool Design Insights

- `Infer.RacerD`: Tools can make localised unsound decisions, which act as assumptions for further sound steps.
- *'From this perspective, the role of logic is not to produce iron-clad unconditional guarantees, but is to clarify assumptions and their role when making sound inferences.'*
- `Infer.Pulse`: 20 disjuncts case was ~2.75x wall clock time faster, ~3.1x user time faster, and found 97% of the issues that the 50 disjuncts case found.
  - Choice is not binary! E.g., deploy fast one at code review time, slow one later in the process.

Flaky Tests - I

- “flaky test” : due to nondeterminism, can give different answers on different runs.
- If π is a program path, then
  - $\mathrm{wp}(\pi)q$: States for which execution of π is guaranteed to terminate and satisfy q.
  - $\mathrm{wpp}(\pi)q$: States for which execution of π is possible to terminate and satisfy q.
- We will use these to obtain pre-assertions, then use forward reasoning to obtain under-approximate post-assertions.
- Why do we need these?
  - Because strongest under-approximate presumptions do not exist in general (see 5.2 in paper).

Flaky Tests - II

```
void foo()
/* sturdy pre [x is even], ach [er: x is even][ok: false]
   flaky pre [x is odd], ach [er: x is odd][ok: x is odd] */
{
    if (x is even) error();
    else { if (nondet()) skip; else error(); }
}

void flakey_client()
/* flaky achieves: [er: x==3 || x==5] */
{   x = 3;
    foo();
    x = x+2;
    assert(x==4);
}
```

- Use $\mathrm{wp}(\mathtt{assume}(x\ \mathtt{is\ even}))\ \mathrm{true}$ for sturdy presumes, $\mathrm{wpp}(\mathtt{assume}(x\ \mathtt{is\ odd}); b = \mathtt{nondet}(); \mathtt{assume}(b))\ \mathrm{true}$ (where b is local) for flaky presumes.

Reasoning about Procedures - I

- For a path without procedure calls - say a sequential composition of assignment, assume and assert statements
  - Can perform strongest post-condition reasoning, which is also under-approximate.
- Can combine together pre/post pairs for a number of paths to get an under-approximate summary for a procedure.
- But then using that summary to reason (*soundly*) about a path containing a procedure call is subtle.
- Even in straight-line code, it is *easy* to get a false positive using strongest post-condition reasoning with Hoare logic.

Reasoning about Procedures - II

```
void inc()
/* presumes1: [x>=0], achieves1: [ok: x>0]
   presumes2: [x==m && m>=0], achieves2: [ok: x==m+1 && m>=0] */
{   assert(x>=0);
    x=x+1;
}

void client()
/* presumes1: [x>=0], wrong achieves1: [ok: x>0]
   presumes2: [x==m && m>=0], achieves2: [ok: x==m+2 && m>=0] */
{   inc(); inc(); }

void test()
/* wrong achieves1: [er: x==1]
   achieves2: [er: false] */
{   x = 0;
    client();
    assert(x>=2);
}
```

Reasoning about Procedures - III

- Incorrectness logic prevents the unsound (for bug catching) inference `presumes1/achieves1` for `client()` and thus `test()`.
- A different spec of `inc()`, given by `presumes2/achieves2`, lets us reason about the composition `inc();inc()` in `client()` more positively, to obtain `presumes2/achieves2` as stated for `client()`.
- Note: A procedure spec or summary should carry information about free variables and modified - for `inc()`, x is free and modified, m is not free in the procedure body.
- This allows us to apply rules of *Substitution* and *Constancy* to get `client()` spec from `inc()` spec.

Context and Conclusions

- *'The theory Infer was based on originally ... does not match its use to find bugs rather than to prove their absence.'*
- Led to `RacerD`, `Pulse` program analysers.
- A more general theory of "incorrectness" logic (starting from reverse Hoare logic by de Vries and Koutavas in 2011).
- Related theoretical notions: `wlp` (weakest liberal precondition), `wpp` (weakest possible precondition), dynamic logic.
- Each form of reasoning is as fundamental as the other, they just have different principles. Recall:
  *For correctness reasoning, you **get to forget information** as you go along a path, but you **must remember** all the paths. For incorrectness reasoning, you **must remember** information as you go along a path, but you **get to forget** some of the paths.*
- Possible extensions to other models, concurrency. Possible reuse of work from termination proving.

Thank You!

Section 6

Appendix

Backwards Variant - Example I

- For any fixed number of iterations, we can just unfold the *Iterate non-zero* rule and use *Iterate zero*. But no. of iterations may be unknown!

```
x = 0;
y = nondet();
while (y != N) do {
    y = y + 1;
    x = x + 1;
}
```

$[x = 0]$ `while (y != N) do  y = y + 1; x = x + 1` $[\mathrm{ok} : \exists n \,.\, x == n \wedge \mathrm{na}$

Backwards Variant - Example II

```
void loop3()
/* achieves: [er: x == 200000] */
{ x = nondet();
  Kleene-star {
    if (x == 200000) error();
    x = x + 1;
} }
```

Backwards Variant - Example II

```
void loop3()
/* achieves: [er: x == 200000] */
{ x = nondet();
  Kleene-star {
    if (x == 200000) error();
    x = x + 1;
} }
```

- Can guess a value k returned by `nondet()` and apply *Sequencing* $200000 - k$ times. Or

Backwards Variant - Example II

```
void loop3()
/* achieves: [er: x == 200000] */
{ x = nondet();
  Kleene-star {
    if (x == 200000) error();
    x = x + 1;
} }
```

- Can guess a value k returned by `nondet()` and apply *Sequencing* $200000 - k$ times. Or
- Can be `cool` and use *Backwards Variant* to derive more general under-approximate assertions than unrolling, and use the original *Iterate non-zero* to derive an error from the general assertion (with just one C statement).