Incorrectness Logic

Peter W. O'Hearn

May 27, 2020

Overview

- Intuitive Introduction to Incorrectness Logic
- Incorrectness Logic: Syntax and Proof Rules
- Incorrectness Logic: Semantic
- Reasoning with the Logic

Incorrectness Logic: Intuitive Introduction

This paper describes a simple logic for program incorrectness which is the other side of the coin to Hoare's logic of correctness.
Hoare's Logic:

$$\{precond\}code\{postcond\}$$

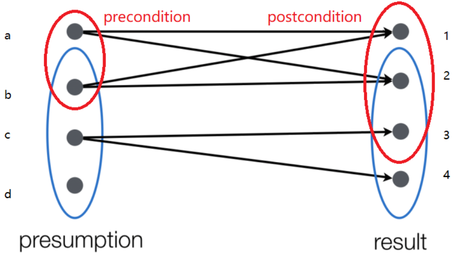
where *postcond* is an **over-approximation(superset)** of the final states reachable.
Incorrectness Logic:

$$[presumption]code[result]$$

where *result* is a **under-approximation(subset)** of the final states reachable.

Why Incorrectness Logic?

As is known, Hoare's logic is used to prove that programs do not contain bugs.
Incorrectness logic, however, is used to prove the presence of bugs but not their absence.



1.pdf

Toolset

Website: fbinfer.com

infer.png

Examples

Under-approximating Triples

For incorrectness logic, we use "presume" as to identify a pre-assertion and use "achieves" to identify the post-assertion. Consider an example below.

```
/* presumes: [z==11] */
  if (x is even) {
     if (y is odd) {
       z=42;
  }  }
/* achieves: [z==42] */
```

Question: is the assertion $z == 42$ an under-approximation of the states reachable from executing code from states statisfying the presumption?

```
/* achieves: [z==42 && (x is even) && (y is odd) ] */
```

Intuition: Inference

Inference rule of Hoare's logic:

$$\frac{\{p\}C\{q \wedge r\}}{\{p\}C\{q\}}$$

where the post condition $q \wedge r$ can be relieved to q.
Inference rule of Incorrectness logic:

$$\frac{[p]C[q_1 \vee q_2]}{[p]C[q_1]}$$

where the result can only be strengthened.

Examples

Specifying Incorrectness
We can reason about errors by distinguishing the result-assertion by some tags. For example, we use “er: predicate” for 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]); }
```

In this paper, the author only considers errors caused by statement `error();`.

Examples

Under-approximate Success

Similar to the incorrectness specification, we also need tags for normal, not exceptional, termination of a program. Here we use "ok" for this purpose.

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

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

Incorrectness Logic: Syntax and Proof Rules

Let ϵ range over a collection of exit conditions, to include at lease "ok" and "er". An *under-approximate triple* is of the form:

$$[p]C[\epsilon : q]$$

meaning q under-approximates the states when C exits via ϵ starting form states in p.
Sometimes, we write $[p]C[ok : q][er : r]$ as shorthand for $[p]C[ok : q]$ and $[p]C[er : r]$
An important point: the triple $[p]C[\epsilon : q]$ express the reachability property that involves termination. **Every state in the result is reachable from some states in the presumption**

Proof Rules

- *Empty under-approximate*: $[p]C[\epsilon : false]$
- *Unit*:

$$[p]skip[ok : p][er : false]$$

- *Consequence*: $\dfrac{p' \Leftarrow p, [p]C[\epsilon : q], q \Leftarrow q'}{[p']C[\epsilon : q']}$
- *Disjunction*

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

- *Sequencing(short-circuit)*

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

- *Sequencing(normal)*

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

Proof Rules

- *Iterate zero*: $[p]C^\star[ok : p]$
- *Iterate non-zero*: $\dfrac{[p]C^\star; C[\epsilon : q]}{[p]C^\star; [\epsilon : q]}$
- *Backwards Variant*:

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

- *Choice*($i = 1$ *or* 2): $\dfrac{[p]C_i[\epsilon : q]}{[p]C_1 + C_2[\epsilon : q]}$
- *Error*: $[p]\texttt{error()}[ok : false][er : p]$
- *Assume*: $[p]\texttt{assume}\ B[ok : p \wedge B][er : false]$

Program Conversion

$$
\begin{aligned}
\texttt{while}\, B\, \texttt{do}\, C \quad &=_{def} \quad (\texttt{assume}(B);\ C)^\star; \texttt{assume}(\neg B) \\
\texttt{if}\, B\, \texttt{then}\, C\, \texttt{else}\, C' \quad &=_{def} \quad (\texttt{assume}(B);\ C) +\ (\texttt{assume}(\neg B);\ C') \\
\texttt{assert}(B) \quad &=_{def} \quad \texttt{assume}(B)\ +\ (\texttt{assume}(\neg B);\ \texttt{error}())
\end{aligned}
$$

Rules for Variables and Mutations

Assignment

$$[p]x = e[ok\colon \exists x'.p[x'/x] \land x = e[x'/x]][er\colon false]$$

Nondet Assignment

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

Constancy

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

Local Variable

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

Substitution I

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

Substitution II

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

where $Mod(C)$ is the set of variables that program C modified, $Free(r)$ means the set of free variables in assertion r and `nondet()` is a nondeterministic value.

Incorrectness Logic: Semantics

A set of *states* Σ is a set of function from the set *Variables* to the set *Value*.(When reason about termination *Value* is the set of natural numbers).

Definition (Post and Semantic Triples)

For any relation $r \subseteq \Sigma \times \Sigma$ and predicate $p, q \subseteq \Sigma$ define

- the *post-image* of r, $post(r) \in P(\Sigma) \to P(\Sigma)$:

$$post(r)p = \{\sigma' \mid \exists \sigma \in p.(\sigma, \sigma') \in r\}$$

- the *under-approximate triple*:
  $[p]r[q]$ is true iff $q \subseteq post(r)p$
- the *over-approximate triple*(Hoare's logic):
  $\{p\}r\{q\}$ is true iff $post(r)p \subseteq q$

Lemma (Characterization)

The following statements are equivalent:

- $[p]r[q]$ *is true.*
- $\forall \sigma_q \in q . \exists \sigma_p \in p . (\sigma_p, \sigma_q) \in r$.

The semantic of the triple $[p]C[\epsilon : q]$? Relations can be regarded as the semantic of a program. Here $\epsilon \in \{ok, er\}$, hence $[\![C]\!]ok$ and $[\![C]\!]er$ are two different relations related to program C.

Semantics

Generic Semantics for arbitrary state sets Σ

$$[\![C]\!]\epsilon \subseteq \Sigma \times \Sigma$$
$$[\![B]\!] : \Sigma \to Bool$$

$$\begin{array}{rclrcl}
[\![\mathtt{skip}]\!]ok & = & \{(\sigma,\sigma) \mid \sigma \in \Sigma\} & [\![\mathtt{skip}]\!]er & = & \emptyset \\
[\![\mathtt{error()}]\!]ok & = & \emptyset & [\![\mathtt{error()}]\!]er & = & \{(\sigma,\sigma) \mid \sigma \in \Sigma\} \\
[\![\mathtt{assume}\ B]\!]ok & = & \{(\sigma,\sigma) \mid [\![B]\!]\sigma = \mathtt{true}\} & [\![\mathtt{assume}\ B]\!]er & = & \emptyset \\
[\![C^\star]\!]\epsilon & = & \bigcup_{i \in Nat} [\![C^i]\!]\epsilon & [\![C_1 + C_2]\!]\epsilon & = & [\![C_1]\!]\epsilon \cup [\![C_2]\!]\epsilon \\
[\![C_1; C_2]\!]\epsilon & = & \multicolumn{4}{l}{\{(\sigma_1,\sigma_3) \mid \exists \sigma_2.(\sigma_1,\sigma_2) \in [\![C_1]\!]ok \text{ and } (\sigma_2,\sigma_3) \in [\![C_2]\!]\epsilon\}} \\
 & & \multicolumn{4}{l}{\cup \left(\mathit{if}\ (\epsilon = ok)\ \mathit{then}\ \emptyset\ \mathit{else}\ \{(\sigma_1,\sigma_2) \mid (\sigma_1,\sigma_2) \in [\![C_1]\!]er\}\right)}
\end{array}$$

Semantics of mutation and local variables

$$\Sigma = Variables \to Values$$
$$[\![e]\!] : \Sigma \to Values$$

$$\begin{array}{rclrcl}
[\![x = e]\!]ok & = & \{(\sigma, (\sigma \mid x \mapsto [\![e]\!]\sigma)) \mid \sigma \in \Sigma\} & [\![x = e]\!]er & = & \emptyset \\
[\![x = \mathtt{nondet()}]\!]ok & = & \{(\sigma, (\sigma \mid x \mapsto v)) \mid \sigma \in \Sigma,\ v \in Values\} & [\![x = \mathtt{nondet()}]\!]er & = & \emptyset \\
[\![\mathtt{local}\ x.C]\!]\epsilon & = & \multicolumn{4}{l}{\{((\sigma \mid x \mapsto v), (\sigma' \mid x \mapsto v)) \mid (\sigma,\sigma') \in [\![C]\!]\epsilon,\ v \in Values\}}
\end{array}$$

Theorem

All the listed properties in last section are true of the semantic.

Soundness and Completeness

Definition (Interpretation of Specifications)

$[p]C[\epsilon : q]$ is true iff the semantic triple $[p](\llbracket C \rrbracket \epsilon)[q]$ holds.

Theorem (Soundness)

The relational semantics last page validates all the rules introduced before: each axiom is true and each inference rule preserves truth.

Theorem (Completeness)

Every true triple involving finitely-supported predicates is provable.

where *finitely-supported* predicate means the predicate only related to finite number of variables in set *Variables*.

Predicate Transformers

Predicate transformers are fundamental semantic tool in program logic and analysis, which are defined as functions that map predicates to predicates.

- Forward Transformers. As is known $post(r)p$ can be given as the strongest predicate satisfying $\{p\}r\{q\}$. Symmetricly, we also have the weakest under-approximation to characterize that.

Definition

For $r \subseteq \Sigma \times \Sigma$:
$StrongestOverPost(r)p = \bigwedge\{q \mid \{p\}r\{q\} holds\}$
$WeakestUnderPost(r)p = \bigvee\{q \mid [p]r[q] holds\}$

Proposition

$StrongestOverPost(r)p = WeakestUnderPost(r)p = post(r)$

Forward Transformer

Iteration:

$$StrongestOverPost([\![C^*]\!]ok)p = \bigwedge\{I \mid p \Rightarrow I \wedge \{I\}C\{I\}\text{is true}\}$$

$$WeakestUnderPost([\![C^*]\!]ok)p = \bigvee_{i \in Nat} \{q \mid [p]C^i[\epsilon : q]\text{is true}\}$$

$$\underline{UnderPost}([\![C^*]\!]\epsilon)p = \bigvee_{i \leq bound} \{q \mid [p]C^i[\epsilon : q]\text{is true}\}$$

Nondeterministic choice:

$$post([\![C_1 + C_2]\!]\epsilon)p = post([\![C_1]\!]\epsilon)p \vee post([\![C_1]\!]\epsilon)p$$

$$\underline{post}([\![C_1 + C_2]\!]\epsilon)p = \underline{post}([\![C_1]\!]\epsilon)p\underline{\vee}\underline{post}([\![C_1]\!]\epsilon)p$$

where $p\underline{\vee}q \Rightarrow p \vee q$

Backward Transformers

Fact (Valid presumptions need not exist)

Given a relation r and assertion q, there needs not exist any p such that $[p]r[q]$ holds.

Strongest under-approximate presumptions do not exists in general.

Example

$C = (\texttt{assume x == 1; x = 88}) + (\texttt{assume x == 2; x = 88})$

Reasoning with the Logic

$$[p]foo()[ok : q][er : s] \vdash [p']C[ok : q'][er : s']$$

Reasoning Details

Reasoning about Loops

```
int x;

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: [err: x==2,000,000 ]   */
   { loop0();
     if (x==2,000,000) {error();}
   }
```

Use consequence rule and implication backward:

$$\frac{[\mathit{true}]\texttt{loop0()}[\mathit{ok} : x \geq 0], x \geq 0 \Leftarrow x{=}{=}2000000}{[\mathit{true}]\texttt{loop0()}[\mathit{ok} : x{=}{=}2000000]}$$

Unroll Loop

```
[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]
```

```
void loop1()
/*  achieves1: [ok: x==0 || x==1 || x==2 || x==3 ]
    achieves2: [ok: x>=0]   */
  { x = 0;
    Kleene-star{
      x = x+1;
  }  }

void client1()
/*  achieves: [er: x==2,000,000]  */
   { loop1();
     if (x==2,000,000) {error();}
   }
```

Conditionals, Expressiveness and Pruning

```
int x,y;

int difficult(int y)
 {  return (y*y); /* or, return hash(y) ... or, unknown code */
 }

 void client()
 /*achieves1: [ok: y==49 && x==1]
   achieves2: [ok: exists z. (y==difficult(z)&& x==1)||(y!=difficult(z)&& x==2)]
   achieves3: [ok: x==1 || x==2]   */
   { int z = nondet();
     if (y == difficult(z))
       {x=1;}
     else
       {x=2;}
   }

void test1()
/*achieves2: [ok: exists z. (y==difficult(z)&& x==1)||(y!=difficult(z)&& x==2)]*/
  { client();
     if (x==1 || x==2) { error(); }
  }

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

Conclusion

This paper,

- described how under-approximate triples are relevant to proving the presence of bugs,
- designed a specific logic along with a semantics and proof theory,
- explored reasoning techniques that are concerned with automatic program analysis.