Program analysis: from proving correctness to proving incorrectness

Roberto Bruni, Roberta Gori
(University of Pisa)
Lecture #06

BISS 2024
March 11-15, 2024
Addendum:
Abstract Interpretation as closure
Expressible elements

\[ c = \gamma(\alpha(c)) \]

\[ \exists a \in A . \ c = \gamma(a) \]

\[ \forall a . \ \alpha(\gamma(a)) = a \]

\[ \Rightarrow \]

\[ \forall a . \ \gamma(\alpha(\gamma(a))) = \gamma(a) \]

\[ \Rightarrow \]

\[ \forall a . \ \gamma(a) \text{ is expressible} \]

\[ \gamma(A) \text{ is the set of expressible elements} \]
Galois insertion as closures

The abstract domain can just be seen as a subset of the concrete domain

We write $A(c)$ as a shorthand for $\gamma(\alpha(c))$

$A(C)$ is the set of expressible elements

Since $A(A(c)) = A(c)$ the map $A : C \rightarrow C$ is a closure operator
Example

\[ \varnothing(\mathbb{Z}) \]

\[ \{-1,0,1,2\} \]

No need of symbolic representations
Example

\( \wp(\mathbb{Z}) \)

\{-1,0,1,2\}

No need of symbolic representations
Examples

\[
\text{Int}([2, 4, 6, \ldots]) = [2, 3, 4, 5, 6, \ldots] = [2, \infty]
\]

\[
\text{Sign}([2, 4, 6, \ldots]) = [1, 2, 3, 4, 5, 6, \ldots] = \mathbb{Z}_{>0}
\]

\[
\text{Int}([0, 2, 4, 6, \ldots]) = [0, 1, 2, 3, 4, 5, 6, \ldots] = [0, \infty]
\]

\[
\text{Sign}([0, 2, 4, 6, \ldots]) = [\ldots, -1, 0, 1, 2, 3, 4, 5, 6, \ldots] = \mathbb{Z}
\]

\[
\text{Sign}^+([0, 2, 4, 6, \ldots]) = [0, 1, 2, 3, 4, 5, 6, \ldots] = \mathbb{Z}_{\geq 0}
\]
Completeness, revisited

\[ \forall P. A([c]P) = [c]_A^#A(P) \subseteq \subseteq A([c]A(P)) \]

Completeness equation: \[ \forall P. A([c]P) = A([c]A(P)) \]
Recap: to be correct or incorrect?
Verification problem

$$\boxed{[c]P \subseteq Spec}$$
Over vs Under

$P \xrightarrow{} [c] P$
An Axiomatic Basis for Computer Programming

C. A. R. Hoare
The Queen's University of Belfast, Northern Ireland

In this paper an attempt is made to explore the logical foundations of computer programming by use of techniques which were first applied in the study of geometry and have later been extended to other branches of mathematics. This involves the elucidation of sets of axioms and rules of inference which can be used in proofs of the properties of computer programs. Examples are given of such axioms and rules, and
In this paper an attempt is made to explore the logical foundations of computer programming by use of techniques which were first applied in the study of geometry and have later been extended to other branches of mathematics. This involves the elucidation of sets of axioms and rules of inference which can be used in proofs of the properties of computer programs. Examples are given of such axioms and rules, and
In this paper an attempt is made to explore the logical foundations of computer programming by use of techniques which were first applied in the study of geometry and have later been extended to other branches of mathematics. This involves the elucidation of sets of axioms and rules of inference which can be used in proofs of the properties of computer programs. Examples are given of such axioms and rules, and
In this paper an attempt is made to explore the logical foundations of computer programming by use of techniques which were first applied in the study of geometry and have later been extended to other branches of mathematics. This involves the elucidation of sets of axioms and rules of inference which can be used in proofs of the properties of computer programs. Examples are given of such axioms and rules, and
An Axiomatic Basis for Computer Programming

C. A. R. Hoare
The Queen's University of Belfast, Northern Ireland

In this paper an attempt is made to explore the logical foundations of computer programming by use of techniques which were first applied in the study of geometry and have later been extended to other branches of mathematics. This involves the elucidation of sets of axioms and rules of inference which can be used in proofs of the properties of computer programs. Examples are given of such axioms and rules, and their use is illustrated.

\[
\{P\} \subset \{Q\} \\
\text{correctness} \checkmark \\
\text{incorrectness} \\
? \\
\]

Incorrectness Logic

PETER W. O'HEARN, Facebook and University College London, UK

Program correctness and incorrectness are two sides of the same coin. As a programmer, even if you would like to have correctness, you might find yourself spending most of your time reasoning about incorrectness. This includes informal reasoning that people do while looking at or thinking about their code, as well as that supported by automated testing and static analysis tools. This paper describes a simple logic for program incorrectness which is, in a sense, the other side of the coin to Hoare's logic of correctness.
In this paper an attempt is made to explore the logical foundations of computer programming by use of techniques which were first applied in the study of geometry and have later been extended to other branches of mathematics. This involves the elucidation of sets of axioms and rules of inference which can be used in proofs of the properties of computer programs. Examples are given of such axioms and rules, and

\[
\{P\} \subseteq \{Q\}
\]

\[
\begin{align*}
P & \rightarrow \ [c]P \\
[P] & \subseteq [Q]
\end{align*}
\]

**Incorrectness Logic**

PETER W. O’HEARN, Facebook and University College London, UK

Program correctness and incorrectness are two sides of the same coin. As a programmer, even if you would like to have correctness, you might find yourself spending most of your time reasoning about incorrectness. This includes informal reasoning that people do while looking at or thinking about their code, as well as that supported by automated testing and static analysis tools. This paper describes a simple logic for program incorrectness which is, in a sense, the other side of the coin to Hoare’s logic of correctness.
An Axiomatic Basis for Computer Programming

In this paper an attempt is made to explore the logical foundations of computer programming by use of techniques which were first applied in the study of geometry and have later been extended to other branches of mathematics. This involves the elucidation of sets of axioms and rules of inference which can be used in proofs of the properties of computer programs. Examples are given of such axioms and rules, and

\[ \{P\} c \{Q\} \]

\[ \{P\} c \{Q\} \]

\[ [P] c [Q] \]

\[ [c]P \]

\[ \text{correctness} \]

\[ \text{incorrectness} \]

**Incorrectness Logic**

PETER W. O’HEARN, Facebook and University College London, UK

Program correctness and incorrectness are two sides of the same coin. As a programmer, even if you would like to have correctness, you might find yourself spending most of your time reasoning about incorrectness. This includes informal reasoning that people do while looking at or thinking about their code, as well as that supported by automated testing and static analysis tools. This paper describes a simple logic for program incorrectness which is, in a sense, the other side of the coin to Hoare’s logic of correctness.
In this paper an attempt is made to explore the logical foundations of computer programming by use of techniques which were first applied in the study of geometry and have later been extended to other branches of mathematics. This involves the elucidation of sets of axioms and rules of inference which can be used in proofs of the properties of computer programs. Examples are given of such axioms and rules, and

\{ P \} \subset \{ Q \}

\{ P \} \subset [c] P

\{ [P] \subset [c] Q \}

**Incorrectness Logic**

PETER W. O’HEARN, Facebook and University College London, UK

Program correctness and incorrectness are two sides of the same coin. As a programmer, even if you would like to have correctness, you might find yourself spending most of your time reasoning about incorrectness. This includes informal reasoning that people do while looking at or thinking about their code, as well as that supported by automated testing and static analysis tools. This paper describes a simple logic for program incorrectness which is, in a sense, the other side of the coin to Hoare’s logic of correctness.

1. This includes informal reasoning that people do while looking at or thinking about their code, as well as that supported by automated testing and static analysis tools.
2. Incorrectness logic adds post-assertions for errors as well as for normal termination, and
3. upon termination when the code is executed starting from states satisfying the pre-condition (the

For example, consider the following scenario:

\( \{ P \} \subset \{ Q \} \)

\( [P] \subset [c] Q \)

\( \{ c \} P \)

**Corrections:**

1. This includes informal reasoning that people do while looking at or thinking about their code, as well as that supported by automated testing and static analysis tools.
2. Incorrectness logic adds post-assertions for errors as well as for normal termination, and
3. upon termination when the code is executed starting from states satisfying the pre-condition (the

For example, consider the following scenario:

\( \{ P \} \subset \{ Q \} \)

\( [P] \subset [c] Q \)

\( \{ c \} P \)

**Corrections:**

1. This includes informal reasoning that people do while looking at or thinking about their code, as well as that supported by automated testing and static analysis tools.
2. Incorrectness logic adds post-assertions for errors as well as for normal termination, and
3. upon termination when the code is executed starting from states satisfying the pre-condition (the
The idea

\[ Q \subseteq \llbracket c \rrbracket P \]

\[ P \to [c] P \]

\[ [P] c [Q] \]

\[ Q \]

\[ \cup \]

\[ \times \]
The idea

\[ Q \subseteq \llbracket c \rrbracket P \]
The idea

\[ Q \subseteq \llbracket c \rrbracket P \]
\[ \llbracket c \rrbracket P \subseteq A(Q) \]
The idea

\[ Q \subseteq [c]P \]
\[ [c]P \subseteq A(Q) \]
\[ Q \subseteq \llbracket c \rrbracket P \]
\[ \llbracket c \rrbracket P \subseteq A(Q) \]
\[ A(Spec) = Spec \]

The idea

\[ \vdash_A [P] c [Q] \]
The idea

Local completeness
The idea

$Q \subseteq [[c]P$

$[[c]P \subseteq A(Q)$

$A(\text{Spec}) = \text{Spec}$

$P \xrightarrow{A} [c]P$

$Q \xrightarrow{\cup I} [[c]P$

$\vdash_A [P] c [Q]$

Local completeness

$A(Q) \subseteq \text{Spec}$

$\iff$

$[[c]P \subseteq \text{Spec}$

$\iff$

$Q \subseteq \text{Spec}$
The idea

\[ Q \subseteq [[c]]P \]
\[ [[c]]P \subseteq A(Q) \]
\[ A(Spec) = Spec \]

\[ A(Q) \subseteq Spec \]
\[ \iff \]
\[ [[c]]P \subseteq Spec \]
\[ \iff \]
\[ Q \subseteq Spec \]
A Logic for Locally Complete Abstract Interpretations

Roberto Bruni1, Roberto Giacobazzi1, Roberto Gorri2, Francesco Ranise3

1University of Pisa, Italy
2University of Verona, Italy
3University of Palermo, Italy

In memory of Anna Maria De Paola and Anna Gorri

Abstract—We introduce the notion of local completeness in abstract interpretations and follow a two-step process for proving that a locally complete abstract interpretation is adequate. Local completeness is a semantic tool to design sound-by-construction program analyses. It coincides with incorrectness logic.

Locally complete under-approximations of program behaviors can be used to provide a strict check of the soundness of an abstract interpretation. In contrast, a locally complete over-approximation of program behaviors coincides with incorrectness logic (IL). In particular, given a program specification Spec and a program P, an IL analysis turns out to be sound when, given any input, the analysis may not return any false alarm. This makes IL a credible support for code review.

Locally complete under-approximations are sound for designing sound-by-construction over-approximations of the program behavior. Given an abstraction A, instead of verifying whether the strongest post-condition post(A) satisifies Spec(A), one can check if post(A[P]) satisfies Spec, which is a decision problem.

Locally complete over-approximations of the program behavior are sound-by-construction. Given a program specification Spec and an abstract interpretation, any violation exposed by the analysis is a true alarm. This makes it a valuable tool for code review.

Introduction

Technology you can’t live without. Any coin has two sides, and software failures are increasingly more frequent and their consequences are more dire. In this section, you describe a framework for writing reliable software.

Locally complete under-approximations are sound for designing sound-by-construction over-approximations of the program behavior. Given an abstraction A, instead of verifying whether the strongest post-condition post(A) satisfais Spec(A), one can check if post(A[P]) satisfies Spec, which is a decision problem.

Locally complete over-approximations of the program behavior are sound-by-construction. Given a program specification Spec and an abstract interpretation, any violation exposed by the analysis is a true alarm. This makes it a valuable tool for code review.

Technically, in a domain $D$ of abstract program states, with abstraction and concretization maps $c$ and $r$, any state property $P$ is, in general, over-approximated by $(r\circ c)(P)$ and under-approximated by $(c\circ r)(P)$. Assuming that $Spec$ is expressible in $D$ means that $Spec = (c\circ r)(Spec)$ holds. For instance, in the abstract domain of intervals let $Int$ be the set of intervals, and let $p(x)$ be an expression in $Int$. If $p(x)$ is satisfiable in $D$, the property $\exists x \in D$ such that $(r\circ c)(p(x)) = \text{True}$ holds.

Sound-by-construction analysis has been successfully used in software engineering and is a major tool in the design of abstract interpreter frameworks and software engineering in producing reliable code. In particular, static analysis is based on symbolic reasoning techniques to prove program properties without running them. Given a program $P$ and a syntax-directed abstract interpreter, the property $\exists x \in D$ such that $(r\circ c)(p(x)) = \text{True}$ holds.

any locally complete under approximation either proves the program correct or incorrect (without false positives)
Local completeness
Expressible specifications

Assume $A(\text{Spec}) = \text{Spec}$

Take a post $Q \in C$

If $Q \not\subseteq \text{Spec}$
then $Q \subseteq A(Q) \not\subseteq \text{Spec}$

If $Q \subseteq \text{Spec}$
then $A(Q) \subseteq A(\text{Spec}) = \text{Spec}$

$Q \subseteq \text{Spec} \iff A(Q) \subseteq \text{Spec}$
Example

\[ \text{Int}(x \geq 0) = [0, \infty] = (x \geq 0) \]

If \( Q_1 \doteq (|x| = 1) \)
then \( \text{Int}(Q_1) = [-1, 1] \not\subseteq (x \geq 0) \)

If \( Q_2 \doteq (x > 0 \land x \% 5 = 0) \)
then \( \text{Int}(Q_2) = [5, \infty] \subseteq (x \geq 0) \)

\( Q \subseteq \text{Spec} \iff A(Q) \subseteq \text{Spec} \)
The role of completeness

correctness of AI
if $[[c]]_A^#A(P) \subseteq Spec$
then $[[c]]P \subseteq Spec$

Expressible specification
$Spec = A(Spec)$

if completeness holds
if $[[c]]_A^#A(P) \not\subseteq Spec$
then $[[c]]P \not\subseteq Spec$

$A([[c]]P) = [[c]]_A^#A(P) \not\subseteq Spec$
$\iff [[c]]P \not\subseteq Spec$

$[[c]]P \subseteq Spec \iff [[c]]_A^#A(P) \subseteq Spec$
Example

+, × complete in \( \text{Int} \)

\[
c \triangleq x := 3x \; ; \; x := x + 2
\]

\[
P \triangleq (x \in \{1,3,6\})
\]

\[
\llbracket c \rrbracket_{\text{Int}}^\# \text{Int}(P) = \llbracket c \rrbracket_{\text{Int}}^\# [1,6] = [5,20] \not\subseteq (x \leq 15)
\]

\[
\iff \llbracket c \rrbracket P \not\subseteq (x \leq 15)
\]

However, not all elements in

\[
[16,20]
\]

are true positives!
Sources of incompleteness

Completeness is preserved by ; , U and fix

Incompleteness can only be introduced by atomic commands \( e \)

assignments: settled for many domains

guards: troublesome

if the bca \( [e]^A \) is incomplete, then any (sound) \( [e]^*_A \) is incomplete

i.e., incompleteness is an intrinsic property of a domain
Completeness for guards

Completeness equation: \( \forall P . A([e]P) = A([e]A(P)) \)

Lemma. [a necessary condition for complete guards]
If a test \( b \) is complete in \( A \), then \( b \) and \( \neg b \) are expressible in \( A \)

Proof. Assume \( b \) not expressible, take \( P = b \) and show \( \neg b \) is not complete.
Examples

Int: the test \((x = 0)\) is not complete \((x \neq 0\) not expressible)  

Int: the test \((x > 5)\) might be complete (but it is not)  

Sign: the test \((x > 5)\) is not complete  

\[\begin{array}{c}
\text{Sign} \\
Z \\
Z_{<0} \\
\emptyset \\
\end{array}\]

Sign\(^+\): the test \((x > 0)\) might be complete (and it is indeed)
Completeness for guards

Th. [a necessary and sufficient condition for complete guards]
Let \( b \) and \( \neg b \) be expressible in \( A \).

The test \( b \) is complete in \( A \)
iff

the join of any two abstract points below \( b \) and \( \neg b \) is expressible.
Completeness illustrated

Does $a_1 \cup a_2 \in A$?
Example

\[ \omega(\mathbb{Z}) \]

\[ \text{Int}\ \{0,1,10,11\} \notin \text{Int} \]
Incompleteness everywhere

Unfortunately, common tests are incomplete in most domains

One possibility:

- take the most abstract domain $A_b$ (called complete shell) that:
  - refines $A$, and
  - is complete for the test $b$

ok, but:

- may cause a blow up (abstract domains are closed under meet)
  - operations that where complete in $A$ may be incomplete in $A_b$
Local completeness

We don’t need completeness for all inputs: e.g., \( b \triangleq (x > 0) \) is complete in Int for \( P \triangleq \{-10,0,1,10\} \)

Local completeness equation: \( \forall P. A([e]P) = A([e]A(P)) \)

We say that \( e \) is locally complete in \( A \) for input \( P \) and write \( C_A^P(e) \)

Idea: we focus on completeness along traversed states (which can be collected as underapproximation)
Local Completeness Logic (LCL)
O’Hearn’s triples

\[ [P] c [Q] \]

any output matching the postcondition can be reached by executing the command on some input matching the precondition

\[ [c]P \supseteq Q \]

under approximation!

includes just reachable states

LCL triples

\[ \vdash_A [P] c [Q] \]

any output matching the postcondition can be reached by executing the command on some input matching the precondition

\[ A(Q) \supseteq [c]P \supseteq Q \]

over approximation!

under approximation!

includes just reachable states

includes just reachable states
Atomic commands

⊢ $A[[e]P] = A([[e]]A(P))$

\[\frac{\mathbb{C}_A^e}{A[P] e [[e]]P} \quad \text{[transfer]}\]

\[\frac{\mathbb{C}_A^a}{A[P] x := a \exists x'. P[x'/x] \land x = a[x'/x]} \quad \text{Floyd's rule for assignments}\]

\[\frac{\mathbb{C}_A^b}{A[P] b? [P \land b]} \quad \text{IL rule for tests}\]
Atomic commands

\[ \vdash_A \exists x'. P[x'/x] \land x = a[x'/x] \]

Floyd's rule for assignments

\[ \vdash_{\text{Int}} \exists x. x \in \{-7, 7\} \implies x := 3x + 1 [x \in \{-20, 22\}] \]
Atomic commands

\[ \frac{C_p^A(b)}{\vdash_A [P] b \quad [P \land b]} \]

IL rule for tests

\[ \vdash_{\text{Int}} \left[ x \in \{-7,7\} \right] x > 0 \quad \left[ x \in \{7\} \right] ? \]

\[
\begin{align*}
\text{Int}(\llbracket x > 0 \rrbracket \{-7,7\}) &= \text{Int}(\{7\}) = [7,7] \\
&\neq [1,7] = \text{Int}(\{1,7\}) = \text{Int}(\llbracket x > 0 \rrbracket [-7,7]) = \text{Int}(\llbracket x > 0 \rrbracket \text{Int}(\{-7,7\}))
\end{align*}
\]

\[
\begin{align*}
\text{Int}(\llbracket x > 0 \rrbracket \{-7,1,7\}) &= \text{Int}(\{1,7\}) = [1,7] = \text{Int}(\{1,7\}) = \text{Int}(\llbracket x > 0 \rrbracket [-7,7]) = \text{Int}(\llbracket x > 0 \rrbracket \text{Int}(\{-7,1,7\}))
\end{align*}
\]

\[ \vdash_{\text{Int}} \left[ x \in \{-7,1,7\} \right] x > 0 \quad \left[ x \in \{1,7\} \right] ? \]

locally incomplete in Int !
Consequence rule

\[ P' \Rightarrow P \Rightarrow A(P') \quad \vdash_A [P] \ r [Q] \quad Q \Rightarrow Q' \Rightarrow A(Q) \]

we can weaken the pre and shrink the post, but not too much!
scalable bug detection
Lemma. [convexity]
If $C_P^A(e)$ and $P \Rightarrow R \Rightarrow A(P)$ then $C_R^A(e)$

Proof.
Assume $A([e]P) = A([e]A(P))$
we want to prove $A([e]R) = A([e]A(R))$

$$A([e]P) \leq A([e]R) \leq A([e]A(R)) = A([e]A(P)) = A([e]P)$$
Consequence rule

\[
P' \Rightarrow P \Rightarrow A(P') \quad \vdash_A [P] \ast [Q] \quad Q \Rightarrow Q' \Rightarrow A(Q)
\]

\[
\vdash_A [P] \ast [Q]
\]

IL style (reversed Hoare)

preserve abstraction

preserve abstraction

[relax]
Consequence rule

\[ P' \Rightarrow P \Rightarrow A(P') \quad \vdash_A [P'] r [Q'] \quad Q \Rightarrow Q' \Rightarrow A(Q) \]

\[ \vdash_A [P] r [Q] \]

IL style (reversed Hoare)

\[ A(P') = A(P) \quad A(Q') \]

preserve abstraction

preserve abstraction

[relax]
Consequence rule

$P' \Rightarrow P \Rightarrow A(P') \quad \vdash_A [P'] \; r \; [Q'] \quad Q \Rightarrow Q' \Rightarrow A(Q) \quad \vdash_A [P] \; r \; [Q]

[relax]

**IL style** (reversed Hoare)

Preserve abstraction

$A(P') = A(P)$

$A(Q) = A(Q')$
Consequence rule

\[ P' \Rightarrow P \Rightarrow A(P') \quad \vdash_A [P'] r [Q'] \quad Q \Rightarrow Q' \Rightarrow A(Q) \]

\[ \vdash_A [P] r [Q] \]

\[ A(P') = A(P) \quad A(Q) = A(Q') \]

IL style (reversed Hoare)

preserve abstraction

preserve abstraction
Consequence rule

\[ P' \Rightarrow P \Rightarrow A(P') \quad \vdash_A [P'] \quad r \quad [Q'] \quad Q \Rightarrow Q' \Rightarrow A(Q) \]

\[ \vdash_A [P] \quad r \quad [Q] \quad [\text{relax}] \]

IL style (reversed Hoare)

\[ \vdash_{\text{Int}} [x \in \{-7,0,7\}] \quad r \quad [x \in \{-5,-2,8\}] \]

\[ \vdash_{\text{Int}} [x \in \{-7,0,3,7\}] \quad r \quad [x \in \{-5,8\}] \]
Consequence rule

\[ P' \Rightarrow P \Rightarrow A(P') \quad \vdash_A [P'] r [Q'] \quad Q \Rightarrow Q' \Rightarrow A(Q) \]

\[ \vdash_A [P] r [Q] \]

[relax]

IL style (reversed Hoare)

\[ \vdash_{\text{Int}} [x \in \{-7,0,7\}] r [x \in \{-5,-2,8\}] \]

\[ \vdash_{\text{Int}} [x \in \{-7,0,7,9\}] r [x \in \{-2,8\}] \]
Fixpoint acceleration

\[ \vdash_A [P] \quad r [R] \quad \vdash_A [P \lor R] \quad r^* [Q] \]

[rec] \[ \vdash_A [P] \quad r^* [Q] \]

[iterate] \[ \vdash_A [P] \quad r^* [P \lor Q] \]

locally complete under-approximation!
scalable bug detection

\[ \vdash \text{Sign}^+ [P] \quad (x \leq 0?; x := x \ast 10)^* \quad [\{-100, -10, -1, 100\}] \]

\[ P \triangleq \{-10, -1, 100\} \]
Sequential composition

\[
\begin{align*}
\vdash_A [P] r_1 [R] &\quad \vdash_A [R] r_2 [Q] \\
\frac{}{\vdash_A [P] r_1 ; r_2 [Q]}\quad \text{[seq]}
\end{align*}
\]

\[
\vdash_{\text{Int}} \begin{array}{l}
[\text{true}] \quad x < 0 \?
\end{array} ;
\begin{array}{l}
x := -x \quad [x \in [1, \infty]]
\end{array}
\]
Choice

\[ \frac{\vdash_A [P] r_1 [Q_1] \quad \vdash_A [P] r_2 [Q_2]}{\vdash_A [P] r_1 + r_2 [Q_1 \lor Q_2]} \] [join]

\[ \vdash_{\text{Int}} [\text{true}] \text{ if } x < 0 \text{ then } x := -x \text{ else skip } [x \in [0, \infty]] \]
Validity, soundness, completeness
The rules of LCL

$$\begin{align*}
\text{transfer:} & \quad \vdash_A [P] \; e \; [[e]]P \\
\text{seq:} & \quad \vdash_A [P] \; r_1 \; [R] \quad \vdash_A [R] \; r_2 \; [Q] \\
& \quad \vdash_A [P] \; r_1; r_2 \; [Q] \\
\text{rec:} & \quad \vdash_A [P] \; r \; [R] \quad \vdash_A [P \lor R] \; qr \; [Q] \\
& \quad \vdash_A [P] \; qr \; [Q] \\
\text{relax:} & \quad \vdash_A [P'] \leq P \leq A(P') \\
& \quad \vdash_A [P'] \; r \; [Q'] \quad Q \leq Q' \leq A(Q) \\
\text{join:} & \quad \vdash_A [P] \; r_1 \; [Q_1] \quad \vdash_A [P] \; r_2 \; [Q_2] \\
& \quad \vdash_A [P] \; r_1 \oplus r_2 \; [Q_1 \lor Q_2] \\
\text{iterate:} & \quad \vdash_A [P] \; r \; [Q] \quad Q \leq A(P) \\
& \quad \vdash_A [P] \; r \; [Q \lor P] \\
\text{invariant:} & \quad \vdash_A [P] \; r \; [Q] \quad Q \leq P \\
& \quad \vdash_A [P] \; qr \; [P] \\
\text{abs-fix:} & \quad \vdash_A [P] \; r \; [Q] \quad A(P) = A(Q) \\
& \quad \vdash_A [P] \; qr \; [Q] \\
\text{limit:} & \quad \forall n \in \mathbb{N}. \; \vdash_A [P_n] \; r \; [P_{n+1}] \\
& \quad \vdash_A [P_0] \; r \; [\bigvee_{i \in \mathbb{N}} P_i]
\end{align*}$$
Validity

A LCL triple $\vdash_A [P] r [Q]$ is **valid** if $Q \subseteq \llbracket r \rrbracket P \subseteq A(Q)$

Is $\vdash_{\text{Int}} [x > 0] x := 10x [x \geq 10]$ valid? $\times$

Is $\vdash_{\text{Int}} [x > 0, y \geq 0] x := yx [x \geq 0]$ valid? $\times$

Is $\vdash_{\text{Sign}} [x > 0, y > 0] x := yx [x = 42, y = 7]$ valid? $\checkmark$

Is $\vdash_{\text{Sign}} [x > 0] (x := x + 1)^* [x > 0]$ valid? $\checkmark$
Logical correctness

Th.
If $\vdash_A [P] r [Q]$ then $Q \subseteq \llbracket r \rrbracket P \subseteq A(Q) = \llbracket r \rrbracket_A^\# A(P)$

Proof.
By induction on the derivation.
Verification

Th.
If $A(\text{Spec}) = \text{Spec}$, then any provable triple $\vdash_A [P] \ r \ [Q]$ either shows the program correct ($Q \subseteq \text{Spec}$) or exposes some true positives ($Q \setminus \text{Spec} \neq \emptyset$)

Proof.
$[[r]]P \subseteq \text{Spec} \iff A[[r]]P \subseteq \text{Spec} \iff A(Q) \subseteq \text{Spec} \iff Q \subseteq \text{Spec}$
Verification

Th.
If $A(Spec) = Spec$, then any provable triple $\vdash_A [P] r [Q]$ either shows the program correct ($Q \subseteq Spec$) or exposes some true positives ($Q \setminus Spec \neq \emptyset$)

Proof.
$[[r]]P \subseteq Spec \iff A[[r]]P \subseteq Spec$
$\iff A(Q) \subseteq Spec$
$\iff Q \subseteq Spec$
Logical completeness

Th.
If $A$ is complete for any atomic command in $r$, then any valid triple $\vdash_A [P] r [Q]$ can be derived

Proof.
We first derive $\vdash_A [P] r [[[r]]P]$, then use [relax] with $Q \subseteq [[r]]P$
Intrinsic logical incompleteness

Th.
For any Turing complete language and any non-trivial abstraction $A$, there are valid triples that cannot be proved.

Proof.
See full version of LICS 2001 paper
IL as LCL
A IL triple \([P] r [Q]\) is **valid** if \(Q \subseteq [[r]]P\)

**Th.** Any valid IL triple can be derived

A LCL triple \(\Gamma_A [P] r [Q]\) is **valid** if \(Q \subseteq [[r]]P \subseteq A(Q)\)

**Th.** If \(A\) is complete for any atomic command in \(r\), then any valid triple \(\Gamma_A [P] r [Q]\) can be derived

**Th.** For any non-trivial abstraction \(A\), there are valid LCL triples that cannot be proved

\([r]P \subseteq A(Q)\) must hold for any \(Q \subseteq [[r]]P\)

\(A \subseteq \{ \top \}\)

\(A\) must be complete

\(A\) must be trivial

\(A = \{ \top \}\)
Consequences

\[ A = \{ \top \} \]

<table>
<thead>
<tr>
<th>Rule</th>
<th>Expression</th>
</tr>
</thead>
<tbody>
<tr>
<td>transfer</td>
<td>[ \mathcal{C}_P^A(e) \vdash_A [P] e [\llbracket e \rrbracket P] ]</td>
</tr>
<tr>
<td>trivial</td>
<td>[ P' \leq P \leq A(P') \vdash_A [P'] r [Q'] \quad Q \leq Q' \leq A(Q) ]</td>
</tr>
<tr>
<td>seq</td>
<td>[ \vdash_A [P] r_1 [R] \quad \vdash_A [R] r_2 [Q] \quad \vdash_A [P] r_1; r_2 [Q] ]</td>
</tr>
<tr>
<td>trivial</td>
<td>[ \vdash_A [P] r_1 [Q_1] \quad \vdash_A [P] r_2 [Q_2] \quad \vdash_A [P] r_1 \oplus r_2 [Q_1 \lor Q_2] ]</td>
</tr>
<tr>
<td>rec</td>
<td>[ \vdash_A [P] r [R] \quad \vdash_A [P \lor R] r^* [Q] \quad \vdash_A [P] r^* [Q] ]</td>
</tr>
<tr>
<td>trivial</td>
<td>[ \vdash_A [P] r [Q] \quad Q \leq A(P) \quad \vdash_A [P] r^* [P \lor Q] ]</td>
</tr>
<tr>
<td>iterate</td>
<td>[ \vdash_A [P] r [Q] \quad Q \leq A(P) \quad \vdash_A [P] r^* [P \lor Q] ]</td>
</tr>
</tbody>
</table>
Consequences

\[ A = \{ \top \} \]

\[
\frac{\vdash_A [P] \ \text{e} \ [[e]P]}{\vdash_A [P] \ e \ [[e]P]} \quad \text{(transfer)}
\]

\[
\frac{P' \leq P}{\vdash_A [P'] \ r \ [Q'] \ \ Q \leq Q'} \quad \text{(relax)}
\]

\[
\frac{\vdash_A [P] \ r_1 \ [R] \ \vdash_A [R] \ r_2 \ [Q]}{\vdash_A [P] \ r_1; r_2 \ [Q]} \quad \text{(seq)}
\]

\[
\frac{\vdash_A [P] \ r \ [R] \ \vdash_A [P \lor R] \ r^* \ [Q]}{\vdash_A [P] \ r^* \ [Q]} \quad \text{(rec)}
\]

\[
\frac{\vdash_A [P] \ r_1 \ [Q_1] \ \vdash_A [P] \ r_2 \ [Q_2]}{\vdash_A [P] \ r_1 \oplus r_2 \ [Q_1 \lor Q_2]} \quad \text{(join)}
\]

\[
\frac{\vdash_A [P] \ r \ [Q]}{\vdash_A [P] \ r^* \ [P \lor Q]} \quad \text{(iterate)}
\]
How to handle \texttt{ok} and \texttt{er}

\[ A = \{ \top \} \]
\[ C \triangleq \wp(\{\texttt{ok,er}\} \times \Sigma) \]
\[ e : Q \text{ shorthand for } \{ e : \sigma \mid \sigma \in Q \} = \{ e \} \times Q \]

\[ \llbracket \text{skip} \rrbracket(\texttt{ok} : Q \cup \texttt{er} : R) \triangleq \texttt{ok} : Q \cup \texttt{er} : R \]
\[ \llbracket x := a \rrbracket(\texttt{ok} : Q \cup \texttt{er} : R) \triangleq \texttt{ok} : \{ \sigma[x \mapsto \llbracket a \rrbracket \sigma] \mid \sigma \in Q \} \cup \texttt{er} : R \]
\[ \llbracket \text{error}() \rrbracket(\texttt{ok} : Q \cup \texttt{er} : R) \triangleq \texttt{er} : Q \cup R \]
\[ \llbracket b? \rrbracket(\texttt{ok} : Q \cup \texttt{er} : R) \triangleq \texttt{ok} : (Q \land b) \cup \texttt{er} : R \]
\[ \llbracket x := \text{nondet}() \rrbracket(\texttt{ok} : Q \cup \texttt{er} : R) \triangleq \texttt{ok} : \{ \sigma[x \mapsto v] \mid \sigma \in Q, v \in \mathbb{Z} \} \cup \texttt{er} : R \]

\textbf{Lemma}

\[ \llbracket r \rrbracket(\texttt{ok} : Q \cup \texttt{er} : R) = \texttt{er} : R \cup \bigcup_{\sigma \in Q} \llbracket r \rrbracket(\texttt{ok} : \sigma) \]

\textit{error preserving semantics}
IL as LCL

Lemma.
\[ \llbracket r \rrbracket (\text{ok} : P) = \text{ok} : \llbracket r \rrbracket \text{ok}(P) \cup \text{er} : \llbracket r \rrbracket \text{er}(P) \]

Corollary. [IL as an instance of LCL]
\[ [P] \ r \ [\text{ok} : Q][\text{er} : R] \text{ in IL} \iff \vdash_{\{\top\}} [\text{ok} : P] \ r \ [\text{ok} : Q \cup \text{er} : R] \]
Questions
Question 1

Which LCL triples are valid for any $r$ and $P$?

\[ \vdash_{\{T\}} [P] \ r \ [false] \quad \checkmark \]

\[ \vdash_{\{T\}} [P] \ r \ [true] \quad \times \]

\[ \vdash_{\text{Sign}} [x > 10] \ r \ [false] \quad \times \]

\[ \vdash_{\{T\}} [wlp(r, P)] \ r \ [P] \quad \times \]
Question 2

Find a derivation for the IL triple

\[ \vdash_{\text{Oct}} [x < 10, y > 20] \text{ if } x \geq y \text{ then } z := x \text{ else } z := y [x < 10, y > 20, z = \max(x, y)] \]

\[ [x < 10, y > 20] \]
if \( x \geq y \) then
  \[ [\text{false}] \]
  \( z := x \)
  \[ [\text{false}] \]
else
  \[ [x < 10, y > 20] \]
  \( z := y \)
  \[ [x < 10, z = y > 20] \equiv [x < 10, y > 20, z = \max(x, y)] \]
\[ [x < 10, y > 20, z = \max(x, y)] \]
Question 3

Are these “mixed” HL+LCL inference rules valid?

\[
\begin{align*}
\vdash_A \{P\} \Rightarrow \{A(Q)\} \\
\vdash_A \{A(P)\} \Rightarrow \{A(Q)\}
\end{align*}
\]

If \(\vdash_A \{P\} \Rightarrow \{Q\}\) then \([r]P \subseteq A(Q)\), hence \(\{P\} \Rightarrow \{A(Q)\}\) is valid.

If \(\vdash_A \{P\} \Rightarrow \{Q\}\) then \([r]A(P) \subseteq [r]^{\#}A(P) = A(Q)\), hence \(\{A(P)\} \Rightarrow \{A(Q)\}\) is valid.
Prove that \([\text{conj}]\) is \textit{unsound} for LCL

\[
\vdash A[P_1] \leftrightarrow [Q_1] \quad \vdash A[P_2] \leftrightarrow [Q_2] \\
\therefore \vdash A[P_1 \land P_2] \leftrightarrow [Q_1 \land Q_2] \quad \text{[conj]}
\]
Show that the following rule is not sound

\[ \vdash_A [P] x := \text{nondet()} [P[v/x]] \]