Winter Systems School 2023

Logic, Hoare Logic, Weakest Precondition, Invariant Inference, Equivalence Checking, Superoptimization, Reflections on Trusting Trust

Sorav Bansal

sbansal@iitd.ac.in

December 4th-8th, 2023

Deductive logic

- Systematic evaluation of arguments Let us say we already know that (*premises*)

  - All WSS23 attendees are motivated to learn more
  - Mr. X is attending WSS23

  Putting these thoughts together, you can infer (*inference step*)

  - Mr. X is motivated to learn more

  You can also infer

  - If Ms. Y is attending WSS23, she must be motivated to learn more

  But you cannot infer

  - If Ms. Y is motivated to learn more, she must be attending WSS23

Generalization

- All WSS23 attendees are motivated to learn more
- Mr. X is attending WSS23
- So, Mr. X is motivated to learn more

- All F are G.
- n is F.
- So, n is G.

Generalization : Another example

- No three-year old understands quantum mechanics.
- Z is a three-year old.
- So, Z does not understand quantum mechanics.

- No F is G.
- n is F.
- So, n is not G.

More deduction examples

- No F is G.
- So, no G is F.

- All F are H.
- No G is H.
- So, no F is G.

- All F are either G or H.
- All G are K.
- All H are K.
- So, all F are K.

Some deductions can be derived from other deductions

- Is the following inference valid?
    - Everyone loves a lover.
    - Romeo loves Juliet.
    - So, everyone loves Juliet.

Some deductions can be derived from other deductions

- Is the following inference valid?
  - Everyone loves a lover.
  - Romeo loves Juliet.
  - So, everyone loves Juliet.

- Proof (sequence of *one-step* inferences)

(1)	Everyone loves a lover	(premiss)
(2)	Romeo loves Juliet	(premiss)
(3)	Romeo is a lover	(from 2)
(4)	Everyone loves Romeo	(from 1,3)
(5)	Juliet loves Romeo	(from 4)
(6)	Juliet is a lover	(from 5)
(7)	Everyone loves Juliet	(from 1,6)

A counterexample disproves an inference

- Is the following inference valid?
    1. All philosophers are logicians
    2. So, all logicians are philosophers?

- Counterexamples (states of the world that satisfy 1 but do not satisfy 2)
    - No philosopher and one logician who is not a philosopher.
    - One philosopher (who is also a logician) and two logicians, neither of whom is a philosopher.
    - ...

Introducing notation : connectives *and*, *or*

- The following are examples of valid inferences

1. Either A or B.
2. Not A.
3. So, B.

1. A and B.
2. So, A.

1. A.
2. B.
3. So, A and B.

1. A.
2. So, either A or B.

Propositional logic

- Consider atomic, or indecomposable, declarative sentences, e.g., "Apple is a fruit"
- We assign distinct symbols to these atomic sentences: p,q,r,...
- Code up complex sentences in a compositional way, using symbols $\neg$ (not), $\wedge$ (and), $\vee$ (or), and $\rightarrow$ (if-then/implies).

Natural deduction

- Calculus for reasoning about propositions.
- Proof rules that can allow us to infer formulas from other formulas.
- $\phi_1, \phi_2, \ldots, \phi_n \vdash \psi$
  (e.g., $(a \land \neg b) \to c, \neg c, a \vdash b$)
- This inference is valid if a proof can be found for it using the proof rules (derivation).
- The proof rules should allow valid arguments and disallow invalid ones.

Example of Natural deduction

- Proof rules

$p \to q \vdash \neg q \to \neg p$	[Contra-positive (CP)]
$p, p \to q \vdash q$	[Modus ponens (MP)]
$\neg(p \wedge q) \vdash \neg p \vee \neq q$	[Neg-over-and (NOA)]
$\neg\neg p \vdash p$	[Double negation elimination (DNE)]
$p \vdash \neg\neg p$	[Double negation introduction (DNI)]
$p \vee q, \neg p \vdash q$	[Or elimination (OE)]

- Example: $(a \wedge \neg b) \to c, \neg c, a \vdash b$

(1) $(a \wedge \neg b) \to c$	(premiss)
(2) $\neg c$	(premiss)
(3) a	(premiss)
(4) $\neg c \to \neg(a \wedge \neg b)$	(CP on 1)
(5) $\neg(a \wedge \neg b)$	(MP on 2,4)
(6) $\neg a \vee \neg\neg b$	(NOA on 5)
(7) $\neg\neg a$	(DNI on 3)
(8) $\neg\neg b$	(OE on 6,7)
(9) b	(DNE on 8)

Logic = formal language + axioms + proof system

- A formal system of logic consists of a formal language together with a set of axioms and a proof system used to draw inferences from these axioms.

- Examples:
  - Propositional logic
  - Intuitionistic propositional logic (proof by contradition is not allowed)
  - First-order logic — allows quantifiers *for all* $\forall$ and *exists* $\exists$ on values, e.g., integer values.
    - $\vdash \forall z_1, z_2 \in \mathbb{Z} : z_1 + z_2 = z_2 + z_1.$
    - $\vdash \forall r_1 \in \mathbb{Z}_{64} : \exists r_2 \in \mathbb{Z}_{64} : (r_1 * r_2 \equiv 1 \mod 2^{64}) \vee (r_1 * r_2 \equiv 0 \mod 2^{64}).$
    - $\vdash \exists f_1, f_2, f_3 \in \mathbb{F}_{64} : f_1 + (f_2 + f_3) \neq (f_1 + f_2) + f_3.$
    - **Hoare logic** (to describe the possible behaviours of an imperative program)
    - ...

Automatic solvers for satisfiability and validity

- Satisfiability (SAT) Solver
  - Given a formula ψ in propositional logic, check its satisfiability
  - Given a formula ψ in propositional logic, check its validity (by checking the satisfiability of $\neg\psi$).
- Satisfiability Modulo Theories (SMT) Solver
  - Check satisfiability and validity of a first-order logic formula ψ in theories:
    - Integers
    - Bounded-integers (of size 2^n)
    - Floating-point numbers (with configurable bitwidths for mantissa and exponent)
    - Uninterpreted functions
    - Arrays (useful for modeling random access memory in computers)
    - ...

Hoare logic

- Reason rigorously about the correctness of a computer program.
- A *predicate* is a boolean condition that must be satisfied by a state of the program
- Example predicates for the program ...; $s_1 : x := v; s_2 : ...$:
    - $(\texttt{PC} = s_2) \rightarrow (x = v)$
    - $(\texttt{PC} = s_1) \rightarrow (y = 0)$
    - $(\texttt{PC} = s_2) \rightarrow (x = v \wedge y = 0)$
    - $(\texttt{PC} = s_2) \rightarrow \texttt{true}$ (not saying much)
- An *assertion* is a predicate connected to a point (PC) in the program.
    - The assertion must evaluate to true when the program is at that PC.

Hoare triple

- A *Hoare triple* is a *triple* that describes how the execution of a piece of code changes the state of the computation.

- $\{P\}C\{Q\}$ represents a Hoare triple where:
  - C is a command.
  - P and Q are assertions connected to the PCs before and after C respectively.
  - P is called a *precondition*, and Q is called a *postcondition*.

  - This triple represents a boolean-valued expression that is equivalent to: *when the precondition P is met, executing the command C (to termination) establishes the postcondition Q*.

Imperative language example

- Five constructors to code a program:

`skip`	Do nothing, just changes the PC value
`x := E`	Evaluate expression `E` and assigns its value to `x`
`S ; T`	Execute `S` followed by `T`
`if B then S else T endif`	Evaluate boolean-valued expression `B` and execute S if it evaluates to true and `T` if it evaluates to false
`while B do S done`	Evaluate boolean-valued expression `B` and execute S if it evaluates to true. Repeat until `B` evaluates to false

Example program : Division

Inputs: x,y s.t. $x \geq 0$, $y > 0$

```
q := 0;
r := x;
while r≥ y do
  r := r - y;
  q := q + 1;
```

Outputs: q,r s.t. $x = q*y + r$, $r \geq 0$, $r < y$

Notation for substitution

- $P[E/x]$ represents a predicate P' that is identical to P except that each free reference to x in P has been replaced with expression E in P'.

Proof rules for Hoare logic

$$
\begin{array}{lcll}
 & \vdash & \{P\}\texttt{skip}\{P\} & \text{(skip)} \\
 & \vdash & \{P[E/x]\}\texttt{x} := \texttt{E}\{P\} & \text{(assignment)} \\
\{P\}S\{Q\}, \{Q\}T\{R\} & \vdash & \{P\}S;T\{R\} & \text{(sequence)} \\
P_1 \to P_2, \{P_2\}S\{Q_2\}, Q_2 \to Q_1 & \vdash & \{P_1\}S\{Q_1\} & \text{(consequence)} \\
\{B \land P\}S\{Q\}, \{\neg B \land P\}T\{Q\} & \vdash & \{P\}\texttt{if B then S else T endif}\{Q\} & \text{(conditional)} \\
\{P \land B\}S\{P\} & \vdash & \{P\}\texttt{while B do S done}\{\neg B \land P\} & \text{(while)}
\end{array}
$$

Example program annotated with assertions

$\{x \geq 0,\ y > 0\}$
q := 0;
$\{x \geq 0,\ y > 0,\ q = 0\}$
r := x;
$\{x \geq 0,\ y > 0,\ q = 0,\ x = r\}$
$\{x \geq 0,\ y > 0,\ x = q * y + r,\ r \geq 0\}$
while r$\geq$ y do
    $\{x \geq 0,\ y > 0,\ x = q * y + r,\ r \geq y\}$
    r := r - y;
    $\{x \geq 0,\ y > 0,\ x = (q+1) * y + r,\ r \geq 0\}$
    q := q + 1;
    $\{x \geq 0,\ y > 0,\ x = q * y + r,\ r \geq 0\}$
$\{x \geq 0,\ y > 0,\ x = q * y + r,\ r \geq 0,\ r < y\}$

Predicate Transformer semantics

- Define the semantics of an *imperative programming paradigm*.

- For a statement s in an imperative programming language, a *predicate transformer* is a function that relates a predicate that holds for a state before the execution of s with a predicate that holds after the execution of s.

- Example: if $s : x := v$, and if ψ holds after the execution of s, then $\psi_{pre_s} \equiv \psi[v/x]$ holds before the execution of s.
  - If $\psi \equiv (x + y) = 0$, then $\psi_{pre_s} \equiv (v + y) = 0$.
  - If $\psi \equiv (x = v)$, then $\psi_{pre_s} \equiv (v = v) \equiv \texttt{true}$.
  - If $\psi \equiv \texttt{false}$, then $\psi_{pre_s} \equiv \texttt{false}$.

Weakest precondition predicate transformer

- For a statement S and a postcondition R, a weakest precondition is a predicate Q such that for any precondition P, PSR holds if and only if $P \rightarrow Q$ holds.
- (Especially for programs without loops) Weakest precondition provide an effective algorithm to reduce the problem of verifying a Hoare triple to the problem of proving a first-order logic formula.

Weakest precondition rules

$$
\begin{array}{lll}
\text{wp}(& \texttt{skip}, & R) = \text{R} \\
\text{wp}(& \texttt{x:=E}, & R) = \text{R[E/x]} \\
\\
\text{wp}(& S;\, T, & R) = \text{wp}(S,\, \text{wp}(T,\, R)) \\
\\
\text{wp}(& \text{if } B \text{ then } S \text{ else } T, & R) = (B \to wp(S, R)) \wedge (\neg B \to wp(T, R)) \\
\\
\text{wp}(& \text{while } B \text{ do } S \text{ done}, & R) \leftarrow I \text{ if } ((B \wedge I) \to wp(S, I)) \wedge ((\neg B \wedge I) \to wp(S, R)) \text{ holds}
\end{array}
$$

Hoare triple and Weakest precondition transformer

- $\{P\}S\{Q\}$ holds iff $P \rightarrow wp(S, Q)$ holds.
- wp() can be computed precisely for acyclic programs.
- If adequate loop invariants are available, the proof for any correct Hoare triple can be constructed automatically.
  - Unfortunately, identifying loop invariants automatically is not possible in general.

Example program annotated with the loop invariant

$\{x \geq 0,\ y > 0\}$

q := 0;

$\{x \geq 0,\ y > 0,\ q=0\}$

r := x;

$\{x \geq 0,\ y > 0,\ q=0,\ x=r\}$

$\boxed{\{x \geq 0,\ y > 0,\ x=q*y+r,\ r \geq 0\} = \mathbf{I}}$

while r$\geq$ y do

$\quad\{x \geq 0,\ y > 0,\ x=q*y+r,\ r \geq y\}$

$\quad$**r := r - y;**

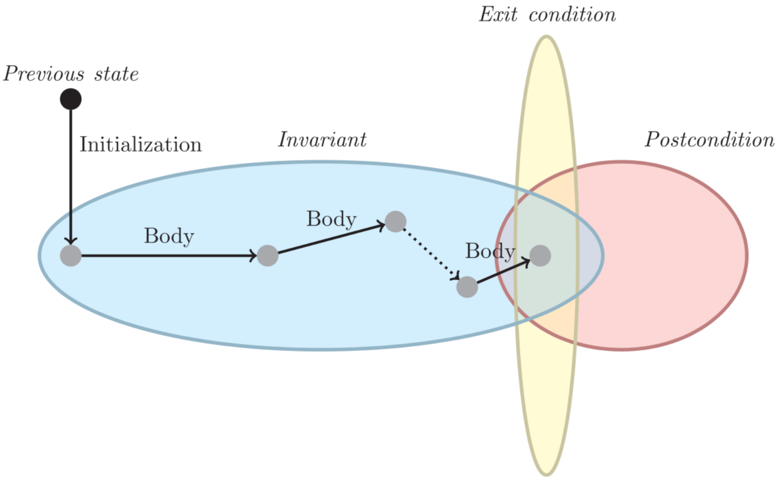
$\quad\{x \geq 0,\ y > 0,\ x=(q+1)*y+r,\ r \geq 0\}$

$\quad$**q := q + 1;**

$\quad\{x \geq 0,\ y > 0,\ x=q*y+r,\ r \geq 0\}$

$\{x \geq 0,\ y > 0,\ x=q*y+r,\ r \geq 0,\ r<y\}$

Loop invariant and its relation to the postcondition



Loop invariants as approximations of a computation. From Furia et. al., Figure 1

Loop invariant example: Euclid's algorithm

Inputs: a,b s.t. $a > 0, b > 0$

x := a;
y := b;
$\{x > 0,\ y > 0,\ \gcd(x,y) = \gcd(a,b)\}$
while $x \neq y$ do
    if $x < y$ then x := x-y else y := y-x

Output: x s.t. $x = \gcd(a,b)$

Loop invariant example: Array maximum

Inputs: Array arr of integers

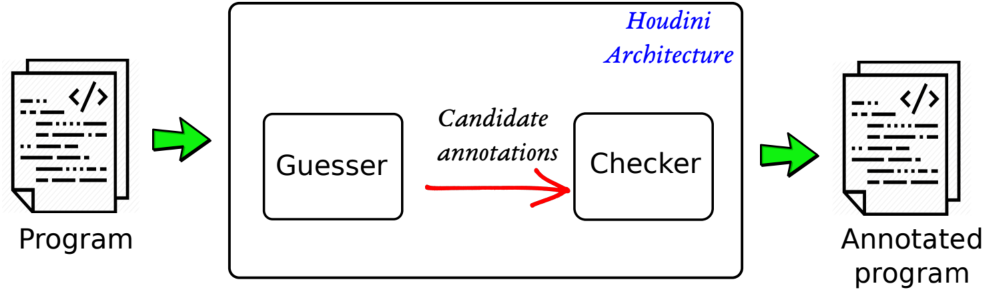
i := 1;
m := arr[0];
{i$\geq$ 0, i$\leq$len(arr), max(arr[0],arr[1],...,arr[i-1])=m}
while i$\neq$len(arr) do
    i := i + 1;
    if m$\leq$arr[i] then m := arr[i];

Output: m s.t. m=max(arr)

Houdini's guess-and-check approach

- Can identify loop invariants, given a (potentially large) set of candidates.
- **Guess-and-check** approach: Guess some (candidate) annotations and then check if they are correct.

Houdini architecture



Houdini architecture. From I. Dillig's slides.

Example program annotated with the candidate loop invariants

$\{x \geq 0, y > 0\}$
q := 0;
r := x;
candidate $C_i \in C_{all} = \{x < 0, x < 1, x \geq 0, y > 0, x = y + 1, q = r, x = q * y + r, r \geq 0, \ldots\}$
while r $\geq$ y do
    r := r - y;
    q := q + 1;

Houdini algorithm

$C_{cur} := C_{all}$; //Initialization
While something changes: //Fixed-point computation
    For each $C_i \in C_{cur}$:
        if $\neg$Verify(C_{cur}, C_i) then
            $C_{cur} := C_{cur} \setminus \{C_i\}$

Verify(C_{cur}, C_i) checks if the Hoare triples generated by using C_{cur} as the precondition and C_i as the postcondition are valid.

Example program and its verification conditions

$\{x \geq 0,\ y > 0\}$
q := 0;
r := x;
candidate $C_i \in C_{all} = \{x < 0,\ x < 1,\ x \geq 0,\ y > 0,\ x = y + 1,\ q = r,\ x = q * y + r,\ r \geq 0, \ldots\}$
while r $\geq$ y do
    r := r - y;
    q := q + 1;

Verify(C_{cur}, C_i) returns false iff either of the following conditions is violated

- $\{x \geq 0,\ y > 0\}$ q := 0; r := x $\{C_i\}$
- $\{C_{cur} \wedge r \geq y\}$ r := r - y; q := q + 1 $\{C_i\}$

Houdini algorithm with its properties

$C_{cur} := C_{all}$; //Initialization
While something changes: //Fixed-point computation
  For each $C_i \in C_{cur}$:
    if $\neg$Verify(C_{cur}, C_i) then
      $C_{cur} := C_{cur} \setminus \{C_i\}$

Verify(C_{cur}, C_i) checks if the Hoare triples generated by using C_{cur} as the precondition and C_i as the postcondition are valid.

Soundness: Upon termination, C_{cur} represents the loop invariants

Termination: Terminates after $\leq |C_{cur}|$ iterations

Largest subset: Finds the largest subset $\subseteq C_{all}$ so the verification conditions are satisfied.

See the playlist on dataflow analysis to understand why.

Limitations of the Houdini algorithm

- The invariants that can be inferred is limited by the set of guessed candidates.
  - The set of required candidates can be infinitely large, e.g., $\texttt{x=3*y+1} \vee \texttt{x=2*z+3}$.
  - The guessed candidates are heuristically chosen to capture the typical properties of the programs being analyzed.

- The running time of the algorithm is proportional to the set of guessed candidates and the size of the program.
  - Practically speaking, this limits the size of the set of guessed candidates to say < 100.

Data-driven invariant inference

- Idea: Execute the program on a set of high-coverage test cases.
- From the execution traces, collect the values of the variables as observed during the execution.
- From these variable values observed during execution, generate candidate guesses for Houdini.
    - A candidate guess is a predicate that evaluates to true for all the execution traces seen so far.

Example program annotated with execution traces

$(x,y) \in \{(0,1),(2,1),(3,2),(9,6),\ldots\}$

```
q := 0;
r := x;
```

$(x,y,q,r) \in \{ (0,1,0,1),$
$(2,1,0,2),(2,1,1,1),(2,1,2,0),$
$(3,2,0,3),(3,2,1,1),$
$(9,6,0,9),(9,6,1,3) \}$

```
while r ≥ y do
    r := r - y;
    q := q + 1;
```

Candidate loop invariants: $\{ \cancel{x=2}$, $x \geq 0$, $\cancel{x=r}$
$x+1 \geq y$, $x+y \geq 0$, $\cancel{x<2*y}$
$\cancel{q \leq y}$, $\cancel{q \geq y}$, $x=q*y+r \}$

Observations on data-driven invariant inference

- While execution traces (data) prune the space of candidate loop invariants, the search space remains uncountably large.

- We still need heuristics to orient the search towards *likely invariants*, perhaps by using domain knowledge about the programs being analyzed.

- This approach requires high-coverage test cases.
  - If a test suite does not cause a loop to get executed, we have no data for that loop.
  - If a test suite only exercises some of the possible values, pruning is ineffective.

- This approach requires sophisticated infrastructure to efficiently generate the required traces upon program execution.

Data-driven inference of affine loop invariants

- An affine loop invariant is a loop invariant of the form $\sum_{i=0}^{n} c_i * x_i = c_0$.
    - $x_0, x_1, \ldots, x_n$ are program variables. $c_0, c_1 \ldots, c_n$ are constants.

- An efficient algorithm to identify the largest set (conjunction) of affine invariants exists.

- For example, let the execution traces for program variables (x, y, z) be $\{(0,1,2), (1,2,4), (2,3,6)\}$. The candidate affine invariants are obtained from the smallest affine space that contains these *points* in 3D space. In this case, the affine space is characterized by basis vectors $\lambda_1(y = x + 1) + \lambda_2(z = 2 * y)$.
    - The same affine space can also be characterized by other basis vectors, e.g., $\lambda_1(y = x + 1) + \lambda_2(z = 2 * x + 2)$.
    - Another affine space that would also contain these points is $\lambda_1(z = x + y + 1)$ (but it would not be the smallest).

- The basis vectors forms the set of candidate invariants.

Observations on affine loop invariant candidate inference

- The basis vectors of the smallest affine space containing a set of n points can be computed in $O(n^3)$ time.
  - Solve $Ax = b$ using LU decomposition of matrix A (to obtain x).
  - Fortunately, n is usually not that large (say 10s of variables that are *live* at a PC).
- The obtained candidates may be stronger (tighter) than what is actually true.
  - For example, the candidates identified for {(0,1,2), (1,2,4), (2,3,6)} may be y=x+1 and z=2*x+2, whereas the actual loop invariant was z=x+y+1.
- There is no guarantee that the inferred affine invariants will help prove the postcondition. All these algorihms are *best effort* algorithms.

Checking a Hoare triple may produce a counterexample

- Consider a proof obligation $\{P\}S\{Q\}$ where
  - P, Q are assertions at $\mathtt{PC}_1$ (just before S) and $\mathtt{PC}_2$ (just after S) respectively.
  - S is acyclic.
- Equivalent to checking the validity of $\pi = (P \rightarrow wp(S, Q))$ (using an SMT solver).
- If the SMT solver determines that π is not valid, it generates a counterexample γ.
- γ is an "execution" trace that evaluates the assertion P (at $\mathtt{PC}_1$) to true, but when executed on S, evaluates the assertion Q (at $\mathtt{PC}_2$) to false.
  - If P is stronger than the actual assertion at $\mathtt{PC}_1$, then γ is as good as a real execution trace.
  - And so its execution over S can be used to obtain an execution trace γ' at $\mathtt{PC}_2$.
  - γ', in turn, can be used to obtain the candidate assertions (e.g., loop invariants) at $\mathtt{PC}_2$.

Example of counterexample generation

$\{x \geq 0,\ y > 0\}$
q := 0;
r := x;
$\{\texttt{false}\}$ assume strongest possible assertion initially
 $(x,y,q,r) \in \{\qquad\}$
while $r \geq y$ do
  r := r - y;
  q := q + 1;

- Check: $\{x \geq 0,\ y > 0\}$ q := 0; r := x $\{\texttt{false}\}$
  - SMT solver returns `invalid` with $\gamma \equiv (x, y) = (0, 1)$.
  - Execution of $\gamma \equiv (x, y) = (0, 1)$ over "`q := 0; r := x`" yields $(x, y, q, r) = (0, 1, 0, 0)$.
  - The assertion at the loop head is *relaxed* (loosened) based on this execution result.

Example of counterexample generation

```
{x≥ 0, y> 0}
q := 0;
r := x;
{x=0,y=1,q=0,r=0}     smallest affine space for the current set of points
 (x,y,q,r)∈{   (0,1,0,0)   }
while r≥ y do
      r := r - y;
      q := q + 1;
```

- Check: $\{x \geq 0,\ y > 0\}$ q := 0; r := x $\{x=0\}$
  - SMT solver returns `invalid` with $\gamma \equiv (x, y) = (1, 1)$.
  - Execution of $\gamma \equiv (x, y) = (1, 1)$ over "`q := 0; r := x`" yields $(x, y, q, r) = (1, 1, 0, 1)$.

Example of counterexample generation

$\{x \geq 0,\ y > 0\}$
q := 0;
r := x;
$\{y=1, x=q, x=r\}$ smallest affine space for the current set of points
$(x,y,q,r) \in \{ \quad (0,1,0,0),\ (1,1,0,1) \quad \}$
while $r \geq y$ do
    r := r - y;
    q := q + 1;

- Check: $\{x \geq 0,\ y > 0\}$ q := 0; r := x $\{x=0\}$
  - SMT solver returns `invalid` with $\gamma \equiv (x, y) = (1, 1)$.
  - Execution of $\gamma \equiv (x, y) = (1, 1)$ over "`q := 0; r := x`" yields $(x, y, q, r) = (1, 1, 0, 1)$.

Counterexample-guided invariant inference algorithm

1. Initialize: set the candidate assertion at the program entry to the P (precondition), and at all other program points to `false`.
2. Check each resulting Hoare triple in sequence. If all Hoare triples are valid, return success. If a Hoare triple fails with counterexample γ, go to the next step.
3. Execute γ to obtain traces at other `PC`s. To ensure termination, upper bound the number of execution steps.
4. Use the newly generated traces to relax the candidate assertions at all `PC`s. Go to step 2.

Program optimization

Original program	**Optimized program**
char mul2(char x) { return 2*x; }	char shl1(char y) { return y<<1; }

- How to check if `shl1` is an optimized implementation of `mul2`, i.e., are they equivalent?
- Construct a *product program* that executes both programs in *lockstep*, and reason about the properties of the product program.
- $\{x=y\}$ mul2(x); shl1(y) $\{ ret_1 = ret_2 \}$

Program optimization with memory operations

Original program	Optimized program
void mul2(char* x) {	void shl1(char* x) {
*x := *x * 2;	*x := *x << 1;
}	}

- A global array M maps `int`$_{64}$ to `int`$_8$.
  - `select(`M`, p)` returns $M.at(p)$.
  - `store(`M`, p, v)` returns a new array M' s.t. $\forall\alpha :((\alpha = p) \rightarrow M'.at(\alpha) = v) \wedge ((\alpha \neq p) \rightarrow M'.at(\alpha) = M.at(\alpha)$.
- "*x" translates to `select`(M, x).
- "*x := v" translates to M := `store`$(M, \mathrm{x}, \mathrm{v})$.
- Check: $\{\mathrm{x}_1{=}\mathrm{x}_2,\ M_1 = M_2\}$ mul2(x$_1$); shl1(x$_2$) $\{\ M_1 = M_2\ \}$
  - Equivalent to $(\mathrm{x}_1{=}\mathrm{x}_2 \wedge M_1 = M_2) \rightarrow (store(M_1, \mathrm{x}_1, select(M_1, \mathrm{x}_1)) = store(M_2, \mathrm{x}_2, select(M_2, \mathrm{x}_2)))$.

Program optimization with loops

Original program	**Optimized program**

void init1(long n) {
  long i = 0;
  while (i < n) {
    foo(i * 3);
    i++;
  }
}
	void init2(long m) {
	  long j = 0;
	  while (j < 3*m) {
	    foo(j);
	    j += 3;
	  }
	}

- The product program needs to correlate the two programs in lockstep.
- The equivalence check becomes easier if **small acyclic fragments of both programs can be correlated**.

Product program examples

Individual programs f() and g()

```
f() {
  while (*)
    A;
  return a; }
```

```
g() {
  while (*)
    B;
  return b; }
```

Product programs X() and Y()

```
X() {
  while (*)
    A;
  while (*)
    B;
  assert(a == b); }
}
```

```
Y() {      Easier to infer invariants here
  assert(Inv);
  while (*) {
    A;
    B;
  }
  assert(a == b); }
```

Convert while to if-then-else and goto

Original program	Optimized program
void init1(long n) {	void init2(char* b, long m) {
long i = 0;	long j = 0;
l1: if (i < n) {	l2: if (j < 3*m) {
foo(i * 3);	foo(j);
i += 1;	j += 3;
goto l1; }	goto l2; }
}	}

Product program with invariants

```
void X(long n, long m) {
        long i = 0;    long j = 0;
    l:  assert(j=3*i);
        if (i < n && j < 3*m) {
              foo(i * 3);
              i += 1;
              foo(j);
              j += 3;
              goto l;   }
         assert(M1 = M2);
}
```

Equivalence checker demonstration

Translation validation