Can LLMs Formally Reason as Abstract Interpreters for Program Analysis?

Anonymous ACL submission

Abstract

LLMs have demonstrated impressive capabilities in code generation and comprehension, but their potential in being able to perform program analysis in a formal, automatic manner remains under-explored. To that end, we systematically investigate whether LLMs can reason about programs using a program analysis framework called abstract interpretation. We prompt LLMs to follow two different strategies, denoted as Compositional and Fixed Point Equation, to formally reason in the style of abstract interpretation, which has never been done before to the best of our knowledge. We validate our approach using state-of-the-art LLMs on 22 challenging benchmark programs from the Software Verification Competition (SV-COMP) 2019 dataset, widely used in program analysis. Our results show that our strategies are able to elicit abstract interpretation-based reasoning in the tested models, but LLMs are susceptible to logical errors, especially while interpreting complex program structures, as well as general hallucinations. This highlights key areas for improvement in the formal reasoning capabilities of LLMs.

1 Introduction

Large language models (LLMs) have undeniably changed the way we interact with software, offering significant benefits in software development, from acting as coding agents (Bouzenia et al., 2024) to aiding program analysis tasks (Cheng et al., 2024). Program analysis mathematically proves certain properties of a program, such as ensuring that safety-critical systems never reach an error state. A program property that always holds true is called a program *invariant*.

Many works have leveraged LLMs for the task of guessing program invariants by performing supervised fine-tuning on examples of program invariants (First et al., 2023; Wu et al., 2024a; Pei et al., 2023). However, these approaches primarily treat LLMs as black-box tools, offloading the formal reasoning to external solvers. Thus, it is still unclear whether the LLMs are capable of proving that the generated invariants are indeed true, or they are simply relying on pattern-matching from the examples seen during training or fine-tuning. Thus, it is of interest to see if LLMs are capable of generating formal proofs as part of their reasoning.

One type of proof comes from running interactive theorem provers (e.g., Lean (Moura and Ullrich, 2021), Coq (Bertot et al., 2014)) based on first-order / higher-order logic and type systems. While many prior works have used LLMs to generate formal proofs (Yang et al., 2024b; Wei et al., 2024), they still rely on some interaction with an external solver. Another type of proof comes from conducting abstract interpretation on a program (Cousot and Cousot, 1977). In contrast to interactive theorem provers, abstract interpretation is a form of automatic (non-interactive) theorem proving that soundly identifies program invariants during all possible program executions. Because abstract interpretation operates on relatively simplified mathematical operations compared to higher-order logic, evaluating LLMs' capabilities in doing this task on their own is a precursor to automating more complex logical reasoning.

To explore these gaps, we evaluate the innate ability of LLMs to conduct formal reasoning in the style of abstract interpretation. These abstract interpretation steps are inherently algorithmic, which suggests that algorithmic approaches (Sel et al., 2023) could be beneficial, but have not been explored yet in the context of program analysis and code semantics. To this end, we design an evaluation pipeline as follows. First, we translate benchmark programs into a simple intermediate representation and annotate them with program locations and control flow directives for context. Then, we prompt LLMs to follow two different strategies for invariant generation in the style of abstract interpretation. Finally, we verify whether or not the generated invariants are sound at each program location and analyze any errors LLMs have made during the process. Our main contributions are:

1. We introduce how to elicit formal reasoning from LLMs under two abstract interpretation-based strategies in the context of invariant generation.

2. We propose an experimental pipeline and evaluate LLMs' formal reasoning abilities on real-world benchmark programs from the Software Verification Competition (SV-COMP) 2019.

3. We quantitatively and qualitatively analyze thematic errors in LLMs' formal reasoning to identify their common pitfalls and discuss how to address them.

To the best of our knowledge, we are the first to systematically evaluate whether LLMs are capable of generating and proving program invariants in a formal, automatic manner. The paper is organized as follows. We first review the preliminaries and define our problem statement in Section 2. Then, we outline our methodology in Section 3. We present and discuss our experimental evaluation in Section 4. Finally, we review the related work in Section 5 and finish with concluding remarks in Section 6.

2 Preliminaries

2.1 Abstract Interpretation

Abstract interpretation is a mathematical framework used to prove properties about programs by approximating their semantics. Consider the program in Figure 1; to verify that a program never reaches an `ERROR` state, one could theoretically execute the program with every possible value of a. However, such an analysis would never terminate, as a can take on infinitely many values.

Motivated by this, abstract interpretation executes the program *abstractly*, such that the analysis can be made to terminate (Cousot and Cousot, 1977). This is done by representing variable values not in a concrete domain, but in an abstract domain (AD). In our work, we use a simple abstract domain called the interval abstract domain.

Definition 1 (Interval Abstract Domain)**.** Each integer variable x is assigned an interval $[a, b]$ (where $a \leq b$), $\top([-\inf, \inf])$, or $\bot$ (the empty interval, or undefined). The partial order $\sqsubseteq$ is defined by $[a, b] \sqsubseteq [c, d]$, if and only if $c \leq a$ and $b \leq d$.

```
int a = input();
if (a > 6)    { a = 0; }
while (a < 6) { a = a + 1; }
if (a > 6)    { ERROR; }
```

Figure 1: Our running example program.

For example, instead of representing x as $x = 1$ or $x = 2$, we can represent it in the interval AD as $x \mapsto [1, 2]$, meaning that that x can take any integer value between 1 and 2.

Definition 2 (Abstract State)**.** An abstract state S maps each variable to an integer interval.

Consider the first line of the program in Figure 1 (`int a = input();`). Just after executing that line, the abstract state (AS) at this location will be $S = \{a \mapsto [-\inf, \inf]\}$, indicating that the value of a can be anything after reading user input.

Definition 3 (Witness Invariant)**.** A witness invariant[1] ϕ maps every location in a program to an abstract state, which maps every variable to an integer interval that contains all possible values the variable may take during execution. If every abstract state does this for every program location, a witness invariant (WI) is considered *sound*.

Using Figure 1 again, let l represent the program location after exiting the while-loop. Consider the following witness invariant (WI) $\phi = \{l \mapsto \{a \mapsto [8, 10]\}\}$. This is unsound because the value of a at l may be 6, which is not represented by the interval. A sound witness invariant could map l to $a \mapsto [6, 6]$ or $a \mapsto [4, 10]$, as these intervals contain all possible values of a at l. Now, consider the following WI: $\phi = \{l \mapsto \{a \mapsto [6, 6]\}\}$. This is not only sound but also sufficiently precise to prove that the `ERROR` state is unreachable.

2.2 Computing Witness Invariants

Computing witness invariants via abstract interpretation requires four pieces of machinery: (1) abstract transfer functions, (2) filtering operations, (3) join operations, and (4) widening operations.

Definition 4 (Abstract Transfer Function)**.** An abstract transfer function lifts a program operation into one that operates on a given abstract domain.

[1] In the context of our paper, we use witness invariants and program invariants interchangeably.

For example, subtraction in the interval AD is defined as $[a, b] - [c, d] = [a - d, b - c]$. Specifically, the resulting interval must include the result of $n_1 - n_2$, for all $n_1 \in [a, b]$ and $n_2 \in [c, d]$.

Definition 5 (Filtering Operation)**.** A filtering operation refines an abstract state to ensure that a specific Boolean expression holds.

In our running example, suppose that the AS before the first if-statement is $a \mapsto [-\inf, \inf]$. When entering the then-branch, a filtering operation refines $a \mapsto [-\inf, \inf]$ to $a \mapsto [7, \inf]$ to account for that $a > 6$ holds.

Definition 6 (Join Operation)**.** A join operation uses the join operator $\sqcup$ to merge two abstract states to over-approximate their union.

For example, if $S = \{a \mapsto [-1, 2]\}$ and $S' = \{a \mapsto [4, 7]\}$, then $S \sqcup S' = \{a \mapsto [-1, 7]\}$.

Tying these things together, we look at the while-loop in our running example. Let F be a function that performs a filtering operation to refine an AS by the loop guard $a < 6$. Let I be a function that applies to an AS an addition abstract transfer function for the instruction inside the loop body $a = a + 1$. Together, $I \circ F$ represents the interpretation of the loop body once. Suppose that the abstract state just before the loop is $S_0 = \{a \mapsto [0, 0]\}$. Now, we can define the sequence of iterations as $S_{k+1} = S_k \sqcup I \circ F(S_k)$ and continue until $S_{k+1} = S_k$. This outputs the following sequence of abstract states: $S_1 = \{a \mapsto [0, 1]\}, \ldots, S_6 = \{a \mapsto [0, 6]\}, S_7 = \{a \mapsto [0, 6]\}$. Thus, our final abstract state that describes all behaviors of the loop is $\{a \mapsto [0, 6]\}$. Note that $S_7 = F(S_6) = S_6$.

This procedure is known as *fixed point computation*, which describes how to reach convergence to derive a sound abstract state in the presence of loops (Cousot and Cousot, 1977). However, convergence can often be slow and may not even terminate. This motivates the widening operation.

Definition 7 (Widening Operation)**.** A widening operation uses the widening operator ∇ to accelerate the convergence of fixed point computation by over-approximating the result as follows: $[a, b] \nabla [c, d] = [\text{if } c < a \text{ then } -\inf \text{ else } a, \text{ if } d > b \text{ then } \inf \text{ else } b]$.

Using the same example as before, we now define the sequence of iterations using the widening operation: $S_{k+1} = S_k \nabla I \circ F(S_k)$. Then, $S_1 = \{a \mapsto [0, 0] \nabla [0, 1]\} = \{a \mapsto [0, \inf]\}$ and $S_2 = \{a \mapsto [0, \inf]\}$. Since $S_2 = S_1$, our final abstract state is $\{a \mapsto [0, \inf]\}$. Note that the final abstract states using $\sqcup$ and ∇ differ slightly in form. Both approaches are sound, but ∇ guarantees that the analysis will terminate, whereas $\sqcup$ provides no such guarantee.

$E := n \mid x \mid E \odot E \mid \texttt{read}()$
$B := x \oplus n \mid B \&\& B \mid B \,||\, B \mid !B \mid (B)$
$C := \texttt{skip} \mid C; C \mid x := E \mid$
$\quad \texttt{if}(B) \text{ then } \{C\} \text{ else } \{C\} \text{ end } \mid \texttt{while} \text{ do } (B)\{C\} \text{ end}$
$P := C$
$\odot \in \{+, -, *, /\}$ (arithmetic operators)
$\oplus \in \{<, <=, ==, >, >=\}$ (comparison operators)
$n \in \mathbb{Z}$ (integers)
$x \in \mathbb{X}$ (program variables)

Figure 2: A simple IMP context-free grammar for some program P.

```
{P0}
a := read();
{P1}
if (a > 6) then
  [then]
  {P2}
  a := 0;
  {P3}
else
  [else]
  {P4}
  skip;
  {P5}
end [endif]
{P6}
while (a < 6) do
  [while_true]
  {P7}
  a := a + 1;
  {P8}
end [while_false]
{P9}
```

Figure 3: Annotated version of our running example program from Figure 1 written in the grammar of Figure 2. {P0} ... {P9} mark program locations, which are included to help LLMs identify where to compute abstract states.

3 Methodology

With these definitions in place, we now introduce our task formally: *Given a program P, can we prompt LLMs to successfully generate a sound witness invariant ϕ for each program location using abstract interpretation?*

In this section, we discuss our methodology to do this task, which consists of annotating input programs for abstract operations (Section 3.1) and prompting LLMs to perform abstract interpretation under two distinct strategies for computing witness invariants (Section 3.2).

3.1 Program Annotation

Grammar In our study, all programs are expressed in an intermediate language derived from the IMP programming language, a small imperative language containing while-loops, if-statements, assignments, and sequential composition of those statements. The grammar used in our work is shown in Figure 2. We translate programs into this grammar to simplify the program representation allow the LLMs to reason like abstract interpreters, since this intermediate representation is common in the area of formal program semantics.

Program Locations Figure 3 shows an annotated version of the running example translated into our grammar. We place program location marks before every program statement defined in IMP with the following annotation: {Px}. We do the translation and annotation manually, but this can also be done automatically.

Control Flow Directives Each program location that deals with control flow, such as `if`-statements and `while`-loops, is annotated with control flow directives, to show LLMs how to use control-flow information for fixed point computation.

`if`**-statements** rely on join ($\sqcup$) operation at the end of the statements to soundly over-approximate all possible program behaviors. Using the example from Figure 3, the abstract state(AS) at {P6} is the result of joining AS at {P3} and at {P5}. This is indicated to the LLM with an [`endif`] directive. The branching statements depend on filtering operation to satisfy the condition to enter either branch. Looking at Figure 3 again, suppose that the abstract state at {P1} is $a \mapsto [-10, 20]$. Then, the abstract state at {P2} is $a \mapsto [7, 20]$, as the condition indicates that $a > 6$. This is indicated to the LLM with a [`then`] directive. Analogously, the same thing is done in the else branch with the negation of the condition. This is indicated with a [`else`] directive.

`while`**-loops** also rely on the join and filtering operations. The AS immediately before the while loop ({P6}) and the AS immediately after the last statement of the loop body ({P8}) are joined together for both at the loop head ({P7}) and outside the statement ({P9}). To account for the condition being true ({P7}), the join operation is filtered by the loop guard; the LLM is instructed to do this with a [`while_true`] directive. To account for the false case ({P9}), the join operation is filtered by the negation of the loop guard; this is indicated to the LLM with a [`while_false`] directive. [`while_true`] directive also indicates to the LLM to perform widening operation to accelerate the convergence of fixed point computation.

3.2 Prompting for Abstract Interpretation

There are two primary strategies to calculate witness invariants in abstract interpretation, Compositional and Fixed Point Equation (Cousot and Cousot, 1977), described in detail later. Since these approaches are inherently algorithmic, we design our prompts inspired by the Algorithm of Thoughts (AoT) (Sel et al., 2023). AoT is similar to Chain of Thought (Wei et al., 2022) but further integrates the search process into their few-shot learning (Brown et al., 2020). In-context examples in AoT are designed to illustrate how to evaluate each solving step and to guide whether the model should explore a problem subtree further or backtrack to find a different viable subtree to make progress towards the solution. Our full prompts for the two strategies including some examples are provided in Appendix A for reference.

3.2.1 Compositional Strategy

Compositional approach interprets each program operation as a function between abstract states, where each program construct is interpreted by *compositionally* applying a corresponding abstract version of the operation. This closely aligns with the mathematical, theoretical perspective of abstract interpretation.

As shown on the top left of Figure 4, we represent the program like a tree for this strategy to guide LLMs to inductively interpret statements. By leveraging the subtree information at each program location, they can perform higher-level operations for locations that depend on previously computed abstract states. The program locations are not explicit in the abstract program semantics, so we model updating the abstract state at a specific location as a side-effect. For example, when `a := read()` is processed in Step 1, we update the abstract state at $\{P1\}$ to be $\{a : [-\inf, \inf]\}$ as a side-effect of interpreting the statement that precedes it.

We show the inner workings of the Compositional strategy in Figure 5. Black arrows represent the flow between program components, and blue arrows represent the flow between the internal machinery for fixed point computation. This approach takes in an initial abstract state S, iteratively transforms it by processing each statement, and returns a final abstract state S'.

First, we interpret the `read()` and then go

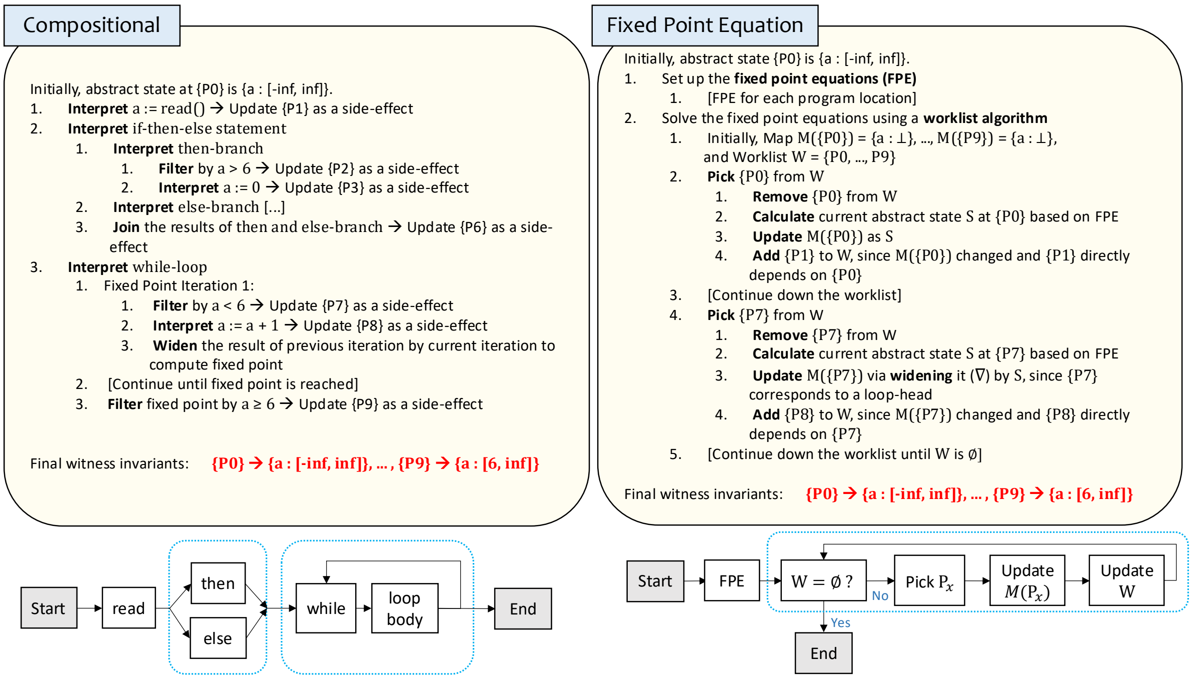

Figure 4: Two strategies for abstract interpretation: Compositional (left) and Fixed Point Equation (right). The texts on the top show the in-context outputs for the two strategies, given the annotated program from Figure 3 as the in-context input. The flowcharts on the bottom visually represent the algorithmic flow of the two strategies.

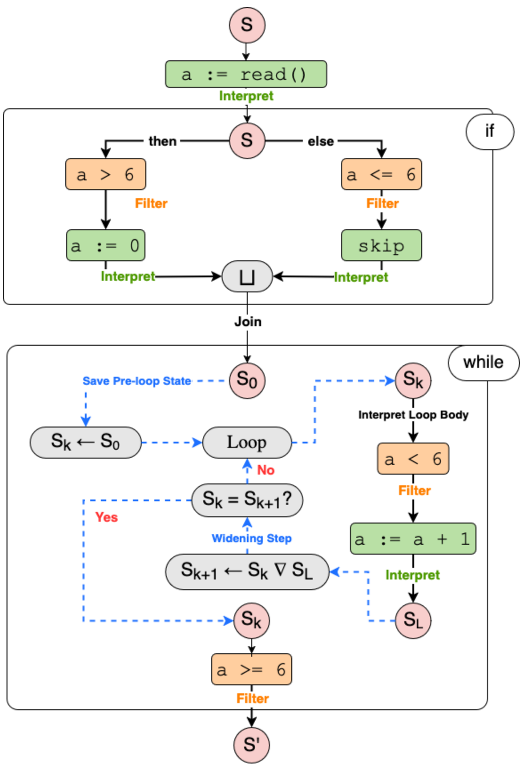

Figure 5: Overall flow of the Compositional strategy for our running example. It corresponds to the high-level workflow shown on the bottom left of Figure 4.

$$
\begin{aligned}
\{P0\} &= \{a : [-\inf, \inf]\} \\
\{P1\} &= Interpret(a := read, P0(a)) \\
\{P2\} &= Filter(a > 6, P1(a)) \\
\{P3\} &= Interpret(a := 0, P2(a)) \\
\{P4\} &= Filter(a \leq 6, P1(a)) \\
\{P5\} &= Interpret(skip, P4(a)) \\
\{P6\} &= \{a : P3(a) \sqcup P5(a)\} \\
\{P7\} &= Filter(a < 6, P6(a) \sqcup P8(a)) \\
\{P8\} &= Interpret(a := a + 1, P7(a)) \\
\{P9\} &= Filter(a \geq 6, P6(a) \sqcup P8(a))
\end{aligned}
$$

Figure 6: Fixed point equations for FPE Prompting for our running example.

through the `if`-statement. The two branches (`then` and `else`) are interpreted separately, and their results are joined at the end ($\sqcup$). Now, we go through the `while`-loop, which is interpreted using fixed point computation in a recursive manner. We first initialize the iteration at $k = 0$, then interpret the loop body, and perform widening (∇), until we reach $S_k = S_{k+1}$. Upon convergence, we go through filtering again to exit the loop and output our final abstract state S'.

3.2.2 Fixed Point Equation Strategy

In contrast to Compositional strategy, which inductively reasons over program statements, Fixed Point Equation strategy explicitly derives and solves a system of *fixed point equations* (FPE). FPEs capture how abstract states are transformed based on the program semantics and control-flow. Each program location has a corresponding FPE, and the system of equations is solved using a standard worklist algorithm. This closely aligns with how abstract interpreters are implemented in practice, known as chaotic iterations.

Figure 6 shows the set of FPEs for our running example. For instance, the AS at $\{P7\}$ (loop head) is the result of filtering the join of the abstract states at $\{P6\}$ (before the loop) and $\{P8\}$ (after the loop body) by the loop guard ($a < 6$).

As shown on the right-hand side of Figure 4, for this strategy, we first ask LLMs to come up with a set of FPEs. Then, it asks to solve it in a linear fashion using a worklist algorithm. Worklist is a list of program locations whose abstract states are not yet converged. Initially, the worklist contains all program locations, and the procedure continues until the worklist is completely empty.

For every program location $\{P_x\}$ that is picked at each step: (1) $\{P_x\}$ is removed from the worklist. (2) The abstract state for $\{P_x\}$, S, is calculated based on its FPE. (3a) S is saved to a map M, where $M(\{P_x\})$ is the most recent AS for $\{P_x\}$. (3b) If $\{P_x\}$ is the first program location inside of a while-loop body (e.g., $\{P_7\}$), the most recent AS for $\{P_x\}$ (i.e., $M(\{P_x\})$) is widened by the current abstract state S to ensure termination. (4) If $M(\{P_x\})$ has changed during the update, then the program locations whose FPEs directly depend on $\{P_x\}$ are to the worklist. This procedure continues until the worklist is finally empty.

4 Experiments

We now describe the experiments conducted on the two strategies. We refer the readers again Appendix A for the full prompts.

4.1 Experimental Setup

Implementation Details We selected 22 C programs from the SV-COMP 2019 dataset (Beyer, 2019) containing complex control flows, such as nested loops and conditionals. C programs were parsed to IMP using a customized parser. Once the models are queried, we automatically verify the soundness of the witness invariant using UAutomizer (Heizmann et al., 2013), a winning tool in the latest SV-COMP.

Models For our main experiment, we selected four models: (1) NVIDIA's Llama 3.1 Nemotron 70B Instruct (Wang et al., 2024), (2) Google's Gemini 2.0 Flash (Kavukcuoglu, 2025), (3) OpenAI's GPT-4o (OpenAI et al., 2024), and (4) Qwen's QwQ 32B Preview (Yang et al., 2024a). All queries were made using their native API libraries, with the exception of Llama which used OpenRouter.[2] We set the temperature to 0 for all models.

4.2 Main Results

Table 1 describes our experimental results. Column 1 shows the program names. Columns 2-5 show the results for Compositional strategy (Section 3.2.1), and Columns 6-13 show the results for FPE strategy (Section 3.2.2). The '-' symbol means that the LLMs did not return a final witness invariant at the end of their responses.

4.2.1 Witness Invariant Soundness

The Witness Invariant (WI) Soundness columns describe whether the witness invariant generated per program by each model is sound or not. The fraction indicates how many of the abstract states within the witness invariant are sound (the number of abstract states corresponds to the number of program locations, denoted by the denominator).

Any fraction less than 1 means that there is an abstract state at some program location that does not capture all possible values of the program variables; for example, an abstract state at $\{P2\}$ maps x to $[6, 8]$, but there is some concrete program execution such that $x = 5$ at $\{P2\}$. When the model returns a final witness invariant that is missing abstract states for certain locations, a point is also deducted per program location missing.

Looking at Table 1, there are fewer '-' cells for Compositional than for FPE strategy. This indicates that LLMs return more outputs that are coherent and not cut off for Compositional than for FPE strategy. We hypothesize that this is due to the fact in Compositional strategy, the program statements are reasoned over inductively, and this additional structure that inductive reasoning enforces may allow for better internal state tracking. In contrast, in FPE strategy, the worklist algorithm simply keeps track of program locations and their corresponding fixed point equations. Upon inspection, we find that for FPE strategy, this is largely due to LLMs not knowing when to terminate the worklist algorithm and continuously adding a program location when the abstract state has already reached convergence, which corroborates our hypothesis. For

[2] https://openrouter.ai/

Table 1: Comparison of different models across the two strategies on 22 C programs.

Program	Compositional				FPE							
	Witness Invariant Soundness				*Witness Invariant Soundness*				*Fixed Point Equation Correctness*			
	Llama	Gemini	GPT-4o	QwQ	Llama	Gemini	GPT-4o	QwQ	Llama	Gemini	GPT-4o	QwQ
afnp2014.c	3/7	7/7	6/7	7/7	3/7	7/7	3/7	-	7/7	7/7	7/7	7/7
as2013-hybrid.c	3/14	14/14	14/14	14/14	11/14	-	3/14	-	12/14	14/14	14/14	13/13
benchmark02_linear.c	11/12	12/12	12/12	12/12	9/12	12/12	12/12	12/12	10/12	12/12	12/12	12/12
benchmark04_conjunctive.c	12/13	13/13	6/13	2/13	9/13	13/13	6/13	-	11/13	13/13	12/13	13/13
cggmp2005.c	5/9	9/9	8/9	6/9	3/9	8/9	-	-	9/9	9/9	9/9	9/9
const.c	14/14	14/14	14/14	14/14	14/14	14/14	14/14	14/14	14/14	14/14	14/14	14/14
count_by_2.c	6/6	6/6	6/6	5/6	4/6	6/6	6/6	5/6	6/6	6/6	6/6	2/6
css2003.c	10/16	16/16	13/16	14/16	8/16	16/16	-	-	14/16	16/16	16/16	16/16
deep-nested.c	10/33	-	10/33	9/33	4/33	21/33	-	-	33/33	33/33	33/33	13/33
eq1.c	14/14	14/14	14/14	14/14	14/14	14/14	14/14	0/14	14/14	14/14	14/14	5/14
eq2.c	9/9	9/9	9/9	9/9	9/9	9/9	9/9	5/9	9/9	9/9	9/9	2/9
even.c	5/5	5/5	5/5	5/5	5/5	5/5	5/5	5/5	5/5	5/5	5/5	3/5
gauss_sum.c	9/14	14/14	13/14	13/14	10/14	14/14	10/14	-	14/14	14/14	14/14	14/14
in-de20.c	-	14/14	14/14	8/14	7/14	13/14	5/14	10/14	14/14	14/14	14/14	10/14
jm2006.c	13/18	18/18	16/18	15/18	6/18	18/18	-	11/18	18/18	18/18	17/18	18/18
loopv3.c	8/11	11/11	11/11	5/11	9/11	11/11	11/11	9/11	11/11	11/11	11/11	6/11
mine-2018-ex4.6.c	5/5	5/5	2/5	5/5	5/5	5/5	3/5	5/5	5/5	5/5	3/5	5/5
mono-crafted_7.c	6/17	14/17	13/17	7/17	13/17	-	-	-	17/17	17/17	17/17	17/17
Mono6_1.c	4/12	12/12	12/12	12/12	7/12	-	6/12	-	12/12	12/12	12/12	12/12
nested_1.c	6/11	11/11	11/11	11/11	11/11	11/11	11/11	11/11	11/11	11/11	11/11	11/11
nested_2.c	10/16	16/16	16/16	5/16	15/16	15/16	10/16	-	16/16	16/16	16/16	13/16
simple_vardp_1.c	9/9	9/9	9/9	-	4/9	9/9	8/9	-	9/9	9/9	9/9	9/9

Compositional strategy, we find that this is due to meeting the output token limit.

For cases where both the Compositional and FPE strategies returned a final witness invariant, more programs are found with more sound abstract states in Compositional than in FPE strategy across the models. For Llama, 8 programs yield better results with Compositional strategy, and 7 programs yield better results with FPE strategy. For Gemini, 3 programs and 0 programs perform better with Compositional and FPE strategy, respectively. For GPT-4o, the scores are 6 programs and 2 programs, respectively. For QwQ, the scores are 2 programs and 2 programs, respectively.

When considering the same cases, the total number of sound abstract states found for Llama in FPE strategy is 1 more than what is found in Compositional. For the rest of the models, Compositional strategy yielded more sound abstract states compared to FPE: there are 3, 38, and 16 more sound abstract states found for Gemini, GPT, and QwQ, respectively in Compositional strategy.

4.2.2 Fixed Point Equation Correctness

The Fixed Point Equation (FPE) Correctness columns found under FPE strategy header describe whether or not the fixed point equations generated per program by each model are correct or not. Recall that for FPE strategy, models are required to derive correct fixed point equations before starting the worklist algorithm. Thus, we want to evaluate if the LLMs find difficulty in either setting up the fixed point equations in the first place or going through the worklist algorithm.

The results show that the fixed point equations are almost always all correct, with the exception of a few, mostly in QwQ. This indicates that LLMs in general have a good understanding of the abstract control flows in the program. Considering that the fractions in the Witness Invariant Soundness columns are in general smaller than the fractions under the FPE Correctness columns, we can conclude that although LLMs can set up the equation systems well at the start, they still struggle during the worklist algorithm.

4.3 Additional Results

We additionally tested our methodology on more advanced models: OpenAI's GPT-o1 (OpenAI, 2024) and DeepSeek's R1 (DeepSeek-AI et al., 2025). They are known to be capable of complex, multi-step reasoning, so we are interested to see if these specific abilities will help the LLMs reason in the style of abstract interpretation.

Table 2 presents the results of these models on a four selected programs that contain challenging nested control-flows. How to interpret the column names are the same as our main table. Overall, GPT-o1 and DeepSeek-R1 perform well, in particular for FPE strategy. This is especially notable for mono-crafted_7, for which three out of four models used in the main evaluation could not even return coherent abstract states at the end. DeepSeek did especially well for deep-nested, with 31 out of 33 abstract states being sound.

What is particularly interesting about these results is that FPE correctness is sometimes lower than WI soundness for DeepSeek-R1. This means that although the model generated wrong fixed point equations in the beginning, it was still able to derive sound abstract states toward the end.

Table 2: Comparison of advanced models on selected C programs. GPT refers to GPT-o1, and DS refers to DeepSeek-R1.

Program	Compositional		FPE			
	WI Soundness		*WI Soundness*		*FPE Correctness*	
	GPT	DS	GPT	DS	GPT	DS
deep-nested.c	10/33	8/33	8/33	31/33	31/33	33/33
mono-crafted_7.c	17/17	17/17	17/17	17/17	17/17	14/17
nested_1.c	11/11	11/11	11/11	11/11	11/11	11/11
nested_2.c	16/16	16/16	16/16	16/16	16/16	13/16

4.4 Thematic Errors Made by LLMs During Reasoning

In most cases, LLMs successfully generated traces of abstract interpretation but exhibited common errors across models. This section outlines these recurring mistakes.

Forgetting Key Operations. LLMs may overlook essential program operations. For instance, in `as2013-hybrid.c`, Gemini incorrectly filters $[0, 0]$ with $i \leq 9$ to $[0, 9]$, overapproximating unintentionally. In the FPE algorithm, LLMs often neglect the widening operator during worklist iterations, especially in complex nested structures requiring prolonged analysis. In `css2003.c`, GPT-4o fails to apply widening after a long iteration.

Mixing State Order. LLMs struggle with control flow in long-context programs. In `deep_nested.c`, which has six nested while-loops, the $R1$ model misorders abstract operations, applying transformations before the corresponding statements.

Short-Circuiting. LLMs sometimes produce final abstract states that would be unreachable if widening were properly applied. Similarly, they may generate abstract states that do not follow correct interpretation steps, suggesting reliance on extraneous information. Despite being an error, short-circuiting could inform heuristics for deciding when to widen (Lakhdar-Chaouch et al., 2011).

5 Related Work

There is a growing body work in the intersection of formal reasoning and LLMs, due to the demonstrated capability of LLMs to reason about code. Some methods use LLMs as agents in program analysis pipelines (Bouzenia et al., 2024; Cheng et al., 2024), while others use LLMs to predict loop invariants to be used in bounded model checkers in an iterative manner, until the bounded model checker approves the invariant (Wu et al., 2024a).

Pei et al. (2023) fine-tune LLMs on Daikon datasets for invariant prediction. While they achieve reasonable test performance, it is unclear whether LLMs are merely pattern-matching, given the similarity between training and test data. Moreover, their approach does not assess LLMs' innate formal reasoning abilities or specific frameworks like abstract interpretation.

First et al. (2023) similarly fine-tune LLMs on Isabelle/HOL-style proofs to generate full proofs rather than individual steps. They show that additional context, such as error messages from failed proof attempts, helps LLMs repair proofs. However, it remains unclear whether their performance stems from training data rather than innate ability, as formal reasoning is left to external proof assistants.

Wu et al. (2024b) propose a formal proof language where LLMs predict invariants within inference rules, with correctness checked by Z3. Incorrect predictions allow for backtracking, but formal reasoning is still delegated to external tools.

These are works closest to ours. Unlike these works, we evaluate LLMs' innate formal reasoning ability without fine-tuning or reliance on external tools.

6 Conclusion

We introduce a new framework to evaluate the *innate* ability of LLMs to conduct formal reasoning for code in the style of abstract interpretation. In particular, we provide two novel prompting strategies inspired by Algorithm of Thoughts that can be used to elicit formal reasoning required to understand code abstractions. Our results indicate that the LLM has the ability to perform formal reasoning under a proper guiding algorithm, but there is still space to potential guiding algorithms to avoid errors. The ability of the LLMs to generate abstract interpretation-based proofs shows great promise for the using LLMs as program verifiers, however, correcting the errors we encountered is critical for this endeavor. We plan to explore avenues to improve upon this in future work.

Limitations

While our results demonstrate that LLMs have promising potential in reasoning as abstract interpreters, we acknowledge several limitations, the first two are regarding the choice of the concepts from abstract interpretation we are testing.

The first limitation is that the in-context examples are limited to a simple widening strategy. In practice, many different widening strategies may be used which may be more precise than the standard widening operator from (Cousot and Cousot, 1977), we introduced. Techniques such as narrowing may be employed in practice to recover precision. This limitation raises questions about the generalizability to the broader landscape of abstract interpretation techniques used in abstract interpreters.

The second is our choice of the interval abstract domain. The interval abstract domain is a very simple domain, which is not as expressive in terms of its ability to capture relationships between program variables. In practice, a variety of much more expressive abstract domains are used in order to prove program properties, including the octagon abstract domain, which can capture linear relationships between program variables (i.e. $x \leq y$) to the symbolic bit-vector domain used to prove properties about cryptographic code and low-level systems software. Our choice of abstract domain was motivated by the possibility that more complex domains would make it more difficult for the LLMs to interpret. For instance, in the polyhedra abstract domain, the join of two polyhedra requires computing the convex hull of two polyhedrons. Instead, in this work, our goal was to focus on understanding if the LLMs can understand the core concepts behind abstract interpretation, rather than specific abstract domains. We leave exploration of other domains to future work.

The third limitation is in that while LLMs may produce valid proofs of program properties by generating a correct sequence of abstract intepretation-based steps, it is unclear if this capability corresponds to internal phenomena in the vein of mechanistic interpretability. Combining our work along with mechanisitic interpretability techniques could provide deeper insights into how LLMs handle the tasks explored in our paper, for instance, if certain neural circuits correspond to handling certain components of abstract interpretation, such as a circuit specialized for widening.

References

Y. Bertot, G. Huet, P. Castéran, and C. Paulin-Mohring. 2014. *Interactive Theorem Proving and Program Development: Coq'Art: The Calculus of Inductive Constructions*. Texts in Theoretical Computer Science. An EATCS Series. Springer Berlin Heidelberg.

Dirk Beyer. 2019. Advances in automatic software verification: Sv-comp 2019. In *Tools and Algorithms for the Construction and Analysis of Systems (TACAS 2019)*, volume 11429 of *Lecture Notes in Computer Science*, pages 194–223. Springer.

Islem Bouzenia, Premkumar T. Devanbu, and Michael Pradel. 2024. Repairagent: An autonomous, llm-based agent for program repair. *CoRR*, abs/2403.17134.

Tom B. Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, Sandhini Agarwal, Ariel Herbert-Voss, Gretchen Krueger, Tom Henighan, Rewon Child, Aditya Ramesh, Daniel M. Ziegler, Jeffrey Wu, Clemens Winter, Christopher Hesse, Mark Chen, Eric Sigler, Mateusz Litwin, Scott Gray, Benjamin Chess, Jack Clark, Christopher Berner, Sam McCandlish, Alec Radford, Ilya Sutskever, and Dario Amodei. 2020. Language models are few-shot learners. *CoRR*, abs/2005.14165.

Yiran Cheng, Lwin Khin Shar, Ting Zhang, Shouguo Yang, Chaopeng Dong, David Lo, Shichao Lv, Zhiqiang Shi, and Limin Sun. 2024. Llm-enhanced static analysis for precise identification of vulnerable OSS versions. *CoRR*, abs/2408.07321.

Patrick Cousot and Radhia Cousot. 1977. Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Proceedings of the 4th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages (POPL)*, pages 238–252. ACM.

DeepSeek-AI, Daya Guo, Dejian Yang, Haowei Zhang, Junxiao Song, Ruoyu Zhang, Runxin Xu, Qihao Zhu, Shirong Ma, Peiyi Wang, Xiao Bi, Xiaokang Zhang, Xingkai Yu, Yu Wu, Z. F. Wu, Zhibin Gou, Zhihong Shao, Zhuoshu Li, Ziyi Gao, Aixin Liu, Bing Xue, Bingxuan Wang, Bochao Wu, Bei Feng, Chengda Lu, Chenggang Zhao, Chengqi Deng, Chenyu Zhang, Chong Ruan, Damai Dai, Deli Chen, Dongjie Ji, Erhang Li, Fangyun Lin, Fucong Dai, Fuli Luo, Guangbo Hao, Guanting Chen, Guowei Li, H. Zhang, Han Bao, Hanwei Xu, Haocheng Wang, Honghui Ding, Huajian Xin, Huazuo Gao, Hui Qu, Hui Li, Jianzhong Guo, Jiashi Li, Jiawei Wang, Jingchang Chen, Jingyang Yuan, Junjie Qiu, Junlong Li, J. L. Cai, Jiaqi Ni, Jian Liang, Jin Chen, Kai Dong, Kai Hu, Kaige Gao, Kang Guan, Kexin Huang, Kuai Yu, Lean Wang, Lecong Zhang, Liang Zhao, Litong Wang, Liyue Zhang, Lei Xu, Leyi Xia, Mingchuan Zhang, Minghua Zhang, Minghui Tang, Meng Li, Miaojun Wang, Mingming Li, Ning Tian, Panpan

Huang, Peng Zhang, Qiancheng Wang, Qinyu Chen, Qiushi Du, Ruiqi Ge, Ruisong Zhang, Ruizhe Pan, Runji Wang, R. J. Chen, R. L. Jin, Ruyi Chen, Shanghao Lu, Shangyan Zhou, Shanhuang Chen, Shengfeng Ye, Shiyu Wang, Shuiping Yu, Shunfeng Zhou, Shuting Pan, S. S. Li, Shuang Zhou, Shaoqing Wu, Shengfeng Ye, Tao Yun, Tian Pei, Tianyu Sun, T. Wang, Wangding Zeng, Wanjia Zhao, Wen Liu, Wenfeng Liang, Wenjun Gao, Wenqin Yu, Wentao Zhang, W. L. Xiao, Wei An, Xiaodong Liu, Xiaohan Wang, Xiaokang Chen, Xiaotao Nie, Xin Cheng, Xin Liu, Xin Xie, Xingchao Liu, Xinyu Yang, Xinyuan Li, Xuecheng Su, Xuheng Lin, X. Q. Li, Xiangyue Jin, Xiaojin Shen, Xiaosha Chen, Xiaowen Sun, Xiaoxiang Wang, Xinnan Song, Xinyi Zhou, Xianzu Wang, Xinxia Shan, Y. K. Li, Y. Q. Wang, Y. X. Wei, Yang Zhang, Yanhong Xu, Yao Li, Yao Zhao, Yaofeng Sun, Yaohui Wang, Yi Yu, Yichao Zhang, Yifan Shi, Yiliang Xiong, Ying He, Yishi Piao, Yisong Wang, Yixuan Tan, Yiyang Ma, Yiyuan Liu, Yongqiang Guo, Yuan Ou, Yuduan Wang, Yue Gong, Yuheng Zou, Yujia He, Yunfan Xiong, Yuxiang Luo, Yuxiang You, Yuxuan Liu, Yuyang Zhou, Y. X. Zhu, Yanhong Xu, Yanping Huang, Yaohui Li, Yi Zheng, Yuchen Zhu, Yunxian Ma, Ying Tang, Yukun Zha, Yuting Yan, Z. Z. Ren, Zehui Ren, Zhangli Sha, Zhe Fu, Zhean Xu, Zhenda Xie, Zhengyan Zhang, Zhewen Hao, Zhicheng Ma, Zhigang Yan, Zhiyu Wu, Zihui Gu, Zijia Zhu, Zijun Liu, Zilin Li, Ziwei Xie, Ziyang Song, Zizheng Pan, Zhen Huang, Zhipeng Xu, Zhongyu Zhang, and Zhen Zhang. 2025. Deepseek-r1: Incentivizing reasoning capability in llms via reinforcement learning. *Preprint*, arXiv:2501.12948.

Emily First, Markus N. Rabe, Talia Ringer, and Yuriy Brun. 2023. Baldur: Whole-proof generation and repair with large language models. In *Proceedings of the 31st ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering, ESEC/FSE 2023, San Francisco, CA, USA, December 3-9, 2023*, pages 1229–1241. ACM.

Matthias Heizmann, Jürgen Christ, Daniel Dietsch, Evren Ermis, Jochen Hoenicke, Markus Lindenmann, Alexander Nutz, Christian Schilling, and Andreas Podelski. 2013. Ultimate automizer with smtinterpol. In *Tools and Algorithms for the Construction and Analysis of Systems*, pages 641–643, Berlin, Heidelberg. Springer Berlin Heidelberg.

Koray Kavukcuoglu. 2025. Gemini 2.0 is now available to everyone.

Lies Lakhdar-Chaouch, Bertrand Jeannet, and Alain Girault. 2011. Widening with thresholds for programs with complex control graphs. In *Automated Technology for Verification and Analysis, 9th International Symposium, ATVA 2011, Taipei, Taiwan, October 11-14, 2011. Proceedings*, volume 6996 of *Lecture Notes in Computer Science*, pages 492–502. Springer.

Leonardo de Moura and Sebastian Ullrich. 2021. The lean 4 theorem prover and programming language. In *Automated Deduction – CADE 28*, pages 625–635, Cham. Springer International Publishing.

OpenAI. 2024. Openai o1 system card. Accessed: 2025-02-15.

OpenAI, Josh Achiam, Steven Adler, Sandhini Agarwal, Lama Ahmad, Ilge Akkaya, Florencia Leoni Aleman, Diogo Almeida, Janko Altenschmidt, Sam Altman, Shyamal Anadkat, Red Avila, Igor Babuschkin, Suchir Balaji, Valerie Balcom, Paul Baltescu, Haiming Bao, Mohammad Bavarian, Jeff Belgum, Irwan Bello, Jake Berdine, Gabriel Bernadett-Shapiro, Christopher Berner, Lenny Bogdonoff, Oleg Boiko, Madelaine Boyd, Anna-Luisa Brakman, Greg Brockman, Tim Brooks, Miles Brundage, Kevin Button, Trevor Cai, Rosie Campbell, Andrew Cann, Brittany Carey, Chelsea Carlson, Rory Carmichael, Brooke Chan, Che Chang, Fotis Chantzis, Derek Chen, Sully Chen, Ruby Chen, Jason Chen, Mark Chen, Ben Chess, Chester Cho, Casey Chu, Hyung Won Chung, Dave Cummings, Jeremiah Currier, Yunxing Dai, Cory Decareaux, Thomas Degry, Noah Deutsch, Damien Deville, Arka Dhar, David Dohan, Steve Dowling, Sheila Dunning, Adrien Ecoffet, Atty Eleti, Tyna Eloundou, David Farhi, Liam Fedus, Niko Felix, Simón Posada Fishman, Juston Forte, Isabella Fulford, Leo Gao, Elie Georges, Christian Gibson, Vik Goel, Tarun Gogineni, Gabriel Goh, Rapha Gontijo-Lopes, Jonathan Gordon, Morgan Grafstein, Scott Gray, Ryan Greene, Joshua Gross, Shixiang Shane Gu, Yufei Guo, Chris Hallacy, Jesse Han, Jeff Harris, Yuchen He, Mike Heaton, Johannes Heidecke, Chris Hesse, Alan Hickey, Wade Hickey, Peter Hoeschele, Brandon Houghton, Kenny Hsu, Shengli Hu, Xin Hu, Joost Huizinga, Shantanu Jain, Shawn Jain, Joanne Jang, Angela Jiang, Roger Jiang, Haozhun Jin, Denny Jin, Shino Jomoto, Billie Jonn, Heewoo Jun, Tomer Kaftan, Łukasz Kaiser, Ali Kamali, Ingmar Kanitscheider, Nitish Shirish Keskar, Tabarak Khan, Logan Kilpatrick, Jong Wook Kim, Christina Kim, Yongjik Kim, Jan Hendrik Kirchner, Jamie Kiros, Matt Knight, Daniel Kokotajlo, Łukasz Kondraciuk, Andrew Kondrich, Aris Konstantinidis, Kyle Kosic, Gretchen Krueger, Vishal Kuo, Michael Lampe, Ikai Lan, Teddy Lee, Jan Leike, Jade Leung, Daniel Levy, Chak Ming Li, Rachel Lim, Molly Lin, Stephanie Lin, Mateusz Litwin, Theresa Lopez, Ryan Lowe, Patricia Lue, Anna Makanju, Kim Malfacini, Sam Manning, Todor Markov, Yaniv Markovski, Bianca Martin, Katie Mayer, Andrew Mayne, Bob McGrew, Scott Mayer McKinney, Christine McLeavey, Paul McMillan, Jake McNeil, David Medina, Aalok Mehta, Jacob Menick, Luke Metz, Andrey Mishchenko, Pamela Mishkin, Vinnie Monaco, Evan Morikawa, Daniel Mossing, Tong Mu, Mira Murati, Oleg Murk, David Mély, Ashvin Nair, Reiichiro Nakano, Rajeev Nayak, Arvind Neelakantan, Richard Ngo, Hyeonwoo Noh, Long Ouyang, Cullen O'Keefe, Jakub Pachocki, Alex Paino, Joe Palermo, Ashley Pantuliano, Giambattista Parascandolo, Joel Parish, Emy Parparita, Alex Passos, Mikhail Pavlov, Andrew Peng, Adam Perelman, Filipe de Avila Belbute Peres, Michael Petrov, Henrique Ponde de Oliveira Pinto, Michael, Pokorny, Michelle Pokrass, Vitchyr H. Pong, Tolly Powell, Alethea Power, Boris Power, Elizabeth Proehl, Raul Puri, Alec Radford, Jack Rae, Aditya Ramesh, Cameron Raymond, Francis Real, Kendra Rimbach, Carl Ross, Bob Rotsted, Henri Roussez, Nick Ryder, Mario Saltarelli, Ted Sanders, Shibani Santurkar, Girish Sastry, Heather Schmidt, David Schnurr, John Schulman, Daniel Selsam, Kyla Sheppard, Toki Sherbakov, Jessica Shieh, Sarah Shoker, Pranav Shyam, Szymon Sidor, Eric Sigler, Maddie Simens, Jordan Sitkin, Katarina Slama, Ian Sohl, Benjamin Sokolowsky, Yang Song, Natalie Staudacher, Felipe Petroski Such, Natalie Summers, Ilya Sutskever, Jie Tang, Nikolas Tezak, Madeleine B. Thompson, Phil Tillet, Amin Tootoonchian, Elizabeth Tseng, Preston Tuggle, Nick Turley, Jerry Tworek, Juan Felipe Cerón Uribe, Andrea Vallone, Arun Vijayvergiya, Chelsea Voss, Carroll Wainwright, Justin Jay Wang, Alvin Wang, Ben Wang, Jonathan Ward, Jason Wei, CJ Weinmann, Akila Welihinda, Peter Welinder, Jiayi Weng, Lilian Weng, Matt Wiethoff, Dave Willner, Clemens Winter, Samuel Wolrich, Hannah Wong, Lauren Workman, Sherwin Wu, Jeff Wu, Michael Wu, Kai Xiao, Tao Xu, Sarah Yoo, Kevin Yu, Qiming Yuan, Wojciech Zaremba, Rowan Zellers, Chong Zhang, Marvin Zhang, Shengjia Zhao, Tianhao Zheng, Juntang Zhuang, William Zhuk, and Barret Zoph. 2024. Gpt-4 technical report. *Preprint*, arXiv:2303.08774.

Kexin Pei, David Bieber, Kensen Shi, Charles Sutton, and Pengcheng Yin. 2023. Can large language models reason about program invariants? In *International Conference on Machine Learning, ICML 2023, 23-29 July 2023, Honolulu, Hawaii, USA*, volume 202 of *Proceedings of Machine Learning Research*, pages 27496–27520. PMLR.

Bilgehan Sel, Ahmad Al-Tawaha, Vanshaj Khattar, Lu Wang, Ruoxi Jia, and Ming Jin. 2023. Algorithm of thoughts: Enhancing exploration of ideas in large language models. *CoRR*, abs/2308.10379.

Zhilin Wang, Alexander Bukharin, Olivier Delalleau, Daniel Egert, Gerald Shen, Jiaqi Zeng, Oleksii Kuchaiev, and Yi Dong. 2024. Helpsteer2-preference: Complementing ratings with preferences. *Preprint*, arXiv:2410.01257.

Chenrui Wei, Mengzhou Sun, and Wei Wang. 2024. Proving olympiad algebraic inequalities without human demonstrations. *Preprint*, arXiv:2406.14219.

Jason Wei, Xuezhi Wang, Dale Schuurmans, Maarten Bosma, Brian Ichter, Fei Xia, Ed H. Chi, Quoc V. Le, and Denny Zhou. 2022. Chain-of-thought prompting elicits reasoning in large language models. In *Advances in Neural Information Processing Systems 35: Annual Conference on Neural Information Processing Systems 2022, NeurIPS 2022, New Orleans, LA, USA, November 28 - December 9, 2022*.

Guangyuan Wu, Weining Cao, Yuan Yao, Hengfeng Wei, Taolue Chen, and Xiaoxing Ma. 2024a. LLM meets bounded model checking: Neuro-symbolic loop invariant inference. In *Proceedings of the 39th IEEE/ACM International Conference on Automated Software Engineering, ASE 2024, Sacramento, CA, USA, October 27 - November 1, 2024*, pages 406–417. ACM.

Haoze Wu, Clark W. Barrett, and Nina Narodytska. 2024b. Lemur: Integrating large language models in automated program verification. In *The Twelfth International Conference on Learning Representations, ICLR 2024, Vienna, Austria, May 7-11, 2024*. OpenReview.net.

An Yang, Baosong Yang, Binyuan Hui, Bo Zheng, Bowen Yu, Chang Zhou, Chengpeng Li, Chengyuan Li, Dayiheng Liu, Fei Huang, Guanting Dong, Haoran Wei, Huan Lin, Jialong Tang, Jialin Wang, Jian Yang, Jianhong Tu, Jianwei Zhang, Jianxin Ma, Jin Xu, Jingren Zhou, Jinze Bai, Jinzheng He, Junyang Lin, Kai Dang, Keming Lu, Keqin Chen, Kexin Yang, Mei Li, Mingfeng Xue, Na Ni, Pei Zhang, Peng Wang, Ru Peng, Rui Men, Ruize Gao, Runji Lin, Shijie Wang, Shuai Bai, Sinan Tan, Tianhang Zhu, Tianhao Li, Tianyu Liu, Wenbin Ge, Xiaodong Deng, Xiaohuan Zhou, Xingzhang Ren, Xinyu Zhang, Xipin Wei, Xuancheng Ren, Yang Fan, Yang Yao, Yichang Zhang, Yu Wan, Yunfei Chu, Yuqiong Liu, Zeyu Cui, Zhenru Zhang, and Zhihao Fan. 2024a. Qwen2 technical report. *arXiv preprint arXiv:2407.10671*.

Kaiyu Yang, Gabriel Poesia, Jingxuan He, Wenda Li, Kristin Lauter, Swarat Chaudhuri, and Dawn Song. 2024b. Formal mathematical reasoning: A new frontier in AI. *CoRR*, abs/2412.16075.

A Prompts Used

A.1 Compositional Strategy

Context: Given a program, analyze the program with abstract interpretation, using the interval abstract domain. Programs are composed of assignment, skip, if-then-else, while-loops, and sequential composition of these statements, where program variables are integer variables. The goal is to output an abstract state for each program location. An abstract state maps each program variable to an interval, or the empty interval $\perp$. For example, $\{x : [1, 4], y : [-1, 3]\}$ means that x can take on values between 1 and 4 and y can take on values between -1 and 3. $\perp$ means that the variable cannot have any concrete value.

Each abstract state should be sound. For instance, if the abstract state at location $\{P\}$ maps x to $[4, 10]$, then in any concrete execution of the program, the value of x should be between 4 and 10 at location $\{P\}$.

Arithmetic expressions are interpreted with interval arithmetic. Be cautious of edge cases in interpreting division with interval arithmetic. For example, $[1, 3]/[0, 0] = \perp$, as no valid value results from a division by 0. Furthermore, $[1, 3]/[-2, 3] = [-\inf, \inf]$, as division by 0 may or may not occur.

`read`() expressions are interpreted as $[-\inf, \inf]$, as reading from the standard input can result in any value.

The abstract state at $\{P0\}$, the program entry point, maps each program variable to $[-\inf, \inf]$, indicating that at the beginning of the program, the variables can have any integer value.

You should abstractly interpret programs in a denotational style. This means that each program statement is interpreted as a function, mapping abstract states to abstract states, and we iteratively interpret each statement on an input abstract state. As the program is being interpreted, we save the abstract state at a program location after interpreting the statement preceding it, as a side-effect of the interpretation process.

There are several directives in the annotated programs that help keep track of control flow.
[`if_then`] means that the input abstract state to the if-statement is filtered to account for the fact that the guard of the if-statement should hold.
[`if_else`] means that the input abstract state to the if-statement is filtered to account for the fact that the negation of the guard of the if-statement should hold.
[`endif`] means the the result of interpreting the then-branch on the input abstract state and the result of interpreting the else-branch on the input abstract state are merged.

[`while_true`] means that the input abstract state to a while-statement is filtered to account for the fact that the loop guard should hold.
[`while`$_{\texttt{f}}$`alse`] means that the abstract state as a result of interpreting the loop body is filtered by the negation of the loop guard, indicating possible behaviors when the while loop is no longer executed.

Some examples of filtering are:

- Filtering abstract state $x : [5, 7], y : [6, 8]$ by !(`read`() == `0`) results in the same abstract state, because we cannot know for certain if the result of reading from standard input is 0.
- Filtering abstract state $\{x : [5, 10], y : [5, inf]\}$ by !(`y` == `6`) results in

$x : [5, 10], y : [5, inf]$. Filtering by !(y == 6) is equivalent to filtering by $y > 6 || y < 6$. Filtering the abstract state by $y > 6$ results in $x : [5, 10], y : [7, \inf]$. Filtering the abstract state by $y < 6$ results in $x : [5, 10], y : [5, 5]$. Joining the resulting abstract states results in $x : [5, 10], y : [5, inf]$.

- Filtering abstract state $x : [5, 9], y : [10, 12]$ by y == 16 results in $\{x : \perp, y : \perp\}$, as it is impossible for y to be 16.

- Filtering the abstract state $x : [5, 10], y : [4, 9]$ by (y <= 8)&&(x <= y) results in the filtering the state by y <= 8 and filtering the state by $x <= y$ and then intersecting the resulting states. Filtering $\{x : [5, 10], y : [4, 9]\}$ by $y <= 8$results in$x : [5, 10], y : [4, 8]$. Filtering $x : [5, 10], y : [4, 9]$ by $x <= y$ results in $x : [5, 9], y : [4, 9]$. Intersecting both states results in $x : [5, 9], y : [4, 8]$.

While loops are interpreted using fixed point computation, by iteratively interpreting the body of the while loop on the input abstract state. At each iteration, the result from the previous iteration is widened by the interpretation of the loop body on the abstract state from the previous iteration. Widening is used to ensure termination of the fixed point computation. The widening of two intervals is defined symbolically as $[a, b] \nabla [c, d] = [\text{if } c < a, \text{ then } -\inf \text{ else } a, \text{ if } d > b \text{ then } \inf \text{ else } b]$. For example, $[6, 7] \nabla [9, 10] = [6, \inf]$. Note that $\perp \nabla [c, d] = [c, d]$ and $[a, b] \nabla \perp = [a, b]$. Finally, when a fixed point is reached, it is filtered by the negation of the loop guard, which is the final result of interpreting the loop.
When all commands have been interpreted, the most recent abstract state at each program location is returned.
Here are some examples:
Example 1:
Input:

```
{P0}
x := read();
{P1}
if (x < 3) then
    [if_then]
    {P2}
    x := x - 1;
    {P3}
    x := x * 2;
    {P4}
else
    [if_else]
    {P5}
    x := x + 2;
    {P6}
end [endif]
{P7}
```

Output:
Initially, the abstract state at $P0$ is $x : [-\inf, \inf]$.
Begin interpreting the program.

1. Interpret `x := read();`

    - The input abstract state is $\{x : [-\inf, \inf]\}$

- The resulting abstract state is $\{x : [-\inf, \inf]\}$
- As a side-effect, the abstract state at $\{P1\}$ is $\{x : [-\inf, \inf]\}$

2. Interpret the if-then-else statement.

   - Interpret the then-branch.
     1. The input abstract state is x : [-inf, inf].
     2. Filter the input state by x < 3. The resulting abstract state is x : [-inf, 2]. As a side-effect, the abstract state at P2 is x : [-inf, 2].
     3. Interpret x := x - 1;
        - The input abstract state is x : [-inf, 2].
        - The resulting abstract state is x: [-inf, 1].
        - As a side-effect, the abstract state at P4 is x : [-inf, 2].
   - Interpret the else-branch.
     1. The input abstract state is x : [-inf, inf].
     2. Filter the input state by x >= 3. The resulting abstract state is x : [3, inf]. As a side-effect, the abstract state at P5 is x : [3, inf].
     3. Interpret x := x + 2;
        - The input abstract state is x : [3, inf].
        - The resulting abstract state is x : [5, inf].
        - As a side-effect the abstract state at P6 is x : [5, inf].
   - Join the results of interpreting the then and else branch:
     - The output of interpreting the then-branch is x : [-inf, 2].
     - The output of interpreting the else-branch is x : [5, inf].
     - The result of joining the two states is x : [-inf, inf]. As a side-effect, the abstract state at P7 is x : [-inf, inf]

There are no more statements to interpret, and the answer is

$$
\begin{aligned}
\{P_0\} &\mapsto \{x : [-\inf, \inf]\} \\
\{P_1\} &\mapsto \{x : [-\inf, \inf]\} \\
\{P_2\} &\mapsto \{x : [-\inf, 2]\} \\
\{P_3\} &\mapsto \{x : [-\inf, 1]\} \\
\{P_4\} &\mapsto \{x : [-\inf, 2]\} \\
\{P_5\} &\mapsto \{x : [3, \inf]\} \\
\{P_6\} &\mapsto \{x : [5, \inf]\} \\
\{P_7\} &\mapsto \{x : [-\inf, \inf]\}
\end{aligned}
$$

Example 2:
Input:

```
{P0}
i := 1;
{P1}
j := 0;
{P2}
while (i <= 5) do
    [while_true]
```

```
    {P3}
    j := j + i;
    {P4}
    i := i + 1;
    {P5}
end [while_false]
{P6}
```

Output:
Initially, the abstract state at P0 is $i : [-\inf, \inf], j : [-\inf, \inf]$

1. Interpret i := 1
   - The input abstract state is $\{i : [-inf, inf], j : [-inf, inf]\}$
   - The resulting abstract state is $\{i : [1, 1], j : [-inf, inf]\}$
   - As a side-effect, the abstract state at $\{P1\}$ is $\{i : [1, 1], j : [-inf, inf]\}$
2. Interpret j := 0
   - The input abstract state is $\{i : [1, 1], j : [-inf, inf]\}$
   - The resulting abstract state is $\{i : [1, 1], j : [0, 0]\}$
   - As a side-effect, the abstract state at $\{P2\}$ is $\{i : [1, 1], j : [0, 0]\}$.
3. Interpret the while loop.
   - The input abstract state (iteration 0) is $\{i : [1, 1], j : [0, 0]\}$.
   - Begin fixed point iteration.
   - Fixed point Iteration 1:
     - The input abstract state to this iteration is $\{i : [1, 1], j : [0, 0]\}$
     - Filtering the state by i <= 5 results in the abstract state $\{i : [1, 1], j : [0, 0]\}$. As a side-effect, the abstract state at $\{P3\}$ is $\{i : [1, 1], j : [0, 0]\}$.
     - Interpret j := j + i;
       - The input abstract state is $\{i : [1, 1], j : [0, 0]\}$
       - The resulting abstract state is $\{i : [1, 1], j : [1, 1]\}$
       - As a side-effect the abstract state at $\{P4\}$ is $\{i : [1, 1], j : [1, 1]\}$.
     - Interpret i := i + 1;
       - The input abstract state is $\{i : [1, 1], j : [1, 1]\}$.
       - The resulting abstract state is $\{i : [2, 2], j : [1, 1]\}$.
       - As a side-effect the abstract state at $\{P5\}$ is $\{i : [2, 2], j : [1, 1]\}$.
     - Widen the input abstract state by the interpretation of the loop body
       - The input abstact state to this iteration is $\{i : [1, 1], j : [0, 0]\}$
       - The result of interpreting the loop body is $\{i : [2, 2], j : [1, 1]\}$.
       - $\{i : [1, 1], j : [0, 0]\} \nabla \{i : [2, 2], j : [1, 1]\}$ results in $\{i : [1, inf], j : [0, inf]\}$.
     - The result of this iteration is $\{i : [1, inf], j : [0, inf]\}$.
   - Fixed point Iteration 2:
     - The input abstract state to this iteration is $\{i : [1, inf], j : [0, inf]\}$.
     - Filtering the state by i <= 5 results in the abstract state $\{i : [1, 5], j : [0, inf]\}$. As a side-effect, the abstract state at $\{P3\}$ is $\{i : [1, 5], j : [0, inf]\}$.
     - Interpret j := j + i;
       - The input abstract state is $\{i : [1, 5], j : [0, inf]\}$.
       - The resulting abstract state is $\{i : [1, 5], j : [1, inf]\}$

927

        - As a side-effect the abstract state at $\{P4\}$ is $\{i : [1, 5], j : [1, inf]\}$
    - Interpret i := i + 1;
        - The input abstract state is $\{i : [1, 5], j : [1, inf]\}$
        - The resulting abstract state is $\{i : [2, 6], j : [1, inf]\}$
        - As a side-effect the abstract state at $\{P5\}$ is $\{i : [2, 6], j : [1, inf]\}$.
    - Widen the abstract state from the previous iteration by the interpretation of the loop body
        - The input abstract state to this iteration is $\{i : [1, inf], j : [0, inf]\}$
        - The result of interpreting the loop body is $\{i : [2, 6], j : [1, inf]\}$.
        - $\{i : [1, inf], j : [0, inf]\}\nabla\{i : [2, 6], j : [1, inf]\}$ results in $\{i : [1, inf], j : [0, inf]\}$.
    - The result of this iteration is $\{i : [1, inf], j : [0, inf]\}$.
- We are at a fixed point. The result of the iteration was the same as the previous one.
- Filter the fixed point by the negation of the loop-guard, i > 5. Filtering $i : [1, \text{inf}], j : [0, \text{inf}]$ by $i > 5$ results in $\{i : [6, inf], j : [0, inf]\}$. As a side effect the abstract state at $\{P6\}$ is $\{i : [6, inf], j : [0, inf]\}$.

There are no more statements to interpret, and the answer is

$$
\begin{aligned}
\{P_0\} &\mapsto \{i : [-\text{inf}, \text{inf}], j : [-\text{inf}, \text{inf}]\} \\
\{P_1\} &\mapsto \{i : [1, 1], j : [-\text{inf}, \text{inf}]\} \\
\{P_2\} &\mapsto \{i : [1, 1], j : [0, 0]\} \\
\{P_3\} &\mapsto \{i : [1, 5], j : [0, \text{inf}]\} \\
\{P_4\} &\mapsto \{i : [1, 5], j : [1, \text{inf}]\} \\
\{P_5\} &\mapsto \{i : [2, 6], j : [1, \text{inf}]\} \\
\{P_6\} &\mapsto \{i : [6, \text{inf}], j : [0, \text{inf}]\}
\end{aligned}
$$

Example 3:
Input:

```
{P0}
y := 7;
{P1}
while (true) do
    [while_true]
    {P2}
    x := read();
    {P3}
    while (x <= y) do
        [while_true]
        {P4}
        x := x + 1;
        {P5}
    end [while_false]
    {P6}
end [while_false]
{P7}
```

Initially, the abstract state at $\{P0\}$ is $\{x : [-inf, inf], y : [-inf, inf]\}$.

1. Interpret y := 7
   - The input abstract state is $\{x : [-inf, inf], y : [-inf, inf]\}$.
   - The resulting abstract state is $\{x : [-inf, inf], y : [7, 7]\}$.
   - As a side-effect, the abstract state at $\{P1\}$ is $\{x : [-inf, inf], y : [7, 7]\}$.

2. Interpret the outer while-loop.
   - The input abstract state (iteration 0) is $\{x : [-inf, inf], y : [7, 7]\}$.
   - Begin fixed point iteration.
   - Outer Loop Fixed Point Iteration 1:
     - The input abstract state to this iteration is $\{x : [-inf, inf], y : [7, 7]\}$.
     - Filtering the state by true results in the abstract state $\{x : [-inf, inf], y : [7, 7]\}$. As a side-effect, the abstract state at $\{P2\}$ is $\{x : [-inf, inf], y : [7, 7]\}$.
     - Interpret x := read();
       - The input abstract state is $\{x : [-inf, inf], y : [7, 7]\}$.
       - The resulting abstract state is $\{x : [-inf, inf], y : [7, 7]\}$.
       - As a side-effect, the abstract state at $\{P3\}$ is $\{x : [-inf, inf], y : [7, 7]\}$.
     - Interpret the inner while-loop:
       - The input abstract state (iteration 0) is $\{x : [-inf, \mathrm{inf}], y : [7, 7]\}$.
       - Begin fixed point iteration.
       - Inner Loop Fixed Point Iteration 1:
         - The input abstract state to this iteration is $\{x : [-inf, \mathrm{inf}], y : [7, 7]\}$.
         - Filtering the state by x <= y results in $\{x : [-inf, 7], y : [7, 7]\}$. As a side-effect, the abstract state at $\{P4\}$ is $\{x : [-inf, 7], y : [7, 7]\}$.
         - Interpret x := x + 1;
           - The input abstract state is $\{x : [-inf, 7], y : [7, 7]\}$.
           - The resulting abstract state is $\{x : [-inf, 8], y : [7, 7]\}$.
           - As a side-effect the abstract state at $\{P5\}$ is $\{x : [-inf, 8], y : [7, 7]\}$
         - Widen the abstract state from the previous iteration by the interpretation of the loop body
           - The input abstract state to this iteration is $\{x : [-inf, \mathrm{inf}], y : [7, 7]\}$.
           - The result of interpreting the loop body is $\{x : [-inf, 8], y : [7, 7]\}$.
           - $\{x : [-inf, \mathrm{inf}], y : [7, 7]\} \nabla \{x : [-inf, 8], y : [7, 7]\} = \{x : [-inf, \mathrm{inf}], y : [7, 7]\}$.
         - The result of this iteration is $\{x : [-inf, \mathrm{inf}], y : [7, 7]\}$
       - We are at a fixed point. The result of this iteration was the same as the previous one.
       - Filter the fixed point by the negation of the loop guard, x > y. Filtering $\{x : [-inf, \mathrm{inf}], y : [7, 7]\}$ by x > y results in $\{x : [8, \mathrm{inf}], y : [7, 7]\}$. As a side-effect, the abstract state at $\{P6\}$ is $\{x : [8, \mathrm{inf}], y : [7, 7]\}$.
     - The result of interpreting the inner while loop is $\{x : [8, \mathrm{inf}], y : [7, 7]\}$.
     - Widen the abstract state from the previous iteration by the interpretation of the loop body
       - The input abstract state to this iteration is $\{x : [-inf, \mathrm{inf}], y : [7, 7]\}$.
       - The result of interpreting the outer loop body is $\{x : [8, \mathrm{inf}], y : [7, 7]\}$.
       - $\{x : [-inf, \mathrm{inf}], y : [7, 7]\} \nabla \{x : [8, \mathrm{inf}], y : [7, 7]\} = \{x : [-inf, \mathrm{inf}], y : [7, 7]\}$.

929

- The result of this iteration for the outer while loop is $\{x : [-inf, \text{inf}], y : [7, 7]\}$.
- We've reached a fixed point for the outer while loop. The input state to the first iteration of the fixed point computation for the outer loop is the same as the abstract state resulting from the first iteration.
- Filter the fixed point for the outer while loop by the negation of the loop guard, false. Filtering $\{x : [-inf, \text{inf}], y : [7, 7]\}$ by false results in $\{x : \perp, y : \perp\}$. As a side-effect, the abstract state at $\{P7\}$ is set to $\{x : \perp, y : \perp\}$.

There are no more statements to interpret, and the answer is

$$
\begin{aligned}
\{P_0\} &\mapsto \{x : [-\inf, \inf], y : [-\inf, \inf]\} \\
\{P_1\} &\mapsto \{x : [-\inf, \inf], y : [7, 7]\} \\
\{P_2\} &\mapsto \{x : [-\inf, \inf], y : [7, 7]\} \\
\{P_3\} &\mapsto \{x : [-\inf, \inf], y : [7, 7]\} \\
\{P_4\} &\mapsto \{x : [-\inf, 7], y : [7, 7]\} \\
\{P_5\} &\mapsto \{x : [-\inf, 8], y : [7, 7]\} \\
\{P_6\} &\mapsto \{x : [8, \inf], y : [7, 7]\} \\
\{P_7\} &\mapsto \{x : \perp, y : \perp\}
\end{aligned}
$$

Now, please solve this, outputting the intermediary steps you take:
[Input Program]

A.2 Fixed Point Equation Strategy

Context:
Given a program, analyze the program with abstract interpretation, using the interval abstract domain. Programs are composed of assignment, skip, if-then-else, while-loops, and sequential composition of these statements, where program variables are integer variables. The goal is to output an abstract state for each program location. An abstract state maps each program variable to an interval, or the empty interval $\perp$. For example, $\{x : [1, 4], y : [-1, 3]\}$ means that x can take on values between 1 and 4 and y can take on values between -1 and 3. $\perp$ means that the variable cannot have any concrete value.

Each abstract state should be sound. For instance if the abstract state at location $\{P\}$ maps x to $[4, 10]$, then in any concrete execution of the program, the value of x should be between 4 and 10 at location $\{P\}$.

Arithmetic expressions are interpreted with interval arithmetic. Be cautious of edge cases in interpreting division with interval arithmetic. For example, $[1, 3]/[0, 0] = \perp$, as no valid value results from a division by 0. Furthermore, $[1, 3]/[-2, 3] = [-inf, inf]$, as division by 0 may or may not occur.

`read()` expressions are interpreted as $[-inf, inf]$, as reading from the standard input can result in any value.

You should abstractly interpret programs by first deriving a set of fixed point equations, where each program location corresponds to one equation. Then, solve the fixed point equations iteratively until you reach a fixed point. The fixed point equation associated with the location at program entry, $\{P0\}$, maps each program variable to $[-\inf, \inf]$, indicating that at the beginning of the program, the variables can have any integer value.

There are several directives in the annotated programs that help keep track of control flow, as well as indicate how the fixed point equations should be defined.

[`if_then`] means that the fixed point equation corresponding to the location after the directive is the result of filtering the abstract state at the location corresponding to the input of the if-then-else statement, by the if guard.
[`if_else`] means that the fixed point equation corresponding to the location after the directive is the result of filtering the abstract state at the location corresponding to the input of the if-then-else statement, by the negation of the if guard.
[`if_end`] means the fixed point equation corresponding to the location after the directive is the result of joining the abstract states at the locations of the end of each branch in the if-statement.

[`while_true`] means that the fixed point equation at the location after the directive first joins the abstract states at the program locations before the while-loop and after the last statement in the loop body, and filters this result by the loop guard.

[`while_false`] means that the fixed point equation at the location after the directive first joins the abstract states at the program locations before the while-loop and after the last statement in the loop body, and filters this result by the negation of the loop guard.

Some examples of filtering are:

- Filtering abstract state $x : [5, 7], y : [6, 8]$ by $!(\texttt{read}() == \texttt{0})$ results in the same abstract state, because we cannot know for certain if the result of reading from standard input is 0.

- Filtering abstract state $\{x : [5, 10], y : [5, inf]\}$ by $!(\texttt{y} == \texttt{6})$ results in $x : [5, 10], y : [5, inf]$. Filtering by $!(\texttt{y} == \texttt{6})$ is equivalent to filtering by $y > 6 || y < 6$. Filtering the abstract state by $y > 6$ results in $x : [5, 10], y : [7, \inf]$. Filtering the abstract state by $y < 6$ results in $x : [5, 10], y : [5, 5]$. Joining the resulting abstract states results in $x : [5, 10], y : [5, inf]$.

- Filtering abstract state $x : [5, 9], y : [10, 12]$ by $\texttt{y} == \texttt{16}$ results in $\{x : \bot, y : \bot\}$, as it is impossible for y to be 16.

- Filtering the abstract state $x : [5, 10], y : [4, 9]$ by $(\texttt{y} <= \texttt{8}) \&\& (\texttt{x} <= \texttt{y})$ results in the filtering the state by $\texttt{y} <= \texttt{8}$ and filtering the state by $x <= y$ and then intersecting the resulting states. Filtering $\{x : [5, 10], y : [4, 9]\}$ by $y <= 8$results in$x : [5, 10], y : [4, 8]$. Filtering $x : [5, 10], y : [4, 9]$ by $x <= y$ results in $x : [5, 9], y : [4, 9]$. Intersecting both states results in $x : [5, 9], y : [4, 8]$.

In the equations, use Interpret(assignment, S) and Interpret(skip, S) to denote interpreting the result of applying an assignment statement to abstract state S and applying a skip statement to abstract state S, respectively. Use Filter(B, S) to filter abstract state S by boolean expression B.
Once the equations are set up, fixed point computation is conducted using a worklist algorithm. Initially, all program locations are added to the worklist. If a location is in the worklist, this indicates that the abstract state at that location has not stabilized yet. When we compute the abstract state at a location just after a [`while_true`] directive, we widen the result of computation by the previous abstract state at the same location. This ensures termination of the analysis. The widening of two intervals is defined symbolically as $[a, b] \nabla [c, d] = [\text{if } c < a, \text{ then } -$

933

inf else a, if $d > b$ then inf else b]. For example, $[6,7]\nabla[9,10] = [6, \text{inf}]$. Note that $\perp\nabla[c,d] = [c,d]$ and $[a,b]\nabla\perp = [a,b]$.
A solution for the fixed point equations is reached after the worklist is empty, and the final abstract states are returned.
Here are some examples:
Example 1:

```
Input: {P0}
    x := read();
    {P1}
    if (x < 3) then
        [if_then]
        {P2}
        x := x - 1;
        {P3}
        x := x * 2;
        {P4}
    else
        [if_else]
        {P5}
        x := x + 2;
        {P6}
    end [endif] {P7}
```

Output:
1. Create a system of fixed point equations.

Here, $M(\{P\})$ denotes the abstract state at location $\{P\}$. $M(\{P\})(x)$ denotes the abstract value of variable x at location $\{P\}$. F_0 denotes the fixed point equation at $\{P0\}$.

$$
\begin{aligned}
F_0(M) &= \{x : [-\text{inf}, \text{inf}]\}\\
F_1(M) &= \texttt{Interpret}(\texttt{x} := \texttt{read}(), M(\{P_0\}))\\
F_2(M) &= \texttt{Filter}(\texttt{x} < 3, M(\{P_1\}))\\
F_3(M) &= \texttt{Interpret}(\texttt{x} := \texttt{x} - 1, M(\{P_2\}))\\
F_4(M) &= \texttt{Interpret}(\texttt{x} := \texttt{x} * 2, M(\{P_3\}))\\
F_5(M) &= \texttt{Filter}(\texttt{x} >= 3, M(\{P_1\}))\\
F_6(M) &= \texttt{Interpret}(\texttt{x} := \texttt{x} + 2, M(\{P_5\}))\\
F_7(M) &= M(\{P_4\}) \sqcup M(\{P_6\})
\end{aligned}
$$

2. Solve the fixed point equations using a worklist algorithm.

Initially, the map of program locations to abstract states looks like:

$$M(\{P0\}) = \{x : \bot\},$$
$$M(\{P1\}) = \{x : \bot\},$$
$$M(\{P2\}) = \{x : \bot\},$$
$$M(\{P3\}) = \{x : \bot\},$$
$$M(\{P4\}) = \{x : \bot\},$$
$$M(\{P5\}) = \{x : \bot\},$$
$$M(\{P6\}) = \{x : \bot\},$$
$$M(\{P7\}) = \{x : \bot\}.$$

The worklist W is $\{\{P0\}, \{P1\}, \{P2\}, \{P3\}, \{P4\}, \{P5\}, \{P6\}, \{P7\}$.

- Pick $\{P_0\}$ from W.
  - Remove $\{P_0\}$ from W.
  - $M(\{P_0\})$ is $\{x : \bot\}$.
  - Compute $F_0(M)$, and update the value of $M(\{P_0\})$, resulting in $M(\{P_0\}) = \{x : [-\inf, \inf]\}$.
  - $M(\{P_0\})$ has changed, so add the program locations whose fixed point equations directly depend on $M(\{P_0\})$ to W.
    - Add $\{P_1\}$ to W.
  - W is now $\{\{P_1\}, \{P_2\}, \{P_3\}, \{P_4\}, \{P_5\}, \{P_6\}, \{P_7\}\}$.
- Pick $\{P_1\}$ from W.
  - Remove $\{P_1\}$ from W.
  - $M(\{P_1\})$ is $\{x : \bot\}$.
  - Compute $F_1(M)$, and update the value of $M(\{P_1\})$, resulting in $M(\{P_1\}) = \{x : [-\inf, \inf]\}$, where
    - $M(\{P_1\})(x) = [-\inf, \inf]$ is the result of interpreting $\texttt{x} := \texttt{read}()$.
  - $M(\{P_1\})$ has changed, so add the program locations whose fixed point equations directly depend on $M(\{P_1\})$ to W.
    - Add $\{P_2\}$ and $\{P_5\}$ to W.
  - W is now $\{\{P_2\}, \{P_3\}, \{P_4\}, \{P_5\}, \{P_6\}, \{P_7\}\}$.
- Pick $\{P_2\}$ from W.
  - Remove $\{P_2\}$ from W.
  - $M(\{P_2\})$ is $\{x : \bot\}$.
  - Compute $F_2(M)$:
    - $M(\{P_1\}) = \{x : [-\inf, \inf]\}$
    - Filtering $M(\{P_1\})$ by $x < 3$ results in:
      - $\{x : [-\inf, 2]\}$
    - Update $M(\{P_2\})$ to be $\{x : [-\inf, 2]\}$.
  - $M(\{P_2\})$ has changed, so add the program locations whose fixed point equations directly depend on $M(\{P_2\})$ to W.
    - Add $\{P_3\}$ to W.
  - W is now $\{\{P_3\}, \{P_4\}, \{P_5\}, \{P_6\}, \{P_7\}\}$.

- Pick $\{P_3\}$ from W.
  - Remove $\{P_3\}$ from W.
  - $M(\{P_3\})$ is $\{x : \bot\}$.
  - Compute $F_3(M)$ and update the value of $M(\{P_3\})$, which results in $M(\{P_3\}) = \{x : [-\inf, 1]\}$, where
    - $M(\{P_3\})(x) = M(\{P_2\})(x) - [1, 1] = [-\inf, 2] - [1, 1] = [-\inf, 1]$
  - $M(\{P_3\})$ has changed, so add the program locations whose fixed point equations directly depend on $M(\{P_3\})$ to W.
    - Add $\{P_4\}$ to W.
  - W is now $\{\{P_4\}, \{P_5\}, \{P_6\}, \{P_7\}\}$.
- Pick $\{P4\}$ from W.
  - Remove $\{P4\}$ from W.
  - $M(\{P4\})$ is $\{x : \bot\}$.
  - Compute $F_4(M)$ and update the value of $M(\{P4\})$, which results in $M(\{P4\}) = \{x : [-\inf, 2]\}$, where
    - $M(\{P4\})(x) = M(\{P3\})(x) * [2, 2] = [-\inf, 1] * [2, 2] = [-\inf, 2]$
  - $M(\{P4\})$ has changed, so add the program locations whose fixed point equations directly depend on $M(\{P4\})$ to W.
    - Add $\{P7\}$ to W.
  - W is now $\{\{P5\}, \{P6\}, \{P7\}\}$.
- Pick $\{P5\}$ from W.
  - Remove $\{P5\}$ from W.
  - $M(\{P5\})$ is $\{x : \bot\}$.
  - Compute $F_5(M)$:
    - $M(\{P1\}) = \{x : [-\inf, \inf]\}$
    - Filtering $M(\{P1\})$ by $x \geq 3$ results in:
      - $\{x : [3, \inf]\}$
    - Update $M(\{P5\})$ to be $\{x : [3, \inf]\}$
  - $M(\{P5\})$ has changed, so add the program locations whose fixed point equations directly depend on $M(\{P5\})$ to W.
    - Add $\{P6\}$ to W.
  - W is now $\{\{P6\}, \{P7\}\}$.
- Pick $\{P6\}$ from W.
  - Remove $\{P6\}$ from W.
  - $M(\{P6\})$ is $\{x : \bot\}$.
  - Compute $F_6(M)$ and update the value of $M(\{P6\})$, which results in $M(\{P6\}) = \{x : [5, \inf]\}$, where
    - $M(\{P6\})(x) = M(\{P5\})(x) + [2, 2] = [3, \inf] + [2, 2] = [5, \inf]$
  - $M(\{P6\})$ has changed, so add the program locations whose fixed point equations directly depend on $M(\{P6\})$ to W.
    - Add $\{P7\}$ to W.
  - W is now $\{\{P7\}\}$.

936

- Pick $\{P7\}$ from W.
  - Remove $\{P7\}$ from W.
  - $M(\{P7\})$ is $\{x : \bot\}$.
  - Compute $F_7(M)$:
    - $M(\{P4\}) \sqcup M(\{P6\}) = \{x : [-\inf, 2]\} \sqcup \{x : [5, \inf]\} = \{x : [-\inf, \inf]\}$
    - Update $M(\{P7\})$ to be $\{x : [-\inf, \inf]\}$
  - $M(\{P7\})$ has changed so add the program locations whose fixed point equations directly depend on $M(\{P7\})$ to W.
    - According to the system of equations, there is no such location, so no location is added to W.
  - W is now $\{\}$.

The worklist is empty, meaning we've finished the analysis and M is

$$M(\{P0\}) = \{x : [-\inf, \inf]\}$$
$$M(\{P1\}) = \{x : [-\inf, \inf]\}$$
$$M(\{P2\}) = \{x : [-\inf, 2]\}$$
$$M(\{P3\}) = \{x : [-\inf, 1]\}$$
$$M(\{P4\}) = \{x : [-\inf, 2]\}$$
$$M(\{P5\}) = \{x : [3, \inf]\}$$
$$M(\{P6\}) = \{x : [5, \inf]\}$$
$$M(\{P7\}) = \{x : [-\inf, \inf]\}$$

Example 2:
Input:

```
{P0}
i := 1;
{P1}
j := 0;
{P2}
while (i <= 5) do
    [while_true]
    {P3}
    j := j + i;
    {P4}
    i := i + 1;
    {P5}
end [while_false]
{P6}
```

Output:
1. Create a system of fixed point equations.
Here, $M(\{P\})$ denotes the abstract state at location $\{P\}$. $M(\{P\})(x)$ denotes the abstract value of variable x at location $\{P\}$.

$$F_0(M) = \{i : [-\inf, \inf], j : [-\inf, \inf]\}$$

937

$$F_1(M) = \text{Interpret}(i := 1, M(\{P_0\}))$$
$$F_2(M) = \text{Interpret}(j := 0, M(\{P_1\}))$$
$$F_3(M) = \text{Filter}(i \leq 5, M(\{P_2\}) \sqcup M(\{P_5\}))$$
$$F_4(M) = \text{Interpret}(j := j + i, M(\{P_3\}))$$
$$F_5(M) = \text{Interpret}(i := i + 1, M(\{P_4\}))$$
$$F_6(M) = \text{Filter}(i > 5, M(\{P_2\}) \sqcup M(\{P_5\}))$$

2. Solve the fixed point equations using a worklist algorithm.
Initially, the map of program locations to abstract states looks like:

$$M(\{P_0\}) = \{i : \bot, j : \bot\}$$
$$M(\{P_1\}) = \{i : \bot, j : \bot\}$$
$$M(\{P_2\}) = \{i : \bot, j : \bot\}$$
$$M(\{P_3\}) = \{i : \bot, j : \bot\}$$
$$M(\{P_4\}) = \{i : \bot, j : \bot\}$$
$$M(\{P_5\}) = \{i : \bot, j : \bot\}$$
$$M(\{P_6\}) = \{i : \bot, j : \bot\}$$

The worklist W is $\{\{P0\}, \{P1\}, \{P2\}, \{P3\}, \{P4\}, \{P5\}, \{P6\}\}$.

- Pick $\{P0\}$ from W.
  - Remove $\{P0\}$ from W.
  - $M(\{P0\})$ is $\{i : \bot, j : \bot\}$.
  - Compute $F_0(M)$, and update the value of $M(\{P0\})$, resulting in $M(\{P0\}) = \{i : [-\inf, \inf], j : [-\inf, \inf]\}$.
  - $M(\{P0\})$ has changed, so add the program locations whose fixed point equations directly depend on $M(\{P0\})$ to W.
    - Add $\{P1\}$ to W.
  - W is now $\{\{P1\}, \{P2\}, \{P3\}, \{P4\}, \{P5\}, \{P6\}\}$.
- Pick $\{P1\}$ from W.
  - Remove $\{P1\}$ from W.
  - $M(\{P1\})$ is $\{i : \bot, j : \bot\}$.
  - Compute $F_1(M)$, and update the value of $M(\{P1\})$, resulting in $M(\{P1\}) = \{i : [1, 1], j : [-\inf, \inf]\}$, where
    - $M(\{P1\})(i) = [1, 1]$
    - $M(\{P1\})(j) = M(\{P0\})(j)$
  - $M(\{P1\})$ has changed, so add the program locations whose fixed point equations directly depend on $M(\{P1\})$ to W.
    - Add $\{P2\}$ to W.
  - W is now $\{\{P2\}, \{P3\}, \{P4\}, \{P5\}, \{P6\}\}$.
- Pick $\{P2\}$ from W.
  - Remove $\{P2\}$ from W.

938

- $M(\{P2\})$ is $\{i : \bot, j : \bot\}$.
- Compute $F_2(M)$ and update the value of $M(\{P2\})$, resulting in $M(\{P2\}) = \{i : [1,1], j : [0,0]\}$, where
  - $M(\{P2\})(i) = M(\{P1\})(i)$
  - $M(\{P2\})(j) = [0,0]$
- $M(\{P2\})$ has changed, so add the program locations whose fixed point equations directly depend on $M(\{P2\})$ to W.
  - Add $\{P3\}$ and $\{P6\}$ to W.
- W is now $\{\{P3\}, \{P4\}, \{P5\}, \{P6\}\}$.

- Pick $\{P3\}$ from W.
  - Remove $\{P3\}$ from W.
  - $M(\{P3\})$ is $\{i : \bot, j : \bot\}$.
  - Compute $F_3(M)$:
    - $M(\{P2\}) \sqcup M(\{P5\}) = \{i : [1,1], j : [0,0]\} \sqcup \{i : \bot, j : \bot\} = \{i : [1,1], j : [0,0]\}$.
    - Filtering $\{i : [1,1], j : [0,0]\}$ by $i \leq 5$ results in:
      - $S = \{i : [1,1], j : [0,0]\}$
  - Because $\{P3\}$ corresponds to a loop head, we widen $M(\{P3\})$ by S.
    - $M(\{P3\}) \nabla S$ results in $S' = \{i : [1,1], j : [0,0]\}$, where
      - $S'(i) = \bot \nabla [1,1]$
      - $S'(j) = \bot \nabla [0,0]$
  - Update $M(\{P3\})$ to $\{i : [1,1], j : [0,0]\}$.
  - $M(\{P3\})$ has changed, so add the program locations whose fixed point equations directly depend on $M(\{P3\})$ to W.
    - Add $\{P4\}$ to W.
  - W is now $\{\{P4\}, \{P5\}, \{P6\}\}$

- Pick $\{P4\}$ from W.
  - Remove $\{P4\}$ from W.
  - $M(\{P4\})$ is $\{i : \bot, j : \bot\}$.
  - Compute $F_4(M)$, and update the value of $M(\{P4\})$, resulting in $M(\{P4\}) = \{i : [1,1], j : [1,1]\}$, where
    - $M(\{P4\})(i) = M(\{P3\})(i) = [1,1]$
    - $M(\{P4\})(j) = M(\{P3\})(j) + M(\{P3\})(i) = [0,0] + [1,1] = [1,1]$
  - $M(\{P4\})$ has changed, so add the program locations whose fixed point equations directly depend on $M(\{P4\})$ to W.
    - Add $\{P5\}$ to W.
  - W is now $\{\{P5\}, \{P6\}\}$.

- Pick $\{P5\}$ from W.
  - Remove $\{P5\}$ from W.
  - $M(\{P5\})$ is $\{i : \bot, j : \bot\}$.
  - Compute $F_5(M)$, and update the value of $M(\{P5\})$, resulting in $M(\{P5\}) = \{i : [2,2], j : [1,1]\}$, where
    - $M(\{P5\})(i) = M(\{P4\})(i) + [1,1] = [1,1] + [1,1] = [2,2]$

939

    - $M(\{P5\})(j) = M(\{P4\})(j)$
  - $M(\{P5\})$ has changed, so add the program locations whose fixed point equations directly depend on $M(\{P5\})$ to W.
    - Add $\{P3\}$ and $\{P6\}$ to W.
  - W is now $\{\{P3\}, \{P6\}\}$.

- Pick $\{P3\}$ from W.
  - Remove $\{P3\}$ from W.
  - $M(\{P3\})$ is $\{i : [1, 1], j : [0, 0]\}$.
  - Compute $F_3(M)$:
    - $M(\{P2\}) \sqcup M(\{P5\}) = \{i : [1, 1], j : [0, 0]\} \sqcup \{i : [2, 2], j : [1, 1]\} = \{i : [1, 2], j : [0, 1]\}$
    - Filtering $\{i : [1, 2], j : [0, 1]\}$ by $i \leq 5$ results in:
      - $S = \{i : [1, 2], j : [0, 1]\}$
  - Because $\{P3\}$ corresponds to a loop head, we widen $M(\{P3\})$ by S.
    - $M(\{P3\})\nabla S$ results in $S' = \{i : [1, \text{inf}], j : [0, \text{inf}]\}$, where
      - $S'(i) = [1, 1]\nabla[1, 2] = [1, \text{inf}]$
      - $S'(j) = [0, 0]\nabla[0, 1] = [0, \text{inf}]$
  - Update $M(\{P3\})$ to $\{i : [1, \text{inf}], j : [0, \text{inf}]\}$.
  - $M(\{P3\})$ has changed, so add the program locations whose fixed point equations directly depend on $M(\{P3\})$ to W.
    - Add $\{P4\}$ to W.
  - W is now $\{\{P4\}, \{P6\}\}$

- Pick $\{P4\}$ from W.
  - Remove $\{P4\}$ from W.
  - $M(\{P4\})$ is $\{i : [1, 1], j : [1, 1]\}$.
  - Compute $F_4(M)$ and update the value of $M(\{P4\})$, resulting in $M(\{P4\}) = \{i : [1, \text{inf}], j : [1, \text{inf}]\}$, where
    - $M(\{P4\})(i) = M(\{P3\})(i) = [1, \text{inf}]$
    - $M(\{P4\})(j) = M(\{P3\})(j) + M(\{P3\})(i) = [0, \text{inf}] + [1, \text{inf}] = [1, \text{inf}]$
  - $M(\{P4\})$ has changed, so add the program locations whose fixed point equations directly depend on $M(\{P4\})$ to W.
    - Add $\{P5\}$ to W.
  - W is now $\{\{P5\}, \{P6\}\}$.

- Pick $\{P5\}$ from W.
  - Remove $\{P5\}$ from W.
  - $M(\{P5\})$ is $\{i : [2, 2], j : [1, 1]\}$.
  - Compute $F_5(M)$, and update the value of $M(\{P5\})$, resulting in $M(\{P5\}) = \{i : [2, \text{inf}], j : [1, \text{inf}]\}$, where
    - $M(\{P5\})(i) = M(\{P4\})(i) + [1, 1] = [1, \text{inf}] + [1, 1] = [2, \text{inf}]$
    - $M(\{P5\})(j) = M(\{P4\})(j) = [1, \text{inf}]$
  - $M(\{P5\})$ has changed, so add the program locations whose fixed point equations directly depend on $M(\{P5\})$ to W.
    - Add $\{P3\}$ and $\{P6\}$ to W.

- W is now $\{\{P3\}, \{P6\}\}$.

- Pick $\{P3\}$ from W.
  - Remove $\{P3\}$ from W.
  - $M(\{P3\})$ is $\{i : [1, \inf], j : [0, \inf]\}$.
  - Compute $F_3(M)$:
    - $M(\{P2\}) \sqcup M(\{P5\}) = \{i : [1, 1], j : [0, 0]\} \sqcup \{i : [2, \inf], j : [1, \inf]\} = \{i : [1, \inf], j : [0, \inf]\}$
    - Filtering $\{i : [1, \inf], j : [0, \inf]\}$ by $i \leq 5$ results in:
      - $S = \{i : [1, 5], j : [0, \inf]\}$
  - Because $\{P3\}$ corresponds to a loop head, we widen $M(\{P3\})$ by S.
    - $M(\{P3\})\nabla S$ results in $S' = \{i : [1, \inf], j : [0, \inf]\}$, where
      - $S'(i) = [1, \inf]\nabla[1, 5] = [1, \inf]$
      - $S'(j) = [0, \inf]\nabla[0, \inf] = [0, \inf]$
    - Now, $M(\{P3\}) = \{i : [1, \inf], j : [0, \inf]\}$.
  - $M(\{P3\})$ has not changed, so do not add anything to the worklist.
  - W is now $\{\{P6\}\}$.

- Pick $\{P6\}$ from W.
  - Remove $\{P6\}$ from W.
  - $M(\{P6\})$ is $\{i : \bot, j : \bot\}$.
  - Compute $F_6(M)$:
    - $M(\{P2\}) \sqcup M(\{P5\}) = \{i : [1, 1], j : [0, 0]\} \sqcup \{i : [2, \inf], j : [1, \inf]\} = \{i : [1, \inf], j : [0, \inf]\}$
    - Filtering $\{i : [1, \inf], j : [0, \inf]\}$ by $i > 5$ results in
      - $\{i : [6, \inf], j : [0, \inf]\}$
    - Now, $M(\{P6\}) = \{i : [6, \inf], j : [0, \inf]\}$
  - $M(\{P6\})$ has changed, so add the program locations whose fixed point equations directly depend on $M(\{P6\})$ to W.
    - According to the system of equations, there is no such location, so no location is added to W.
  - W is now $\{\}$.

The worklist is empty, meaning we've finished the analysis and M is

$$M(\{P_0\}) = \{i : [-\inf, \inf], j : [-\inf, \inf]\}$$
$$M(\{P_1\}) = \{i : [1, 1], j : [-\inf, \inf]\}$$
$$M(\{P_2\}) = \{i : [1, 1], j : [0, 0]\}$$
$$M(\{P_3\}) = \{i : [1, \inf], j : [0, \inf]\}$$
$$M(\{P_4\}) = \{i : [1, \inf], j : [1, \inf]\}$$
$$M(\{P_5\}) = \{i : [2, \inf], j : [1, \inf]\}$$
$$M(\{P_6\}) = \{i : [6, \inf], j : [0, \inf]\}$$

Example 3:
Input:

941

```
{P0}
y := 7;
{P1}
while (true) do
    [while_true]
    {P2}
    x := read();
    {P3}
    while (x <= y) do
        [while_true]
        {P4}
        x := x + 1;
        {P5}
    end [while_false]
    {P6}
end [while_false]
{P7}
```

1. Create a system of fixed point equations.
Here, $M(\{P\})$ denotes the abstract state at location $\{P\}$. $M(\{P\})(x)$ denotes the abstract value of variable x at location $\{P\}$.

$$F_0(M) = \{x : [-\inf, \inf], y : [-\inf, \inf]\}$$
$$F_1(M) = \text{Interpret}(y := 7, M(\{P_0\}))$$
$$F_2(M) = \text{Filter}(\text{true}, M(\{P_1\}) \sqcup M(\{P_6\}))$$
$$F_3(M) = \text{Interpret}(x := \text{read}(), M(\{P_2\}))$$
$$F_4(M) = \text{Filter}(x \leq y, M(\{P_3\}) \sqcup M(\{P_5\}))$$
$$F_5(M) = \text{Interpret}(x := x + 1, M(\{P_4\}))$$
$$F_6(M) = \text{Filter}(x > y, M(\{P_3\}) \sqcup M(\{P_5\}))$$
$$F_7(M) = \text{Filter}(\text{false}, M(\{P_1\}) \sqcup M(\{P_6\}))$$

2. Solve the fixed point equations using a worklist algorithm.
Initially, the map of program locations to abstract states looks like:

$$M(\{P0\}) = \{x : \bot, y : \bot\}$$
$$M(\{P1\}) = \{x : \bot, y : \bot\}$$
$$M(\{P2\}) = \{x : \bot, y : \bot\}$$
$$M(\{P3\}) = \{x : \bot, y : \bot\}$$
$$M(\{P4\}) = \{x : \bot, y : \bot\}$$
$$M(\{P5\}) = \{x : \bot, y : \bot\}$$
$$M(\{P6\}) = \{x : \bot, y : \bot\}$$
$$M(\{P7\}) = \{x : \bot, y : \bot\}$$

The worklist W is $\{P0\}, \{P1\}, \{P2\}, \{P3\}, \{P4\}, \{P5\}, \{P6\}, \{P7\}\}$.

- Pick $\{P0\}$ from W.

942

- Remove $\{P0\}$ from W.
- $M(\{P0\})$ is $\{i : \bot, j : \bot\}$.
- Compute $F_0(M)$, and update the value of $M(\{P0\})$, resulting in $M(\{P0\}) = \{x : [-\inf, \inf], y : [-\inf, \inf]\}$
- $M(\{P0\})$ has changed, so add the program locations whose fixed point equations directly depend on $M(\{P0\})$ to W.
  - Add $\{P1\}$ to W.
- W is now $\{\{P1\}, \{P2\}, \{P3\}, \{P4\}, \{P5\}, \{P6\}, \{P7\}\}$.

- Pick $\{P1\}$ from W.
  - Remove $\{P1\}$ from W.
  - $M(\{P1\})$ is $\{x : \bot, y : \bot\}$.
  - Compute $F_1(M)$ and update the value of $M(\{P1\})$, resulting in $M(\{P1\}) = \{x : [-\inf, \inf], y : [7, 7]\}$, where
    - $M(\{P1\})(x) = M(\{P0\})(x)$
    - $M(\{P1\})(y) = [7, 7]$
  - $M(\{P1\})$ has changed, so add the program locations whose fixed point equations directly depend on $M(\{P1\})$ to W.
    - Add $\{P2\}$ and $\{P7\}$ to W.
  - W is now $\{\{P2\}, \{P3\}, \{P4\}, \{P5\}, \{P6\}, \{P7\}\}$.

- Pick $\{P2\}$ from W.
  - Remove $\{P2\}$ from W.
  - $M(\{P2\})$ is $\{x : \bot, y : \bot\}$.
  - Compute $F_2(M)$:
    - $M(\{P1\}) \sqcup M(\{P6\}) = \{x : [-\inf, \inf], y : [7, 7]\} \sqcup \{x : \bot, y : \bot\} = \{x : [-\inf, \inf], y : [7, 7]\}$
    - Filtering $\{x : [-\inf, \inf], y : [7, 7]\}$ by true results in:
      - $S = \{x : [-\inf, \inf], y : [7, 7]\}$
  - Because $\{P2\}$ corresponds to a loop head, we widen $M(\{P2\})$ by S.
    - $M(\{P2\})\nabla S$ results in $S' = \{x : [-\inf, \inf], y : [7, 7]\}$, where
      - $S'(x) = \bot\nabla[-\inf, \inf] = [-\inf, \inf]$
      - $S'(y) = \bot\nabla[7, 7] = [7, 7]$
    - Update $M(\{P2\})$ to be $\{x : [-\inf, \inf], y : [7, 7]\}$.
  - $M(\{P2\})$ has changed, so add the program locations whose fixed point equations directly depend on $M(\{P2\})$ to W.
    - Add $\{P3\}$ to W.
  - W is now $\{\{P3\}, \{P4\}, \{P5\}, \{P6\}, \{P7\}\}$.

- Pick $\{P3\}$ from W.
  - Remove $\{P3\}$ from W.
  - $M(\{P3\})$ is $\{x : \bot, y : \bot\}$.
  - Compute $F_3(M)$, resulting in $M(\{P3\}) = \{x : [-\inf, \inf], y : [7, 7]\}$, where
    - $M(\{P3\})(x) = [-\inf, \inf]$, which is the result of interpreting $x := \text{read}()$.
    - $M(\{P3\})(y) = M(\{P2\})(y)$
  - $M(\{P3\})$ has changed, so add the program locations whose fixed point equations directly depend on $M(\{P3\})$ to W.

- Add $\{P4\}$ and $\{P6\}$ to W.
- W is now $\{\{P4\}, \{P5\}, \{P6\}, \{P7\}\}$.

- Pick $\{P4\}$ from W.
  - Remove $\{P4\}$ from W.
  - $M(\{P4\})$ is $\{x : \bot, y : \bot\}$
  - Compute $F_4(M)$:
    - $M(\{P3\}) \sqcup M(\{P5\}) = \{x : [-\inf, \inf], y : [7, 7]\} \sqcup \{x : \bot, y : \bot\} = \{x : [-\inf, \inf], y : [7, 7]\}$
    - Filtering $\{x : [-\inf, \inf], y : [7, 7]\}$ by $x \leq y$ results in:
      - $S = \{x : [-\inf, 7], y : [7, 7]\}$.
  - Because $\{P4\}$ corresponds to a loop head, we widen $M(\{P4\})$ by S.
    - $M(\{P4\})\nabla S$ results in $S' = \{x : [-\inf, 7], y : [7, 7]\}$, where
      - $S'(x) = \bot\nabla[-\inf, 7] = [-\inf, 7]$
      - $S'(y) = \bot\nabla[7, 7] = [7, 7]$
    - Update $M(\{P4\})$ to be $\{x : [-\inf, 7], y : [7, 7]\}$.
  - $M(\{P4\})$ has changed, so add the program locations whose fixed point equations directly depend on $M(\{P4\})$ to W.
    - Add $\{P5\}$ to W.
  - W is now $\{\{P5\}, \{P6\}, \{P7\}\}$.

- Pick $\{P5\}$ from W.
  - Remove $\{P5\}$ from W.
  - $M(\{P5\})$ is $\{x : \bot, y : \bot\}$.
  - Compute $F_5(M)$ and update the value of $M(\{P5\})$, resulting in $M(\{P5\}) = \{x : [-\inf, 8], y : [7, 7]\}$, where
    - $M(\{P5\})(x) = M(\{P4\})(x) + [1, 1] = [-\inf, 7] + [1, 1] = [-\inf, 8]$
    - $M(\{P5\})(y) = M(\{P4\})(y) = [7, 7]$
  - $M(\{P5\})$ has changed, so add the program locations whose fixed point equations directly depend on $M(\{P5\})$ to W.
    - Add $\{P4\}$ and $\{P6\}$ to W.
  - W is now $\{\{P4\}, \{P6\}, \{P7\}\}$.

- Pick $\{P4\}$ from W.
  - Remove $\{P4\}$ from W.
  - $M(\{P4\}) = \{x : [-\inf, 7], y : [7, 7]\}$.
  - Compute $F_4(M)$:
    - $M(\{P3\}) \sqcup M(\{P5\}) = \{x : [-\inf, \inf], y : [7, 7]\} \sqcup \{x : [-\inf, 8], y : [7, 7]\} = \{x : [-\inf, \inf], y : [7, 7]\}$
    - Filtering $\{x : [-\inf, \inf], y : [7, 7]\}$ by $x \leq y$ results in:
      - $S = \{x : [-\inf, 7], y : [7, 7]\}$.
  - Because $\{P4\}$ corresponds to a loop head, we widen $M(\{P4\})$ by S.
    - $M(\{P4\})\nabla S$ results in $S' = \{x : [-\inf, 7], y : [7, 7]\}$, where
      - $S'(x) = [-\inf, 7]\nabla[-\inf, 7] = [-\inf, 7]$
      - $S'(y) = [7, 7]\nabla[7, 7] = [7, 7]$
    - Update $M(\{P4\})$ to be $\{x : [-\inf, 7], y : [7, 7]\}$.

944

- $M(\{P4\})$ has not changed, so we don't add anything to W.
- W is now $\{\{P6\},\{P7\}\}$.

- Pick $\{P6\}$ from W.
  - Remove $\{P6\}$ from W.
  - $M(\{P6\}) = \{x : \bot, y : \bot\}$.
  - Compute $F_6(M)$:
    - $M(\{P3\}) \sqcup M(\{P5\}) = \{x : [-\inf, \inf], y : [7,7]\} \sqcup \{x : [-\inf, 8], y : [7,7]\} = \{x : [-\inf, \inf], y : [7,7]\}$
    - Filtering $\{x : [-\inf, \inf], y : [7,7]\}$ by $x > y$ results in $\{x : [8, \inf], y : [7,7]\}$.
    - Update $M(\{P6\})$ to be $\{x : [8, \inf], y : [7,7]\}$.
  - $M(\{P6\})$ has changed, so add the program locations whose fixed point equations directly depend on $M(\{P6\})$ to W.
    - Add $\{P2\}$ and $\{P7\}$ to W.
  - W is now $\{\{P2\},\{P7\}\}$.

- Pick $\{P2\}$ from W.
  - Remove $\{P2\}$ from W.
  - $M(\{P2\}) = \{x : [-\inf, \inf], y : [7,7]\}$.
  - Compute $F_2(M)$:
    - $M(\{P1\}) \sqcup M(\{P6\}) = \{x : [-\inf, \inf], y : [7,7]\} \sqcup \{x : [8, \inf], y : [7,7]\} = \{x : [-\inf, \inf], y : [7,7]\}$
    - Filtering $\{x : [-\inf, \inf], y : [7,7]\}$ by true results in:
      - $S = \{x : [-\inf, \inf], y : [7,7]\}$.
  - Because $\{P2\}$ corresponds to a loop head, we widen $M(\{P2\})$ by S.
    - $M(\{P2\}) \nabla S$ results in $S' = \{x : [-\inf, \inf], y : [7,7]\}$, where
      - $S'(x) = [-\inf, \inf] \nabla [-\inf, \inf] = [-\inf, \inf]$
      - $S'(y) = [7,7] \nabla [7,7] = [7,7]$
    - Update $M(\{P2\})$ to be $\{x : [-\inf, \inf], y : [7,7]\}$.
  - $M(\{P2\})$ has not changed, so don't add anything to W.
  - W is now $\{\{P7\}\}$.

- Pick $\{P7\}$ from W.
  - Remove $\{P7\}$ from W.
  - $M(\{P7\}) = \{x : \bot, y : \bot\}$.
  - Compute $F_7(M)$
    - $M(\{P1\}) \sqcup M(\{P6\}) = \{x : [-\inf, \inf], y : [7,7]\} \sqcup \{x : [8, \inf], y : [7,7]\} = \{x : [-\inf, \inf], y : [7,7]\}$.
    - Filtering $\{x : [-\inf, \inf], y : [7,7]\}$ by false results in:
      - $S = \{x : \bot, y : \bot\}$.
    - Update $M(\{P7\})$ to be $\{x : \bot, y : \bot\}$.
  - $M(\{P7\})$ has not changed, so don't add anything to W.
  - W is now $\{\}$.

The worklist is empty, meaning we've finished the analysis and M is

$$M(\{P_0\}) = \{x : [-\inf, \inf], y : [-\inf, \inf]\}$$

945

$$M(\{P_1\}) = \{x : [-\inf, \inf], y : [7, 7]\}$$
$$M(\{P_2\}) = \{x : [-\inf, \inf], y : [7, 7]\}$$
$$M(\{P_3\}) = \{x : [-\inf, \inf], y : [7, 7]\}$$
$$M(\{P_4\}) = \{x : [-\inf, 7], y : [7, 7]\}$$
$$M(\{P_5\}) = \{x : [-\inf, 8], y : [7, 7]\}$$
$$M(\{P_6\}) = \{x : [8, \inf], y : [7, 7]\}$$
$$M(\{P_7\}) = \{x : \perp, y : \perp\}$$

Now, please solve this, outputting the intermediary steps you take:
[Input Program]

B Case Study

In this section we will provide the case study of LLM's output for each behavior.

B.1 Misunderstanding of Abstract Interpretation

For program 'as2013-hybrid', the LLM has

```
Now, widen the input state {i: [0, inf], j: [10, inf]} with the new state {i:
↪  [0,9], j: [10, inf]}

Widening:

- For i: [0, inf] $\nabla$ [0,9] = [0, inf]

- For j: [10, inf] $\nabla$ [10, inf] = [10, inf]

- So, widened state: {i: [0, inf], j: [10, inf]}

This is the same as the input state to this iteration, so we've reached a
↪  fixed point for the outer loop.

Now, filter this fixed point by the negation of the loop guard, which is false
↪  (since the loop condition is true always).

Filtering {i: [0, inf], j: [10, inf]} by false results in bottom ($\bot$)

- Side-effect: abstract state at {P13} is {i: $\bot$, j: $\bot$}
```

Here, we can see that at the last program location, the LLM misunderstands the definition and wrongly output the result as bottom ($\perp$).