ABSINT-AI: LANGUAGE MODELS FOR ABSTRACT INTERPRETATION

Michael Wang
MIT CSAIL
mi27950@mit.edu

Kexin Pei
University of Chicago
kpei@uchicago.edu

Armando Solar-Lezama
MIT CSAIL
asolar@csail.mit.edu

ABSTRACT

Static program analysis is a popular technique in software engineering. Traditional static analysis algorithms treat programs as sets of logical statements with well-defined semantics. These traditional analyzers can provide guarantees of their performance, such as guaranteeing that they will never miss a bug. However, they leave out lots of very rich information such as variable and field names. Language models for code on the other hand, take full advantage of information such as variable names, but it is extremely difficult to provide guarantees of their output. In this work, we present ABSINT-AI, a language model augmented static analyzer based on abstract interpretation that combines the best of both worlds. Using a language model in ABSINT-AI achieves up to a 70% decrease in false positives for bug detection while providing guarantees of never missing a bug.

1 INTRODUCTION

As dynamic languages like JavaScript find their way into more backend applications with strong performance requirements, there has been a growing interest in compiling them down to more optimal forms ang; Serrano (2022); Chandra et al. (2016). An important obstacle for these approaches is the difficulty of performing static program analysis on these languages due to their dynamic behavior and extensive use of complex heap allocated data Feldthaus et al. (2013); Antal et al. (2023); Sridharan et al. (2012). This is a problem because sound program analysis is an essential element of compiler optimization Hind (2001); Schneck (1973). Soundness ensures that the analysis captures all possible runtime behaviors of the program; without it, compilers cannot guarantee the safety of specific transformations.

In this paper, we show that Language Models (LMs) can play an important role in supporting sound program analysis for dynamic languages. Our effort focuses on JavaScript because it is widely used and representative of other dynamic languages in its extensive use of high order functions, closures and dynamically created objects represented as heap allocated dictionaries. Our approach is not to replace traditional analysis with LMs, but rather to leverage the extensive knowledge captured by LMs to improve the precision of traditional static analysis.

It is not a new idea to combine machine learning and program analysis. One technique that has become popular recently is to augment traditional static analysis with Language Models applied specifically for areas that are difficult to analyze. LLift Li et al. (2024a) for example, uses LM's to analyze use-before-initialization cases that are too expensive for traditional static analysis. While this improves the overall performance of the analyzers, the introduction of a neural network almost ubiquitously loses any guarantees of soundness. To the best of our knowledge, LMs have not been used in program analysis without losing soundness guarantees.

Our approach instead seeks to preserve the strong guarantees provided by traditional static analysis techniques while addressing some of their major limitations. Static analysis techniques analyze programs by treating them as sets of logical statements with well-defined semantics Cousot & Cousot (1977). This type of analysis can provide guarantees of soundness, but these methods leave out a lot of rich information, such as variable names, comments, general programming design patterns, and background knowledge. LM's on the other hand, are able to take advantage of this information very well, but lack the robustness and rigor of traditional static analysis techniques. For example, changing variable names has been shown to have a drastic impact on the performance of these models Zeng

et al. (2022); Srikant et al. (2021). ABSINT-AI combines the best of both worlds by using LM's to provide background information to a static analyzer without losing soundness guarantees.

Static analysis algorithms achieve scalability and soundness by using *abstractions* in their analysis. Programs often manipulate unbounded resources (e.g., integers, heap structures). Abstractions merge a potentially infinite set of objects into a single *summary* object to ensure convergence and for scalability. A key challenge is choosing *what* to abstract in the target program to ensure convergence while retaining as much important information as possible. In this work, we use an LM to decide what should be abstracted in the target program.

Deciding what to abstract has two important heuristics: (1) deciding which objects are similar and can be merged together and (2) deciding what details are unimportant to the analysis at hand and can be abstracted away. As an example for (1), an abstraction based on an object's allocation site is based on the intuition that objects allocated at the same program location often represent the same conceptual entity in the program Chase et al. (1990).

These are two areas where natural language and knowledge of general concepts is very helpful. For example, in a learning management system, it may make sense to group all the teachers into one summary object and all the students into another (an example can be seen in Figure 1a). These are things that are intuitive to an LM, but are difficult to encode symbolically into a static analysis. We use the LM *only* to determine what parts of the input program are suitable for applying abstractions and what areas should be kept concrete. The static analysis and abstraction process itself is hidden from the LM. This way, the LM *only* brings background knowledge.

Achieving soundness with ABSINT-AI. An important point is that ABSINT-AI only uses the LM to decide *where* to apply its abstractions. The LM can go wrong in two ways, neither of which results in unsoundness. (1), the LM *decides something should be abstracted when it should not.* In this case, our abstractions lose precision and we report more false positives than necessary. However, because each abstraction is a superset of the concrete state, this affects *precision*, but not soundness. (2), The LM *decides something should not be abstracted when it should.* This can impede convergence and scalability, as abstractions are necessary to converge, but does not affect soundness.

In summary, our contributions are:

- A novel static analysis, ABSINT-AI, that integrates background information and variable names using LM's without losing soundness.
- An evaluation showing that incorporating an LM increases precision up to 70%.

2 MOTIVATING EXAMPLE

Figure 1a shows a basic example of a learning management system, and illustrates how background knowledge can help with forming abstractions. First, consider the most common method of summarization: using an object's allocation site.

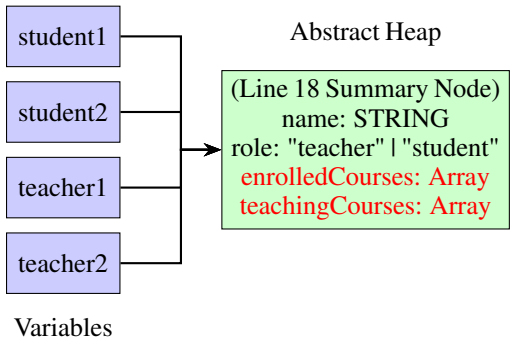
The `createUser` function defined on line 14 is used to populate the application with users, with the object representing the new user being allocated on line 18. In a real-world application, there could be an infinite number of objects being allocated, so one way of making a static analysis tractable is to create a single *summary node* for all objects allocated by the statement on line 18. However, this results in `student1, student2, teacher1, and teacher2` all pointing to the same summary node, illustrated by Figure 1b. This is very imprecise, as both the field accesses to `teachingCourses` on line 10 and the access to `enrolledCourses` on line 3 will be potential invalid accesses. There have been many previous works on heuristics to decide how to create summary nodes and different predicates to improve the precision Kanvar & Khedker (2016), such as using the call stack at as well as heap connectivity to distinguish objects.

As a reader of the code, however, one abstraction that would make sense is to have one summary node for teachers, and another summary node for students. Intuitively, programmers are likely to write their code based on the abstract concepts their objects represent. A more intuitive abstraction may be something similar to the one in Figure 1c, where all the objects representing teachers point to one summary node, and all objects representing students point to another summary node. This type of background knowledge and natural language hints are often very difficult to incorporate heuristics for

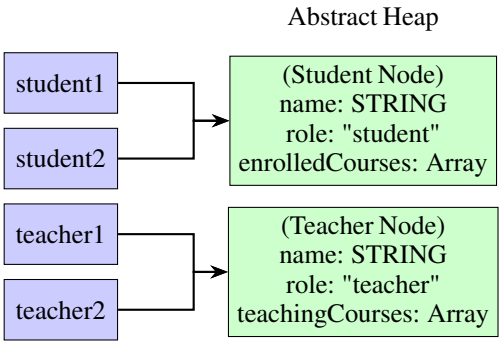
```
// Sample user data
function enrollStudent(student, courseId) {
    if (student.role === "student") {
        student.enrolledCourses.push(courseId);
    }
}

function addCourse(instructor, courseId) {
    if (instructor.role !== "teacher") {
        instructor.teachingCourses.push(courseId);
    }
}
// Create a new user
function createUser(name, email, role) {
    if (!["student", "teacher"].includes(role)) {
      throw new Error("Invalid role specified.");
    }
    const newUser = {
      name:name,
      role:role,
    };
    // Add role-specific fields
    if (role === "student") {
      newUser.enrolledCourses = [];
    } else if (role === "teacher") {
      newUser.teachingCourses = [];
    }
}
//create students
const student1 = createUser("Alice", "student");
const student2 = createUser("John", "student");
//create teachers
const teacher1 = createUser("Dr. Bob", "teacher");
const teacher2 = createUser("Dr. Jane", "teacher");
//assign students
enrollStudent(student1, "101");
enrollStudent(student2, "102");
//assign teachers
assignInstructor(teacher1, "101");
assignInstructor(teacher2, "102");
...
```

(a) A learning-management-system application where incorporating background information into abstractions would be helpful.



(b) An allocation-site based abstraction for Figure 1a. Fields that may or may not exist are in red.



(c) An abstraction for Figure 1a which takes into account variable names and background knowledge.

Figure 1: A learning-management-system application and how summarization based on allocation sites differ from summarization based on natural language and background knowledge.

summarization. In this work, we use an LM to decide which objects should be merged into a summary node and which parts of the object need further summarization.

3 METHODOLOGY

ABSINT-AI is based on traditional abstract interpretation, but queries an LM to decide how to merge summary nodes at key points in the analysis. In this section, we describe the abstract interpretation at a high level as well as how we query the LM. A high-level overview of the architecture of ABSINT-AI can be found in Figure 2.

3.1 ABSTRACT INTERPRETATION

In this section, we briefly describe our abstract interpretation at a high level. Abstract interpretation requires an abstract domain as well as modeling of the heap. In this section, we briefly describe our abstract domain, our two-level representation of the heap, and when we invoke the LM for summarization. The full analysis supports prototypal inheritance, recursion, loops, and closures. Additional details can be found in Appendix A. **Abstract Domain.** Our abstract domain keeps track of heap objects using concrete nodes and summary nodes. Summary nodes represent a set of possible concrete nodes. Each node is a dictionary from primitive or abstract values to other values. Our domain of primitive values is based off of TAJS Jensen et al. (2009), one of the first abstract interpretation based analyses for Javascript. Additional details on our abstract domain can be found in Appendix A. The most important runtime decision of ABSINT-AI is deciding when summarize heap nodes. We keep two separate heap structures, referred to as the local heap and global heap.

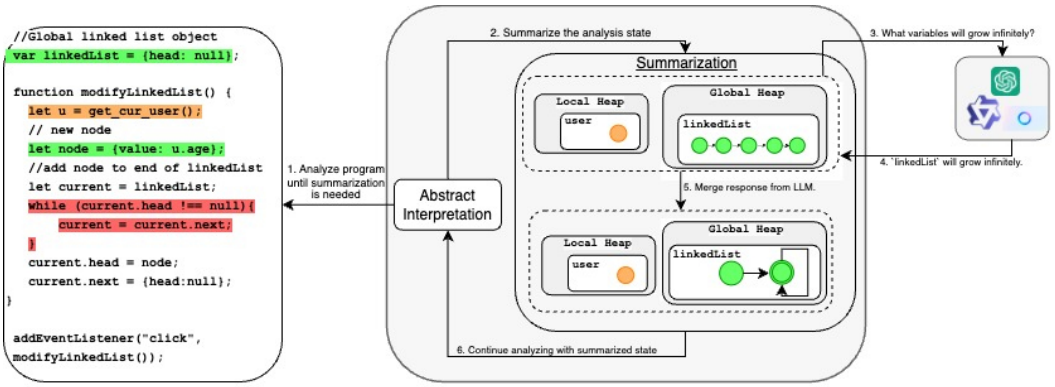


Figure 2: An overview of ABSINT-AI's architecture. Objects stored in the local heap are highlighted in orange. Objects stored in the global heap are highlighted in green. ABSINT-AI calls an external LM during the summarization process to simplify the local and global heaps.

Local heap. The local heap is used for precise representation for objects within local procedures, such as a local object allocation in a function call. It is flow-sensitive Kildall (1973), taking into account the order of statements. For example, in Figure 7, `obj` on line 4 is tracked in the local heap. After line 5 is executed, `obj.f` will be 2.

Global heap. The global heap is a much less precise representation for objects that are accessed and manipulated by multiple functions. The global heap captures all possible relationships between globally visible objects at any point in the execution. The global heap is motivated by flow-insensitive analysis Weihl (1980); Cousot & Cousot (1977). This has two benefits: (1) It is much cheaper, as we don't have to keep track of a separate heap for each program location, and (2) it allows different functions to be analyzed independently; the global heap considers all the possible heap states at the point when the function is invoked, and the analysis of the function can reveal if any additional relationships need to be added to the global heap. Summarization only happens in the global heap.

We draw a distinction between the local and global heap because JavaScript programs tend to be reactive, with execution driven largely by external events. This has important implications for analysis, because the analysis can't assume the program will simply execute starting at the beginning from a well defined initial state. Take the example in Figure 7, where `inc_global` is invoked by an event handler and must be executed at least 10 times in order to trigger the bug on line 11. Keeping two separate heaps allows us to to track global dependencies while not losing precision for local procedures. At the end of each iteration on the program, ABSINT-AI invokes an LM to summarize the global state. If it remains unchanged from the last iteration, we terminate the analysis.

Loop summarization. A key challenge in abstract interpretation is knowing when to terminate when analyzing potentially unbounded loops. Typically, abstract interpretation analyzes the loop body and summarizes the abstract state at the end of the loop until the analysis state reaches a fixpoint.

Take the example in Figure 8, adapted from the original example in Figure 1a. There is an unbounded loop on line 3, and two distinct objects are pushed to the `users` array. In order to terminate, this requires summarizing the `users` array as well as the objects created in the loop. At the end of each iteration of an unbounded loop, ABSINT-AI invokes an LM to summarize the state as described in Section 3.2. If the state remains unchanged from the last iteration of the loop after summarization, ABSINT-AI exits the loop and continues analyzing the rest of the program.

3.2 SUMMARIZATION WITH LMS

In this section, we describe how we use LM's to summarize the abstract heap. At a high level, we ask the LM for the variables that are important to summarize. As explained in the introduction, using the LM to identify the variables to summarize does not lose soundness, and incorporates additional information such as variable names which are difficult to encode in traditional heuristics for summarization.

We split our prompting into two parts. First, we ask for any variable names that need to be summarized. Next, if the variable is an object, we ask for any fields that need to be summarized. For each step,

Algorithm 1 Variable Selection with LM

Require: Program P, Analysis state S, Prompt Template T
Identify all variables V_{all} in S
Construct query Q using T, V_{all}, and S
Send Q to LM and receive response R
Extract V_{abstract} from response R
for each v in V_{abstract} **do**
  if v in V_{all} **then**
    summarize(v)
  end if
end for

Algorithm 2 Object summarization with LM

Require: Program P, Analysis state S, Object O, Prompt Template T
Construct query Q using T, O, and S
Send Q to LM and receive response R
if R == "ALL FIELDS" **then**
  summarize(O)
else
  Extract fields F_{all} from R
  for each f in F_{all} **do**
    summarize($O.f$)
  end for
end if

```
1  let userId = 100; // userId abstracted to NUMBER.
2  let names = {100: "Jane"}
3  names[userId]; // False positive
```

Figure 3: An example of a false positive due to `userId` getting abstracted to the abstract NUMBER type.

we give the LM some instructions on some heuristics to use in its decisions. All the prompts for this section can be found in Appendix B.

Variable selection. First, we ask the LM for high level information about what variables should be summarized. Our procedure can be seen in Algorithm 1. The exact prompt can be seen in Figure 10 in Appendix B. At a high level, we provide the LM with the state of the code as well as the state of the current variables, and ask for which variables should be summarized. For each variable that is returned, we merge all primitive values (if there are integers, we summarize it to the INTEGER type etc.). If there are objects, we first join all the fields and values as a union into a summary node, and query the LM further for what parts of the object to summarized.

Object summarization. If an object needs to be summarized, we use the procedure in Algorithm 2 to determine which fields need to be summarized. If the LM returns all fields, we summarize all fields and values recursively. If not, we extract the fields that should be summarized and only summarized the values corresponding to the selected fields. The exact prompts can be found in Figures 11 and 12 in Appendix B.

3.3 Detecting type errors.

As a downstream task to test the precision of AbsInt-AI, we detect the following situations:

- Accessing a property of `null` or `undefined`.
- Reading an absent property of an object.

Abstracting unnecessarily can lead to false positives. Take the example in Figure 3. If `userId` on line 1 gets abstracted to the abstract NUMBER type, then the object access on line 3 is reported as a possible read of an absent property. `userId` could take the value of all possible numbers, but `names` only has the the property `100`.

Intersection of multiple runs. Different abstraction choices within the program lead to different sets of reported bugs. For example, if we run AbsInt-AI on the program in Figure 3 multiple times, it might not decide to abstract `userId` every single time. However, soundness guarantees ensures that we will never miss any true bugs; as long as a bug is not present in *any single run*, we know it is not a true bug. This allows us to improve precision by taking the intersection of reported bugs from several runs (similar to self-consistency Wang et al. (2022b)) while still guaranteeing soundness.

4 Implementation

Dataset. We collected 12 Javascript programs from the Big Code dataset Raychev et al. (2016). We filtered for programs that were self-contained and did not use builtins excessively, as this greatly increases the imprecision of the analysis (`Math.floor`, for example, requires modeling the `Math`

Table 1: Each program and a small description.

Program	#Lines	Description
CGOL.js	65	Conway's Game of Life.
equal.js	304	Methods of comparing equality.
base64.js	158	base64 encoding and decoding.
datepattern.js	91	Testing date string equality
confetti.js	400	Confetti animations in the DOM.
pong.js	243	Pong game in the DOM.
snake_game.js	102	Snake game in the DOM.
books.js	504	A library for storing books.
FlashSort.js	84	Flash Sort.
math_sprint.js	345	Math calculations in the DOM.
drawing-app.js	442	A drawing app in the DOM.
TimSort.js	113	Tim Sort.

Table 2: Overall performance across the Dataset. #FP stands for False Positives. Fewer is better.

	Model	#FP↓	% Improve
Baselines	Baseline	271	0%
	Lower Bound	147	100%
Mean	Llama-3.3	204	54.0%
	GPT-4o-mini	216	44.3%
	Qwen2.5	205	53.2%
Intersection	Llama-3.3	184	70.1%
	GPT-4o-mini	183	70.9%
	Qwen2.5	189	66.1%
	Full Intersection	162	87.9%

library to analyze precisely). A description of the dataset can be found in Table 1. Each programs has unbounded loops or a global state which requires some form of abstraction in order to converge.

Implementation. We implemented ABSINT-AI in 8049 lines of Python, and use Espree brettz9 to parse the Javascript into an AST. We conducted the experiments on a Linux server with two AMD EPYC 7763 64-Core Processors, 128 cores, 1024GB RAM, and 4 NVIDIA RTX 6000 Ada Generation GPUs.

4.1 BASELINES

Allocation-site based abstractions. Our baseline that we compare against is an abstraction which merges all objects allocated at the same location in the code into one object. Details of our baseline can be found in Appendix C. At a high level, whenever we encounter an unbounded loop and at the end of a full iteration, we keep track of any addresses that are modified and merge them together with other objects allocated at the same location until the abstract state reaches a fixpoint.

Lower bound. There are additional sources of imprecision and false positives beyond just our abstractions such as lack of builtin modeling. For example, `Math.floor` is reported as an access to an uninitialized object if the `Math` library is not included in the static analyzer.

To get a lower bound on the number of false positives returned (lower is better), we perform a single run executing each loop once and computing no abstractions, and only running once on the whole program. This is not sound, but returns the number of bugs from sources like unmodeled builtins and DOM accesses. The number of false positives returned is the lower bound that a perfect abstraction scheme can return.

5 EVALUATION

We aim to answer the following questions:

- **RQ1: Performance.** How well does using an LM perform compared to the baseline? How sensitive is this to the choice of LM? How does taking the intersection improve performance?
- **RQ2: Runtime.** How does the runtime of using an LM compare to using allocation-site based abstractions?
- **RQ3: Guided vs. Direct.** Does using the LM as a high-level guide for program analysis work better than having the LM directly predict the abstraction?
- **RQ4: Interactivity.** Can the LM take advantage of comments in the code to improve precision?

5.1 RQ1: PERFORMANCE.

We tested three different models, GPT-4o-mini, Qwen2.5-72B Instruct, and Llama-3.3-70B Instruct. We ran each model 10 times across the dataset listed in Table 1, and measure the performance as an improvement in percentage from the baseline to the lower bound for the number of buggy lines reported.

Each model performs well, with the mean performance resulting 44-54% compared to the baseline of using allocation sites. Each LM performs either about the same or better than the baseline across all

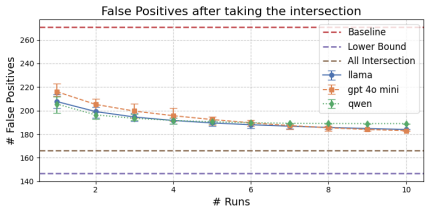


Figure 4: Performance of taking the intersection. The X-axis denotes how many runs, and the Y-axis denotes the number of false positives reported by taking the intersection of the number of runs.

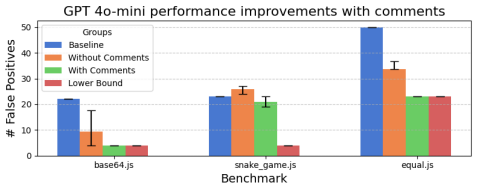


Figure 5: Performance improvements by adding comments.

Table 3: Comparison of the average runtime in seconds of using LM's compared to the baseline.

Benchmark	Baseline	Llama3.3	4o-mini	Qwen
CGOL.js	3	151	81	166
equal.js	11093	82	88	91
base64.js	15	331	250	605
datepattern.js	1	384	146	426
confetti.js	3	2510	498	2516
pong.js	1	307	202	387
snake_game.js	1	313	390	833
books.js	3	643	139	585
FlashSort.js	18	1075	466	1348
math_sprint.js	4	1671	769	1521
drawing-app.js	3	348	194	503
TimSort.js	32	1408	861	1846

Table 4: Directly predicting the abstraction does not converge on more than half the samples.

Benchmark	Llama-3.3		4o-mini		Qwen	
	Guide	Dir.	Guide	Dir	Guide	Dir.
CGOL.js	✓	✓	✓	✗	✓	✗
equal.js	✓	✓	✓	✓	✓	✓
base64.js	✓	✗	✓	✗	✓	✗
datepattern.js	✓	✓	✓	✗	✓	✗
confetti.js	✓	✓	✓	✗	✓	✓
pong.js	✓	✗	✓	✗	✓	✗
snake_game.js	✓	✓	✓	✓	✓	✓
books.js	✓	✗	✓	✗	✓	✗
FlashSort.js	✓	✗	✓	✗	✓	✗
math_sprint.js	✓	✗	✓	✗	✓	✗
drawing-app.js	✓	✗	✓	✗	✓	✗
TimSort.js	✓	✓	✓	✗	✓	✗

```
1  // The snake MUST be abstracted!!
2  var snake = [{x:10,y:10}]
```

Figure 6: A comment being added to guide the abstraction.

benchmarks. All three models perform well, indicating that our result is generalizable across different model architectures.

Intersection. As stated in Section 3.3, a benefit of soundness is being able to take the intersection of all of the positives. The results of this can be seen in Figure 4. As expected, the LM frequently makes different abstraction decisions. We also took the intersection of all bugs from the 10 runs for each model as well as taking the full intersection of all 30 runs. Most notably, taking the intersection of all the runs together leads to an 87.9% improvement in precision.

5.2 RQ2: Runtime.

Running an LM is notably slower than the baseline, as querying the LM takes much longer than a simple merge. Table 3 shows the average runtime performance of each LM in seconds compared to the baseline. However, An interesting case is `equal.js`, where using an LM is actually much faster than the baseline. This is due to `equal.js` having an object with many functions that get merged together with the baseline, leading to a state explosion in the abstract space. The LMs are able to come up with an abstraction that avoid this state explosion, leading to an *improvement* in runtime. GPT-4o-mini is notably faster, likely due to better hardware. The longest benchmark is `TimSort.js`, where it still finishes in under 15 minutes.

5.3 RQ3: Guided vs. Direct.

AbsInt-AI uses the LM to provide high-level information about the target program to determine *what* to abstract, but the LM does not perform the abstraction itself. One question is whether using the LM to perform the abstraction directly and then verifying the abstraction's validity would perform better. This tests the LM's capability of performing the analysis itself, rather than providing high-level information. We define not converging if an unbounded loop executes more than 10 times without

reaching a fixpoint, or if ABSINT-AI runs over the entire program more than 25 times. The prompts used for direct abstraction can be found in Appendix D.

Table 4 shows the difference between using the LMs to provide high level guidance compared to directly predicting the abstraction output. Directly performing the abstraction itself results in the LM not being able to converge in half the benchmarks. This supports the notion that LM's are best suited for providing high level guidance, and leaving the abstractions themselves to the symbolic program analysis.

5.4 RQ4: INTERACTIVITY.

There have been several past works on user-guided program analysis, where certain analysis choices are made by the analysis user rather than the analysis writer Mangal et al. (2015). However, this requires the user to learn a specific language and configuration for the analysis tool. In SLEEK Chin et al. (2011), for instance, the user must provide inductive predicates as separation logic formulas to describe the shape of data structures. Using an LM for the abstractions allows users to provide information using natural language comments, allowing for a more intuitive user interface.

To show this, we took three examples from our benchmark where the LM had some trouble forming abstractions, and annotated them with comments detailing what should and shouldn't be abstracted. An example of a comment being added can be seen in Figure 6. In all 3 examples, adding comments notably reduced the number of false positives returned, as show in Figure 5.

6 LIMITATIONS AND FUTURE WORK

Scalability. Querying an LM whenever summarization is needed is very expensive. Larger programs with lots of loops could take prohibitively long. In the future, we will investigate combining LM's with traditional heuristics to improve speed.

More advanced baselines. There is a rich literature on improving heap abstractions with additional predicates and heuristics Kanvar & Khedker (2016), and it has been shown that using only allocation sites is very imprecise Liang et al. (2010). It is possible that incorporating these would increase the false positive rate more than using an LM. We will leave investigating these to future work.

7 RELATED WORK

LMs for program analysis. LMs have been used for many program analysis task such as type inference Peng et al. (2023); Wei et al. (2023); Wang et al. (2023b), fuzzing Xia et al. (2024); Yang et al. (2023b;a); Deng et al. (2023), vulnerability detection Mathews et al. (2024); Liu et al. (2023), resource leak detection Wang et al. (2023a); Mohajer et al. (2023), code summarization Cai et al. (2023); Geng et al. (2024); Ahmed et al. (2024); Wang et al. (2022a), and fault localisation Wu et al. (2023). They differ from traditional machine learning applied to code in that they use pre-trained language models, and typically do not need large amounts of training data. However, they have not been applied to static analysis while maintaining soundness guarantees.

Neurosymbolic approaches for program analysis. LLift Li et al. (2024a) specifically applies LM's on the Linux kernel to detect Use-Before-Initialization. They run UBITect Zhai et al. (2020), which returns many possible false positives, and apply an LM to filter out false positives that cannot be handled easily by static analysis. IRIS Li et al. (2024b) augments CodeQL Avgustinov et al. (2016) with an LM to improve taint analysis. InferROI Wang et al. (2024) uses LM's to augment a static analysis to detect resource leaks in Java programs. While all of these approaches use LMs to improve analysis precision, the introduction of a neural network loses soundness.

8 CONCLUSION

We propose a method to augment static analyzers with LM's to provide background information and take advantage of variable names. We present ABSINT-AI as a proof-of-concept and an evaluation showing that augmenting static analysis with LM's can have a dramatic improvement on the precision without losing soundness guarantees.

REFERENCES

URL https://angular.dev/tools/cli/aot-compiler.

Toufique Ahmed, Kunal Suresh Pai, Premkumar Devanbu, and Earl T. Barr. Automatic semantic augmentation of language model prompts (for code summarization), 2024.

Gábor Antal, Péter Hegedűs, Zoltán Herczeg, Gábor Lóki, and Rudolf Ferenc. Is javascript call graph extraction solved yet? a comparative study of static and dynamic tools. *IEEE Access*, 11: 25266–25284, 2023. doi: 10.1109/ACCESS.2023.3255984.

Pavel Avgustinov, Oege de Moor, Michael Peyton Jones, and Max Schäfer. Ql: Object-oriented queries on relational data. In *European Conference on Object-Oriented Programming*, 2016. URL https://api.semanticscholar.org/CorpusID:13385963.

brettz9. Brettz9/espree: An esprima-compatible javascript parser. URL https://github.com/brettz9/espree.

Yufan Cai, Yun Lin, Chenyan Liu, Jinglian Wu, Yifan Zhang, Yiming Liu, Yeyun Gong, and Jin Song Dong. On-the-fly adapting code summarization on trainable cost-effective language models. In A. Oh, T. Neumann, A. Globerson, K. Saenko, M. Hardt, and S. Levine (eds.), *Advances in Neural Information Processing Systems*, volume 36, pp. 56660–56672. Curran Associates, Inc., 2023.

Satish Chandra, Colin S Gordon, Jean-Baptiste Jeannin, Cole Schlesinger, Manu Sridharan, Frank Tip, and Youngil Choi. Type inference for static compilation of javascript. *ACM SIGPLAN Notices*, 51(10):410–429, 2016.

David R Chase, Mark Wegman, and F Kenneth Zadeck. Analysis of pointers and structures. *ACM SIGPLAN Notices*, 25(6):296–310, 1990.

Wei-Ngan Chin, Cristina David, and Cristian Gherghina. A hip and sleek verification system. In *Proceedings of the ACM international conference companion on Object oriented programming systems languages and applications companion*, pp. 9–10, 2011.

Patrick Cousot and Radhia Cousot. 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*, pp. 238–252, 1977.

Yinlin Deng, Chunqiu Steven Xia, Haoran Peng, Chenyuan Yang, and Lingming Zhang. Large language models are zero-shot fuzzers: Fuzzing deep-learning libraries via large language models, 2023.

Asger Feldthaus, Max Schäfer, Manu Sridharan, Julian Dolby, and Frank Tip. Efficient construction of approximate call graphs for javascript ide services. In *2013 35th International Conference on Software Engineering (ICSE)*, pp. 752–761, 2013. doi: 10.1109/ICSE.2013.6606621.

Mingyang Geng, Shangwen Wang, Dezun Dong, Haotian Wang, Ge Li, Zhi Jin, Xiaoguang Mao, and Xiangke Liao. Large language models are few-shot summarizers: Multi-intent comment generation via in-context learning. In *Proceedings of the 46th IEEE/ACM International Conference on Software Engineering*, ICSE '24, New York, NY, USA, 2024. Association for Computing Machinery. ISBN 9798400702174. doi: 10.1145/3597503.3608134. URL https://doi.org/10.1145/3597503.3608134.

Michael Hind. Pointer analysis: Haven't we solved this problem yet? In *Proceedings of the 2001 ACM SIGPLAN-SIGSOFT workshop on Program analysis for software tools and engineering*, pp. 54–61, 2001.

Simon Holm Jensen, Anders Møller, and Peter Thiemann. Type analysis for javascript. In *Static Analysis: 16th International Symposium, SAS 2009, Los Angeles, CA, USA, August 9-11, 2009. Proceedings 16*, pp. 238–255. Springer, 2009.

Vini Kanvar and Uday P. Khedker. Heap abstractions for static analysis. *ACM Computing Surveys*, 49(2):1–47, June 2016. ISSN 1557-7341. doi: 10.1145/2931098. URL http://dx.doi.org/10.1145/2931098.

Gary A Kildall. A unified approach to global program optimization. In *Proceedings of the 1st annual ACM SIGACT-SIGPLAN symposium on Principles of programming languages*, pp. 194–206, 1973.

Haonan Li, Yu Hao, Yizhuo Zhai, and Zhiyun Qian. Enhancing static analysis for practical bug detection: An llm-integrated approach. *Proc. ACM Program. Lang.*, 8(OOPSLA1), April 2024a. doi: 10.1145/3649828. URL https://doi.org/10.1145/3649828.

Ziyang Li, Saikat Dutta, and Mayur Naik. Llm-assisted static analysis for detecting security vulnerabilities, 2024b. URL https://arxiv.org/abs/2405.17238.

Percy Liang, Omer Tripp, Mayur Naik, and Mooly Sagiv. A dynamic evaluation of the precision of static heap abstractions. *ACM Sigplan Notices*, 45(10):411–427, 2010.

Puzhuo Liu, Chengnian Sun, Yaowen Zheng, Xuan Feng, Chuan Qin, Yuncheng Wang, Zhi Li, and Limin Sun. Harnessing the power of llm to support binary taint analysis, 2023.

Ravi Mangal, Xin Zhang, Aditya V Nori, and Mayur Naik. A user-guided approach to program analysis. In *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering*, pp. 462–473, 2015.

Noble Saji Mathews, Yelizaveta Brus, Yousra Aafer, Meiyappan Nagappan, and Shane McIntosh. Llbezpeky: Leveraging large language models for vulnerability detection, 2024.

Mohammad Mahdi Mohajer, Reem Aleithan, Nima Shiri Harzevili, Moshi Wei, Alvine Boaye Belle, Hung Viet Pham, and Song Wang. Skipanalyzer: A tool for static code analysis with large language models, 2023.

Yun Peng, Chaozheng Wang, Wenxuan Wang, Cuiyun Gao, and Michael R. Lyu. Generative type inference for python, 2023.

Veselin Raychev, Pavol Bielik, Martin Vechev, and Andreas Krause. Learning programs from noisy data. *SIGPLAN Not.*, 51(1):761–774, January 2016. ISSN 0362-1340. doi: 10.1145/2914770.2837671. URL https://doi.org/10.1145/2914770.2837671.

Paul B Schneck. A survey of compiler optimization techniques. In *Proceedings of the ACM annual conference*, pp. 106–113, 1973.

Manuel Serrano. On javascript ahead-of-time compilation performance (keynote). In *Proceedings of the 19th International Conference on Managed Programming Languages and Runtimes*, pp. 1–1, 2022.

Manu Sridharan, Julian Dolby, Satish Chandra, Max Schäfer, and Frank Tip. Correlation tracking for points-to analysis of javascript. In James Noble (ed.), *ECOOP 2012 – Object-Oriented Programming*, pp. 435–458, Berlin, Heidelberg, 2012. Springer Berlin Heidelberg. ISBN 978-3-642-31057-7.

Shashank Srikant, Sijia Liu, Tamara Mitrovska, Shiyu Chang, Quanfu Fan, Gaoyuan Zhang, and Una-May O'Reilly. Generating adversarial computer programs using optimized obfuscations. *arXiv preprint arXiv:2103.11882*, 2021.

Chaozheng Wang, Yuanhang Yang, Cuiyun Gao, Yun Peng, Hongyu Zhang, and Michael R. Lyu. No more fine-tuning? an experimental evaluation of prompt tuning in code intelligence. In *Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, ESEC/FSE 2022, pp. 382–394, New York, NY, USA, 2022a. Association for Computing Machinery. ISBN 9781450394130. doi: 10.1145/3540250.3549113. URL https://doi.org/10.1145/3540250.3549113.

Chong Wang, Jianan Liu, Xin Peng, Yang Liu, and Yiling Lou. Llm-based resource-oriented intention inference for static resource leak detection, 2023a.

Chong Wang, Jianan Liu, Xin Peng, Yang Liu, and Yiling Lou. Boosting static resource leak detection via llm-based resource-oriented intention inference, 2024. URL https://arxiv.org/abs/2311.04448.

Xuezhi Wang, Jason Wei, Dale Schuurmans, Quoc Le, Ed Chi, Sharan Narang, Aakanksha Chowdhery, and Denny Zhou. Self-consistency improves chain of thought reasoning in language models. *arXiv preprint arXiv:2203.11171*, 2022b.

Yue Wang, Hung Le, Akhilesh Deepak Gotmare, Nghi D. Q. Bui, Junnan Li, and Steven C. H. Hoi. Codet5+: Open code large language models for code understanding and generation, 2023b.

Jiayi Wei, Greg Durrett, and Isil Dillig. Typet5: Seq2seq type inference using static analysis, 2023.

William E Weihl. Interprocedural data flow analysis in the presence of pointers, procedure variables, and label variables. In *Proceedings of the 7th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pp. 83–94, 1980.

Yonghao Wu, Zheng Li, Jie M. Zhang, Mike Papadakis, Mark Harman, and Yong Liu. Large language models in fault localisation, 2023.

Chunqiu Steven Xia, Matteo Paltenghi, Jia Le Tian, Michael Pradel, and Lingming Zhang. Fuzz4all: Universal fuzzing with large language models, 2024.

Chenyuan Yang, Yinlin Deng, Runyu Lu, Jiayi Yao, Jiawei Liu, Reyhaneh Jabbarvand, and Lingming Zhang. White-box compiler fuzzing empowered by large language models, 2023a.

Chenyuan Yang, Zijie Zhao, and Lingming Zhang. Kernelgpt: Enhanced kernel fuzzing via large language models, 2023b.

Zhengran Zeng, Hanzhuo Tan, Haotian Zhang, Jing Li, Yuqun Zhang, and Lingming Zhang. An extensive study on pre-trained models for program understanding and generation. In *Proceedings of the 31st ACM SIGSOFT international symposium on software testing and analysis*, pp. 39–51, 2022.

Yizhuo Zhai, Yu Hao, Hang Zhang, Daimeng Wang, Chengyu Song, Zhiyun Qian, Mohsen Lesani, Srikanth V. Krishnamurthy, and Paul Yu. Ubitect: a precise and scalable method to detect use-before-initialization bugs in linux kernel. In *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, ESEC/FSE 2020, pp. 221–232, New York, NY, USA, 2020. Association for Computing Machinery. ISBN 9781450370431. doi: 10.1145/3368089.3409686. URL https://doi.org/10.1145/3368089.3409686.

A ABSTRACT INTERPRETATION DETAILS

```
1  var global = 0;
2  var global_obj = {};
3  function inc_global() {
4      let obj = {f: 1};
5      obj.f += 1;
6      global = global + obj.f;
7  }
8
9  function access_obj() {
10     if (global > 10) {
11         var f = global_obj.foo.bar; // bug
12     }
13 }
14 var button1 = document.createElement("button");
15 var button2 = document.createElement("button");
16 button1.addEventListener("click", inc_global);
17 button2.addEventListener("click", access_obj);
```

Figure 7: `inc_global` needs to be run at least 10 times before the bug on line 11 is triggered.

```
1  var users = []
2  let n = user_input();
3  for (let i = 0; i < n; i++) {
4      let student = createUser(i, "student");
5      if (i % 10 == 0) {
6          let teacher = createUser(i, "teacher");
7      }
8      users.push(student);
9      users.push(teacher);
10 }
```

Figure 8: An unbounded loop that requires abstraction to terminate.

A.1 ANALYSIS DETAILS

Functions In Javascript, functions are stored as objects on the heap. We include a `__code__` property storing the function body to be executed. At the beginning of the analysis, ABSINT-AI scans the entire program, and generates a *schema* for each function. The schema for each function contains which variables are local to the function and which variables are accessed by other functions. We refer to variables that are local as *private*, and variables that are accessed by other functions as *shared*. Each time a function is executed, an environment is initialized according to the schema for that function. When a function is defined, is initialized with a `__hf__` field set to the current heap frame. The `__hf__` field is used to model scopes and closures. When the function returns, the stack frame σ is popped from the stack, and the stack pointer is decremented.

Scopes and Closures Whenever a function is called, a new stack frame σ is pushed, along with a corresponding heap frame. The stack pointer for the current stack frame is updated to point to σ. The private variables for that function are stored in the stack frame σ, and any shared variables are stored in the heap frame. The heap frame is initialized with a parent field `__parent__` which is used to model the scope chain. The `__parent__` field points to the `__hf__` field for the function being initialized.

To lookup a variable name in the environment, ABSINT-AI first checks the current stack frame. If it finds a value for the variable, it returns the value. If it doesn't, it checks the corresponding heap frame for the stack frame, and then follows the chain of `__parent__` pointers until it finds the variable.

Recursion ABSINT-AI keeps track of all functions that have been called but have not finished executing yet. Whenever it encounters a recursive call, ABSINT-AI sets the return value to a recursive placeholder and stores a hash of the function that is called. When the function returns, ABSINT-AI checks the return values and any allocated heap objects for recursive placeholders for the function and fills them in with the return values.

A.2 ENVIRONMENT

In this section we describe how ABSINT-AI represents the abstract state. We define concrete and abstract values. H_L refers to the concrete heap, H_G refers to the global heap, and σ refers to the stack. τ is an abstract type, C refers to constants, obj and $\widetilde{obj}$ refer to concrete and abstract objects. val and

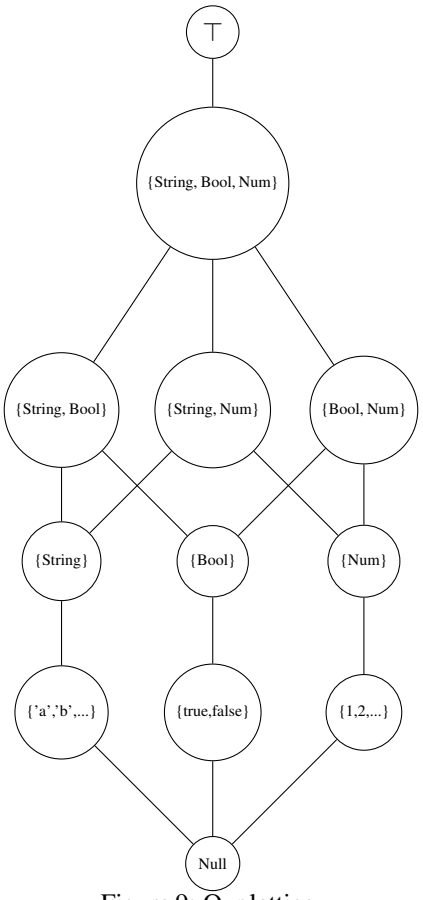


Figure 9: Our lattice.

$\widetilde{val}$ refer to the values that a variable can take.

$$
\begin{array}{rcl}
val & ::= & a \,|\, obj \,|\, \widetilde{val} \\
\widetilde{val} & ::= & C \,|\, \widetilde{a} \,|\, \tau \,|\, \widetilde{obj} \\
\tau & ::= & Bool \,|\, Null \,|\, Num \,|\, String \\
obj & ::= & \tau \rightarrow val \,|\, C \rightarrow val \\
\widetilde{obj} & ::= & \tau \rightarrow \widetilde{val} \,|\, C \rightarrow \widetilde{val} \\
H_L & ::= & a \rightarrow val \\
H_G & ::= & \widetilde{a} \rightarrow \widetilde{val} \\
\sigma & ::= & C \rightarrow val
\end{array}
$$

A.3 SYNTAX

$$
\begin{array}{rcl}
op & ::= & + \,|\, - \,|\, \div \,|\, \cdot \,|\, ... \\
E & ::= & id \,|\, E.field \,|\, E[E] \,|\, foo(E) \,|\, E_1[E_2](E_3,E4,...) \,|\, \text{function}(x_0,x_1,...)\{S\} \\
 & & |\, \text{new } foo(E_1,E_2,...) \,|\, C \,|\, \{f:E\} \\
varDef & ::= & \text{var } id=E \,|\, \text{let } id=E \,|\, \text{const } id=E \\
Stmt & ::= & varDef \,|\, id=E \,| \\
 & & E.f=E \,|\, E[E]=E \,|\, \text{def } foo(x_1,x_2,...,x_n)\{Stmt\} \,| \\
 & & \text{if } (E)\{Stmt\} \text{ else } \{Stmt\} \,|\, \text{class } foo\{Stmt\} \,| \\
 & & \text{return } E \,|\, \text{for (varDef; E; Stmt)}\{Stmt\} \\
 & & \text{for (varDef in E)}\{Stmt\} \,|\, \text{while } (E)\{Stmt\} \,|\, Stmt;Stmt
\end{array}
$$

A.4 SEMANTICS

A.4.1 FUNCTIONS

This section is several functions we use, such looking up a variable name and initializing a new schema for a function.

lookup(id)

$$\frac{s \equiv \emptyset \quad \theta = \emptyset}{\langle lookup(H_L, H_G, s, id) \to \theta \rangle}$$

$$\frac{s \in H_L \quad id \in H_L(s) \quad \theta = s}{\langle lookup(H_L, H_G, s, id) \to \theta \rangle}$$

$$\frac{s \in H_G \quad id \in H_G(s) \quad \theta = s}{\langle lookup(H_L, H_G, s, id) \to \theta \rangle}$$

$$\frac{s \in H_L \quad id \notin H_L(s) \quad \theta = lookup(H_L, H_G, H_L(s).par, id)}{\langle lookup(H_L, H_G, s, id) \to \theta \rangle}$$

$$\frac{s \in H_G \quad id \notin H_G(s) \quad \theta = lookup(H_L, H_G, H_G(s).par, id)}{\langle lookup(H_L, H_G, s, id) \to \theta \rangle}$$

initialize(schema)

$$\frac{H_L[a \mapsto \{schema.public, par \mapsto \sigma.hf\}] \quad \sigma'.\_secret \mapsto \{schema.secret\} \quad \sigma'.hf \mapsto a}{initialize(schema) \to H_L, H_G, \sigma :: \sigma'}$$

return_from_schema

$$\frac{\sigma \equiv \sigma' :: v}{return\_from\_schema \to H_L, H_G, \sigma'}$$

A.4.2 SMALL-STEP SEMANTICS

$$\langle H_L, H_G, \sigma, S \rangle \to \langle H'_L, H'_G, \sigma', S' \rangle$$

id $\dfrac{}{\langle H_L,H_G,\sigma,id\rangle \to \langle H_L,H_G,\sigma,lookup(id)\rangle}$

E.field $\dfrac{\langle H_L,H_G,\sigma,E\rangle \to \langle H'_L,H'_G,\sigma',V\rangle}{\langle H_L,H_G,\sigma,\text{E.field}\rangle \to \langle H'_L,H'_G,\sigma',\text{get(V,field)}\rangle}$

$E_1[E_2]$ $\dfrac{\langle H_L,H_G,\sigma,E_2\rangle \to \langle H'_L,H'_G,\sigma',V_2\rangle \quad \langle H'_L,H'_G,\sigma',E_1\rangle \to \langle H''_L,H''_G,\sigma'',V_1\rangle}{\langle H_L,H_G,\sigma,E_1[E_2]\rangle \to \langle H'_L,H'_G,\sigma',\text{get}(V_1,V_2)\rangle}$

$foo(E_0,E_1,...)$ $\dfrac{\langle lookup(foo)\to V, V.\_\_type \equiv Function\rangle \quad \langle H_L,H_G,\sigma,E_0,E_1,...\rangle \to \langle H'_L,H'_G,\sigma',V_0,V_1,...\rangle}{\langle H_L,H_G,\sigma,foo(E_0,E_1,...)\rangle \to \langle H'_L[x_0\mapsto V_0,x_1\mapsto V_1,...],H'_G,\sigma',initialize(V.\_\_code);V.\_\_code\rangle}$

$E_1[E_2](E_3,E_4,...)$ $\dfrac{\langle H_L,H_G,\sigma,E_0,E_1,...\rangle \to \langle H'_L,H'_G,\sigma',V_0,V_1,V_2,...\rangle \quad \langle get(V_0,V_1)\to V, V.\_\_type \equiv Function\rangle}{\langle H_L,H_G,\sigma,foo(E_0,E_1,...)\rangle \to \langle H'_L[x_0\mapsto V_0,x_1\mapsto V_1,...],H'_G,\sigma'[this\mapsto V_0],V.\_\_code\rangle}$

function$(x_0,x_1,...)\{S\}$ $\dfrac{}{\langle H_L,H_G,\sigma,\text{function}(x_0,x_1,...)\rangle \to \langle H'_L[a'\mapsto\{...,prototype:a\},a\mapsto],H'_G,\sigma',a'\rangle}$

new foo$(E_0,E_1,...)$ $\dfrac{\langle lookup(foo)\to V\rangle \quad \langle V.\_\_type\equiv Class\rangle \quad \langle E_0,E_1,...\rangle \to \langle V_0,V_1,...\rangle}{\langle H_L,H_G,\sigma,\text{new foo}(E_0,E_1,...)\rangle \to \langle H'_L,H'_G,\sigma'[this\mapsto V],init();get(prototype(V),constructor)(V}$

$\dfrac{\langle lookup(foo)\to V\rangle \quad \langle V.\_\_type\equiv Function\rangle \quad \langle E_0,E_1,...\rangle \to \langle V_0,V_1,...\rangle}{\langle H_L,H_G,\sigma,\text{new foo}(E_0,E_1,...)\rangle \to \langle H'_L,H'_G,\sigma',V.\_\_code(V_0,V_1,...)\rangle}$

$\{f_1:E_1,f_2:E_2,...\}$ $\dfrac{\langle H_L,H_G,\sigma,E_1,E_2,...\rangle \to \langle H'_L,H'_G,\sigma',V_1,V_2,...\rangle}{\langle H_L,H_G,\sigma,\{f_1:E_1,f_2:E_2,...\}\rangle \to \langle H_L[a\mapsto\{f_1:V_1,f_2:V_2,...,\_\_type:\text{object}\}],H_G,\sigma,a\rangle}$

(var x = E) $\dfrac{\langle H_L,H_G,\sigma,E\rangle \to \langle H'_L,H'_G,\sigma',V\rangle \quad \theta = lookup(x) \quad \theta\in H_L \quad \text{fr} = H_L[\theta] \quad \text{fr'} = fr[id\mapsto V]}{\langle H_L,H_G,\sigma,x=E\rangle \to \langle H'_L[\theta\mapsto fr'],H'_G,\sigma',skip\rangle}$

$\dfrac{\langle H_L,H_G,\sigma,E\rangle \to \langle H'_L,H'_G,\sigma',V\rangle \quad \theta = lookup(x) \quad \theta\in H_G \quad \text{fr} = H_G[\theta] \quad \text{fr'} = fr[id\mapsto V\cup fr[id]]}{\langle H_L,H_G,\sigma,x=E\rangle \to \langle H'_L,H'_G[\theta\mapsto fr'],\sigma',skip\rangle}$

(x.f = E) $\dfrac{lookup(x)\equiv a \quad \theta = H_L(a) \quad \langle H_L,H_G,\sigma,E\rangle \to \langle H'_L,H'_G,\sigma',V\rangle}{\langle H_L,H_G,\sigma,x.f=E\rangle \to \langle H'_L[\theta[f\mapsto V]],H'_G,\sigma'],skip\rangle}$

$\dfrac{lookup(x)\equiv \widetilde{a} \quad \widetilde{\theta} = H_G(\widetilde{a}) \quad \langle H_L,H_G,\sigma,E\rangle \to \langle H'_L,H'_G,\sigma',V\rangle}{\langle H_L,H_G,\sigma,x=E\rangle \to \langle H'_L,H'_G[\widetilde{\theta}[f\mapsto V],\sigma',skip\rangle}$

(x[E] = E')
$$\frac{lookup(x) \equiv a \quad \theta = H_L(a) \quad \langle H_L, H_G, \sigma, E, E' \rangle \rightarrow \langle H'_L, H'_G, \sigma', V, V' \rangle}{\langle H_L, H_G, \sigma, x[f] = E \rangle \rightarrow \langle H'_L[\theta[V \mapsto V']], H'_G, \sigma' \rangle, skip \rangle}$$

$$\frac{lookup(x) \equiv \widetilde{a} \quad \widetilde{\theta} = H_G(\widetilde{a}) \quad \langle H_L, H_G, \sigma, E, E' \rangle \rightarrow \langle H'_L, H'_G, \sigma', V, V' \rangle}{\langle H_L, H_G, \sigma, x = E \rangle \rightarrow \langle H'_L, H'_G[\widetilde{\theta}[V \mapsto V'], \sigma', skip \rangle}$$

(def foo($x_0, x_1, \ldots, x_n$){Stmt})
$$\frac{\theta = lookup(foo) \quad \theta \in \sigma}{\langle H_L, H_G, \sigma, x[f] = E \rangle \rightarrow \langle H_L[a \mapsto ..., prototype : a', a' \mapsto \{\}], H_G, \sigma[\theta \mapsto a], skip \rangle}$$

$$\frac{\theta = lookup(foo) \quad \theta \in H_L}{\langle H_L, H_G, \sigma, x[f] = E \rangle \rightarrow \langle H_L[a \mapsto ..., prototype : a', a' \mapsto \{\}, \theta \mapsto a], H_G, \sigma, skip \rangle}$$

$$\frac{\theta = lookup(foo) \quad \theta \in H_G}{\langle H_L, H_G, \sigma, x[f] = E \rangle \rightarrow \langle H_L, H_G[a \mapsto ..., prototype : a', a' \mapsto \{\}, \theta \mapsto \theta \cup a], \sigma, skip \rangle}$$

(x[E] = E')
$$\frac{lookup(x) \equiv a \quad \theta = H_L(a) \quad \langle H_L, H_G, \sigma, E, E' \rangle \rightarrow \langle H'_L, H'_G, \sigma', V, V' \rangle}{\langle H_L, H_G, \sigma, x[f] = E \rangle \rightarrow \langle H'_L[\theta[V \mapsto V']], H'_G, \sigma' \rangle, skip \rangle}$$

$$\frac{lookup(x) \equiv \widetilde{a} \quad \widetilde{\theta} = H_G(\widetilde{a}) \quad \langle H_L, H_G, \sigma, E, E' \rangle \rightarrow \langle H'_L, H'_G, \sigma', V, V' \rangle}{\langle H_L, H_G, \sigma, x = E \rangle \rightarrow \langle H'_L, H'_G[\widetilde{\theta}[V \mapsto V'], \sigma', skip \rangle}$$

if (E) { Stmt }) else { Stmt' }
$$\frac{\langle H_L, H_G, \sigma, E \rangle \rightarrow \langle H'_L, H'_G, \sigma', False \vee \emptyset \rangle}{\langle H_L, H_G, \sigma, \text{if (E)} \{Stmt\} \text{ else } \{Stmt;\} \rangle \rightarrow \langle H'_L, H'_G, \sigma' \rangle, Stmt \rangle}$$

$$\frac{\langle H_L, H_G, \sigma, E \rangle \not\rightarrow \langle H'_L, H'_G, \sigma', False \vee \emptyset \rangle}{\langle H_L, H_G, \sigma, \text{if (E)} \{Stmt\} \text{ else } \{Stmt'\} \rangle \rightarrow \langle H'_L, H'_G, \sigma' \rangle, Stmt' \rangle}$$

class foo$[M_1, M_2, ..., M_N]$
$$\frac{class\_obj = \{M_1, M_2, ..., M_N\}}{\langle H_L, H_G, \sigma, \text{class foo}[M_1, M_2, ..., M_N] \rangle \rightarrow \langle H_L[a \mapsto class\_obj], H_G, \sigma, skip \rangle}$$

(return E)
$$\frac{\langle H_L, H_G, \sigma, E \rangle \rightarrow \langle H'_L, H'_G, \sigma', V \rangle}{\langle H_L, H_G, \sigma, returnE \rangle \rightarrow \langle H'_L, H'_G, \sigma'[returns \mapsto \sigma'[returns] \cup V], skip \rangle}$$

for ([let | var] id in E) $\{Stmt\}$
$$\frac{\langle H_L, H_G, \sigma, E \rangle \rightarrow \langle H'_L, H'_G, \sigma', V \rangle \quad V.\_\_proto\_\_ \equiv \emptyset \quad isEmpty(V) \equiv True}{\langle H_L, H_G, \sigma, \text{for ([let | var] id in E) } \{Stmt\} \rangle \rightarrow \langle H'_L, H'_G, \sigma', skip \rangle}$$

$$\frac{\langle H_L, H_G, \sigma, E \rangle \rightarrow \langle H'_L, H'_G, \sigma', V \rangle \quad V.\_\_proto\_\_ \not\equiv \emptyset \quad isEmpty(V) \equiv True}{\langle H_L, H_G, \sigma, \text{for ([let | var] id in E) } \{Stmt\} \rangle \rightarrow \langle H'_L, H'_G, \sigma', \text{for ([let | var] id in V.\_\_proto\_\_) } \{\text{Stmt}\} \rangle}$$

$$\frac{\langle H_L, H_G, \sigma, E \rangle \rightarrow \langle H'_L, H'_G, \sigma', V \rangle \quad V \equiv X :: V' \quad \text{varDef.type} \equiv let}{\langle H_L, H_G, \sigma, \text{for (let id in E) } \{Stmt\} \rangle \rightarrow \langle H''_L, H''_G, \sigma'', \text{initialize(Stmt);let id=X;Stmt; for (let id in V')}}$$

$$\frac{\langle H_L, H_G, \sigma, E \rangle \rightarrow \langle H'_L, H'_G, \sigma', V \rangle \quad V \equiv X :: V' \quad \text{varDef.type} \equiv var}{\langle H_L, H_G, \sigma, \text{for (let id in E) } \{Stmt\} \rangle \rightarrow \langle H'_L, H'_G, \sigma', \text{var id=X;Stmt; for (let id in V') } \{ \text{ Stmt } \} \rangle}$$

$$\text{while (E) } \{Stmt\} \quad \frac{\langle H_L,H_G,\sigma,E\rangle \to \langle H'_L,H'_G,\sigma',V\rangle \quad V \in \text{Falsey}}{\langle H_L,H_G,\sigma,\text{while (E) } \{Stmt\}\rangle \to \langle H'_L,H'_G,\sigma',skip\rangle}$$

$$\frac{\langle H_L,H_G,\sigma,E\rangle \to \langle H'_L,H'_G,\sigma',V\rangle \quad V \notin \text{Falsey} \quad \langle H'_L,H'_G,\sigma',\text{Stmt;summarize()}\to \langle H'_L,H'_G,\sigma'\rangle}{\langle H_L,H_G,\sigma,\text{while (E) } \{Stmt\}\rangle \to \langle H'_L,H'_G,\sigma',skip\rangle}$$

$$\frac{\langle H_L,H_G,\sigma,E\rangle \to \langle H'_L,H'_G,\sigma',V\rangle \quad V \notin \text{Falsey} \quad \langle H'_L,H'_G,\sigma',\text{Stmt;summarize()}\to \langle H''_L,H''_G,\sigma''\rangle}{\langle H_L,H_G,\sigma,\text{while (E) } \{Stmt\}\rangle \to \langle H''_L,H''_G,\sigma'',\text{while (E) } \{Stmt\}\rangle}$$

B PROMPTING FOR HIGH-LEVEL GUIDANCE

This section contains the prompts used for high-level guidance for the LLM.

```
You are part of a static analyzer, where you are responsible for deciding which
variables need to be abstracted and which ones can be kept concrete. Your goal is
to abstract the necessary variables so that the analysis can converge to a fixpoint
, but not to lose too much information through the abstractions. You are only
allowed to abstract variable names, not fields. Here are some basic heuristics for
when to abstract a variable and when to keep it concrete:
- Abstract a variable if the precise value of the variable is not important for the
 analysis.
- Abstract a variable if re-executing the program infinitely will cause the
variable to increase in size. If re-executing the program does not change the shape
 or values of the variable, keep it concrete.
- Keep a variable concrete if the current values provided above would not change by
 re-executing the program.
- Keep a variable concrete if it is used frequently in control flow decisions.
- Whenever possible and if you are unsure, keep variables concrete and do not
abstract. We can always abstract later if needed.


I will provide the code and the current state of the variables.
The code is:
'''javascript
[code]
'''

The variables you can abstract are: [all variable names]. The values for the
variables in JSON form so far are:
'''json
[all variable values as JSON]
'''
 What variables should be abstracted? You can abstract both primitives and objects.
  The program will converge when the variables do not change between executions.
 Please perform enough abstractions for the program converge but do not abstract
 too much information. We are about to re-execute the code I provided above with
 the variables initialized to the values above with no destructive updates to the
 variables. This means that any updates will be performed on the variables above.

           [MODEL RESPONSE]
```

Figure 10: The initial prompt for getting the variable names to abstract.

```
You chose to abstract the following variable: ‘ variable name ‘. It has multiple
fields, the value of the object is:
‘‘‘json
 variable value
‘‘‘
As a reminder, the code is:
‘‘‘javascript
 code
‘‘‘
Do you want to pick individual fields to abstract, or should I abstract the whole
thing? You would abstract individual fields if:
    - You know that only some fields are relevant to the analysis
    - Some fields are not being updated, and can stay concrete
    - Some fields are being updated, and need to be abstracted
You would abstract all fields if:
    - You are not sure which fields are relevant
    - The fields are being added dynamically
If there are fields or properties being added dynamically, you’ll have to abstract
all fields.
Please respond with "INDIVIDUAL FIELDS" or "ALL FIELDS". DO NOT say anything else.

           [MODEL RESPONSE]
```

Figure 11: The prompt determining whether the entire object should be abstracted or only certain fields.

```
You chose to abstract the following variable: ‘ variable name ‘. It has multiple
fields, the value of the object is:
‘‘‘json
 variable value
‘‘‘
As a reminder, the entire code is:
‘‘‘javascript
 code
‘‘‘
What fields should be abstracted, and why? The possible fields are:
 comma separated fields .
You should abstract a field if:
    - The value for the field is being updated dynamically, and abstraction will
    help the analysis converge.
You should NOT abstract a field if:
    - The precise value of the field is important for the analysis, for example if
    it is important for control flow.
    - The value of the field is not being updated
```

Figure 12: If the LLM says to abstract only certain fields, select what fields to abstract.

C ALLOCATION SITE ABSTRACTIONS

This is the baseline. We keep a map of allocation sites to all the objects that are allocated there. When `summarize` is called, it merges all of the allocation sites into a single node. There is a separation between merging objects, heap frames, and functions. The rough steps of the algorithm for merging a collection of objects or classes:

1. Combine fields and values of the objects into summary node in the abstract heap. If there is already a node for the allocation site, we merge in the objects for the allocation site. If not, we create an empty object for that allocation site and use it for future merges. Update all pointers for the previous objects to the old objects to the new summary node. We then recurse through all values in the new summary node and move any objects referenced in the concrete heap to the abstract heap. If the object has been modified, continue to step 2. If not, exit. This way we don't prematurely run our abstractions on objects that haven't been modified.
2. Merge all the fields into abstract types. So if there are fields that are strings, merge all of them into the STRING object, etc. Now the summary object has a maximum of 3 keys (STRING, BOOL, NUM).
3. For the values of each field, if there are primitives merge them into the abstract primitives. For any objects in the values, perform steps 1-3 on the collection of objects. If there are functions, perform the function merging algorithm.

The rough steps of the algorithm for merging heap frames:

1. Combine fields and values of the heap frames into one summary frame, similar to objects. If there is already a node for the allocation site, we merge in the existing heap frames for the allocation site. We then recurse through all values in the new summary frame and move any objects referenced in the concrete heap to the abstract heap.
2. Each "field" in the heap frame is a variable in the program, so we don't need to merge them together. For all the values for each variable, if there are primitives merge them into the abstract primitives. For any objects in the values, perform steps 1-3 on the collection of objects. If there are functions, perform the function merging algorithm.

The rough steps for merging a collection of functions. The only thing that gets dynamically allocated and needs to be summarized is the heap frames for closures which are encapsulated by the `__parent__` field for the function.

1. Merge the function objects into a single one, where everything is the same except the `__parent__` field which is now a collection of heap frames. If there is only one value for the `__parent__` field, exit.
2. For the collection of parent objects, the fields are all the same since these are just the variable names. Merge all of the values following step 3 of the object merging function.
3. Repeat for each parent.

D PROMPTS FOR DIRECT ABSTRACTION PREDICTION

````
I am going to provide you with the code, the environment, and the state of the
concrete heap and abstract heap. The environment and heaps will be provided in JSON
 format. Please merge nodes in the heap to achieve the best balance of scalability
and precision for detecting 'TypeError: Cannot read properties of null or undefined
'.
Here is the code: \n
```javascript
[code]
```
Here is the loop body we are analyzing:
 ```javascript
[Loop Body]
 ```

Here is the current state of the environment:
```json
[Current Primitive Values]
```

Here is the current state of the concrete heap:
```json
[Concrete Heap]
```

Here is the current state of the abstract heap:
```json
[Abstract Heap]
```

Please provide an abstraction by merging nodes in the concrete and abstract heap,
and provide a mapping between which nodes have been merged in your output. Only
specify the addresses you want to merge.
The keys in your JSON object should be abstract address strings, and the values
should be lists of concrete addresses that you want to merge. For example, if you
want to merge addresses ["Concrete Address(1)", "Concrete Address(2)"] into an
abstract address "Abstract Address(1)", you would provide the following JSON object
:
{{"Abstract Address(1)": ["Concrete Address(1)", "Concrete Address(2)"]}}
You can also return an empty object if you do not want to merge any nodes.
[MODEL RESPONSE]
````

Figure 13: The initial prompt for merging objects without abstracting.

````
You are performing heap abstractions to find a fixpoint in order to finish
analyzing a loop. I am going to provide you with the code surrounding the loop body
 we are analyzing, the loop body we are analyzing, and the state of the heap.
Please provide a heap abstraction that results in a fixpoint while keeping as much
relevant information as possible.

1. Abstract a list of strings into an abstract STRING object. For example, ["foo1",
 "foo2", "foo3"] can become ["STRING"]
2. Abstract a list of numbers into an abstract NUMBER object. For example,
[1,2,3,4,5] can become ["NUMBER"]
3. Merging keys in an object following the above rules. For example, {{1: ["var"],
2: ["var"], 3: ["var"]}} can become {{"NUMBER": ["var"]}}. It can also become {{"
NUMBER": ["STRING"]}}.
4. Leave the object as is and do not perform an abstraction.

Here is the code:
```javascript
[code]
```

[Loop Body]

Here is the current state of the environment:
```json
[Current Primitive Values]
```

Here is the current state of the concrete heap:
```json
[Concrete Heap]
```

Here is the current state of the abstract heap:
```json
[Abstract Heap]
````


Please provide a general heap abstraction for the relevant variables that results
in a fixpoint. Perform as few abstractions as are necessary to reach a fixpoint.

You can return an empty object if you do not want to perform any abstractions.
Please use an abstraction that is as precise as possible while still maintaining a
fixpoint and include a JSON of the abstraction in your response.
[MODEL RESPONSE]
```

Figure 14: The prompt for directly abstracting the heap.

```
You are abstracting variables in order to finish analyzing a loop. I am going to
provide you with the code and the state of the heap. Please provide an abstraction
of the primitive variables that results in a fixpoint while keeping as much
relevant information as possible.

1. Abstract a list of strings into an abstract STRING object. For example, ["foo1",
 "foo2", "foo3"] can become ["STRING"]
2. Abstract a list of numbers into an abstract NUMBER object. For example,
[1,2,3,4,5] can become ["NUMBER"]
3. Merging keys in an object following the above rules. For example, {{1: ["var"],
2: ["var"], 3: ["var"]}} can become {{"NUMBER": ["var"]}}. It can also become {{"
NUMBER": ["STRING"]}}.
4. Leave the object as is and do not perform an abstraction.

Here is the code:
'''javascript
[code]
'''
Here is the loop body we are analyzing:
'''javascript
[Loop Body]
'''

Here is the current state of the environment:
'''json
[Current Primitive Values]
'''

Please provide a general abstraction for the relevant variables that results in a
fixpoint. Perform as few abstractions as are necessary to reach a fixpoint. You can
 return an empty object if you do not want to perform any abstractions. Please use
an abstraction that is as precise as possible while still maintaining a fixpoint,
and include a JSON of the abstraction in your response.
[MODEL RESPONSE]
```

Figure 15: The prompt for directly abstracting any primitive values in the environment.