

Partial Evaluation + Global Analyser = Separate Analyser

GYUHYEOK OH, Seoul National University, Korea

1 PROBLEM

Scalable static analyzer needs a separate analysis technique. Given open program fragments, a separate analysis computes summaries for the fragments in isolation then links the analysis results to conclude the analysis for the whole program.

However, the design of these separate static analyses often relies on ad hoc techniques; design of a systematic framework is still an open problem.

We propose using partial evaluation [6, 7] as the basis for a framework for separate static analysis design. By partially evaluating a global abstract interpretation [4] with respect to each module, we obtain its separate analysis result. Linking of these results is, in turn, achieved again by partially evaluating the linking operator over the available separate analysis results.

2 APPROACH

Our core idea is that the summary of a module m should not be an abstract value, but the residual analysis code sa_m derived from partially evaluating the abstract interpreter $I^\#$ against m .

2.1 Step 1: Module Analyzer Generation

The first stage specializes the generic abstract interpreter $I^\#$ for a specific module m . $I^\#$ is a function that takes a module and an abstract state, returning a new abstract state: $I^\#: M \rightarrow \Sigma^\# \rightarrow \Sigma^\#$. We treat the module m as static and the input state $\sigma^\#$ as dynamic.

Definition 2.1 (Separate Analyzer by partial evaluation). Let $\mathcal{PE}$ be a partial evaluator. Given $I^\#$ and a module m , the residual analyzer sa_m is

$$sa_m = \mathcal{PE}(I^\#, m) \quad \text{where } sa_m \in C_{\Sigma^\# \rightarrow \Sigma^\#}.$$

The residual code sa_m is semantically equivalent to the original interpreter specialized for m ($\forall \sigma^\# \in \Sigma^\#. \llbracket sa_m \rrbracket(\sigma^\#) = I^\#(m, \sigma^\#)$). The PE process has already executed all m -dependent logic, leaving only operations dependent on the dynamic input state $\sigma^\#$.

2.2 Step 2: Analyzer Composition

Given these residual analyzers, a naive linker would simply execute them sequentially, passing the concrete abstract state between them. Here, sa_a and sa_b are the respective analysis results for modules ma and mb , and $m_a \bowtie m_b$ represents the relationship $m_a \rightarrow m_b$ between them. However, executing a chain $m_1 \rightarrow m_2 \rightarrow m_3$ as $\llbracket sa_3 \rrbracket(\llbracket sa_2 \rrbracket(\llbracket sa_1 \rrbracket(\sigma_{init}^\#)))$ is not truly compositional, as sa_2 and sa_3 cannot be evaluated together without the concrete value $\sigma_1^\#$.

For compositional linking, we need to treat the *link* program itself as a target for partial evaluation. We apply *pe* again, this time treating analyzer code sa_a and sa_b as static data and the input state σ_{in} as dynamic data. This allows us to statically *link* the two residual analyzers sa_a and sa_b to generate a new analyzer $sa_{a,b}$.

Definition 2.2 (Analyzer Composition with partial evaluation). For compositional linking, we partially evaluate the *link* function, which is not compositional.

$$sa_{a,b} = \mathcal{PE}(\text{link}, (sa_a \bowtie sa_b))$$

Definition 2.3 (Semantics of link). The semantics of *link*, which is the target of $\mathcal{PE}$, is to connect the input and output and execute them. Given residual analyzers sa_a and sa_b and an input state $\sigma_{in}^\#$, we can get the final abstract state:

$$link(sa_a \bowtie sa_b, \sigma_{in}^\#) = \llbracket sa_b \rrbracket(\llbracket sa_a \rrbracket(\sigma_{in}^\#))$$

The logic of sa_a is inlined into sa_b by $\mathcal{PE}$ which perform static reduction on the code of the analyzers. Any static output of sa_a (e.g., a constant) is propagated statically into sa_b 's logic, enabling inter-module optimization before any abstract state is computed.

With this linker, we can link m_2 and m_3 first, independently of m_1 :

$$sa_{2,3} = \mathcal{PE}(link, (sa_2, sa_3))$$

The result $sa_{2,3} \in C$ is a new residual analyzer whose logic fuses sa_2 and sa_3 . The dynamic input of this new analyzer is $\sigma_1^\#$.

$$\forall \sigma_1^\#. \llbracket sa_{2,3} \rrbracket(\sigma_1^\#) = link(sa_2, sa_3, \sigma_1^\#) = \llbracket sa_3 \rrbracket(\llbracket sa_2 \rrbracket(\sigma_1^\#))$$

The final analyzer for the whole program can then be computed by composing sa_1 with the $sa_{2,3}$ artifact:

$$sa_{1,(2,3)} = \mathcal{PE}(link, (sa_1, sa_{2,3}))$$

The result $sa_{1,(2,3)}$ is a single, fused, highly optimized residual program representing the whole analysis, whose execution on the initial abstract state yields the final abstract state:

$$\forall \sigma_{init}^\#. \llbracket sa_{1,(2,3)} \rrbracket(\sigma_{init}^\#) = link(sa_1, sa_{2,3}, \sigma_{init}^\#)$$

3 SMALL EXAMPLE

To illustrate this approach, consider a simple imperative language where a program is a sequence of modules. Each module can define variables and perform arithmetic operations, including accessing variables from preceding modules (e.g., $m_A.x$):

Module $M ::= m : S$

Stmt $S ::= x := e \mid S_1; S_2 \mid \text{ifz } e \{S_1\} \{S_2\}$

Exp $e ::= x \mid m.x \mid n \mid e_1 \oplus e_2$

We define an abstract interpreter $I^\#$ that analyzes these modules using a standard abstract domain (e.g., intervals).

let $I^\#(M, \sigma_{in}^\#) = \text{match } M \text{ with}$

$\mid (m_{cur}, S) \rightarrow eval_S(m_{cur}, S, \sigma_{in}^\#)$

and $eval_S(m_{cur}, S, \sigma_{in}^\#) = \text{match } S \text{ with}$

$\mid (x := e) \rightarrow \text{let } v = eval_E(m_{cur}, e, \sigma_{in}^\#) \text{ in } \sigma_{in}^\#[m_{cur}.x \mapsto v]$

$\mid (S_1; S_2) \rightarrow \text{let } \sigma'^\# = eval_S(m_{cur}, S_1, \sigma_{in}^\#) \text{ in } eval_S(m_{cur}, S_2, \sigma'^\#)$

$\mid (\text{ifz } e \{S_1\} \{S_2\}) \rightarrow \text{let } v_{cond} = eval_E(m_{cur}, e, \sigma_{in}^\#) \text{ in}$

$\mathcal{F}_{\neq 0}^\#(v_{cond}, eval_S(m_{cur}, S_1, \sigma_{in}^\#)) \sqcup^\# \mathcal{F}_{=0}^\#(v_{cond}, eval_S(m_{cur}, S_2, \sigma_{in}^\#))$

and $eval_E(m_{cur}, e, \sigma_{in}^\#) = \text{match } e \text{ with}$

$\mid x \rightarrow \sigma_{in}^\#[m_{cur}.x] \mid m.x \rightarrow \sigma_{in}^\#(m.x) \mid n \rightarrow \eta(n)$

$\mid (e_1 \oplus e_2) \rightarrow \text{let } v_1 = eval_E(m_{cur}, e_1, \sigma_{in}^\#) \text{ in let } v_2 = eval_E(m_{cur}, e_2, \sigma_{in}^\#) \text{ in } v_1 \oplus^\# v_2$

Here, η abstracts constants into abstract values into a chosen abstract domain (e.g., intervals). This choice parameterizes the definitions of the various abstract operators used in the analysis. These operators include, for instance, the abstract arithmetic operations $\oplus^\#$ (like $+$) which are defined to correspond to their concrete counterparts ($\oplus$). Other critical abstract operators shown

are the join operator $\sqcup^\sharp$, which merges abstract states from branches and the filter functions $F^\sharp$, which enhance precision by refining the state based on the branch conditions (e.g., $e = 0$) [9].

Let's analyze a program with three modules $m_A \bowtie m_B \bowtie m_C$:

$$m_A: x := 5; y := 10 \quad m_B: z := m_A.x + m_A.y \quad m_C: w := m_B.z * 2$$

First, we apply the partial evaluator $\mathcal{PE}$ to the abstract interpreter $I^\sharp$ and each module independently. This generates residual analyzers sa_A , sa_B , and sa_C . The resulting residual analyzers:

$$\text{let } sa_A(\sigma_{\text{in}}) = \sigma_{\text{in}}[m_A.x \mapsto 5^\sharp][m_A.y \mapsto 10^\sharp]$$

$$\text{let } sa_B(\sigma_{\text{in}}) = \text{let } v_1 = \sigma_{\text{in}}(m_A.x) \text{ in let } v_2 = \sigma_{\text{in}}(m_A.y) \text{ in let } v_z = v_1 +^\sharp v_2 \text{ in } \sigma_{\text{in}}[m_B.z \mapsto v_z]$$

$$\text{let } sa_C(\sigma_{\text{in}}) = \text{let } v_z = \sigma_{\text{in}}(m_B.z) \text{ in let } v_w = v_z *^\sharp 2^\sharp \text{ in } \sigma_{\text{in}}[m_C.w \mapsto v_w]$$

Then, we can link these residual analyzers by executing them, passing the abstract state.

- M_A analysis: $\llbracket sa_A \rrbracket(\sigma_{\text{init}}^\sharp) = \{m_A.x \mapsto 5^\sharp, m_A.y \mapsto 10^\sharp\} = \sigma_A^\sharp$
- M_B analysis: $\llbracket sa_B \rrbracket(\sigma_A^\sharp) = \{m_A.x \mapsto 5^\sharp, m_A.y \mapsto 10^\sharp, m_B.z \mapsto 15^\sharp\} = \sigma_B^\sharp$
- M_C analysis: $\llbracket sa_C \rrbracket(\sigma_B^\sharp) = \{m_A.x \mapsto 5^\sharp, m_A.y \mapsto 10^\sharp, m_B.z \mapsto 15^\sharp, m_C.w \mapsto 30^\sharp\} = \sigma_C^\sharp$

The problem with this approach is its lack of compositionality for re-analysis. If m_A changes (e.g., to $x := 7$), we must re-run $\llbracket sa_A \rrbracket$. This produces a new $\sigma_A^\sharp$, which then forces us to re-run $\llbracket sa_B \rrbracket$ to get a new $\sigma_B^\sharp$, which in turn forces us to re-run $\llbracket sa_C \rrbracket$. For solving this problem, we can use our PE-based linker to fuse sa_B and sa_C into a single analyzer $sa_{B,C} = \mathcal{PE}(\text{link}, (sa_B, sa_C))$:

$$\text{let } sa_{B,C}(\sigma_{\text{in}}^\sharp) =$$

$$\text{let } v_1 = \sigma_{\text{in}}^\sharp(m_A.x) \text{ in let } v_2 = \sigma_{\text{in}}^\sharp(m_A.y) \text{ in let } v_z = v_1 +^\sharp v_2 \text{ in let } v_w = v_z *^\sharp 2^\sharp \text{ in } \sigma_{\text{in}}[m_B.z \mapsto v_z][m_C.w \mapsto v_w]$$

This fused analyzer $sa_{B,C}$ represents the complete analysis for modules B and C, optimized as a single unit. The intermediate abstract state $\sigma_B^\sharp$ is completely eliminated from the computation. Now, when m_A changes: We re-run the new sa_A with $\sigma_{\text{init}}^\sharp$ to get the new $\sigma_A^\sharp$ and single fused analyzer $sa_{B,C}$. We have reduced the re-analysis chain from two steps— sa_B then sa_C —to a single, optimized step— $sa_{B,C}$, improving efficiency for incremental and compositional analysis.

4 RELATED WORK

We propose a compositional analysis framework centered on analyzer fusion, which leverages partial evaluation to specialize an abstract interpreter against a module, rather than computing a simplified summary. This approach has strength in preserving the full precision of the original interpreter without using expensive abstraction like symbolic execution, compared to prior work on modular abstract interpretation [5].

This separate analysis based on partial evaluation fundamentally differs from logic-based approaches like Infer, which rely on inferring logical specifications (e.g., separation logic footprints, incorrectness assertions) via bespoke engines like bi-abduction [2, 3, 8]. Our framework provides greater generality, as it is applicable to any abstract domain (e.g., intervals, polyhedra) implemented in a language amenable to partial evaluation.

Furthermore, analyzer fusion has similarities to incremental analysis techniques [10, 11]. While traditional methods cache intermediate values and trigger a re-analysis cascade upon change [12], our approach caches the fused computation itself. This pre-compiles the re-analysis work, eliminating the propagation chain and enabling more efficient updates.

This methodology also shares conceptual similarities with staged abstract interpretation and abstract compilation [1, 13]. However, their primary objective differs: they typically focus on acceleration to generate a faster analyzer for entire programs, whereas our goal is to enable modularity and compositionality in analysis through partial evaluation.

REFERENCES

- [1] Dominique Boucher and Marc Feeley. 1996. Abstract compilation: A new implementation paradigm for static analysis. In *Compiler Construction*, Tibor Gyimóthy (Ed.). Springer Berlin Heidelberg, Berlin, Heidelberg, 192–207.
- [2] Cristiano Calcagno and Dino Distefano. 2011. Infer: an automatic program verifier for memory safety of C programs. In *Proceedings of the Third International Conference on NASA Formal Methods* (Pasadena, CA) (NFM’11). Springer-Verlag, Berlin, Heidelberg, 459–465.
- [3] Cristiano Calcagno, Dino Distefano, Peter O’Hearn, and Hongseok Yang. 2009. Compositional shape analysis by means of bi-abduction. In *Proceedings of the 36th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (Savannah, GA, USA) (POPL ’09). Association for Computing Machinery, New York, NY, USA, 289–300. <https://doi.org/10.1145/1480881.1480917>
- [4] 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* (Los Angeles, California) (POPL ’77). Association for Computing Machinery, New York, NY, USA, 238–252. <https://doi.org/10.1145/512950.512973>
- [5] Patrick Cousot and Radhia Cousot. 2002. Modular Static Program Analysis. In *Proceedings of the 11th International Conference on Compiler Construction* (CC ’02). Springer-Verlag, Berlin, Heidelberg, 159–178.
- [6] Yuichiro Futamura. 1971. Partial evaluation of computation process—an approach to a compilercompiler. *Systems and Computers in Japan* 2, 5 (1971), 45–50. <https://doi.org/10.1002/sys.10102>
- [7] Yoshihiko Futamura. 1999. Partial Evaluation of Computation Process, Revisited. *Higher-Order and Symbolic Computation* 12, 4 (01 Dec 1999), 377–380. <https://doi.org/10.1023/A:1010043619517>
- [8] Peter W. O’Hearn. 2019. Incorrectness logic. *Proc. ACM Program. Lang.* 4, POPL, Article 10 (Dec. 2019), 32 pages. <https://doi.org/10.1145/3371078>
- [9] Xavier Rival and Kwangkeun Yi. 2020. *Introduction to Static Analysis: An Abstract Interpretation Perspective*. The MIT Press, Cambridge, Massachusetts.
- [10] Helmut Seidl, Julian Erhard, and Ralf Vogler. 2020. *Incremental Abstract Interpretation*. Springer International Publishing, Cham, 132–148. https://doi.org/10.1007/978-3-030-41103-9_5
- [11] Tamás Szabó, Gábor Bergmann, Sebastian Erdweg, and Markus Voelter. 2018. Incrementalizing lattice-based program analyses in Datalog. *Proc. ACM Program. Lang.* 2, OOPSLA, Article 139 (Oct. 2018), 29 pages. <https://doi.org/10.1145/3276509>
- [12] Noah Van Es, Jens Van der Plas, Quentin Stiévenart, and Coen De Roover. 2020. MAF: A Framework for Modular Static Analysis of Higher-Order Languages. In *2020 IEEE 20th International Working Conference on Source Code Analysis and Manipulation (SCAM)*. 37–42. <https://doi.org/10.1109/SCAM51674.2020.00009>
- [13] Guannan Wei, Yuxuan Chen, and Tiark Rompf. 2019. Staged abstract interpreters: fast and modular whole-program analysis via meta-programming. *Proc. ACM Program. Lang.* 3, OOPSLA, Article 126 (Oct. 2019), 32 pages. <https://doi.org/10.1145/3360552>