Retargeting an Abstract Interpreter for a New Language by Partial Evaluation (Short Paper)

Jay Lee
jhlee@ropas.snu.ac.kr
Department of Computer Science and Engineering
Seoul National University
Korea

Joongwon Ahn
jwahn@ropas.snu.ac.kr
Department of Computer Science and Engineering
Seoul National University
Korea

Kwangkeun Yi
kwang@ropas.snu.ac.kr
Department of Computer Science and Engineering
Seoul National University
Korea

Abstract

Implementing sound static analyzers for new languages requires significant time and effort. We show that it is possible by partial evaluation to automatically retarget existing abstract interpreters for new languages. Given an existing abstract interpreter for a source language and a definitional interpreter for a new target language, we can derive a correct abstract interpreter for the target language by partially evaluating the abstract interpreter with respect to the definitional interpreter. We demonstrate our method on mini-languages and show that it is possible to mechanize the process with MetaOCaml. Our approach suggests a promising direction to reduce the burden of building new analyzers from scratch.

Keywords: Partial evaluation, Abstract interpretation

1 Introduction

Abstract interpretation [Cousot and Cousot 1977] provides a systematic framework to construct sound static analyzers by overapproximating a concrete semantics for a language. While abstract interpreters can be designed in a principled manner following a recipe [Rival and Yi 2020], it is time-consuming to model both a concrete semantics of a language and a corresponding abstract semantics. This burden is compounded when an analysis designer repeats the process for every new target language.

We propose automatically deriving abstract interpreters for new languages from an existing abstract interpreter. We show that partial evaluation [Futamura 1971, 1999] of an abstract interpreter for a source language, using the semantics of target languages written in the source language in a form of a definitional interpreter [Reynolds 1972], leads to a retargeted sound abstract interpreter for the new target. This eliminates the need to build analyzers for new languages from scratch.

Furthermore, we demonstrate using mini-languages that our approach is mechanizable by retargeting an abstract interpreter written in (BER) MetaOCaml [Calcagno et al. 2003; Kiselyov 2014].

2 Our Approach

There are three languages in play: a Meta-language, a **S**ource language, and a new T**T**arget language. The meta-language M is the implementation language of the abstract interpreter for language **S**, which also serves as the source language of a definitional interpreter for the new analysis target T.

Our goal is

- **given** a concrete interpreter $\mathbf{I}_{\mathbf{S}}^{\mathsf{T}}$ for T written in **S** (a "concrete T-interpreter," see section 3.1), and
- **given** an abstract interpreter $\mathsf{I}_{\mathsf{M}}^{\sharp\mathbf{S}}$ for **S** written in M (an "abstract **S**-interpreter," see section 3.2), to
- **derive** an abstract interpreter $\mathsf{I}_{\mathsf{M}}^{\sharp\mathsf{T}}$ for T written in M (an "abstract T-interpreter," see section 4).

A key observation is that partial evaluation $\mathcal{PE}$ in the M-language (section 3.3) allows for a *fully automatic* derivation of an abstract interpreter for T. In section 4, we show that the retargeted abstract interpreter is indeed sound:

Main Theorem (Correctness of a Retargeted Abstract Interpreter)**.** *Retargeting an abstract interpreter* $\mathsf{I}_{\mathsf{M}}^{\sharp\mathbf{S}}$ *with respect to an interpreter* $\mathbf{I}_{\mathbf{S}}^{\mathsf{T}}$ *results in a sound abstract interpreter* $\mathsf{I}_{\mathsf{M}}^{\sharp\mathsf{T}} \triangleq \mathcal{PE}(\mathsf{I}_{\mathsf{M}}^{\sharp\mathbf{S}}, \mathbf{I}_{\mathbf{S}}^{\mathsf{T}})$.

We finish by presenting a mechanized mini-example in section 5 to illustrate our approach with MetaOCaml.

3 The Ingredients

Notation. We distinguish programs and values of each language using different typefaces and colors, e.g., p for M-, **p** for **S**-, and p for T-programs. Explicit language name subscripts may be used when they are helpful, e.g., $\mathbb{P}_{\mathsf{M}}$, $\mathbb{P}_{\mathbf{S}}$, and $\mathbb{P}_{\mathsf{T}}$. Given domains $\mathbb{D}_1$ and $\mathbb{D}_2$, we denote the product as $\mathbb{D}_1 \times \mathbb{D}_2$, disjoint union as $\mathbb{D}_1 + \mathbb{D}_2$, and (Scott-)continuous function domain as $\mathbb{D}_1 \to \mathbb{D}_2$, where $\times$ has the highest precedence, $+$ has lower precedence, and $\to$ the lowest; $\times$ and $+$ are left-associative, whereas $\to$ is right-associative [Reynolds 1998]. We denote the set of partial functions from X to Y as $X \rightharpoonup Y$. Angle brackets denote semantic tuples, e.g., $\langle v_1, v_2 \rangle \in \mathbb{D}_1 \times \mathbb{D}_2$ when $v_1 \in \mathbb{D}_1$ and $v_2 \in \mathbb{D}_2$. $(\!|\cdot ? \cdot : \cdot|\!)$ is the semantic conditional operator.

We now present the ingredients of our recipe. While our recipe is tuned to be readily applied to the mini-example of section 5, exact signatures for semantic functions and domains may be adapted to other forms.

Definition 1 (Syntax and Semantics). Let $\mathbb{V}_M$ be the set of values in M, where $\mathbb{V}_M$ is closed under product $\mathbb{V}_M \times \mathbb{V}_M \subseteq \mathbb{V}_M$ and continuous functions $(\mathbb{V}_M \to \mathbb{V}_M) \subseteq \mathbb{V}_M$; and $\mathbb{S}_S$ and $\mathbb{S}_T$ be the sets of states in S and T. Let $\mathbb{P}_M$, $\mathbb{P}_S$, and $\mathbb{P}_T$ be programs of each language M, S, and T. Then we have concrete semantics functions

$$[\![p]\!] \in \mathbb{V}_M, \qquad [\![p]\!] \in \mathbb{S}_S \to \mathbb{S}_S, \text{ and } \qquad [\![p]\!] \in \mathbb{S}_T \to \mathbb{S}_T$$

for all $p \in \mathbb{P}_M$, $\mathbf{p} \in \mathbb{P}_S$, and $\mathsf{p} \in \mathbb{P}_T$. □

Moreover, each domain of defining-language values should be able to encode the programs and the values of the defined-language—this is expressed in definition 2.

3.1 An Interpreter for the New Language

The first ingredient is the semantics of the new language T: a definitional interpreter $\mathbf{I}_S^T$. This is written in the source language S for which we have an abstract interpreter $\mathsf{I}_M^{\sharp S}$.

For an interpreter to accept programs of the target language, we need to embed the program and the initial state of the target language into the source language.

Definition 2 (Encoders and Decoders). An *input encoder* $\lceil \cdot, \cdot \rceil_T^S \in \mathbb{P}_T \times \mathbb{S}_T \to \mathbb{S}_S$ is a function that encodes a T-program and a T-state into an S-state. This induces an *output decoder* $\lfloor \cdot \rfloor_T^S \in \mathbb{S}_S \rightharpoonup \mathbb{S}_T$, which is a partial function, where

$$\left\lfloor \lceil \mathsf{p}, \sigma \rceil_T^S \right\rfloor_T^S = \sigma$$

for all $\mathsf{p} \in \mathbb{P}_T$ and $\sigma \in \mathbb{S}_T$. We overload the notation for the lifted decoder, that is, for any $\overline{\boldsymbol{\sigma}} \in \wp(\mathbb{S}_S)$ we have

$$\lfloor \overline{\boldsymbol{\sigma}} \rfloor_T^S = \{ \lfloor \boldsymbol{\sigma} \rfloor_T^S \mid \boldsymbol{\sigma} \in \overline{\boldsymbol{\sigma}} \}. \qquad \square$$

In practice, encoders and decoders are usually part of an interpreter as parsers, and we can assume that they are available. For brevity, we shall omit the subscripts and superscripts of encoders and decoders as they can be unambiguously inferred.

Definition 3 (Concrete Interpreter). A concrete T-interpreter $\mathbf{I}_S^T \in \mathbb{P}_S$ is an S-program that accepts an S-state that encodes a T-program and a T-state, and returns an S-value that encodes the evaluation result. That is, $\mathbf{I}_S^T$ satisfies

$$\left\lfloor [\![\mathbf{I}_S^T]\!] \lceil \mathsf{p}, \sigma \rceil \right\rfloor = [\![\mathsf{p}]\!]\, \sigma \tag{1}$$

for all $\mathsf{p} \in \mathbb{P}_T$ and $\sigma \in \mathbb{S}_T$. □

Note that equation (1) enforces a few requirements on S. S-language should be expressive enough to encode the programs and values of T, and to write an interpreter for T.

3.2 An Existing Abstract Interpreter

Now we need the second ingredient: an existing abstract interpreter for S, $\mathsf{I}_M^{\sharp S}$. This is written in the meta-language M for which we have a partial evaluator $\mathcal{PE}$.

We define the abstract domain and the concretization function for the source language S in which the abstract interpreter $\mathsf{I}_M^{\sharp S}$ operates.

Definition 4 (Abstract Domain and Concretization). The abstract domain $\mathbb{S}_S^\sharp$ of S approximates the concrete domain $\mathbb{S}_S$ with an approximation partial order $\sqsubseteq_S$. An abstract state can be concretized to a set of concrete states using a monotonic concretization function $\boldsymbol{\gamma} \in \mathbb{S}_S^\sharp \to \wp(\mathbb{S}_S)$, where for any $\boldsymbol{\sigma}_1^\sharp, \boldsymbol{\sigma}_2^\sharp \in \mathbb{S}_S^\sharp$, we have

$$\boldsymbol{\sigma}_1^\sharp \sqsubseteq_S \boldsymbol{\sigma}_2^\sharp \implies \boldsymbol{\gamma}\, \boldsymbol{\sigma}_1^\sharp \subseteq \boldsymbol{\gamma}\, \boldsymbol{\sigma}_2^\sharp.$$

Moreover, we overload the notation of encoder and decoder for programs and abstract states, that is, we have $\lceil \cdot \rceil \in \mathbb{P}_S + \mathbb{S}_S^\sharp \to \mathbb{V}_M$, $\lceil \cdot, \cdot \rceil \in \mathbb{P}_S \times \mathbb{S}_S^\sharp \to \mathbb{V}_M \times \mathbb{V}_M$ and $\lfloor \cdot \rfloor \in \mathbb{V}_M \rightharpoonup \mathbb{S}_S^\sharp$, where

$$\lceil \mathbf{p}, \boldsymbol{\sigma}^\sharp \rceil = \langle \lceil \mathbf{p} \rceil, \lceil \boldsymbol{\sigma}^\sharp \rceil \rangle$$

and

$$\left\lfloor \lceil \boldsymbol{\sigma}^\sharp \rceil \right\rfloor = \boldsymbol{\sigma}^\sharp. \qquad \square$$

With definition 4, we can define a correctness condition for an abstract S-interpreter $\mathsf{I}_M^{\sharp S}$.

Definition 5 (Abstract Interpreter). An abstract S-interpreter $\mathsf{I}_M^{\sharp S} \in \mathbb{P}_M$ is an M-program that accepts an S-program and an S-state, and returns a sound approximation of the evaluation result with respect to the concretization $\boldsymbol{\gamma}$, as defined in definition 4. That is, $\mathsf{I}_M^{\sharp S}$ satisfies

$$\boldsymbol{\sigma} \in \boldsymbol{\gamma}\, \boldsymbol{\sigma}^\sharp \implies [\![\mathbf{p}]\!]\, \boldsymbol{\sigma} \in \boldsymbol{\gamma} \left\lfloor [\![\mathsf{I}_M^{\sharp S}]\!] \lceil \mathbf{p}, \boldsymbol{\sigma}^\sharp \rceil \right\rfloor \tag{2}$$

for all $\mathbf{p} \in \mathbb{P}_S$, $\boldsymbol{\sigma}^\sharp \in \mathbb{S}_S^\sharp$ and $\boldsymbol{\sigma} \in \mathbb{S}_S$. □

Again, equation (2) imposes requirements on the expressiveness of M. The meta-language M should be expressive enough to encode the programs and values of S, and write an abstract interpreter for S. In addition, M should be able to construct and represent tuples and functions, which explains the requirements in definition 1.

3.3 Retargeting using a Partial Evaluator

The final ingredient for automatically deriving a retargeted abstract interpreter is a partial evaluator for M: $\mathcal{PE} \in \mathbb{P}_M \times \mathbb{V}_M \to \mathbb{P}_M$.

Definition 6 (Partial Evaluator). A partial evaluator $\mathcal{PE} \in \mathbb{P}_M \times \mathbb{V}_M \to \mathbb{P}_M$ accepts two inputs,

1. a program $p \in \mathbb{P}_M$ that accepts $\langle v_1, v_2 \rangle \in \mathbb{V}_M$, and
2. a value $v_1 \in \mathbb{V}_M$,

and returns a specialized program that accepts the remaining value $v_2 \in \mathbb{V}_M$. That is, $\mathcal{PE}$ satisfies

$$[\![\mathcal{PE}(p, v_1)]\!]\, v_2 = [\![p]\!] \langle v_1, v_2 \rangle \tag{3}$$

for all $p \in \mathbb{P}_M$ and $v_1, v_2 \in \mathbb{V}_M$. □

From equation (3), we can specialize an abstract interpreter with a target program by substituting $\mathsf{I}_{\mathsf{M}}^{\sharp\mathbf{S}}$ for p:

$$[\![\mathcal{PE}(\mathsf{I}_{\mathsf{M}}^{\sharp\mathbf{S}}, \ulcorner \mathsf{p} \urcorner)]\!]\ \ulcorner \sigma^{\sharp} \urcorner = [\![\mathsf{I}_{\mathsf{M}}^{\sharp\mathbf{S}}]\!]\ \ulcorner \mathsf{p}, \sigma^{\sharp} \urcorner. \tag{4}$$

This is essentially the first Futamura projection extended to abstract interpreters.

4 The Correctness Theorem

Before we show the main theorem, we prove the correctness of a meta-level analysis.

Lemma 1 (Correctness of a Meta-level Analysis). *Analyzing a* T*-program using an abstract interpreter* $\mathsf{I}_{\mathsf{M}}^{\sharp\mathbf{S}}$ *along with the concrete interpreter* $\mathbf{I}_{\mathbf{S}}^{\mathsf{T}}$ *is sound.*

Proof. Suppose we are given a T-program p and a T-state σ. Then by substituting $\mathbf{I}_{\mathbf{S}}^{\mathsf{T}}$ for $\mathbf{p}$ and $\ulcorner \mathsf{p}, \sigma \urcorner$ for $\boldsymbol{\sigma}$ in equation (2),

$$\begin{aligned} \ulcorner \mathsf{p}, \sigma \urcorner \in \boldsymbol{\gamma}\ \boldsymbol{\sigma}^{\sharp} &\implies [\![\mathbf{I}_{\mathbf{S}}^{\mathsf{T}}]\!] \ulcorner \mathsf{p}, \sigma \urcorner \in \boldsymbol{\gamma} \lfloor [\![\mathsf{I}_{\mathsf{M}}^{\sharp\mathbf{S}}]\!]\ \ulcorner \mathbf{I}_{\mathbf{S}}^{\mathsf{T}}, \boldsymbol{\sigma}^{\sharp} \urcorner \rfloor \\ &\implies \lfloor [\![\mathbf{I}_{\mathbf{S}}^{\mathsf{T}}]\!] \ulcorner \mathsf{p}, \sigma \urcorner \rfloor \in \lfloor \boldsymbol{\gamma} \lfloor [\![\mathsf{I}_{\mathsf{M}}^{\sharp\mathbf{S}}]\!]\ \ulcorner \mathbf{I}_{\mathbf{S}}^{\mathsf{T}}, \boldsymbol{\sigma}^{\sharp} \urcorner \rfloor \rfloor \end{aligned}$$

Then from equation (1), $\lfloor [\![\mathbf{I}_{\mathbf{S}}^{\mathsf{T}}]\!] \ulcorner \mathsf{p}, \sigma \urcorner \rfloor$ is precisely $[\![\mathsf{p}]\!]\ \sigma$. Thus the analysis $[\![\mathsf{I}_{\mathsf{M}}^{\sharp\mathbf{S}}]\!]\ \ulcorner \mathbf{I}_{\mathbf{S}}^{\mathsf{T}}, \boldsymbol{\sigma}^{\sharp} \urcorner$ subsumes the concrete evaluation $[\![\mathsf{p}]\!]\ \sigma$ when decoded. ∎

Main Theorem (Correctness of a Retargeted Abstract Interpreter). *Retargeting an abstract interpreter* $\mathsf{I}_{\mathsf{M}}^{\sharp\mathbf{S}}$ *with respect to an interpreter* $\mathbf{I}_{\mathbf{S}}^{\mathsf{T}}$ *results in a sound abstract interpreter* $\mathsf{I}_{\mathsf{M}}^{\sharp\mathsf{T}} \triangleq \mathcal{PE}(\mathsf{I}_{\mathsf{M}}^{\sharp\mathbf{S}}, \ulcorner \mathbf{I}_{\mathbf{S}}^{\mathsf{T}} \urcorner)$.

Proof. Given a partial evaluator $\mathcal{PE}$ that satisfies equation (3), we can partially evaluate the abstract interpreter $\mathsf{I}_{\mathsf{M}}^{\sharp\mathbf{S}}$ with respect to the interpreter $\mathbf{I}_{\mathbf{S}}^{\mathsf{T}}$ as

$$\mathsf{I}_{\mathsf{M}}^{\sharp\mathsf{T}} \triangleq \mathcal{PE}(\mathsf{I}_{\mathsf{M}}^{\sharp\mathbf{S}}, \ulcorner \mathbf{I}_{\mathbf{S}}^{\mathsf{T}} \urcorner).$$

We now show that the specialized $\mathsf{I}_{\mathsf{M}}^{\sharp\mathsf{T}}$ is indeed a correct static analyzer of T. Fix a T-program p and a T-state σ. Given $\boldsymbol{\sigma}^{\sharp}$ such that $\ulcorner \mathsf{p}, \sigma \urcorner \in \boldsymbol{\gamma}\ \boldsymbol{\sigma}^{\sharp}$, from equation (4),

$$[\![\mathsf{I}_{\mathsf{M}}^{\sharp\mathsf{T}}]\!]\ \ulcorner \boldsymbol{\sigma}^{\sharp} \urcorner = [\![\mathcal{PE}(\mathsf{I}_{\mathsf{M}}^{\sharp\mathbf{S}}, \ulcorner \mathbf{I}_{\mathbf{S}}^{\mathsf{T}} \urcorner)]\!]\ \ulcorner \boldsymbol{\sigma}^{\sharp} \urcorner = [\![\mathsf{I}_{\mathsf{M}}^{\sharp\mathbf{S}}]\!]\ \ulcorner \mathbf{I}_{\mathbf{S}}^{\mathsf{T}}, \boldsymbol{\sigma}^{\sharp} \urcorner$$

and from lemma 1, this soundly approximates the evaluation

$$[\![\mathsf{p}]\!]\ \sigma \in \lfloor \boldsymbol{\gamma} \lfloor [\![\mathsf{I}_{\mathsf{M}}^{\sharp\mathbf{S}}]\!]\ \ulcorner \mathbf{I}_{\mathbf{S}}^{\mathsf{T}}, \boldsymbol{\sigma}^{\sharp} \urcorner \rfloor \rfloor$$

Thus the specialized abstract interpreter $\mathsf{I}_{\mathsf{M}}^{\sharp\mathsf{T}}$ is correct. ∎

5 A Mechanized Example

We now present a concrete example that demonstrates how our approach retargets an abstract interpreter $\mathsf{I}_{\mathsf{M}}^{\sharp\mathbf{S}}$ for a source language **S** to an abstract interpreter $\mathsf{I}_{\mathsf{M}}^{\sharp\mathsf{T}}$ for a target language T. For mechanization, we use MetaOCaml to be the meta-language M. This example illustrates how our method works in practice and highlights the properties of the resulting retargeted abstract interpreter.

Consider mini-languages T and **S**:

$$\mathbb{P}_{\mathsf{T}}\ \mathsf{p} ::= \texttt{ADD n} \mid \texttt{MUL n} \mid \mathsf{p};\ \mathsf{p}$$

$$\mathbb{E}_{\mathbf{S}}\ \mathbf{e} ::= \mathbf{n} \mid \mathbf{*e} \mid \mathbf{e + e} \mid \mathbf{e * e} \mid \mathbf{e == e}$$

$$\mathbb{P}_{\mathbf{S}}\ \mathbf{p} ::= \mathbf{skip} \mid \mathbf{*e = e} \mid \mathbf{p; p} \mid \mathbf{if\ (e)\ \{p\}\ else\ \{p\}} \mid \mathbf{while\ (e)\ \{p\}}$$

T is an assembly-like language with a single register, e.g., `ADD 42; MUL 3` returns $(\sigma + 42) \times 3$, given an initial register value σ. **S** is a structured language with a memory that allows arbitrary assignment at natural number addresses, e.g, `*(13 + 37) = 0` transforms the value at address 50 to 0.

Given the integer value domains $\mathbb{V}_{\mathsf{T}} = \mathbb{V}_{\mathbf{S}} = \mathbb{Z}$ and state domains $\mathbb{S}_{\mathsf{T}} = \mathbb{V}_{\mathsf{T}}$ as a single register value and $\mathbb{S}_{\mathbf{S}} = \overline{\mathbb{V}}_{\mathbf{S}}$ as an array of memory values, the semantics $[\![\cdot]\!]$ and $[\![\cdot]\!]$ and the encoder $\ulcorner \cdot, \cdot \urcorner$ are given as follows:

$$[\![\mathsf{p}]\!] \in \mathbb{V}_{\mathsf{T}} \to \mathbb{V}_{\mathsf{T}} \qquad \ulcorner \mathsf{p}, \sigma \urcorner \in \overline{\mathbb{V}}_{\mathbf{S}}$$

$$[\![\texttt{ADD n}]\!] \triangleq \lambda\sigma.\, \mathsf{n} + \sigma \qquad [\![\texttt{MUL n}]\!] \triangleq \lambda\sigma.\, \mathsf{n} \times \sigma$$

$$[\![\mathsf{p}_1;\ \mathsf{p}_2]\!] \triangleq [\![\mathsf{p}_2]\!] \circ [\![\mathsf{p}_1]\!]$$

$$\ulcorner \mathsf{p}_1;\ \ldots;\ \mathsf{p}_k, \sigma \urcorner \triangleq \sigma :: 2 :: \mathop{+\!\!+}_{i=1}^{k} \left[(\mathsf{p}_i = \texttt{ADD n}\ ?\ 1 : 2), \mathsf{n} \right] +\!\!+ [0]$$

$$[\![\mathbf{e}]\!] \in \mathbb{S}_{\mathbf{S}} \to \mathbb{V}_{\mathbf{S}} \qquad [\![\mathbf{p}]\!] \in \mathbb{S}_{\mathbf{S}} \to \mathbb{S}_{\mathbf{S}}$$

$$[\![\mathbf{n}]\!] \triangleq \lambda\boldsymbol{\sigma}.\, \mathbf{n} \qquad [\![\mathbf{*e}]\!] \triangleq \lambda\boldsymbol{\sigma}.\, \boldsymbol{\sigma}[[\![\mathbf{e}]\!]\boldsymbol{\sigma}]$$

$$[\![\mathbf{e}_1 + \mathbf{e}_2]\!] \triangleq \lambda\boldsymbol{\sigma}.\, [\![\mathbf{e}_1]\!]\boldsymbol{\sigma} + [\![\mathbf{e}_2]\!]\boldsymbol{\sigma}$$

$$[\![\mathbf{e}_1 * \mathbf{e}_2]\!] \triangleq \lambda\boldsymbol{\sigma}.\, [\![\mathbf{e}_1]\!]\boldsymbol{\sigma} \times [\![\mathbf{e}_2]\!]\boldsymbol{\sigma}$$

$$[\![\mathbf{e}_1 == \mathbf{e}_2]\!] \triangleq \lambda\boldsymbol{\sigma}.\, ([\![\mathbf{e}_1]\!]\boldsymbol{\sigma} = [\![\mathbf{e}_2]\!]\boldsymbol{\sigma}\ ?\ 1 : 0)$$

$$[\![\mathbf{skip}]\!] \triangleq \lambda\boldsymbol{\sigma}.\, \boldsymbol{\sigma} \qquad [\![\mathbf{*e}_1 = \mathbf{e}_2]\!] \triangleq \lambda\boldsymbol{\sigma}.\, \boldsymbol{\sigma}[[\![\mathbf{e}_1]\!] \mapsto [\![\mathbf{e}_2]\!]]$$

$$[\![\mathbf{p}_1; \mathbf{p}_2]\!] \triangleq [\![\mathbf{p}_2]\!] \circ [\![\mathbf{p}_1]\!]$$

$$[\![\mathbf{if\ (e)\ \{p_1\}\ else\ \{p_2\}}]\!] \triangleq \lambda\boldsymbol{\sigma}.\, ([\![\mathbf{e}]\!]\boldsymbol{\sigma} \neq 0\ ?\ [\![\mathbf{p}_1]\!]\boldsymbol{\sigma} : [\![\mathbf{p}_2]\!]\boldsymbol{\sigma})$$

$$[\![\mathbf{while\ (e)\ \{p\}}]\!] \triangleq \lambda\boldsymbol{\sigma}.\, ([\![\mathbf{e}]\!]\boldsymbol{\sigma} = 0\ ?\ \boldsymbol{\sigma} : [\![\mathbf{while\ (e)\ \{p\}}]\!] \circ [\![\mathbf{p}]\!]\ \boldsymbol{\sigma})$$

To elaborate on the encoding $\ulcorner \mathsf{p}, \sigma \urcorner$, we place the register value σ and the program counter—initially 2—at the first and second addresses. Each instruction is stored with an alignment of two words: the first cell stores the operation code—`ADD` is 1 and `MUL` is 2—and the second stores the argument. The program finishes with the operation code 0 for halting. We omit the presentation of $\mathbb{P}_{\mathsf{M}}$, $\mathbb{V}_{\mathsf{M}}$, $\ulcorner \cdot \urcorner$, and $[\![\mathsf{e}]\!]$ of MetaOCaml (M) for brevity. The encoder $\ulcorner \cdot \urcorner$ is a typical parser that embeds the AST of an **S**-program or an **S**-value into its corresponding M-datatype representation.

Now we can define the concrete T-interpreter

```
1 I_S^T = while (*PC) {
2   if (*PC == 1) { R = R + *(PC + 1) }
3   else if (*PC == 2) { R = R * *(PC + 1) }
4   else { skip };
5   PC = PC + 2 }
```

where macros `R` = `*0` and `PC` = `*1` are used for convenience.

The abstract interpreter $\mathsf{I}^{\sharp\mathbf{S}}_{\mathsf{M}}$ parameterized by the base value abstraction is shown below in (pseudo-)MetaOCaml. Brackets .<...>. wraps code to generate, and an escape .~ represents a hole inside the code template.

```
1  let I^#S_M(p, σ#) = evalp#(p, σ#)
2  where evale#(e, σ#) = match e with
3    | n → .<η(n)>.
4    | *e → .< .~σ#[.~evale#(e, σ#)] >.
5    | e1 + e2 → .< .~evale#(e1, σ#) +# .~evale#(e2, σ#) >.
6    | e1 * e2 → .< .~evale#(e1, σ#) ×# .~evale#(e2, σ#) >.
7    | e1 == e2 → .< .~evale#(e1, σ#) =# .~evale#(e2, σ#) >.
8  and evalp#(p, σ#) = match p with
9    | skip → σ#
10   | *e1 = e2 → .< .~σ#[ .~evale#(e1, σ#) ↦# .~evale#(e2, σ#) ] >.
11   | p1; p2 → .< .~evalp#(p2, evalp#(p1, σ#)) >.
12   | if (e) {pt} else {pf} →
13     .< let v# = .~evale#(e, σ#) in
14        F#_{≠0}(v#, .~evalp#(pt, σ#)) ⊔# F#_{=0}(v#, .~evalp#(pf, σ#)) >.
15   | while (e) {pb} →
16     .< let v# = .~evale#(e, σ#) in
17       .~evalp#(p, evalp#(pb, .< F#_{≠0}(v#, σ#) >.)) ⊔# F#_{=0}(v#, σ#) >.
```

The parametrizable knob is given by the extraction function η [Darais and Horn 2019; Nielson et al. 1999] that extracts the abstract value from a single concrete value. Abstract operators are defined correspondingly: abstract operators $+^\sharp$, $\times^\sharp$, $=^\sharp$ correspond to $+$, $\times$, $=$, respectively; $\sqcup^\sharp$ is an abstract join; and $\mathcal{F}^\sharp_{\neq 0}$ and $\mathcal{F}^\sharp_{=0}$ filter abstract values based on the predicate $\neq 0$ and $= 0$, respectively [Rival and Yi 2020].

To ensure termination, we have also modified the above code into an open-recursive style in the actual implementation, *à la* Abstracting Definitional Interpreters [Darais et al. 2017].

Performing the specialization by applying $\mathbf{I}^\top_{\mathbf{S}}$ to the above function, we get the retargeted abstract interpreter $\mathsf{I}^{\sharp\top}_{\mathsf{M}} = \mathcal{PE}(\mathsf{I}^{\sharp\mathbf{S}}_{\mathsf{M}}, \lceil \mathbf{I}^\top_{\mathbf{S}} \rceil)$ as[1]

```
1  let I^#S_M [I^⊤_S](σ#) =
2    let v1 = σ#[σ#[η(1)]] in
3    let σ#_2 = F#_{≠0}(v1, σ#) in
4    let v3 = σ#_2[σ#_2[η(1)]] =# η(0) in
5    let σ#_4 = F#_{≠0}(v3, σ#_2) in let σ#_5 = F#_{=0}(v3, σ#_2) in
6    let v6 = σ#_5[σ#_5[η(1)]] =# η(2) in
7    let σ#_7 = F#_{≠0}(v6, σ#_5) in let σ#_8 = F#_{=0}(v6, σ#_5) in
8    let σ#_9 =
9      σ#_4[η(0) ↦# σ#_4[η(0)] +# σ#_4[σ#_4[η(1)] +# η(1)]] ⊔#
10     σ#_7[η(0) ↦# σ#_7[η(0)] ×# σ#_7[σ#_7[η(1)] +# η(1)]] ⊔# σ#_8 in
11   F#_{=0}(v1, σ#) ⊔# I^#S_M [I^⊤_S]([σ#_9[η(1) ↦# σ#_9[η(1)] +# η(2)]])
```

where the subscript $\mathbf{I}^\top_{\mathbf{S}}$ indicates that the source code $\mathsf{I}^{\sharp\mathbf{S}}_{\mathsf{M}}$ is specialized with respect to the concrete interpreter $\mathbf{I}^\top_{\mathbf{S}}$.

[1]The open-recursive style has been manually restored to a recursive style for presentation.

Notice that while the retargeted abstract interpreter $\mathsf{I}^{\sharp\top}_{\mathsf{M}}$ is written in the meta-language M, the (abstract-)interpretative overhead is eliminated and the structure of the interpreter $\mathbf{I}^\top_{\mathbf{S}}$ is closely mirrored: two abstract joins in lines 9–10 mirror the if-elif-else structure of the loop body, and the recursive call in line 11 mirrors the while loop of $\mathbf{I}^\top_{\mathbf{S}}$. This is a classical phenomenon in partial evaluation [Jones 1996].

Moreover, observe the key advantage of our approach: $\mathsf{I}^{\sharp\top}_{\mathsf{M}}$ reuses the abstract operators $+^\sharp$ (line 9) and $\times^\sharp$ (line 10) from the existing $\mathsf{I}^{\sharp\mathbf{S}}_{\mathsf{M}}$ to analyze the T-programs `ADD n` and `MUL n`, resulting in a sound analyzer (main theorem). Our approach enjoys the inherited properties from the existing (abstract) interpreter, as observed by Reynolds [1972].

6 Related Work

Meta-level Analysis. The effort to automatically keep the analysis of JavaScript (JS) programs in sync with the rapidly evolving ECMA-262 [Guo et al. 2025] specification motivated the development of a meta-level static analyzer JSAVER [Park et al. 2022]. JSAVER targets a low-level language IR_{ES}, which is used to describe the semantics of JS. By applying a meta-level analysis to a definitional interpreter of JS written in IR_{ES}, JSAVER can analyze JS programs. In this specialized setting, JS and IR_{ES} share the same base value domain.

We extend this work by

1. providing a language-agnostic framework, even allowing source and target languages to have different value domains; and
2. addressing the performance limitations mentioned in their work through partial evaluation.

Partial Evaluation. It is well known that partial evaluation of an interpreter with respect to a program leads to compilation of the program—this is called the first Futamura [1971, 1999] projection. The compiled program is essentially a specialized interpreter that does not need to traverse the AST of the original program, which eliminates interpretative overhead [Jones 1996].

The same idea of improving analysis performance by applying the first Futamura projection to abstract definitional interpreters [Darais et al. 2017] was demonstrated in staged abstract interpreters [Wei et al. 2019]. Since the analyzer is specialized to the target program, the (abstract) interpretative overhead is eliminated, resulting in efficient analysis.

Peralta et al. [1998] proposed a method for partially evaluating a definitional interpreter for an imperative language written in constraint logic programming. The resulting specialized interpreter—an imperative program compiled into a constraint logic program—can then be analyzed using an existing static analyzer for logic programming languages. While their approach produces an efficient representation of individual target programs, our approach yields a general analyzer applicable to any program in the target language.

Note that we stage an abstract interpreter with respect to a *definitional interpreter* for a new language rather than an analysis target program, representing a fundamentally different use of partial evaluation compared to prior work.

Reusing Abstractions. Reusing abstractions for static analysis is an ongoing challenge. Skeletal semantics [Bodin et al. 2019] can be used to derive abstract interpreters by integrating meta-language abstractions with language-specific ones [Jensen et al. 2023]. Composing modular analysis components has been proposed by Keidel and Erdweg [2019], which employs arrows meta-language [Hughes 2000]. An open-source static analysis framework Mopsa [Journault et al. 2020] leverages OCaml's extensible variants to easily define new targets and reuse abstractions compositionally.

Our method follows a top-down approach of recycling an entire abstract interpreter to retarget to a new one, instead of composing analysis components in a bottom-up fashion.

7 Discussion

Our recipe for retargeting abstract interpreters can drastically reduce the effort required to develop static analyzers for various applications. For example, a JavaScript static analyzer such as JSAVER [Park et al. 2022] can be retargeted to analyze React applications using a React definitional interpreter like React-tRace [Lee et al. 2025].

We are currently working on identifying the properties of the retargeted abstract interpreter—for example, the effect of the semantic gap between the source and target languages, its precision and performance characteristics, and approaches to inheriting analysis sensitivity for the target language. We believe that our recipe can be extended to provide a knob for analysis designers, enabling sensitivity control over the analyses of target programs.

Acknowledgments

We thank John Patrick Gallagher and anonymous reviewers for their valuable feedback.

A preliminary version of this work was presented at the Student Research Competition (SRC) at the 46th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '25) [Lee 2025].

This work was supported by the National Research Foundation of Korea (NRF) grant funded by the Korea government (MSIT) (0536-20250107), BK21 FOUR Intelligence Computing (Dept. of Computer Science and Engineering, SNU) funded by National Research Foundation of Korea (NRF) (4199990214639), Greenlabs Co., Ltd. (0536-20220078), and Samsung Electronics Co., Ltd. (0536-20230088).

References

Martin Bodin, Philippa Gardner, Thomas Jensen, and Alan Schmitt. 2019. Skeletal Semantics and Their Interpretations. *Proc. ACM Program. Lang.* 3, POPL, Article 44 (Jan. 2019), 31 pages. doi:10.1145/3290357

Cristiano Calcagno, Walid Taha, Liwen Huang, and Xavier Leroy. 2003. Implementing multi-stage languages using ASTs, Gensym, and reflection. In *Proceedings of the 2nd International Conference on Generative Programming and Component Engineering* (Erfurt, Germany) *(GPCE '03)*. Springer-Verlag, Berlin, Heidelberg, 57–76.

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. doi:10.1145/512950.512973

David Darais and David Van Horn. 2019. Constructive Galois Connections. *J. Funct. Program.* 29 (2019), e11. doi:10.1017/S0956796819000066

David Darais, Nicholas Labich, Phúc C. Nguyễn, and David Van Horn. 2017. Abstracting Definitional Interpreters (Functional Pearl). *Proc. ACM Program. Lang.* 1, ICFP, Article 12 (Aug. 2017), 25 pages. doi:10.1145/3110256

Yoshihiko Futamura. 1971. Partial Evaluation of Computation Process—An Approach to a Compiler-Compiler. *Systems. Computers. Controls* 2, 5 (1971), 45–50.

Yoshihiko Futamura. 1999. Partial Evaluation of Computation Process—An Approach to a Compiler-Compiler. *Higher Order Symbol. Comput.* 12, 4 (Dec. 1999), 381–391. doi:10.1023/A:1010095604496

Shu-yu Guo, Michael Ficarra, and Kevin Gibbons. 2025. ECMAScript® 2025 Language Specification. https://tc39.es/ecma262/

John Hughes. 2000. Generalising Monads to Arrows. *Sci. Comput. Program.* 37, 1-3 (May 2000), 67–111. doi:10.1016/S0167-6423(99)00023-4

Thomas Jensen, Vincent Rébiscoul, and Alan Schmitt. 2023. Deriving Abstract Interpreters from Skeletal Semantics. *Electron. Proc. Theo. Comput. Sci.* 387 (Sept. 2023), 97–113. doi:10.4204/EPTCS.387.8

Neil D. Jones. 1996. An Introduction to Partial Evaluation. *ACM Comput. Surv.* 28, 3 (Sept. 1996), 480–503. doi:10.1145/243439.243447

Matthieu Journault, Antoine Miné, Raphaël Monat, and Abdelraouf Ouadjaout. 2020. Combinations of Reusable Abstract Domains for a Multilingual Static Analyzer. In *Verified Software. Theories, Tools, and Experiments*, Supratik Chakraborty and Jorge A. Navas (Eds.). Vol. 12031. Springer, Cham, 1–18. doi:10.1007/978-3-030-41600-3_1

Sven Keidel and Sebastian Erdweg. 2019. Sound and Reusable Components for Abstract Interpretation. *Proc. ACM Program. Lang.* 3, OOPSLA, Article 176 (Oct. 2019), 28 pages. doi:10.1145/3360602

Oleg Kiselyov. 2014. The Design and Implementation of BER MetaOCaml. In *Functional and Logic Programming*, Michael Codish and Eijiro Sumii (Eds.). Springer International Publishing, Cham, 86–102.

Jay Lee. 2025. Retargeting an Abstract Interpreter for a New Language by Partial Evaluation. arXiv:2507.04316 [cs.PL] https://arxiv.org/abs/2507.04316

Jay Lee, Joongwon Ahn, and Kwangkeun Yi. 2025. React-tRace: A Semantics for Understanding React Hooks: An Operational Semantics and a Visualizer for Clarifying React Hooks. *Proc. ACM Program. Lang.* 9, OOPSLA2, Article 289 (Oct. 2025), 28 pages. doi:10.1145/3763067

Flemming Nielson, Hanne Riis Nielson, and Hankin Chris. 1999. *Principles of Program Analysis.* Springer, Berlin, Heidelberg.

Jihyeok Park, Seungmin An, and Sukyoung Ryu. 2022. Automatically Deriving JavaScript Static Analyzers from Specifications using Meta-level Static Analysis. In *Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering* (Singapore, Singapore) *(ESEC/FSE 2022)*. Association for Computing Machinery, New York, NY, USA, 1022–1034. doi:10.1145/3540250.3549097

Julio C. Peralta, John P. Gallagher, and Hüseyin Sağlam. 1998. Analysis of Imperative Programs through Analysis of Constraint Logic Programs. In *Static Analysis*, Gerhard Goos, Juris Hartmanis, Jan Van Leeuwen, and Giorgio Levi (Eds.). Vol. 1503. Springer Berlin Heidelberg, Berlin, Heidelberg, 246–261. doi:10.1007/3-540-49727-7_15

John C. Reynolds. 1972. Definitional Interpreters for Higher-Order Programming Languages. In *Proceedings of the ACM Annual Conference - Volume 2* (Boston, Massachusetts, USA) *(ACM '72)*. Association for Computing Machinery, New York, NY, USA, 717–740. doi:10.1145/800194.805852

John C. Reynolds. 1998. *Theories of Programming Languages*. Cambridge University Press, Cambridge, UK.

Xavier Rival and Kwangkeun Yi. 2020. *Introduction to Static Analysis: An Abstract Interpretation Perspective*. The MIT Press, Cambridge, Massachusetts.

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. doi:10.1145/3360552