On the Structure of Abstract Interpreters

Wonyeol Lee, Matthieu Lemerre, Xavier Rival, Hongseok Yang

On the Structure of Abstract Interpreters

Wonyeol Lee
Computer Science & Engineering, POSTECH
South Korea

Matthieu Lemerre
Université Paris-Saclay, CEA, List
France

Xavier Rival
DIENS
INRIA Paris & DIENS, ENS, CNRS, and PSL University
France

Hongseok Yang
School of Computing
KAIST
South Korea

Abstract

Static analysis aims at computing semantic properties of programs. Abstract interpretation provides a framework to design static analyses and allows one to divide the construction of a static analysis into the definition of abstract domains that describe families of logical predicates with operations to reason on them, and the semantic-guided formalisation of abstract interpreters. The latter relies on the abstract domains, that describe semantic properties, and on the concrete semantics. A large part of the research on static analysis focuses on the design of novel abstract domains, with ever more expressive and/or efficient computer representation for semantic properties. In this short paper, we consider more specifically the core of the abstract interpreters (also called the abstract iterators) and discuss several techniques to build them, that are inspired by functional programming. First, we briefly discuss common iteration techniques based on control flow graphs, which have often been used for program analyses aimed at computing state properties. Second, we consider iteration techniques that borrow principles from denotational semantics and are typically defined by induction over the syntax of programs. Last, we extend the latter family of techniques to relational abstractions that capture relations between program inputs and outputs.

CCS Concepts: • **Theory of computation** → **Denotational semantics**; **Operational semantics**; **Program analysis**; **Abstraction**; **Invariants**.

Keywords: semantics, static analysis, abstract interpretation.

ACM Reference Format:
Wonyeol Lee, Matthieu Lemerre, Xavier Rival, and Hongseok Yang. 2025. On the Structure of Abstract Interpreters. In *Proceedings of the Workshop Dedicated to Olivier Danvy on the Occasion of His 64th Birthday (OLIVIERFEST '25), October 12–18, 2025, Singapore, Singapore.* ACM, New York, NY, USA, 7 pages. https://doi.org/10.1145/3759427.3760368

OLIVIERFEST '25, Singapore, Singapore

ACM ISBN 979-8-4007-2150-2/25/10
https://doi.org/10.1145/3759427.3760368

1 Introduction

In this short paper, we review several iteration techniques to be used in abstract interpretation based static analysers. At a high-level, a static analyser can generally be split into two parts: (1) the abstract domain implements the basic properties that the analysis manipulates and the operations on them, and (2) the abstract interpreter that computes an approximation of the semantics of the program. The latter calls the abstract domain operations to analyse each basic program step and typically performs some abstract (post-)fixpoint computation for loops and recursive functions.

Among classical abstract interpreter designs, a common approach proceeds by iteration over the control flow graph (CFG) of the program to analyse [3, 4, 7, 21]. This technique boils down to an abstract interpretation of an operational semantics. Another popular approach abstracts the semantics of each statement without requiring its CFG [5, 11, 19] and simply uses the abstract syntax tree (AST) instead. This second technique directly performs an abstract interpretation of the collecting semantics of commands. It is particularly well adapted to the use of functional programming languages (such as ML, OCaml, or Haskell) for the implementation of static analysers. The AST-based approach also naturally leads to the design of *definitional abstract interpreters* [14], which are built upon abstract machines [1, 2].

Both the CFG-based and the AST-based approaches to static analysis design have compelling use cases [22]. In this short paper, we discuss the advantages of both techniques. More specifically, we illustrate the merits of the AST-based approach. In particular, it allows one to remain close to a concrete interpreter while computing over abstract states. It also leads to compact and elegant definitions of static analysers, which is the main motivation for us to present this approach in this essay, dedicated to Olivier Danvy, who brought major contributions to the design of elegant and compositional interpreters.

In Section 2, we fix the syntax, operational semantics, and collecting semantics of a toy imperative programming

language. In Section 3, we discuss target properties and semantic abstractions. Then, we describe iteration techniques for state abstractions in the next two sections. We consider iteration on control-flow graphs in Section 4 and iteration on abstract syntax trees in Section 5. Finally, the last sections study two iteration techniques for relational abstractions. In Section 6, we consider bottom up static analysis, where the abstraction of a command is computed based on that of the components of the command. In Section 7, we extend the iteration technique based on abstract syntax trees to relational abstractions, so as to compute a relational abstraction for a command using a classical forward analysis.

2 Syntax, Operational Semantics, and Collecting Semantics

In this paper, we use a very restricted basic imperative language as it provides a sufficient setup to illustrate several families of iteration techniques.

Syntax. We assume a fixed, finite set of variables $\mathbb{X}$, a set of scalar values $\mathbb{V}$, and a set of operations $\mathbb{O}$ (which are assumed deterministic and without errors for simplicity). We consider a basic programming language with expressions and commands. An expression E may be a constant value, the reading of a variable, or the application of an operation to a tuple of expressions (in examples, we write binary operators in the more usual inline notation). We write $\mathbf{fv}(E)$ for the free variables of expression E. A command C may be the skip command that does nothing, an assignment, a sequence, a condition, or a loop. The syntax of expressions and commands is defined formally by the grammar below:

$$\begin{array}{rcll} E & ::= & c & (c \in \mathbb{V}) \\ & | & \mathsf{x} & (\mathsf{x} \in \mathbb{X}) \\ & | & f(E, \ldots) & (f \in \mathbb{O}) \\ C & ::= & \textbf{skip} & \\ & | & \mathsf{x} := E & (\mathsf{x} \in \mathbb{X}) \\ & | & C; C & \\ & | & \textbf{if}(E)\{C\}\textbf{else}\{C\} & \\ & | & \textbf{while}(E)\{C\} & \end{array}$$

Operational semantics. A memory state is a function m from variables to values. We write $\mathbb{M} = [\mathbb{X} \to \mathbb{V}]$ for the set of memory states. The semantics $[\![E]\!]$ of expression E maps a memory state into a value, and can be defined trivially by induction over the syntax of expressions. Since all operations are assumed deterministic, $[\![E]\!]$ has signature $\mathbb{M} \to \mathbb{V}$.

An *operational semantics* for commands can be defined as a transition relation that describes all the basic execution steps that a command defines. A step is defined by a pair of states, each of which is made of a control state and a memory state. For simplicity, we let a control state be a command. Therefore, the operational semantics is defined by a transition relation $\to$, such that $(C, m) \to (C', m')$ means that one possible execution step of command C starting from memory state m produces the memory state m' with the command C' remaining to be executed. The full definition of $\to$ is shown in Figure 1(a).

The transition relation $\to$ allows one to derive (finite or infinite) program execution traces or reachable states, by straightforward fixpoint computation. As usual, we let $\to^\star$ be the reflexive and transitive closure of $\to$.

Collecting semantics. A *collecting semantics* for commands can be defined by induction over the syntax, as a function mapping a set of states fed as input to a command into a set of states that can be observed as output. To formalise it, we introduce a couple of notations. We write $m[\mathsf{x} \mapsto c]$ for the update of variable x with value c in memory state m. Given a set of memory states M, we write $[M?E]$ (resp., $[M?\neg E]$) for $\{m \in M \mid [\![E]\!](m) \neq 0\}$ (resp., $\{m \in M \mid [\![E]\!](m) = 0\}$).

Moreover, the semantics of loops requires the use of fixpoints. We recall that, for any continuous function F from a complete partial order E to itself, there exists a unique element $e \in E$ such that $F(e) = e$ (i.e., e is a *fixpoint* of F) and any other element $e' \in E$ such that $F(e') = e'$ is greater than e. This element is called the *least fixpoint* of F and is denoted by $\mathbf{lfp}F$. Moreover, it can be computed as the least upper-bound of the iterates of F.

Based on the above definitions, the semantics $[\![C]\!]$ of any command C can be defined by induction over the syntax as shown in Figure 1(b). To explain more precisely the case of loops, we consider the loop commmand $\textbf{while}(E)\{C\}$ and let $G(\mathbb{M}) = [\![C]\!]([\mathbb{M}?E])$. Then, for all integer n, we have:

$$F^n(\emptyset) = \bigcup_{0 \le k < n} G^k(\mathbb{M}).$$

For all command C, the collecting semantics $[\![C]\!]$ satisfies the following property:

$$\forall m, m' \in \mathbb{M},\ (C, m) \to^\star (\textbf{skip}, m') \iff m' \in [\![C]\!](\{m\}).$$

More generally, these two semantics (and others) can be connected with abstraction relations [8, 9].

3 Target Properties and Common Abstractions

In the following, we discuss techniques for the static analysis of programs so as to compute several families of properties. More precisely, we consider the two following families:

- A *state property* is a semantic property that boils down to a set of states S and that expresses that all the states reached after running a given program should belong to that set S. More precisely, a command C satisfies S if and only if $[\![C]\!](\mathbb{M}) \subseteq S$ or, equivalently:
$$\forall m, m' \in \mathbb{M},\ (C, m) \to (\textbf{skip}, m') \implies m' \in S.$$
- A *relational property* is a semantic property that consists of a set $\mathcal{R}$ of valid input state–output state relations. More precisely, a command C satisfies $\mathcal{R}$ if and

$$\frac{}{(\textbf{skip}; C, m) \to (C, m)} \qquad \frac{}{(\mathtt{x} := E, m) \to (\textbf{skip}, m[x \mapsto [\![E]\!]])} \qquad \frac{(C_0, m) \to (C_0', m')}{(C_0; C_1, m) \to (C_0'; C_1, m')}$$

$$\frac{[\![E]\!](m) \neq 0}{(\textbf{if}(E)\{C_0\}\textbf{else}\{C_1\}, m) \to (C_0, m)} \qquad \frac{[\![E]\!](m) = 0}{(\textbf{if}(E)\{C_0\}\textbf{else}\{C_1\}, m) \to (C_1, m)}$$

$$\frac{[\![E]\!](m) \neq 0}{(\textbf{while}(E)\{C\}, m) \to (C; \textbf{while}(E)\{C\}, m)} \qquad \frac{[\![E]\!](m) = 0}{(\textbf{while}(E)\{C\}, m) \to (\textbf{skip}, m)}$$

(a) Operational semantics transition relation

$$\begin{aligned}
[\![C]\!] &: \mathcal{P}(\mathbb{M}) \longrightarrow \mathcal{P}(\mathbb{M}) \\
[\![\textbf{skip}]\!](M) &:= M \\
[\![\mathtt{x} := E]\!](M) &:= \{m[\mathtt{x} \mapsto [\![E]\!](m)] \mid m \in M\} \\
[\![C_0; C_1]\!](M) &:= ([\![C_1]\!] \circ [\![C_0]\!])(M) \\
[\![\textbf{if}(E)\{C_0\}\textbf{else}\{C_1\}]\!](M) &:= [\![C_0]\!]([M?E]) \cup [\![C_1]\!]([M?\neg E]) \\
[\![\textbf{while}(E)\{C\}]\!](M) &:= [M_{lfp}?\neg E] \qquad \text{where} \quad M_{lfp} = \textbf{lfp}\ F_M \\
&\qquad\qquad\quad \text{with} \quad F_M(M') = M \cup [\![C]\!]([M'?E])
\end{aligned}$$

(b) Collecting semantics

Figure 1. Syntax and semantics (operational semantics and collecting semantics)

only if:

$$[\![C]\!] \in \mathcal{R}.$$

As an example, state properties include constraints over output states, that describe that whenever a command returns, output states should satisfy a given set of scalar constraints. Note that non-termination (i.e., the property that a command should never terminate) is of this form (the set of acceptable final states—which is empty—can be described by any unsatisfiable set of constraints). Another case of state property is the verification of absence of runtime errors as in, e.g., Astrée [5] (the formal definition of the absence of runtime errors as a state property slightly differs from the above definition though; indeed, while the above definition focuses on the final states, absence of errors characterises all states—whether intermediate or final).

Among relational properties we can cite dependence properties or security properties such as non-interference [23]. When scalar values are assumed real numbers, we can also cite smoothness properties such as continuity, continuous differentiability or Lipschitz continuity [17].

Static analysis attempts to infer such properties by the means of *abstractions* [9] that enable the computation of an over-approximation of the semantics. The next sections utilise both state abstractions and relational abstractions.

Definition 1 (State abstraction)**.** *A* state abstraction *consists of an abstract domain $\mathbb{D}_s$ with a concretisation function $\gamma_s : \mathbb{D}_s \to \mathcal{P}(\mathbb{M})$. In this set up a sound* abstract state semantics *maps each command C to a function $[\![C]\!]^\sharp : \mathbb{D}_s \to \mathbb{D}_s$ such that, for all $M^\sharp$, we have*

$$[\![C]\!] \circ \gamma_s(M^\sharp) \subseteq \gamma_s \circ [\![C]\!]^\sharp(M^\sharp).$$

Numerical abstract domains provide many examples of state abstractions. For instance, the elements of the abstract domain of intervals [9] are functions which map each variable into an over-approximation of its range (therefore, an abstract state $M^\sharp$ defines at most two constraints per variable). The elements of the abstract domain of convex polyhedra [12] consist of finite conjunctions of linear inequalities. In both cases, an abstract element $M^\sharp$ concretises into the set of memory states that satisfy all the constraints within $M^\sharp$.

Definition 2 (Relational abstraction)**.** *A* relational abstraction *consists of an abstract domain $\mathbb{D}_r$ with a concretisation function $\gamma_r : \mathbb{D}_r \to \mathcal{P}(\mathcal{P}(\mathbb{M} \times \mathbb{M}))$. In this set up, a sound* relational abstract semantics *maps each command C to an element $[\![C]\!]_r^\sharp$ of $\mathbb{D}_r$ and such that:*

$$\{(m, m') \mid m \in \mathbb{M} \wedge m' \in [\![C]\!](\{m\})\} \subseteq \gamma_r([\![C]\!]_r^\sharp).$$

Abstract domains such as convex polyhedra can be extended to capture relations between inputs and outputs, using the classical variable duplication technique, where each variable is duplicated into an instance holding the initial value and an instance holding the final value [20]. In this setup, by convention, the primed symbol $\mathtt{x}'$ denotes the new value of variable $\mathtt{x}$ whereas a non-primed symbol denotes the old value of the corresponding variable. For example, the constraint $\mathtt{x}' = \mathtt{x} + 1$ describes the effect of command $\mathtt{x} := \mathtt{x} + 1$.

Another typical example of relational abstraction is the dependency abstraction which collects, for each variable x, the set of variables the initial value of which may have an influence on the final value of x.

In the following, we study several techniques to compute state and relational abstract semantics under the assumption that an abstract domain is provided. We also point to references of analyses of each form.

4 State Abstraction and Iteration over Control Flow Graphs

We first consider the case of state properties and state abstractions. A standard approach to static analysis for state properties computes the control flow graph (CFG) of programs and performs abstract iteration over the CFG [7]. This approach attaches one abstract state to each CFG node, so that the abstract interpreter boils down to a fixpoint solver over a set of equations defined by the CFG edges.

In the setup of Section 3, such an abstract interpreter relies on an over-approximation of the transition relation $\rightarrow$ of the program, whereas each node of the CFG corresponds to a command. For instance, the execution step corresponding to an assignment $\mathtt{x} := E$ boils down to a transition over an edge from command $\mathtt{x} := E; C$ to the command C.

This technique naturally extends to backwards static analysis, where one is interested in an over-approximation of the set of initial states from which executions may reach a given set of final states.

While this technique is easy to port to lower level languages (such as assembly), it is less appropriate for higher level programming languages. It also requires care to mitigate the cost of storing abstract states at each control point or possibly redundant computations of analyses of specific program steps.

5 State Abstraction and Iteration over the Abstract Syntax Tree

Another technique for inferring state properties proceeds by abstraction of the collecting semantics of Section 2. In this approach, the abstract semantics is defined by induction over the syntax of commands and associates to each command a function that maps abstract pre-conditions to abstract post-conditions. To formally define this abstract semantics, we fix a few notations. Given an abstract element $M^\sharp$, a variable x, and an expression E, we write $M^\sharp[\mathtt{x} \mapsto E]^\sharp$ to denote the abstract post-condition for assignment $\mathtt{x} := E$ in pre-condition $M^\sharp$. Likewise, we let $[M^\sharp ? E]^\sharp$ and $[M^\sharp ? \neg E]^\sharp$ be the abstract post-conditions for condition tests $E \neq 0$ and $E = 0$. These operations are assumed to satisfy the following soundness conditions:

$$\forall M^\sharp \in \mathbb{D},\ \forall m \in \gamma_s(M^\sharp),\quad \left\{ \begin{array}{ll} & m[\mathtt{x} \mapsto [\![E]\!](m)] \in \gamma_s(M^\sharp[\mathtt{x} \mapsto E]^\sharp \\ \wedge & [\![E]\!](m) \neq 0 \Longrightarrow m \in \gamma_s([M^\sharp ? E]^\sharp) \\ \wedge & [\![E]\!](m) = 0 \Longrightarrow m \in \gamma_s([M^\sharp ? \neg E]^\sharp) \end{array} \right.$$

Finally, we let $\sqcup$ be a binary operation over $\mathbb{D}$ that returns an over-approximation of concrete unions and we let $\triangledown$ be a widening operator in $\mathbb{D}$. We recall $\triangledown$ is a widening operator if and only if it satisfies the following two conditions:

1. $\triangledown$ computes an over approximation of concrete least upper bound in the sense that,
$$\forall M_0^\sharp, M_1^\sharp,\ \gamma_s(M_0^\sharp) \cup \gamma_s(M_1^\sharp) \subseteq \gamma_s(M_0^\sharp \triangledown M_1^\sharp);$$
2. $\triangledown$ ensures the termination of abstract iterates in the sense that, for any sequence $(M_n^\sharp)_{n\in\mathbb{N}}$ of abstract states, the second sequence defined as follows is ultimately stationary:
$$(N_n^\sharp)_{n\in\mathbb{N}} \text{ where } \left\{ \begin{array}{rcl} N_0^\sharp & = & M_0^\sharp \\ \forall n,\ N_{n+1}^\sharp & = & N_n^\sharp \triangledown M_{n+1}^\sharp. \end{array} \right.$$

The definition of each of these operations is dependent on the choice of $\mathbb{D}$ and not discussed here. Then, a sound abstract semantics $[\![.]\!]_s^\sharp$ can be defined as shown in Figure 2.

This static analysis definition offers several advantages. It is natural and based on the program abstract syntax tree thus, it provides a perfect basis for a direct implementation in a functional programming language. Moreover, storage requirements are minimal. Indeed, the analysis of a **while** command only requires to store at any given time at most two consecutive elements of the sequence of abstract iterates $(M_n^\sharp)_{n\in\mathbb{N}}$. The analysis of an **if** command requires to store the final results of the analyses of both branches till their abstract union is computed. In the case of any other command, intermediate abstract states may be discarded as soon as the analysis moves to the next command. Thus, the number of abstract states that need to be stored at any given time is a linear factor of the maximal nesting depth of conditions and loops. Another advantage is that the results that are produced are intuitive to interpret since analysis output describes program post-conditions.

This technique has been used in various static analysis tools such as Astrée [5] and MemCAD [19] for C programs, and Pyppai [18] for probabilistic programs in Pyro. We remark that a similar analysis structure would also apply to backward analysis.

Support of non-locally branching commands. One apparent limitation of the above definition of $[\![.]\!]_s^\sharp$ is that non-structured branching statements such as gotos do not trivially fit in the structured iteration scheme. However, this limitation can be easily lifted as we show in the following paragraph. Let us consider the addition of a **break** statement that branches to the exit of the closest enclosing loop.

$$[\![C]\!]^\sharp_s : \mathbb{D} \longrightarrow \mathbb{D}$$

$$\begin{array}{rcl}
[\![\textbf{skip}]\!]^\sharp_s(M^\sharp) & := & M^\sharp \\
[\![\mathsf{x} := E]\!]^\sharp_s(M^\sharp) & := & M^\sharp[\mathsf{x} \mapsto E]^\sharp \\
[\![C_0; C_1]\!]^\sharp_s(M^\sharp) & := & ([\![C_1]\!]^\sharp_s \circ [\![C_0]\!]^\sharp_s)(M^\sharp) \\
[\![\textbf{if}(E)\{C_0\}\textbf{else}\{C_1\}]\!]^\sharp_s(M^\sharp) & := & [\![C_0]\!]^\sharp_s([M^\sharp ? E]^\sharp) \sqcup [\![C_1]\!]^\sharp_s([M^\sharp ? \neg E]^\sharp)) \\
[\![\textbf{while}(E)\{C\}]\!]^\sharp_s(M^\sharp) & := & [M^\sharp_\infty ? \neg E]^\sharp \qquad \text{where} \qquad M^\sharp_\infty = \lim(M^\sharp_n) \\
& & \text{with} \quad \left\{ \begin{array}{rcl} M^\sharp_0 & = & M^\sharp \\ M^\sharp_{n+1} & = & M^\sharp_n \triangledown (M^\sharp_0 \sqcup [\![C]\!]^\sharp_s([M^\sharp_n ? E]^\sharp)) \end{array} \right.
\end{array}$$

Figure 2. Syntax directed abstract iteration

Its operational semantics is trivial to express with a transition relation $\rightarrow$, but can also be captured in an extension of $[\![.]\!]$. Indeed, while $[\![.]\!]$ was defined in Section 2 as a function from $\mathcal{P}(\mathbb{M})$ to itself, we let it be defined as a function over $\mathcal{P}(\mathbb{M}) \times (\mathcal{P}(\mathbb{M}))^*$ in the remaining part of this section, where A^* denotes sequences of elements of set A, and where the second component denotes a stack of sets of states, that respectively branch to each of the currently enclosing loops. Intuitively $(M, (M_0, M_1, M_2, M_3))$ describes a set of state-continuation pairs, where the states in M are associated to the current continuation (i.e., are going to run through the next command), where the states in M_0 are meant to branch to the exit of the closest enclosing loop, where the states in M_1 are meant to branch to the exit of the next enclosing loop, and where the conditions for M_2, M_3 are expressed similarly, with respective depths 2 and 3. In particular, the semantics of **break** becomes:

$$[\![\textbf{break}]\!](M, M_0 \cdot \ldots \cdot M_n) = (\emptyset, (M \cup M_0) \cdot M_1 \cdot \ldots \cdot M_n)$$

Commands such as assignments and conditions only operate over the first component.

This new definition of $[\![.]\!]$ directly leads to an extension of $[\![.]\!]^\sharp_s$ that relies on the same principle, and which operates on $\mathbb{D} \times (\mathbb{D})^*$ instead of $\mathbb{D}$. This extension of $[\![.]\!]^\sharp_s$ can be defined in a similar syntax directed style as in Figure 2. The analysis of the **break** command proceeds as follows:

$$[\![\textbf{break}]\!]^\sharp_s(M^\sharp, M^\sharp_0 \cdot \ldots \cdot M^\sharp_n) = (\bot, (M^\sharp \sqcup M^\sharp_0) \cdot M^\sharp_1 \cdot \ldots \cdot M^\sharp_n)$$

where $\bot$ is the element of $\mathbb{D}$ such that $\gamma(\bot) = \emptyset$. This discussion shows that the abstract syntax tree based analysis $[\![.]\!]^\sharp_s$ also extends to statements with non-structured branching. As an example, Astrée [5] and MemCAD [19] use this technique to support control flow branching. Likewise, static analyses for higher-order languages [13, 24] also use abstract continuations.

6 Bottom-up Relational Abstraction

We now turn to the over-approximation of relational properties using relational abstraction. A common method to compute such an abstraction relies on inference rules that operate by induction over the syntax in a bottom-up style [6, 15]. To illustrate this approach, we consider the dependency abstraction that maps variable x to a super-set of the variables that may affect the final value of x after executing a command. In this set-up, $\mathbb{D}_r = \mathbb{X} \rightarrow \mathcal{P}(\mathbb{X})$ and γ maps $R^\sharp \in \mathbb{D}_r$ into the set of all relations R such that the following condition holds:

$$\begin{array}{l}
\forall \mathsf{x} \in \mathbb{X}, \forall (m_0, m'_0), (m_1, m'_1) \in R, \\
\qquad (\forall \mathsf{y} \in R^\sharp(\mathsf{x}),\ m_0(\mathsf{y}) = m_1(\mathsf{y})) \Longrightarrow m'_0(\mathsf{x}) = m'_1(\mathsf{x})
\end{array}$$

Then, we can derive a sound $[\![C]\!]^\sharp_\uparrow$ by induction over the syntax of C, such as:

$$\begin{array}{rcll}
[\![\textbf{skip}]\!]^\sharp_\uparrow & := & \lambda \mathsf{x} \cdot \{\mathsf{x}\} & \\
[\![\mathsf{x} := E]\!]^\sharp_\uparrow & := & \lambda \mathsf{y} \cdot \left\{ \begin{array}{ll} \textbf{fv}(E) & \text{if } \mathsf{x} = \mathsf{y} \\ \{\mathsf{y}\} & \text{otherwise} \end{array} \right. & \\
[\![C_0; C_1]\!]^\sharp_\uparrow & ::= & \lambda \mathsf{x} \cdot \bigcup \{ [\![C_0]\!]^\sharp_\uparrow(\mathsf{y}) \mid \mathsf{y} \in [\![C_1]\!]^\sharp_\uparrow(\mathsf{x}) \} &
\end{array}$$

The definition of this abstract semantics is compositional and by induction over the syntax of programs. It also applies to the computation of an abstraction for other smoothness properties [17] (continuity, Lipschitz continuity, differentiability) that was mentioned in Section 3.

7 Relational Abstraction and Iteration over the Abstract Syntax Tree

One apparent limitation of the bottom-up iteration technique shown in Section 6 is the inability to use state information during the computation of relational abstractions. As an example, consider the command $\mathsf{c} = \mathsf{x} * \mathsf{x} + 1; \textbf{if}(\mathsf{c})\{\mathsf{y} = 0\}\textbf{else}\{\mathsf{y} = \mathsf{z}\}$. The first assignment will always store value 0 into y thus the dependency in the **else** branch is spurious. However, to disprove this dependency, one needs to establish state property $\mathsf{c} \geq 1$ after the first statement. Therefore, a static analysis that combines state and relational abstractions (unlike the analysis defined in Section 6) is required. To remedy this issue, we propose in this section an iteration technique that is inspired by that of Section 5 but works for relational abstraction and show that it enables such a combined analysis.

AST-based relational static analysis. The core idea to achieve this is to define a version of $[\![.]\!]^\sharp$ that operates on a relational abstraction rather than a state abstraction. Intuitively, the abstract semantics $[\![C]\!]^\sharp_r$ inputs an *abstract pre-continuation*, which abstracts a computation performed "before running C" and outputs an *abstract post-continuation* that accounts for the effect of running C on top of this abstract pre-continuation. To define the soundness condition that $[\![C]\!]^\sharp_r$ should meet, we introduce the following notation for the application $[\![C]\!] \circ R$ of the semantics of a command C to a pre-continuation $R \in \mathcal{P}(\mathbb{M} \times \mathbb{M})$ (and returning another element of $\mathcal{P}(\mathbb{M} \times \mathbb{M})$):

$$[\![C]\!] \circ R ::= \{(m, m') \mid \exists m'' \in \mathbb{M},\ (m, m'') \in R \wedge m' \in [\![C]\!](\{m''\})\}$$

The corresponding soundness relation boils down to the following definition:

Definition 3 (Forward relational abstraction)**.** *A sound* forward abstract relational semantics *maps each command C into a function $[\![C]\!]^\sharp_r : \mathbb{D}_r \to \mathbb{D}_r$ such that, for all abstract relation $R^\sharp$ and for all concrete relation $R \in \gamma(R^\sharp)$, we have*

$$[\![C]\!](R) \in \gamma_r([\![C]\!]^\sharp_r(R^\sharp)).$$

Interestingly, the definition of a sound forward relational abstraction $[\![C]\!]^\sharp_r$ follows the same inductive, syntax directed structure as that of $[\![C]\!]^\sharp_s$ in Section 5. This approach can be applied to dependence or more generally to smoothness properties [17] (continuity, Lipschitz-continuity, differentiability...). It also enables the computation of abstract relations that tie data-structures before and after function calls [16].

From state abstraction to relational abstraction. Moreover, forward state abstraction may also be encoded in the same way. Indeed, let us assume an abstraction of states defined by $\mathbb{D}_s$ and γ_s. Then, we can let an abstraction for relations be defined as follows:

$$\begin{array}{rcl} \mathbb{D}_r & = & \mathbb{D}_s \\ \gamma_r(M^\sharp) & = & \mathcal{P}(\{(m, m') \in \mathbb{M}^2 \mid m' \in \gamma_s(M^\sharp)\}). \end{array}$$

Therefore, both state and relational abstractions can be computed using a same abstract interpreter $[\![C]\!]^\sharp_r$, the structure of which follows that of $[\![C]\!]^\sharp_s$ (Figure 2). Moreover, this observation implies that the well-known abstract domain *reduced product* technique [10] naturally applies in this case. To illustrate its effectiveness, we consider the case of the command $\mathtt{c = x * x + 1; if(c)\{y = 0\}else\{y = z\}}$ that was mentioned earlier in this section and assume a product relational abstract domain that combines an interval abstraction and a dependence abstraction. As the first component shows that c is strictly positive after the first assignment, the analysis of the condition needs only to consider the first branch. This branch stores a constant in y, thus this variable does not depend on z. In a security context, this entails that there is no information flow from z to y although a less precise analysis would miss this fact. More generally, the Pyppai extension that infers smoothness properties in [17] uses a similar reduced product of relational and state abstraction, so as to infer state properties that are required to refine smoothness information. For instance, it may use range information to show that an operation is always applied within its continuous domain.

8 Conclusion

In this short paper, we have discussed some of the advantages of the syntax directed technique for designing abstract interpreters. In particular, we have shown that it can be adapted for programming languages with non-local control flow branchings and we have presented an application of the syntax directed technique for the computation of relational abstraction of commands. The latter result implies that AST-based iterators apply not only to state properties but also to relational properties such as some classes of security properties or smoothness properties.

Acknowledgments

This paper is dedicated to Olivier Danvy for his deeply inspiring contributions in both research and teaching in programming languages, semantics, partial evaluation and so many topics in computer science. Many of the notions discussed in this paper are directly or indirectly influenced by Olivier's works.

We also thank the anonymous reviewers for their careful reading, and their numerous suggestions and helpful comments on an earlier version of this paper.

References

[1] Mads Sig Ager, Dariusz Biernacki, Olivier Danvy, and Jan Midtgaard. 2003. A functional correspondence between evaluators and abstract machines. In *PPDP*. ACM, 8–19. doi:10.1145/888251.888254

[2] Mads Sig Ager, Olivier Danvy, and Jan Midtgaard. 2005. A functional correspondence between monadic evaluators and abstract machines for languages with computational effects. *TCS* 342, 1 (2005), 149–172. doi:10.1016/J.TCS.2005.06.008

[3] Martin Helmut Alt, Christian Ferdinand, Florian Martin, and Reinhard Wilhelm. 1996. Cache Behavior Prediction by Abstract Interpretation. In *SAS (Lecture Notes in Computer Science, Vol. 1145)*. Springer, 52–66. doi:10.1007/3-540-61739-6_33

[4] Martin Helmut Alt and Florian Martin. 1995. Generation of Efficient Interprocedural Analyzers with PAG. In *SAS (Lecture Notes in Computer Science, Vol. 983)*. Springer, 33–50. doi:10.1007/3-540-60360-3_31

[5] B. Blanchet, P. Cousot, R. Cousot, J. Feret, L. Mauborgne, A. Miné, D. Monniaux, and X. Rival. 2003. A static analyzer for large safety-critical software. In *PLDI*. 196–207. doi:10.1007/978-3-540-39910-0_11

[6] Ghila Castelnuovo, Mayur Naik, Noam Rinetzky, Mooly Sagiv, and Hongseok Yang. 2015. Modularity in Lattices: A Case Study on the Correspondence Between Top-Down and Bottom-Up Analysis. In *SAS (Lecture Notes in Computer Science, Vol. 9291)*. Springer, 252–274. doi:10.1007/978-3-662-48288-9_15

[7] Patrick Cousot. 1981. Semantic Foundations of Program Analysis. In *Program Flow Analysis: Theory and Applications*, S.S. Muchnick and N.D. Jones (Eds.). Chapter 10, 303–342.

[8] Patrick Cousot. 1997. Constructive design of a hierarchy of semantics of a transition system by abstract interpretation. In *MFPS (ENTCS, Vol. 6)*. 77–102. doi:10.1016/S1571-0661(05)80168-9

[9] Patrick Cousot and Radhia Cousot. 1977. Abstract Interpretation: A Unified Lattice Model for Static Analysis of Programs by Construction or Approximation of Fixpoints. In *POPL*. ACM, 238–252. doi:10.1145/512950.512973

[10] Patrick Cousot and Radhia Cousot. 1979. Systematic Design of Program Analysis Frameworks. In *POPL*. ACM. doi:10.1145/567752.567778

[11] Patrick Cousot, Radhia Cousot, Jérôme Feret, Laurent Mauborgne, Antoine Miné, and Xavier Rival. 2009. Why does Astrée scale up? *FMSD* 35, 3 (2009), 229–264. doi:10.1007/S10703-009-0089-6

[12] Patrick Cousot and Nicolas Halbwachs. 1978. Automatic Discovery of Linear Restraints Among Variables of a Program. In *POPL*. 84–96. doi:10.1145/512760.512770

[13] Olivier Danvy and Jan Midtgaard. 2011. Abstracting abstract machines: technical perspective. *Commununications of the ACM* 54, 9 (2011), 100. doi:10.1145/1995376.1995399

[14] David Darais, Nicholas Labich, Phuc C. Nguyen, and David Van Horn. 2017. Abstracting definitional interpreters (functional pearl). *Proceedings of the ACM (PACM)* 1, ICFP (2017), 12:1–12:25. doi:10.1145/3110256

[15] Roger Hindley. 1969. The principal type-scheme of an object in combinatory logic. *Transactions of the American Mathematical Society, 146:* 146 (1969), 29–60. doi:10.2307/1995158

[16] Hugo Illous, Matthieu Lemerre, and Xavier Rival. 2021. A relational shape abstract domain. *FMSD* 57, 3 (2021), 343–400. doi:10.1007/S10703-021-00366-4

[17] Wonyeol Lee, Xavier Rival, and Hongseok Yang. 2023. Smoothness Analysis for Probabilistic Programs with Application to Optimised Variational Inference. *Proceedings of the ACM (PACM)* 7, POPL (2023), 335–366. doi:10.1145/3571205

[18] Wonyeol Lee, Hangyeol Yu, Xavier Rival, and Hongseok Yang. 2020. Towards verified stochastic variational inference for probabilistic programs. *Proceedings of the ACM (PACM)* 4, POPL (2020), 16:1–16:33. doi:10.1145/3371084

[19] Huisong Li, Francois Berenger, Bor-Yuh Evan Chang, and Xavier Rival. 2017. Semantic-directed clumping of disjunctive abstract states. In *POPL*. ACM, 32–45. doi:10.1145/3009837.3009881

[20] Corneliu Popeea and Wei-Ngan Chin. 2006. Inferring Disjunctive Postconditions. In *ASIAN*. Springer, 331–345. doi:10.1007/978-3-540-77505-8_26

[21] Thomas W. Reps, Susan Horwitz, and Shmuel Sagiv. 1995. Precise Interprocedural Dataflow Analysis via Graph Reachability. In *POPL*. 49–61. doi:10.1145/199448.199462

[22] Xavier Rival and Kwangkeun Yi. 2020. *Introduction to Static Analysis: An Abstract Interpretation Perspective*. MIT Press.

[23] Andrei Sabelfeld and Andrew C. Myers. 2003. Language-based information-flow security. *IEEE J. Sel. Areas Commun.* 21, 1 (2003), 5–19. doi:10.1109/JSAC.2002.806121

[24] Guannan Wei, James M. Decker, and Tiark Rompf. 2018. Refunctionalization of abstract abstract machines: bridging the gap between abstract abstract machines and abstract definitional interpreters (functional pearl). *Proceedings of the ACM (PACM)* 2 (2018), 105:1–105:28. doi:10.1145/3236800

Received 2025-06-03; accepted 2025-07-31