A Modular Soundness Theory for the Blackboard Analysis Architecture

Sven Keidel[1], Dominik Helm[1,2], Tobias Roth[1,2], and Mira Mezini[1,2,3]

[1] Technische Universität Darmstadt, Darmstadt, Germany
keidel,helm,roth,mezini@cs.tu-darmstadt.de
[2] National Research Center for Applied Cybersecurity (ATHENE), Darmstadt, Germany
[3] Hessian Center for Artificial Intelligence (hessian.AI), Darmstadt, Germany

Abstract. Sound static analyses are an important ingredient for compiler optimizations and program verification tools. However, mathematically proving that a static analysis is sound is a difficult task due to two problems. First, soundness proofs relate two complicated program semantics (the static and the dynamic semantics) which are hard to reason about. Second, the more the static and dynamic semantics differ, the more work a soundness proof needs to do to bridge the impedance mismatch. These problems increase the effort and complexity of soundness proofs. Existing soundness theories address these problems by deriving both the dynamic and static semantics from the same artifact, often called *generic interpreter*. A generic interpreter provides a common structure along which a soundness proof can be composed, which avoids having to reason about the analysis as a whole. However, a generic interpreter restricts which analyses can be derived, as all derived analyses must roughly follow the program execution order.
To lift this restriction, we develop a soundness theory for the blackboard analysis architecture, which is capable of describing backward, demand-driven, and summary-based analyses. The architecture describes static analyses with small independent modules, which communicate via a central store. Soundness of a compound analysis follows from soundness of all of its modules. Furthermore, modules can be proven sound independently, even though modules depend on each other. We evaluate our theory by proving soundness of four analyses: a pointer and call-graph analysis, a reflection analysis, an immutability analysis, and a demand-driven reaching definitions analysis.

1 Introduction

Developing static analyses is a laborious and complicated task due to the complexity of modern programming languages. A significant part of the complication pertains to ensuring that static analyses are *sound*, i.e., over-approximate the runtime behavior of analyzed programs. Unfortunately, even well-established static analyses are shown to be unsound, e.g., since 2010, more than 80 soundness bugs have been found in different analyses used in the LLVM compiler [46].

Testing helps finding soundness bugs but cannot prove their absence, leaving the trustworthiness of these analyses in question.

Mathematical soundness proofs ensure the absence of soundness bugs. However, such proofs are difficult for two reasons: First, soundness proofs relate two program semantics: the static semantics and the dynamic semantics [12]—each in its own can individually be complex. Especially modern programming language features such as reflection [30], concurrency [29], or native code [1] are notoriously difficult to analyze and hard to reason about. Second, the style of static and dynamic semantics can differ significantly, e.g., the static semantics of Doop [7], which is described in Datalog, differs significantly from dynamic semantics described with small-step rules [6]. This impedance mismatch makes soundness proofs *monolithic*, i.e., it is difficult to determine which parts of the static semantics relate to which parts of the dynamic semantics, requiring the soundness proofs to reason about both semantics as a whole. These problems complicate soundness proofs such that only leading experts with multiple years of experience can conduct them [13, 26].

To deal with the complexity of soundness proofs, existing works modularize static and dynamic semantics [5, 14, 28]. This modularization allows to compose a soundness proof for the entire analysis from soundness lemmas of small parts of the analysis. This allows reasoning about small parts of the analysis one at a time. These existing works require that both the static and dynamic semantics are derived from the same artifact, often called a *generic interpreter*. A generic interpreter describes the operational semantics of a language, without referring to details of dynamic or static semantics, and provides a common structure along which a soundness proof can be composed. However, generic interpreters restrict what types of analyses can be derived. In particular, generic interpreters derive analyses that follow the program execution order, specifically, forward whole-program abstract interpreters. But it is unclear how other types of analyses can be derived that do not follow the program execution order, such as backward, demand-driven/lazy, or summary-based analyses.

The work presented in this paper lifts this restriction by developing a soundness theory for the *blackboard analysis architecture*. The architecture is the foundation of the *OPAL framework* [21], which has been used to develop different kinds of analyses, including backward analyses [17], on-demand/lazy analyses [19,41], and summary-based analyses [21]. In the architecture, complex static analyses are modularly composed from smaller, simpler *static modules* that handle individual language features, e.g., reflection, or program properties, e.g., immutability. These modules are decoupled—they are not allowed to call each other directly; instead, they communicate with each other by exchanging information via a central data store called *blackboard* [39] orchestrated by a fixpoint solver.

To develop a soundness theory for the blackboard analysis architecture, we define a dynamic semantics, which follows the same style as the static semantics and thus avoids the impedance mismatch problem. Specifically, the dynamic semantics is composed of dynamic modules that communicate with each other via a store. Our soundness theory is *compositional*, which means that each static module can be proven sound individually and soundness for the compound analysis follows from a meta theorem. Furthermore, we extend the theory to make soundness proofs of existing static modules *reusable* across different analyses. In particular, we prove that the soundness proof of an static module remains valid, even if (a) the compound analysis processes source code elements unknown to the module and (b) the store contains other types of analysis information unknown to the module. Furthermore, our proofs are polymorphic in the lattices on which static modules operate, i.e., the lattices can be changed without affecting soundness. For instance, we can reuse a pointer-static module, which typically depends on an allocation-site lattice, in a reflection analysis to propagate string information by extending this lattice without invalidating the pointer-static modules' soundness proof.

We demonstrate the applicability of our theory by implementing four different analyses and their dynamic semantics in the blackboard analysis architecture: (1) a pointer and call-graph analysis, (2) an analysis for reflection, (3) an immutability analysis, and (4) a demand-driven reaching-definitions analysis. Our choice of analyses is inspired by existing state-of-the-art analyses for Java implemented in the OPAL framework [21, 41]. We implemented and tested each analysis and dynamic semantics in Scala to ensure they are executable. Furthermore, we used our theory to prove each analysis sound, where each analysis exercises a different aspect of our theory: (1) static modules can be proven sound independently despite mutually depending on each other, (2) soundness of modules remains valid even though the lattice changes, (3) soundness of a module remains valid even though different source code elements are analyzed, and (4) our theory applies to analyses which do not follow the program execution order.

In summary, we make the following contributions:

- We give the first formalization of the blackboard analysis architecture (Section 2).
- We develop a theory of compositional soundness proofs for the formal model of the blackboard analysis architecture. We prove that soundness of an analysis follows from independent soundness proofs for each of its modules (Section 3).
- We show how to make soundness proofs reusable by extending our theory (Section 4).
- We demonstrate the applicability of our theory on four different types of analyses (Section 5).

All proofs of theorems, lemmas, and case studies are provided in the paper's supplementary material.

2 Blackboard Analysis Architecture

In this section, we introduce and formalize the static and dynamic semantics of the blackboard analysis architecture used in the OPAL framework [21].

2.1 Static Semantics

Static analyses in the blackboard analysis architecture consist of multiple *static modules* exchanging information via a central data store called *blackboard* [39]. This avoids coupling between modules as they are not allowed to call each other directly: Modules store analysis results in the blackboard using keys. These keys allow other modules to retrieve results without needing to know their producer.

Definition 1 (Static Semantics). *We define basic notions and datatypes of the static semantics of the blackboard analysis architecture:*

1. *Entities* ($\widehat{e} \in \widehat{\mathsf{Entity}}$)[4] *are parts of programs an analysis can compute information for. For example, entities could be classes, methods, statements, fields, variables, or allocation sites of objects. Entities are ordered discretely:* $\widehat{e}_1 \sqsubseteq \widehat{e}_2$ *iff* $\widehat{e}_1 = \widehat{e}_2$.
2. *Kinds* ($\kappa \in \mathsf{Kind}$) *identify analysis information that can be computed for an entity. For example, a class entity could have kinds for its immutability or thread safety, a variable entity could have kinds for its definition site or approximations of its value. Kinds are also ordered discretely.*
3. *Properties* ($\widehat{p} \in \widehat{\mathsf{Property}}[\kappa]$ *where* $\widehat{\mathsf{Property}} : \mathsf{Kind} \rightarrow \mathsf{Lattice}$) *denote analysis information which is identified by a kind* κ*. For instance, a class entity could have an immutability property "*mutable*" or "*immutable*". Properties of a kind are partially ordered and form a lattice.*
4. *A central store* ($\widehat{\sigma} \in \widehat{\mathsf{Store}} \subseteq \widehat{\mathsf{Entity}} \times (\kappa : \mathsf{Kind}) \rightharpoonup \widehat{\mathsf{Property}}[\kappa]$)[5] *contains all properties for each entity and kind. We use the notation* $\widehat{\sigma}(\widehat{e}, \kappa)$ *for a store lookup of an entity* $\widehat{e}$ *and kind* κ*, which results in the bottom element* $\perp$ *in case the property is not present. Furthermore, we use the notation* $\widehat{\sigma} \sqcup [\widehat{e}, \kappa \mapsto \widehat{p}]$ *for writing a new property* $\widehat{p}$ *to the store. If a property for the entity* $\widehat{e}$ *and* κ *already exists in the store, then the old property is joined with the new property. Stores are ordered point-wise.*
5. *Static modules* ($\widehat{f} \in \widehat{\mathsf{Module}} = \widehat{\mathsf{Entity}} \times \widehat{\mathsf{Store}} \rightarrow \widehat{\mathsf{Store}}$) *are monotone functions that compute properties of a given entity. The store allows multiple static modules to communicate and exchange information without having to call each other directly. Each static module has access to the entire store and can contribute to one or more properties.*
6. *The fixpoint algorithm* ($\mathsf{fix} : \mathcal{P}(\widehat{\mathsf{Module}}) \times \widehat{\mathsf{Store}} \rightarrow \widehat{\mathsf{Store}}$) *computes a fixpoint of a compound analysis* $\widehat{F} \in \mathcal{P}(\widehat{\mathsf{Module}})$ *for an initial store* $\widehat{\sigma}_1$*. More specifically, the fixpoint* $\mathsf{fix}(\widehat{F}, \widehat{\sigma}_1)$ *is a store* $\widehat{\sigma}_n \sqsupseteq \widehat{\sigma}_1$ *such that static modules* $\widehat{f} \in \widehat{F}$ *do not add new information, i.e.,* $\widehat{f}(\widehat{e}, \widehat{\sigma}_n) = \widehat{\sigma}_n$ *for all* $\widehat{e} \in dom(\widehat{\sigma}_n)$*. The fixpoint is unique and guaranteed to exist when all properties are lattices of finite height [10].*

The types $\widehat{\mathsf{Entity}}$, Kind, *and* $\widehat{\mathsf{Property}}$ *are defined by analysis developers, whereas the other types and functions are fixed by this definition.* □

[4] We use a hat symbol $\widehat{\ }$ to disambiguate static definitions from dynamic definitions with the same name but without hat.

[5] The syntax $A \rightharpoonup B$ denotes a partial function from A to B. Furthermore, $dom(f)$ is the set of all inputs for which a partial function f is defined.

We illustrate Definition 1 at the example of a text-book reaching-definitions analysis [38] for an imperative language with labeled assignments and expressions:

$$\begin{aligned}
&\widehat{\mathsf{Entity}} = \mathsf{Stmt}\\
&\widehat{\mathsf{Property}}[\kappa_{\texttt{ControlFlowPred}}] = \mathcal{P}(\mathsf{Stmt})\\
&\widehat{\mathsf{Property}}[\kappa_{\texttt{ReachingDefs}}] = \mathsf{Var} \rightharpoonup \mathcal{P}(\mathsf{Assign})\\
&\widehat{\mathsf{Store}} = [\mathsf{Stmt} \times \kappa_{\texttt{ControlFlowPred}} \rightharpoonup \mathcal{P}(\mathsf{Stmt})]\\
&\qquad \cup [\mathsf{Stmt} \times \kappa_{\texttt{ReachingDefs}} \rightharpoonup (\mathsf{Var} \rightharpoonup \mathcal{P}(\mathsf{Assign}))]
\end{aligned}$$

```
$\widehat{\mathsf{reachingDefs}}$(stmt: $\widehat{\mathsf{Entity}}$, $\widehat{\sigma}$: $\widehat{\mathsf{Store}}$): $\widehat{\mathsf{Store}}$ =
  predecessors = $\widehat{\sigma}$(stmt, $\kappa_{\texttt{ControlFlowPred}}$)
  $\widehat{\texttt{in}}$ = $\bigsqcup_{\texttt{p}\in\texttt{predecessors}} \widehat{\sigma}$(p, $\kappa_{\texttt{ReachingDefs}}$)
  $\widehat{\texttt{out}}$ = stmt match
    case Assign(x,_,_) => $\widehat{\texttt{in}}[\texttt{x} \mapsto \{\texttt{stmt}\}]$
    case _ => $\widehat{\texttt{in}}$
  $\widehat{\sigma} \sqcup [\texttt{stmt}, \kappa_{\texttt{ReachingDefs}} \mapsto \widehat{\texttt{out}}]$
```

The static module $\widehat{\mathsf{reachingDefs}}$ is implemented with Scala-like pseudo code. Module $\widehat{\mathsf{reachingDefs}}$ computes for every statement of the program which variable definitions reach it. Therefore, entities are statements and the module's property is a mapping from variables to assignments that may have defined it. Module $\widehat{\mathsf{reachingDefs}}$ joins the reaching definitions of all control-flow predecessors and then updates them on variable assignments. Note that module $\widehat{\mathsf{reachingDefs}}$ neither computes the control-flow predecessors directly nor does it call another module which computes this information. Instead, it retrieves this information from the store $\widehat{\sigma}$. This decoupling avoids dependencies between static modules and enables compositional soundness proofs.

2.2 Dynamic Semantics

Static analyses in the blackboard analysis architecture are proven sound with respect to a dynamic semantics in the same style, which we define formally in this subsection:

Definition 2 (Dynamic Semantics). *We define the dynamic semantics used to prove soundness of analyses in the blackboard analysis architecture:*

1. *The dynamic semantics depends on concrete versions of* entities *($e \in \mathsf{Entity}$),* properties *($p \in \mathsf{Property}[\kappa]$ where $\mathsf{Property} : \mathsf{Kind} \to \mathsf{Set}$) and* stores *($\sigma \in \mathsf{Store} \subseteq \mathsf{Entity} \times (\kappa : \mathsf{Kind}) \to \mathsf{Property}[\kappa]$). The kinds are the same as for static modules.*
2. *Dynamic modules ($f \in \mathsf{Module} = \mathsf{Entity} \times \mathsf{Store} \rightharpoonup \mathsf{Store}$) are partial functions which may only be defined for a subset of entities. Furthermore, the partial function is undefined in case it tries to lookup an element from the store which is not present.*
3. *Static analyses are proven sound with respect to a dynamic reachability semantics. The reachability semantics ($\mathsf{reachable} : \mathcal{P}(\mathsf{Module}) \times \mathsf{Store} \to \mathcal{P}(\mathsf{Store})$)*

returns the set of all reachable stores by iteratively applying a set of dynamic modules. More specifically, the set $\mathsf{reachable}(F, \sigma_1)$ *contains store* σ_1 *and for all* $f \in F$*, reachable stores* σ*, and for entities* $e \in dom(\sigma)$*, the set contains* $f(e, \sigma)$*, if it is defined.* □

We illustrate these definitions again at the example of the reaching-definitions analysis which we introduced in the previous subsection:

$$
\begin{aligned}
&\mathsf{Entity} = \mathsf{Stmt} \mid \mathsf{Unit}\\
&\mathsf{Property}[\kappa_{\texttt{ControlFlowPred}}] = \mathsf{Stmt}\\
&\mathsf{Property}[\kappa_{\texttt{ReachingDefs}}] = \mathsf{Var} \rightharpoonup \mathsf{Assign}\\
&\mathsf{Property}[\kappa_{\texttt{State}}] = \texttt{ProgramState}\\
&\mathsf{Store} = [\mathsf{Stmt} \times \kappa_{\texttt{ControlFlowPred}} \rightharpoonup \mathsf{Stmt}] \cup [\mathsf{Stmt} \times \kappa_{\texttt{ReachingDefs}} \rightharpoonup (\mathsf{Var} \rightharpoonup \mathsf{Assign})]\\
&\qquad \cup [\mathsf{Unit} \times \kappa_{\texttt{State}} \rightharpoonup \mathsf{ProgramState}]
\end{aligned}
$$

```
reachingDefs(stmt: Entity, $\sigma$: Store): Store =
  predecessor = $\sigma$(stmt, $\kappa_{\texttt{ControlFlowPred}}$)
  in = $\sigma$(predecessor, $\kappa_{\texttt{ReachingDefs}}$)
  out = stmt match
    case Assign(x,_,_) => in[x $\mapsto$ stmt]
    case _ => in
  $\sigma$[stmt, $\kappa_{\texttt{ReachingDefs}}$ $\mapsto$ out]

controlFlow(stmt1: Entity, $\sigma$: Store): Store =
  state1 = $\sigma$[Unit, $\kappa_{\texttt{State}}$]
  (stmt2, state2) = step(stmt1, state1)
  $\sigma$[stmt2, $\kappa_{\texttt{ControlFlowPred}}$ $\mapsto$ stmt1][Unit, $\kappa_{\texttt{State}}$ $\mapsto$ state2]
```

Dynamic module reachingDefs is analogous to its static counterpart $\widehat{\mathsf{reachingDefs}}$, but computes the *most recent* definition of a variable instead of all possible definitions. The dynamic module depends on the control-flow predecessor, which is the most recently executed statement. The control-flow predecessors are computed by module controlFlow, which is based on a small-step operational semantics $\texttt{step} : \texttt{Stmt} \times \texttt{ProgramState} \rightharpoonup \texttt{Stmt} \times \texttt{ProgramState}$. Module controlFlow demonstrates that the blackboard architecture is capable to integrate existing dynamic operational semantics, such as those for Java [6] or WebAssembly [18].

The blackboard analysis architecture not only modularizes the static semantics but also the dynamic semantics, which is crucial for enabling compositional and reusable soundness proofs. In particular, each static module is proven sound with respect to exactly one dynamic module, which limits the proof scope and guarantees proof independence. Furthermore, for analyses that approximate non-standard dynamic semantics, the standard dynamic semantics can be modularly extended with further modules (e.g., Section 5.1).

To summarize, in this section we formally defined the blackboard analysis architecture, which allows to implement static analyses modularly. Furthermore, we defined a dynamic semantics in the same style against which analyses are proven sound.

3 Compositional Soundness Proofs

In this section, we develop a theory of compositional soundness proofs for analyses in the blackboard style: Soundness of a compound analysis follows directly from soundness of the individual static modules. This soundness theory simplifies the soundness proof, because it allows analysis developers to focus on soundness of individual static modules, instead of having to reason about soundness of the interaction of all static modules with each other. Furthermore, the soundness theory makes the proofs more maintainable, as a change to a module only affects the proof of that module and nothing else.

We start the section by defining soundness of static modules and then work up to soundness of whole analyses. The definitions of soundness are standard and build upon the theory of *abstract interpretation* [12]:

Definition 3 (Soundness of Static Modules). *An static module* $\widehat{f} \in \widehat{\mathsf{Module}}$ *is sound if it overapproximates its dynamic counterpart* $f \in \mathsf{Module}$*:*

$$\begin{aligned}\mathsf{sound}(f, \widehat{f}) \;\; \textit{iff}\; \forall \widehat{e} \in \widehat{\mathsf{Entity}}, \widehat{\sigma} \in \widehat{\mathsf{Store}}, e \in \gamma_{\mathsf{Entity}}(\widehat{e}), \sigma \in \gamma_{\mathsf{Store}}(\widehat{\sigma}).\\ f(e, \sigma) \in \gamma_{\mathsf{Store}}(\widehat{f}(\widehat{e}, \widehat{\sigma}))\end{aligned} \qquad \square$$

The expression $x \in \gamma(\widehat{y})$ reads as "element $\widehat{y}$ soundly overapproximates the concrete element x." Function $\gamma : \widehat{L} \to \mathcal{P}(L)$ is a monotone function from an abstract domain $\widehat{L}$ to a powerset of a concrete domain L and is called *concretization function.* We do not require that an *abstraction function* $\alpha : \mathcal{P}(L) \to \widehat{L}$ in the opposite direction exists nor that γ and α form a Galois connection, both of which are not necessary for soundness proofs.

The soundness definition above requires that analysis developers define concretizations for entities ($\gamma_{\mathsf{Entity}} : \widehat{\mathsf{Entity}} \to \mathcal{P}(\mathsf{Entity})$) and properties ($\gamma_{\mathsf{Property}} : \widehat{\mathsf{Property}}[\kappa] \to \mathcal{P}(\mathsf{Property}[\kappa])$). Often the abstract and concrete entities are of the same type ($\widehat{\mathsf{Entity}} = \mathsf{Entity}$). In this case, the concretization functions map to singleton sets ($\gamma_{\mathsf{Entity}}(e) = \{e\}$). Based on concretization functions for entities, kinds, and properties, we define a point-wise concretization on stores. The definition can be found in the supplementary material.

In the following, we define soundness of compound analyses.

Definition 4 (Soundness of a Compound Analysis). *Let* $\Phi \subseteq \mathsf{Module} \times \widehat{\mathsf{Module}}$ *be a set of static modules paired with corresponding dynamic modules. A compound analysis is sound if the fixpoint of all of its static modules overapproximates the reachability semantics of the corresponding dynamic modules:*

$$\begin{aligned}\mathsf{sound}(\Phi) \;\; \textit{iff} \;\; \forall \widehat{\sigma} \in \widehat{\mathsf{Store}}.\; \mathsf{reachable}(F, \gamma_{\mathsf{Store}}(\widehat{\sigma})) \subseteq \gamma_{\mathsf{Store}}(\mathsf{fix}(\widehat{F}, \widehat{\sigma}))\\ \text{where } F = \{f \mid (f, \_) \in \Phi\} \text{ and } \widehat{F} = \{\widehat{f} \mid (\_, \widehat{f}) \in \Phi\}.\end{aligned} \qquad \square$$

The compound analysis approximates the dynamic reachability semantics (Definition 2.3), which collects the set of all stores reachable by applying dynamic modules. The dynamic reachability semantics is a collecting semantics, commonly used to prove soundness of abstract interpreters [12].

We are now ready to state the main theorem of this work:

Theorem 1 (Soundness Composition). *Let $\Phi \subseteq \mathsf{Module} \times \widehat{\mathsf{Module}}$ be a set of static modules paired with corresponding dynamic modules. Soundness of a compound analysis follows from soundness of all of its static modules:*

$$\text{If } \mathsf{sound}(f, \widehat{f}) \text{ for all } (f, \widehat{f}) \in \Phi \text{ then } \mathsf{sound}(\Phi).$$

Proof. *We show* $\mathsf{reachable}(F, \gamma_{\mathsf{Store}}(\widehat{\sigma}_1)) \subseteq \gamma_{\mathsf{Store}}(\mathsf{fix}(\widehat{F}, \widehat{\sigma}_1))$ *by well-founded induction on* $X \preceq \mathsf{reachable}(F, X)$.

- *Base case:* $\mathsf{reachable}(F, \varnothing) = \varnothing \subseteq \gamma_{\mathsf{Store}}(\mathsf{fix}(\widehat{F}, \widehat{\sigma}_1))$
- *Inductive case: Suppose* $X \subseteq \gamma_{\mathsf{Store}}(\mathsf{fix}(\widehat{F}, \widehat{\sigma}_1))$ *and* $\widehat{\sigma}_n = \mathsf{fix}(\widehat{F}, \widehat{\sigma}_1)$*. Then for all* $\sigma \in X \subseteq \gamma_{\mathsf{Store}}(\mathsf{fix}(\widehat{F}, \widehat{\sigma}_1))$*, we get* $dom(\sigma) \subseteq \gamma_{\mathsf{Entity} \times \mathsf{Kind}}(dom(\widehat{\sigma}_n))$ *and* $\sigma(e, k) \in \gamma_{\mathsf{Property}}(\widehat{\sigma}_n(\widehat{e}, \kappa))$ *for all* $\forall(\widehat{e}, \kappa) \in dom(\widehat{\sigma}_n)$ *and* $e \in \gamma_{\mathsf{Entity}}(\widehat{e})$*. Furthermore, since* $\widehat{\sigma}_n$ *is a fixpoint, it holds* $\widehat{f}(\widehat{e}, \widehat{\sigma}_n) \sqsubseteq \widehat{\sigma}_n$ *for all* $\widehat{f} \in \widehat{F}$ *and* $\widehat{e} \in dom(\widehat{\sigma}_n)$*. From* $\mathsf{sound}(f, \widehat{f})$ *we conclude* $f(e, \sigma) \in \gamma_{\mathsf{Store}}(\widehat{f}(\widehat{e}, \widehat{\sigma}_n)) \subseteq \gamma_{\mathsf{Store}}(\widehat{\sigma}_n) = \gamma_{\mathsf{Store}}(\mathsf{fix}(\widehat{F}, \widehat{\sigma}_1))$ *for all* $(f, \widehat{f}) \in \Phi$*,* $(e, \_) \in dom(\sigma)$*,* $(\widehat{e}, \_) \in dom(\widehat{\sigma}_n)$ *with* $e \in \gamma_{\mathsf{Entity}}(\widehat{e})$*. It follows* $\mathsf{reachable}(F, X) \subseteq \gamma_{\mathsf{Store}}(\mathsf{fix}(\widehat{F}, \widehat{\sigma}_1))$.

□

We illustrate this theorem by applying it to the reaching definitions analysis from Section 2.1. Specifically, soundness of the compound analysis follows from soundness of module $\widehat{\mathsf{reachingDefs}}$ module $\widehat{\mathsf{controlFlow}}$ by Theorem 1:

$$\frac{\mathsf{sound}(\mathsf{reachingDefs}, \widehat{\mathsf{reachingDefs}}) \qquad \mathsf{sound}(\mathsf{controlFlow}, \widehat{\mathsf{controlFlow}})}{\mathsf{sound}(\{(\mathsf{reachingDefs}, \widehat{\mathsf{reachingDefs}}), (\mathsf{controlFlow}, \widehat{\mathsf{controlFlow}})\})}$$

This means $\widehat{\mathsf{reachingDefs}}$ can be proven sound independently from $\widehat{\mathsf{controlFlow}}$, even though the modules interact with each other in the compound analysis. The proof independence is possible because neither module $\mathsf{reachingDefs}$ nor $\widehat{\mathsf{reachingDefs}}$ call the control-flow modules directly. Instead, both the static and dynamic module read the control-flow information from the stores, which are guaranteed to be a sound overapproximation initially (assumption $\sigma \in \gamma_{\mathsf{Store}}(\widehat{\sigma})$). Furthermore, only properties that the reaching-definitions modules themselves wrote to the store need to be sound overapproximations. Properties that other modules wrote to the store are not subject of the soundness proof of the reaching-definitions modules. The soundness proof of module $\widehat{\mathsf{reachingDefs}}$ is found in the supplementary material.

To summarize, in this section we developed a theory of compositional soundness proofs for analyses described in the blackboard architectural style. Each static module can be proven sound independently from other modules. Furthermore, soundness of a whole analysis follows directly from soundness of each module. In particular, no reasoning about the analysis as a whole is required.

4 Reusable Soundness Proofs

As of now, static modules refer to a specific type of entities, kinds, properties, and stores. However, adding new modules to an analysis may require extending these types. This invalidates the soundness proofs of existing modules and they need to be re-established. In this section, we extend our theory to make static modules and their soundness proofs reusable.

4.1 Extending the Type of Entities and Kinds

We start by explaining how entities and kinds can be extended without invalidating existing soundness proofs.

For example, if we were to add a taint static module to an existing analysis over types $\widehat{\mathsf{Entity}}$, Kind, and $\widehat{\mathsf{Store}}$, we needed to extend these types to hold the new analysis information:

$$\widehat{\mathsf{Entity}}' = \widehat{\mathsf{Entity}} \mid \mathsf{Var} \qquad\qquad \mathsf{Kind}' = \mathsf{Kind} \mid \kappa_{\mathtt{Taint}}$$

But this invalidates the proofs of existing modules that depend on the subsets $\widehat{\mathsf{Entity}}$ and Kind. To solve this problem, we first parameterize the type of modules to make explicit what types of entities and kinds they depend on:

Definition 5 (Parameterized Modules (Preliminary)). *We define a type of module that is parameterized by the types of entities E, kinds K, and store S:*

$$f \in \mathsf{Module}[E, K] = \forall S : \mathsf{Store}[E, K].\ \ E \times S \to S \qquad \square$$

Interface $\mathsf{Store}[E, K]$ defines read and write operations for an abstract store type S, that restricts access to entities of type E and kinds of type K. The store interface allows us to call parameterized modules with stores containing supersets of the type of entities and kinds.

For these parameterized modules, we define a sound *lifting* to supersets of entities and kinds:

```
lift : $\forall E', K', E \subseteq E', K \subseteq K', \mathsf{Module}[E, K] \to \mathsf{Module}[E', K']$
lift($f$)($e'$, $\sigma$) = $e'$ match
  case $e : E$ => $f$($e$, $\sigma$)
  case _ => $\sigma$
```

The lifting calls module f on all entities of type E on which f is defined and simply ignores all other entities, returning the store unchanged. For example, the lifted reaching-definitions module $\mathsf{lift}[\mathsf{Stmt} \mid \mathsf{Var},\ \kappa_{\mathtt{ReachingDefs}} \mid \kappa_{\mathtt{ControlFlowPred}} \mid \kappa_{\mathtt{Taint}}](\widehat{\mathsf{reachingDefs}})$ operates on the entities Stmt and the kinds $\kappa_{\mathtt{ReachingDefs}} \mid \kappa_{\mathtt{ControlFlowPred}}$, but ignores entities Var and kinds $\kappa_{\mathtt{Taint}}$.

The lifting preserves soundness of the lifted modules for disjoint extensions of entities.

Definition 6 (Disjoint Extension). *Entities $\widehat{E'} \supseteq \widehat{E}$ and $E' \supseteq E$ are a disjoint extension iff $\gamma_{\mathsf{Entity}}(\widehat{E}) \subseteq E$ and $\gamma_{\mathsf{Entity}}(\widehat{E'} \setminus \widehat{E}) \subseteq E' \setminus E$.* $\square$

In other words, the concretization function γ_{Entity} does not mix up entities in $\widehat{E}$ and $\widehat{E'} \setminus \widehat{E}$.

Lemma 1 (Lifting preserves Soundness). *Let $\widehat{f} \in \mathsf{Module}[\widehat{E}, K]$ and $f \in \mathsf{Module}[E, K]$ be a parameterized static module and dynamic module, $\widehat{E}' \supseteq \widehat{E}$ and $E' \supseteq E$ be a disjoint extension of entities, and $K' \supseteq K$ a superset of kinds.*

$$\text{If } \mathsf{sound}(f, \widehat{f}) \text{ then } \mathsf{sound}(\mathsf{lift}[E', K'](f), \mathsf{lift}[\widehat{E}', K'](\widehat{f})).$$

Proof. Let $\widehat{f} : \mathsf{Module}[\widehat{E}, K]$ and $f : \mathsf{Module}[E, K]$ be an analysis and dynamic module. Furthermore, let $\widehat{e} : \widehat{E}'$ and $e \in \gamma_{\mathsf{Entity}}(\widehat{e})$ be an entity and $\widehat{\sigma} : \mathsf{Store}[\widehat{E}', K']$ and $\sigma \in \gamma_{\mathsf{Store}}(\widehat{\sigma})$ be an abstract and concrete store.

- *In case $\widehat{e} \in \widehat{E}$, then also $e \in E$. Hence, $\mathsf{lift}(\widehat{f})(\widehat{e}, \widehat{\sigma}) = \widehat{f}(\widehat{e}, \widehat{\sigma})$ and $\mathsf{lift}(f)(e, \sigma) = f(e, \sigma)$. Soundness follows by $\mathsf{sound}(f, \widehat{f})$.*
- *In case $\widehat{e} \in \widehat{E}' \setminus \widehat{E}$, then also $e \in E' \setminus E$ for all $e \in \gamma_{\mathsf{Entity}}(\widehat{e})$. Hence $\mathsf{lift}(\widehat{f})(\widehat{e}, \widehat{\sigma}) = \widehat{f}(\widehat{e}, \widehat{\sigma})$ and $\mathsf{lift}(f)(e, \sigma) = f(\widehat{e}, \widehat{\sigma})$.* □

This lemma means that we can prove the soundness of static modules once for specific types of entities and kinds. Later, we can reuse the modules in a compound analysis with extended entities and kinds without having to prove soundness again.

4.2 Changing the Type of Properties

Next, we extend our theory to allow changing the type of properties without invalidating the soundness proofs of existing modules that use them.

For example, consider we already have a pointer-static module that propagates object allocation information $\widehat{\mathsf{Property}}[\kappa_{\mathtt{Val}}] = \widehat{\mathsf{Obj}}$. We may want to track string information as well. This could be done with a independent string-tracking static module with its own lattice. However, since tracking strings is mostly identical to tracking pointer information, such an additional module would duplicate significant amounts of code and require a new proof from scratch.

Instead, we can thus reuse the same pointer-static module to propagate string information $\widehat{\mathsf{Str}}$ by changing its lattice to $\widehat{\mathsf{Property}'}[\kappa_{\mathtt{Val}}] = \widehat{\mathsf{Obj}} \times \widehat{\mathsf{Str}}$. However, this invalidates the soundness proof of the pointer-static module as it depends on type $\widehat{\mathsf{Property}}[\kappa_{\mathtt{Val}}]$.

To solve this problem, we generalize the type of static modules again to be polymorphic over the type $\widehat{\mathsf{Property}}$:

Definition 7 (Parameterized Modules (Final)). *We define a type of module that is parameterized by the type of entities E, kinds K, properties P, and stores S:*

$$f \in \mathsf{Module}[E, K, I] = \forall P : I, S : \mathsf{Store}[E, K, P], E \times S \to S \qquad \square$$

Interface $\mathsf{Store}[E, K, P]$ restricts access to entities of type E and type K and contains properties of type P. Interface I defines operations on properties P.

For example, a pointer-static module may depend on the Scala-like interface Objects in Listing 1.1. Interface Objects depends on a type variable `Value`, which refers to possible values of variables. Function newObj creates a new object of a

```
trait Objects[Value]:
  newObj(class: Class, ctx: Context): Value
  forObj[$S$](Value, $S$)(f: (Class, Context, $S$) => $S$): $S$

object $\widehat{\mathsf{AllocationSite}}$ extends Objects[$\widehat{\mathsf{Obj}}$]:
  newObj(class, ctx) = {(class, ctx)}
  forObj[$S$]($\widehat{\mathsf{Obj}}$(objs), $\widehat{\sigma}$)(f) = $\bigsqcup_{\texttt{(class,ctx)}\in\texttt{objs}}$ f(class,ctx,$\widehat{\sigma}$)

object $\widehat{\mathsf{AllocationSiteAndStrings}}$ extends Objects[$\widehat{\mathsf{Obj}} \times \widehat{\mathsf{Str}}$]:
  newObj(class, ctx) = ({(class, ctx)}, $\bot$)
  forObj[$S$](value, $\widehat{\sigma}$)(f) = value match
    case (objs,_) => $\bigsqcup_{\texttt{(class,ctx)}\in\texttt{objs}}$ f(class, ctx, $\widehat{\sigma}$)
```

Listing 1.1: Interface for different Object Abstractions

certain class and context. Function forObj iterates over all such objects applying continuation `f`. Continuation `f` takes a class name, context, and store and returns a modified store. Interface Objects can be instantiated to support different value abstractions. For example, instance $\widehat{\mathsf{AllocationSite}}$ implements the interface with an allocation-site abstraction $\widehat{\mathsf{Obj}} = \widehat{\mathsf{Obj}}(\mathcal{P}(\mathsf{Class} \times \mathsf{Context}))$ which abstracts object allocations by their class names and a call string to their allocation site. Instance $\widehat{\mathsf{AllocationSiteAndStrings}}$ implements a reduced product [9] of objects $\widehat{\mathsf{Obj}}$ and strings $\widehat{\mathsf{Str}} = \texttt{Constant}[\texttt{String}]$, which abstracts the value of strings with a constant abstraction. This allows us to reuse the same pointer-static module to propagate string information.

Note that certain interfaces may restrict what instances can be implemented. For example, an abstract domain that only approximates strings but not objects, cannot soundly implement operation `forObj` in interface `Objects`. In this case, interfaces need to be generalized to allow a wider range of instances.

4.3 Soundness of Parameterized Modules

In this subsection, we define soundness of parameterized static modules and prove a generalized soundness composition theorem.

Definition 8 (Soundness of Parameterized static Modules). *A parameterized static module $\widehat{f} : \widehat{\mathsf{Module}}[\widehat{E}, K, I]$ is sound w.r.t. a parameterized dynamic module $f : \mathsf{Module}[E, K, I]$ iff all their instances are sound:*

$$\begin{aligned}\mathsf{sound}(f, \widehat{f}) \text{ iff } &\forall P : I, \widehat{P} : I, S : \mathsf{Store}[E, K, P], \widehat{S} : \mathsf{Store}[\widehat{E}, K, \widehat{P}].\\ &\mathsf{sound}(P, \widehat{P}) \implies \mathsf{sound}(f[P, S], \widehat{f}[\widehat{P}, \widehat{S}]).\end{aligned} \qquad \square$$

Parameterized static modules are proven sound for all sound instances of property interface I. A static instance $\widehat{P} : I$ is sound w.r.t. to a dynamic instance $P : I$, if all of its operations are sound. Soundness for dynamic and static instances of interface Objects in Listing 1.1 is defined as follows:

$$\begin{aligned}&\mathsf{sound}(\mathsf{newObj}, \widehat{\mathsf{newObj}}) \;\text{ iff }\; \forall c, \widehat{h}, h \in \gamma(\widehat{h}), \mathsf{newObj}(c,h) \in \gamma_{\mathsf{Obj}}(\widehat{\mathsf{newObj}}(c,\widehat{h}))\\ &\mathsf{sound}(\mathsf{forObj}, \widehat{\mathsf{forObj}}) \;\text{ iff }\; \forall f, \widehat{f}, \mathsf{sound}(f,\widehat{f}) \implies \mathsf{sound}(\mathsf{forObj}(f), \widehat{\mathsf{forObj}}(\widehat{f}))\end{aligned}$$

Soundness of first-order operations like $\widehat{\mathsf{newObj}}$ is similar to that of static modules (Definition 3). Soundness of higher-order operations like $\widehat{\mathsf{forObj}}$ is proven w.r.t. all sound functions $\widehat{f}$.

Finally, we generalize the soundness composition Theorem 1 to parameterized static modules. In particular, an analysis composed of parameterized static modules is sound if all of its modules are sound and the instance of its property interface is sound.

Theorem 2 (Soundness Composition for Parameterized Static Modules). *Let Φ be parameterized static modules paired with corresponding dynamic modules over families of entities $\widehat{E}' = \bigcup_i \widehat{E}_i, E' = \bigcup_i E_i$, kinds $K' = \bigcup_i K_i$, properties $\widehat{P}$, P.*

$$\begin{gathered}\text{If } \mathsf{sound}(f,\widehat{f}) \text{ for all } (f,\widehat{f}) \in \Phi \text{ and } \mathsf{sound}(P,\widehat{P}) \text{ then } \mathsf{sound}(\Phi'),\\ \text{where } \Phi' = \{(\mathsf{lift}[E',K'](f), \mathsf{lift}[\widehat{E}',K'](\widehat{f})) \mid (f,\widehat{f}) \in \Phi\}\end{gathered}$$

Proof. We instantiate the polymorphic modules $f, \widehat{f}$ with the compound types to obtain $\mathsf{sound}[E',K'](\mathsf{lift}(f), \mathsf{lift}[E',K'](\widehat{f}))$. Then the soundness composition Theorem 3.4 for monomorphic modules applies. □

To summarize, in this section we explained how the type of entities, kinds, and properties can be changed without invalidating the soundness proofs of existing modules. To this end, we generalized the type of modules to be parametric over the type of entities, kinds, and properties. The parameterized modules access properties via an interface. The instances of this interface are specific to certain types of properties and require a soundness proof.

5 Applicability of the Theory

In this section, we demonstrate the applicability of our theory by first developing four analyses in the blackboard architecture and then proving them sound compositionally.

5.1 Case Studies

We developed four different analyses in the blackboard architecture (Section 2) together with their dynamic semantics (Section 2.2). We proved each analysis sound and discuss the proofs in Section 5.2. Each analysis exercises a specific part of our soundness theory:

- A pointer analysis which mutually depends on a call-graph analysis (exercises the part of our theory presented in Section 3).

- A reflection analysis which reuses the pointer analysis to propagate string information (exercises Section 4.2).
- A field and object immutability analysis depending on all above analyses (exercises Section 4.1).
- A demand-driven reaching-definitions analysis which demonstrates that our theory applies to this type of analyses.

Our choice of analyses was inspired by similar but more complex analyses for JVM-bytecode implemented in OPAL, which scale to real-world applications [21, 41]. Our analyses operate on a simpler object-oriented language with the following abstract syntax:

$$\begin{aligned}
&\mathsf{Class} = \mathsf{Class}(\mathsf{ClassName}, \mathsf{ClassName}, \mathsf{Field}^*, \mathsf{Method}^*)\\
&\mathsf{Method} = \mathsf{SourceMethod}(\mathsf{MethodName}, \mathsf{Var}^*, \mathsf{Stmt}^*) \mid \mathsf{NativeMethod}(\mathsf{MethodName})\\
&\mathsf{Stmt} = \mathsf{Assign}(\mathsf{Ref}, \mathsf{Expr}) \mid \mathsf{Return}(\mathsf{Method}, \mathsf{Expr}) \mid \mathsf{If}(\mathsf{Expr}, \mathsf{Stmt}^*, \mathsf{Stmt}^*)\\
&\qquad \mid \mathsf{While}(\mathsf{Expr}, \mathsf{Stmt}^*)\\
&\mathsf{Expr} = \mathsf{Ref} \mid \mathsf{New}(\mathsf{ClassName}, (\mathsf{Field} \times \mathsf{Expr})^*) \mid \mathsf{StringLit}(\mathsf{String}) \mid \mathsf{Concat}(\mathsf{Expr}, \mathsf{Expr})\\
&\qquad \mid \mathsf{Call}(\mathsf{Expr}, \mathsf{MethodName}, \mathsf{Expr}^*) \mid \mathsf{BoolLit}(\mathsf{Bool}) \mid \mathsf{Equals}(\mathsf{Expr}, \mathsf{Expr})\\
&\mathsf{Ref} = \mathsf{VarRef}(\mathsf{Var}) \mid \mathsf{FieldRef}(\mathsf{Ref}, \mathsf{Field})
\end{aligned}$$

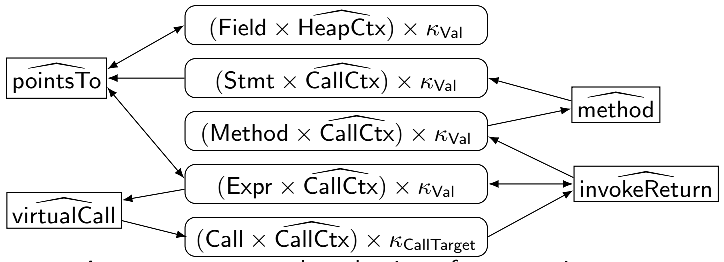
The language features inheritance, mutable memory, class fields, virtual method calls, and Java-like reflection [35]. Reflection is modeled as virtual calls to native methods. We also deliberately added features such as control-flow constructs and boolean operations. These are ignored by the analyses, but need to be modeled by dynamic semantics, complicating the soundness proof of the analyses.

We implemented and tested each analysis in Scala to ensure they are executable. Furthermore, we implemented and tested the corresponding dynamic semantics to ensure they are sensible. The code of analyses and dynamic semantics can be found in the supplementary material accompanying this paper. In the following, we discuss the implementation of each analysis in more detail.

Pointer and Call-Graph Analysis A pointer analysis for an object-oriented language computes which objects a variable or field may point to. A call-graph analysis determines which methods may be called at specific call sites. Pointer and call-graph analyses are the foundation which many other analyses build upon.

The analyses are composed from four static modules, whose dependencies are visualized in Figure 1. An arrow from a store entry to a module represents a read, an arrow in the other direction represents a write. Even though all modules implicitly depend on each other, they can be proven sound independently from each other (Section 3). This is possible because they do not call other modules directly, instead, all communication happens via the store.

Module $\widehat{\mathsf{method}}$ registers each statement of a method in the store to trigger other modules. It disregards control flow as the analysis is flow-insensitive and hence also registers statements that can never be executed. Flow-insensitive


$(\mathsf{Field} \times \widehat{\mathsf{HeapCtx}}) \times \kappa_{\mathsf{Val}}$
$\widehat{\mathsf{pointsTo}}$
$(\mathsf{Stmt} \times \widehat{\mathsf{CallCtx}}) \times \kappa_{\mathsf{Val}}$
$\widehat{\mathsf{method}}$
$(\mathsf{Method} \times \widehat{\mathsf{CallCtx}}) \times \kappa_{\mathsf{Val}}$
$(\mathsf{Expr} \times \widehat{\mathsf{CallCtx}}) \times \kappa_{\mathsf{Val}}$
$\widehat{\mathsf{invokeReturn}}$
$\widehat{\mathsf{virtualCall}}$
$(\mathsf{Call} \times \widehat{\mathsf{CallCtx}}) \times \kappa_{\mathsf{CallTarget}}$

Arrows represent reads and writes of store entries

Fig. 1: Points-To and Call-Graph Static Modules

analyses can be more performant than flow-sensitive ones, but traditional approaches using generic abstract interpreters do not allow for flow-insentitive analyses. Module $\widehat{\mathsf{pointsTo}}$ analyzes `New` expressions and assignments of variable and field references. Module $\widehat{\mathsf{virtualCall}}$ resolves target methods of virtual calls based on the receiver object. Once a call is resolved, module $\widehat{\mathsf{invokeReturn}}$ extends the call context, assigns the method parameters and return value. Finally, it registers the called method as an entity in the store, triggering module $\widehat{\mathsf{method}}$.

The entities of the analyses are fields, statements, expressions, methods, and calls:

$$\begin{aligned}
\widehat{\mathsf{Entity}} &= (\mathsf{Field} \times \widehat{\mathsf{HeapCtx}}) \mid (\mathsf{Stmt} \times \widehat{\mathsf{CallCtx}}) \mid (\mathsf{Expr} \times \widehat{\mathsf{CallCtx}}) \\
&\quad \mid (\mathsf{Method} \times \widehat{\mathsf{CallCtx}}) \mid (\mathsf{Call} \times \widehat{\mathsf{CallCtx}}) \\
\widehat{\mathsf{Property}}[\kappa_{\mathsf{Val}}] &= \bot \mid \widehat{\mathsf{Obj}} \\
\widehat{\mathsf{Property}}[\kappa_{\mathsf{CallTarget}}] &= \widehat{\mathsf{CallTarget}} \\
\widehat{\mathsf{Obj}} &= \widehat{\mathsf{Obj}}(\mathcal{P}(\mathsf{Class} \times \widehat{\mathsf{HeapCtx}})) \\
\widehat{\mathsf{CallTarget}} &= \widehat{\mathsf{CallTarget}}(\mathcal{P}(\mathsf{Class} \times \widehat{\mathsf{HeapCtx}} \times \mathsf{Method} \times \mathsf{Expr}^*))
\end{aligned}$$

Each entity is paired with a call context or heap context, which allows to tune the precision of the analysis. The static modules communicate via two kinds: Kind κ_{Val} refers to possible values of expressions and fields and the return value of methods. Values are abstract objects containing information about where objects were allocated. Kind $\kappa_{\mathsf{CallTarget}}$ refers to possible targets of method calls. Call targets are sets of receiver objects paired with the target method and their arguments.

To illustrate the analysis, Listing 1.2 shows the code of modules $\widehat{\mathsf{virtualCall}}$ and $\widehat{\mathsf{invokeReturn}}$. They implicitly communicate with each other via the store but do not call each other directly. Module $\widehat{\mathsf{virtualCall}}$ resolves virtual method calls by first fetching the points-to set of the receiver reference from the store. Afterwards, it iterates over all possible receivers and fetches possible target methods from the class table. Finally, it writes the new call target to the store. Storing the receiver object and argument expressions as part of the call target allows to reuse module $\widehat{\mathsf{invokeReturn}}$ for different types of calls. If the entity is a `Call` expression, module $\widehat{\mathsf{invokeReturn}}$ first fetches the targets of the call from the store. Then, it iterates over all targets, extends the call context with function

```
$\widehat{\mathsf{virtualCall}}$($e$, $\widehat{\sigma}$) = $e$ match
  case (call@Call(receiver, methodName, args), callCtx) =>
    receiverVal = $\widehat{\sigma}$((receiver, callCtx), $\kappa_{\mathsf{Val}}$)
    $\widehat{\mathsf{forObj}}$(receiverVal, $\widehat{\sigma}$) { (class, heapCtx, $\widehat{\sigma}'$) =>
      method = classTable(class, methodName)
      $\widehat{\sigma}' \sqcup [(\mathtt{call}, \mathtt{callCtx}), \kappa_{\mathsf{CallTarget}} \mapsto \widehat{\mathsf{newCallTarget}}(\mathtt{class}, \mathtt{heapCtx}, \mathtt{method}, \mathtt{args})]$
    }
  case _ => $\widehat{\sigma}$

$\widehat{\mathsf{invokeReturn}}$($e$, $\widehat{\sigma}$) = $e$ match
  case (call@Call(_,_,_), callCtx) =>
    targets = $\widehat{\sigma}((\mathtt{call}, \mathtt{callCtx}), \kappa_{\mathsf{CallTarget}})$
    $\widehat{\mathsf{forCallTarget}}$(targets,$\widehat{\sigma}$){(class,heapCtx,method,args,$\widehat{\sigma}'$) => method match
      case SourceMethod(_,params,_) =>
        newCallCtx = $\widehat{\mathsf{extendCtx}}$(call.label,heapCtx,callCtx)
        $\widehat{\sigma}' \sqcup [(\mathtt{call}, \mathtt{callCtx}), \kappa_{\mathsf{Val}} \mapsto \widehat{\sigma}'((\mathtt{method}, \mathtt{newCallCtx}), \kappa_{\mathsf{Val}})]$
           $\sqcup [(\mathtt{p}, \mathtt{newCallCtx}), \kappa_{\mathsf{Val}} \mapsto \widehat{\sigma}'((\mathtt{a}, \mathtt{callCtx}), \kappa_{\mathsf{Val}}) \mid (\mathtt{p}, \mathtt{a}) \in \mathsf{zip}(\mathtt{params}, \mathtt{args})]$
           $\sqcup [(\mathsf{VarRef}(\text{"}\mathtt{this}\text{"}), \mathtt{newCallCtx}), \kappa_{\mathsf{Val}} \mapsto \widehat{\mathsf{newObj}}(\mathtt{class}, \mathtt{heapCtx})]$
           $\sqcup [(\mathtt{method}, \mathtt{callCtx}), \kappa_{\mathsf{Val}} \mapsto \widehat{\mathsf{nullPointer}}()]$
           $\sqcup [(\mathtt{call}, \mathtt{callCtx}), \kappa_{\mathsf{Val}} \mapsto \widehat{\sigma}((\mathtt{method}, \mathtt{newCallCtx}), \kappa_{\mathsf{Val}})]$
      case NativeMethod(_,_,_) => $\widehat{\sigma}'$
    }
  case Return(method,expr) =>
      $\widehat{\sigma} \sqcup [(\mathtt{method}, \mathtt{callCtx}), \kappa_{\mathsf{Val}} \mapsto \widehat{\sigma}(\mathtt{expr}, \mathtt{callCtx}, \kappa_{\mathsf{Val}})]$
  case _ => $\widehat{\sigma}$
```

Listing 1.2: Static modules for invoking calls and resolving virtual calls.

$\widehat{\mathsf{extendCtx}}$, binds the parameters to the values of the arguments and variable `this` to the receiver object. Furthermore, it registers the called method as an entity in the store, which in turn triggers module $\widehat{\mathtt{method}}$ to process the statements of the called method. Lastly, module $\widehat{\mathsf{invokeReturn}}$ writes the return value of a method to the method entity in the store and copies it to call entities of this method.

The modules depend on interface Objects shown in Listing 1.1 and an analogous interface for call targets. Operations $\widehat{\mathsf{newObj}}$ and $\widehat{\mathsf{newCallTarget}}$ create new abstract objects and call targets. Operations $\widehat{\mathsf{forObj}}$ and $\widehat{\mathsf{forCallTarget}}$ iterate over all objects and call targets. Interface Objects also includes an operation $\widehat{\mathsf{nullPointer}}$ not shown in the listing, which returns an empty set of object allocation-sites ($\widehat{\mathsf{Obj}}(\varnothing)$). The dynamic instances are analogous except that they operate on singleton types.

The dynamic modules compute a program's *heap* and describe its changes during execution. They are analogous to their static counterparts except that they operate on singleton types $\mathsf{Obj}(\mathsf{Class} \times \mathsf{HeapCtx})$ and $\mathsf{CallTarget}(\mathsf{Class} \times \mathsf{HeapCtx} \times \mathsf{Method} \times \mathsf{Expr}^*)$.

All dynamic modules combined do not cover the entire language. In particular, there are no dynamic modules that cover reflective calls. This means, as of now, the dynamic semantics of reflection is undefined, and the soundness proof

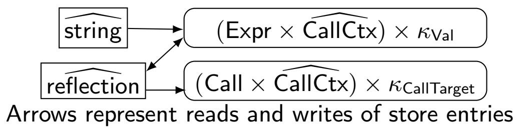

$\widehat{\text{string}}$
$\widehat{\text{reflection}}$
$(\text{Expr} \times \widehat{\text{CallCtx}}) \times \kappa_{\text{Val}}$
$(\text{Call} \times \widehat{\text{CallCtx}}) \times \kappa_{\text{CallTarget}}$
Arrows represent reads and writes of store entries

Fig. 2: Reflection Static Modules

only covers programs without reflective calls. We address this point with the following case study.

Reflection Analysis Reflection is a language feature that allows to query information about classes and methods at runtime [35]. Our language supports three reflective methods: Methods Class.forName and Class.getMethod retrieve classes and methods by a string, respectively. Method.invoke invokes a method, where the target method is determined at runtime. Reflection is notoriously difficult to statically analyze soundly and precisely [30]: analyses need to approximate the content of the string passed into a reflective call. If the analysis cannot determine the string precisely, it needs to overapproximate or risk unsoundness. In this case study, we choose the former to be able to prove the analysis sound.

This case study demonstrates two important features of our formalization: First, the reflection analysis reuses all modules of the pointer and call-graph analysis of the previous section ($\widehat{\texttt{pointsTo}}$, $\widehat{\texttt{method}}$, $\widehat{\texttt{virtualCall}}$, and $\widehat{\texttt{invokeReturn}}$). It extends the value lattice to propagate new types of analysis information about strings. Even though the pointer analysis propagates new information, it does not require any changes and its soundness proof remains valid (Section 4.2). Second, the reflection analysis *cooperates* with the call-graph static module $\widehat{\texttt{virtualCall}}$ as reflective calls are regular virtual calls. For example, a call m.invoke(...) where variable m is of type Method is first resolved by virtual call resolution and its target Method.invoke is then resolved by reflective call resolution. Thus, both analyses add elements to the same set of call targets but can be proven sound independently from each other (Section 3).

The reflection analysis extends the $\widehat{\texttt{Obj}}$ values of the pointer analysis with three new types of values—$\widehat{\text{Str}}$, $\widehat{\text{Class}}$, and $\widehat{\text{Method}}$—as a reduced product [9]:

$$\widehat{\text{Property}}[\kappa_{\text{Val}}] = \bot \mid (\widehat{\text{Obj}} \times \widehat{\text{Str}} \times \widehat{\text{Class}} \times \widehat{\text{Method}})$$
$$\widehat{\text{Str}} = \bot \mid \text{String} \mid \top$$
$$\widehat{\text{Class}} = \mathcal{P}(\text{Class}) \mid \top$$
$$\widehat{\text{Method}} = \mathcal{P}(\text{Method}) \mid \top$$

String values are approximated with a constant lattice. Class and method values are approximated with a finite set of classes/methods or $\top$. We reuse the modules of the pointer and call-graph analysis by implementing a new instance of interface Objects in Listing 1.1 for the new values. The new instance is similar to $\text{Allocation}\widehat{\text{SiteAnd}}\text{Strings}$ and iterates over all allocation-site information in strings, class/method values, and other objects.

The reflection analysis adds two new modules to the existing analysis in Figure 1. The new modules and their dependencies are visualized in Figure 2.

```
$\widehat{\text{reflection}}$($e$, $\widehat{\sigma}$) = $e$ match
  case (call@Call(receiver, method, _), callCtx) =>
    target = $\widehat{\sigma}$((call, callContext), $\kappa_{\text{CallTarget}}$)
    for$\widehat{\text{CallTarget}}$(target, $\widehat{\sigma}$) { (_,heapCtx,method,args,$\widehat{\sigma}'$) =>
      method match
      case NativeMethod("invoke") => arguments match
          case (invokeReceiver :: invokeArgs) =>
            invokeRecVal = $\widehat{\sigma}'$((invokeReceiver, heapCtx), $\kappa_{\text{Val}}$)
            methodVal = $\widehat{\sigma}'$((receiver, callContext), $\kappa_{\text{Val}}$)
            reflectiveTarget = $\widehat{\text{methodInvoke}}$(invokeRecVal, methodVal,
                invokeArgs)
            $\widehat{\sigma}' \sqcup [(\texttt{call}, \texttt{callCtx}), \kappa_{\text{CallTarget}} \mapsto \texttt{reflectiveTarget}]$
      ... }
  case _ => $\widehat{\sigma}$

$\widehat{\text{methodInvoke}}$(recv:$\widehat{\text{Value}}$,methodVal:$\widehat{\text{Value}}$,invokeArgs:Expr$^*$) = methodVal match
  case (_,_,_,methods) => $\widehat{\text{CallTarget}}$({(c,h,m,invokeArgs) |
       (c,h)$\in$recv, m $\in$ methods, m $\in$ classTable(c)})
  case (_,_,_,$\top$) => $\widehat{\text{CallTarget}}$({ (c,h,m,invokeArgs) |
       (c,h) $\in$ recv, method $\in$ classTable(class) })
  case $\bot$ => $\bot$
```

Listing 1.3: Static modules and operations for reflection.

Module $\widehat{\text{reflection}}$ analyzes reflective calls to Class.forName, Class.getMethod, and Method.invoke. Module $\widehat{\text{string}}$ analyzes string literals and concatenation. Listing 1.3 shows an excerpt of module $\widehat{\text{reflection}}$ for Method.invoke. Module $\widehat{\text{reflection}}$ first fetches the targets of a call resolved by module $\widehat{\text{virtualCall}}$. If the call target is the native method `invoke`, module $\widehat{\text{reflection}}$ matches on the arguments of the virtual call to extract the receiver and arguments of the reflective call target. Finally, it calls operation $\widehat{\text{methodInvoke}}$ which returns the set of call targets.

Operation $\widehat{\text{methodInvoke}}$ is part of an interface for reflective calls. The interface contains two other operations for retrieving class names and methods. $\widehat{\text{methodInvoke}}$ matches on the call receiver and the method value. If the method value contains a finite set of methods, the operation checks if the receiver class has these methods and adds them as call targets. If the method value contains $\top$, the operation adds all methods of the receiver class to the set of call targets. This over-approximates the dynamic module reflection where only one method is added as a call target.

The dynamic reflection modules are analogous except that different types of values are alternatives. In contrast to Section 5.1, the dynamic pointer and call-graph modules combined with the reflection and string modules now cover the entire language. Thus, the analysis is sound for all programs, even those using reflection.

Field and Object Immutability Analysis The analysis of this case study computes the immutability of objects and their fields inspired by a class and field immutability analysis by Roth et al. [41]. This information is useful for assessing the thread safety of programs, where multiple threads have access to the same objects.

This case study highlights two important features of our formalization. First, the core dynamic semantics of our language does not describe the immutability property. Therefore, we need to prove the static immutability analysis sound with respect to a dynamic immutability analysis. The case study demonstrates that the immutability concern can be encapsulated with analysis and dynamic modules, added modularly to the existing analysis and dynamic semantics, and reasoned about independently (Section 3). It is unclear how this can be achieved with a non-modular, monolithic analysis implementation. Second, the immutability analysis adds new types of entities and kinds to the store and reuses all modules of the pointer, call-graph, and reflection analysis. Even though the reused modules can be called with the new entities and have access to new kinds in the store, their soundness proofs remain valid (Section 4.1).

The immutability analysis adds objects ($\mathsf{Class} \times \mathsf{HeapCtx}$) to the types of entities and adds kinds κ_{Mut} and κ_{Assign} for their immutability and the assignability of their fields:

$$\widehat{\mathsf{Entity}}' = \widehat{\mathsf{Entity}} \mid (\mathsf{Class} \times \mathsf{HeapCtx})$$
$$\widehat{\mathsf{Property}}[\kappa_{\mathsf{Mut}}] = \widehat{\mathsf{TransitivelyImmutable}} \mid \widehat{\mathsf{NonTransitivelyImmutable}} \mid \widehat{\mathsf{Mutable}}$$
$$\widehat{\mathsf{Property}}[\kappa_{\mathsf{Assign}}] = \widehat{\mathsf{Assignable}} \mid \widehat{\mathsf{NonAssignable}}$$

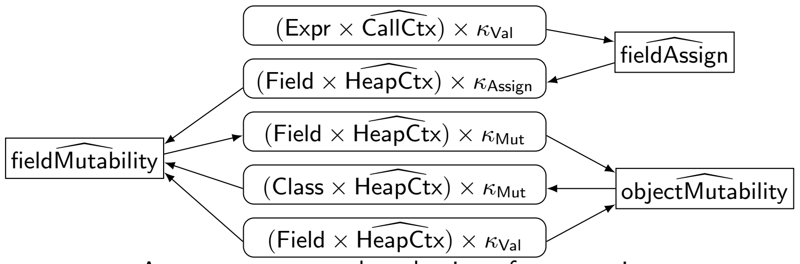
$\widehat{\mathsf{Mutable}}$ describes objects whose fields are reassigned. $\widehat{\mathsf{NonTransitivelyImmutable}}$ describes objects whose fields are not reassigned, but some objects transitively reachable via fields are mutated. $\widehat{\mathsf{TransitivelyImmutable}}$ describes objects whose fields are not reassigned and no transitively reachable objects are mutated. κ_{Assign} uses two elements for reassigned and not reassigned fields.

The immutability analysis consists of three modules shown in Figure 3. Module $\widehat{\mathsf{fieldAssign}}$ sets fields `f` of objects `o` to $\widehat{\mathsf{Assignable}}$ for every assignment of the form `x.f = e`, where `x` may point to `o`. Module $\widehat{\mathsf{fieldMutability}}$ sets a field to $\widehat{\mathsf{Mutable}}$ if the field is assignable, to $\widehat{\mathsf{NonTransitivelyImmutable}}$ if it is non-assignable but one of the pointed-to objects is mutable, and to $\widehat{\mathsf{TransitivelyImmutable}}$ otherwise. Lastly, module $\widehat{\mathsf{objectMutability}}$ sets an object's immutability to the least upper bound of the immutability of all of its fields.

The dynamic modules are analogous except that they operate on concrete objects instead of abstract objects.

Demand-Driven Reaching-Definitions Analysis As a final case study, we developed a demand-driven intra-procedural reaching-definitions analysis for our object-oriented language. This case study demonstrates that our theory lifts a restriction of existing soundness theories for generic interpreters. In particular, our theory also applies to analyses that do not follow the program execution order.

The analysis computes which definitions of variables and fields reach a statement without being overwritten. The analysis is demand-driven, as it performs


$(\mathsf{Expr} \times \widehat{\mathsf{CallCtx}}) \times \kappa_{\mathsf{Val}}$
$(\mathsf{Field} \times \widehat{\mathsf{HeapCtx}}) \times \kappa_{\mathsf{Assign}}$
$(\mathsf{Field} \times \widehat{\mathsf{HeapCtx}}) \times \kappa_{\mathsf{Mut}}$
$(\mathsf{Class} \times \widehat{\mathsf{HeapCtx}}) \times \kappa_{\mathsf{Mut}}$
$(\mathsf{Field} \times \widehat{\mathsf{HeapCtx}}) \times \kappa_{\mathsf{Val}}$
$\widehat{\mathsf{fieldAssign}}$
$\widehat{\mathsf{fieldMutability}}$
$\widehat{\mathsf{objectMutability}}$

Arrows represent reads and writes of store entries

Fig. 3: Immutability Static Modules

the minimum amount of work to compute the reaching definitions of a query statement: the analysis only computes the reaching definitions of the query statement and its predecessors. Also, the analysis does not compute the entire control-flow graph, but only the query statement's predecessors.

The analysis consists of two modules $\widehat{\mathsf{reachingDefs}}$ and $\widehat{\mathsf{controlFlow}}$ similar to these discussed in Section 2. Module $\widehat{\mathsf{controlFlow}}$ calculates the set of control-flow predecessors of a given statement by computing the set of control-flow exits of the preceding statement within the abstract syntax tree. For example, the control-flow exits of an `if` statement are the exits of the last statements of both branches. The dynamic module controlFlow computes the predecessor immediately executed before the given statement. To this end, the module remembers the most recently executed statement in a mutable variable and only updates it if the given statement is the control-flow successor.

The main challenge in this case study was to find a dynamic module controlFlow that closely corresponds to the static module and still computes the correct control-flow predecessor. With a suitable dynamic module, the soundness proof of the static module became easier. Furthermore, we validated the correctness of the dynamic module with several unit tests.

5.2 Soundness Proofs of the Case Studies

We apply our theory to compositionally prove the analyses from the previous section sound. The proofs can be found in the supplementary material accompanying this paper. They are pen-and-paper proofs and do not make use of mechanization; but due to modularization, they are small and easy to verify.

Proving each analysis sound includes (a) proving each of its modules sound (Definition 8), (b) proving the instances of the property interface sound, and (c) verifying that Theorem 2 applies. To ensure the latter, we checked that there are no dependencies between modules and that all communication between them happens via the store (Definition 1). This can be easily checked by inspecting the code of the modules. Furthermore, we verified that modules do not make any assumption about abstract domains and are polymorphic in the store (Definition 7). This can be easily checked by inspecting the polymorphic type of the modules.

To prove the individual modules of an analysis sound, step (a) in the overall soundness proof, we use two techniques. The first uses the observation that static modules and their corresponding dynamic modules are often very similar, except for the types of entities and properties. We can abstract over these differences with a generic module, from which we derive both a dynamic and static module. Then, soundness follows immediately as a free theorem from parametricity [28]. In cases where abstracting with a generic module is not possible or desirable, we resort to a manual proof. We were able to use the first technique for all modules, except for $\widehat{\textsf{method}}$, $\widehat{\textsf{reachingDefs}}$, and $\widehat{\textsf{controlFlow}}$. For illustrating cases where we need manual proofs, consider the flow-insensitive static module $\widehat{\textsf{method}}$ of the pointer analysis and its corresponding dynamic module $\textsf{method}$. While we could potentially derive them from the same generic module, the derived static module would be less performant, because it would trigger the analysis of parts of the code, e.g., `if` conditions, which our current flow-insensitive module does not. This is an example where our approach leads to more freedom in the design of static analyses than the existing approach based on a generic interpreter (Section 6.1).

The soundness proofs of the static modules are reusable across different analyses, because the modules can be soundly lifted to supersets of entities and kinds (Lemma 1). For example, the immutability analysis adds class entities, requiring to lift the modules of the pointer and reflection analysis. Furthermore, the soundness proofs of static modules can be reused because the proofs are independent of the lattices used (Definition 8). For example, the reflection analysis reuses all modules of the pointer analysis, extending the value lattice with string, class, and method information. The soundness proofs of the pointer static modules remain valid because they do not depend on a specific value lattice. Instead, the proofs of the pointer modules depend on soundness lemmas of the $\textsf{newObj}$ and $\textsf{forObj}$ operations of $\textsf{Objects}$ interface.

Finally, we consider step (b) in the overall soundness proof – the soundness proof of the instances of the property interface. These instances need to be proven sound manually, as the proof cannot be decomposed any further. To prove them sound, we proved each of their operations sound. For the pointer analysis we needed to prove 7 operations sound, for the reflection analysis 6 operations, for the immutability analysis 6 operations, and for the reaching-definitions analysis 0 operations. Of these 19 operations, 13 could be proven sound trivially, requiring only a single proof step after unfolding the definitions. The remaining 6 operations required more elaborate proofs with multiple steps and case distinctions. These include $\widehat{\textsf{forObj}}$ from the pointer analysis, $\widehat{\textsf{classForName}}$, $\widehat{\textsf{getMethod}}$, and $\widehat{\textsf{methodInvoke}}$ from the reflection analysis, and $\widehat{\textsf{getFieldMutability}}$ and $\widehat{\textsf{joinMutability}}$ from the immutability analysis.

6 Related Work

In this section, we discuss work related to compositional and reusable soundness proofs as well as to modular analysis architectures.

6.1 Theories for Compositional and Reusable Soundness Proofs

All works discussed in this subsection, including our own, build upon the theory of *abstract interpretation*. Abstract interpretation is a formal theory of sound static analyses, first conceived by Cousot et al. [12] but since then has found wide spread adoption in academia and industry [13, 16, 22, 25, 33, 44]. Abstract interpretation defines soundness of static analyses but does not explain how soundness can be proved. As we elaborate in the introduction, soundness proofs of practical analyses for real-world languages are difficult because they relate two complicated semantics often described in different styles. Proof attempts of such analyses often fail due to high proof complexity and effort. Furthermore, existing proofs are prone to become invalid if the static or dynamic semantics change and reestablishing proofs is laborious and complicated.

Domain constructions, such as *reduced products* and *reduced cardinal powers* [12], combine multiple existing abstract domains to improve their precision. They can be used to compose the soundness proof of operations on the abstract domain, e.g, primitive arithmetic, boolean, or string operations. However, they cannot be used to compose the soundness proof of the analysis of statements, e.g., assignments, loops, or procedure calls. In contrast, the blackboard architecture is capable to compose soundness proofs of both of these types of operations.

Darais et al. [14] developed a theory for soundness proofs, in which the static and dynamic semantics are derived from a *small-step generic interpreter* that describes the operational semantics of the language without mentioning details of static or dynamic semantics. The small-step generic interpreter is instantiated with reusable *Galois transformers* that capture aspects such as flow- or path-sensitivity and allow to change an existing analysis while preserving soundness. Galois transformers can be proven sound once and for all and their soundness proofs are reusable across different analyses. However, the approach does not compose soundness proofs of static semantics derived from the generic interpreter.

Keidel et al. [28] developed a theory for *big-step abstract interpreters*, deriving both the static and dynamic semantics from a generic big-step interpreter. The theory enables soundness composition [28, Theorem 4 and 5] if the generic interpreter is implemented with *arrows* [23] or in a meta-language which enjoys *parametricity*. But there is no theory how parts of soundness proofs can be reused between different analyses. Keidel et al. [27] later refined the theory by introducing reusable *analysis components* that capture different aspects of the language such as values, mutable state, or exceptions and are described with *arrow transformers* [23]. While components can be proven sound independently from each other, their *composition* requires glue code, which needs to be proven sound. Furthermore, the composition creates large arrow transformer stacks – that, unless optimized away by the compiler, may lead to inefficient analysis code. For example, a taint analysis for WebAssembly developed by using the approach depends on a stack of 18 arrow transformers.Eliminating the overhead of an arrow transformer stack of this size requires aggressive inlining and optimizations causing binary bloat and excessive compile times.

Bodin et al. [5] developed a theory of compositional soundness proofs for a style of semantics called *skeletal semantics*, which consists of hooks (recursive calls to the interpreter), filters (tests if variables satisfy a condition), and branches. The dynamic and static semantics are derived from the same *skeleton*. Also, soundness of the instantiated skeleton follows from soundness of the dynamic and static instance [5, Lemma 3.4 and 3.5]. However, their work does not describe how proofs can be reused across different analyses.

To recap, in all theories above the static and dynamic semantics *must* be derived from the same generic interpreter. This restricts what types of analyses can be derived. In particular, the static analysis must closely follow the program execution order dictated by the generic interpreter and it is unclear how static analyses can be derived that do not closely follow the program execution order. For example, backward analyses process programs in reverse order, flow-insensitive analyses may process statements in any order, and summary-based analyses construct summaries in bottom-up order. Our work lifts the restriction that static and dynamic semantics must be derived from the same artifact. static modules and corresponding dynamic modules must follow the blackboard architecture style, but else do not need to share any commonalities. This gives greater freedom as to which types of analyses can be implemented. For example, the blackboard analysis architecture has been used in prior work to develop backward analyses [17], on-demand/lazy analyses [19, 41], and summary-based analyses [21]. We also demonstrated in Section 5.1 that our theory applies to a demand-driven reaching definitions analysis. It is unclear how such an analysis can be derived from a generic interpreter.

6.2 Modular Analysis Architectures

These architectures describe how to implement static analyses modularly. Modular analysis architectures are a necessary requirement to develop theories for compositional and reusable soundness proofs. The theories give formal guarantees about proof independence, composition, and reuse.

Our work formally defines the blackboard analysis architecture used in the *OPAL framework* [15, 21]. In the past, OPAL has been used to implement state-of-the-art analyses for method purity [19], class- and field-immutability [41], and call-graphs [40] for Java Virtual Machine bytecode. Furthermore, OPAL features escape analyses and a solver for IFDS analyses [21] as well as a fixpoint algorithm that parallelizes the analysis execution [20].

Prior to the work presented in this paper, no formalization of the blackboard analysis architecture and no theory for its soundness existed. Our formalization captures the core of the OPAL framework, while deliberately ignoring implementation details. For example, our formalization does not describe the fixpoint algorithm and the order in which it executes static modules to resolve their dependencies. Proving the fixpoint algorithm correct is a separate concern compared to proving analyses sound, which is the focus of our formalization. That said, our formalization covers a variety of OPAL's features described by Helm et al. [21]. For example, OPAL supports *default* and *fallback* properties for missing properties in the store. Fallback properties can be described by our formalization by adding them to the initial store passed to the fixpoint algorithm. We deliberately leave out default properties, which are an edge case in OPAL to mark properties not computed, e.g., because of dead code. They could be added to our formalization by extending analyses with a second set of static modules to be executed after the fixpoint is reached. Furthermore, OPAL supports *optimistic* analyses which ascend the lattice and *pessimistic* analyses which descend the lattice during fixpoint iteration. Both of these are covered by our formalization which describes analyses as monotone functions that ascend or descend the lattice. However, we deliberately do not cover OPAL's mechanisms for allowing interaction between optimistic and pessimistic analyses, another edge case.

Configurable program analysis (CPA) [4] is a modular analysis architecture that describes analyses with transfer relation between control-flow nodes. CPAs can be systematically composed with reduced products. Furthermore, soundness of a component-wise transfer relation follows directly from soundness of its constituents. However, it is unclear how soundness proofs of primitive CPAs can be composed or how proof parts can be reused across analyses.

Doop [7] is a framework which describes analysis with relations in Datalog. Each relation is defined as a set of rules. These rules can be modularly added or replaced, without requiring changes to other rules. While individual analyses in Doop have been proven sound [43], the proofs are not compositional or reusable. In particular, if one rule changes, the proof becomes invalid and needs to be reestablished. This is because the proof reasons about soundness of all rules at once instead of individual rules or relations. The *IncA* framework [45] also describes analyses in Datalog, but allows relations over lattices instead of only sets. However, no soundness theory for its analyses exists. Similar to IncA, the *Flix* framework [37] describes analyses with lattice-based Datalog relations and functions. Flix proves individual functions sound with an automated theorem prover [36]. While an automated theorem prover reduces the proof effort and increases proof trustworthiness, there is no guarantee that the automated theorem prover is able to conduct a proof. Furthermore, the automated theorem prover does not establish a soundness proof of Datalog relations.

Verasco [26] is a modular analysis for *C#minor* [32], an intermediate language used by the *CompCert* C compiler [33]. Verasco is proven sound with the *Coq* proof assistant [3]. The soundness proof of the abstract C#minor semantics is independent of the abstract domain, which makes the proof reusable for other abstract domains. However, the abstract semantics is proven sound w.r.t. the *standard* concrete semantics. Thus, the proof cannot be reused for abstract semantics which approximate *non-standard* concrete semantics, such as information flow analyses [2] or liveness analyses [11].

Several other modular analysis architectures [24, 31, 42] do not have formal theories for soundness.

6.3 Monolithic Soundness Proofs

In this subsection, we compare compositional and reusable soundness proof theories to ad-hoc monolithic proofs and discuss their trade-offs.

Monolithic soundness proofs consider the entire analysis and dynamic semantics as a whole. This complicates the proof because there is no separation of concerns to manage the complexity of modern programming languages. Furthermore, monolithic soundness proofs are harder to maintain. In particular, whenever the analysis needs to be updated to support a new version of the language, or whenever the analysis is fine-tuned to improve precision and scalability, the soundness proof becomes invalid and needs to be reestablished. However, reestablishing the soundness proof is difficult because it is unclear which parts of the proof have become invalid and need to be updated. In contrast, compositional soundness proofs narrow the proofscope to individual modules, which decreases the proofs' complexity. Furthermore, compositional soundness proofs are easier to maintain as changes to individual modules only invalidate their particular soundness proof, while the proofs of other modules remain valid.

The main benefit of monolithic soundness proofs over compositional proofs is that analyses may be proven sound with respect to existing formal dynamic semantics.However, often no suitable formal dynamic semantics exists and analyses still have to be proven sound with respect to customly defined or modified dynamic semantics. For example, *HornDroid* [8] is proven sound with respect to a custom instrumented JVM small-step semantics and *Jaam*[6] is proven sound with respect to a custom JVM semantics in form of an abstract machine [22]. Furthermore, analyses of properties not present in standard language semantics need to be proven sound with respect to instrumented dynamic semantics. For example, a static taint analysis needs to be proven sound with respect to an instrumented dynamic semantics with taint information. In contrast, compositional soundness proofs require a one-time cost of formalizing a modular dynamic semantics for a language. Once this is done, several analyses can be proven sound with respect to this dynamic semantics. Furthermore, the dynamic semantics can be modularly extended to describe new aspects such as taint information.

7 Future Work

In this section, we discuss limitations of our work and how these limitations can be addressed in the future.

First, our soundness theory requires that static analyses and dynamic semantics are described in the blackboard analysis architecture. It is unclear how easily existing analyses and dynamic semantics be adapted to the architecture. In Section 2.2, we showed how existing small-step dynamic semantics can be described as a module and Helm et al. [21] implemented a wide range of static analyses in the architecture. In the future, we want to investigate how other styles of static and dynamic semantics can be adapted to the architecture.

[6] `https://github.com/Ucombinator/jaam`

Second, our soundness theory requires that all static modules are sound. However, in practice static analyses are deliberately unsound due to complicated language features [34]. In the future, we want to investigate how the blackboard analysis architecture can be used to localize unsoundness. Specifically, unsound analysis results could be tagged with the name of the module that produced them. All results derived from unsound results then propagate the tags. This way, it is always clear which results are potentially unsound and which modules caused unsoundness.

Lastly, our work has focused on soundness, i.e., analyses do not produce false-negative results. A complementary property to soundness is completeness, i.e., analyses do not produce false-positives results. No false-positive results are especially important if analyses produce warnings that are to be inspected by developers. In the future, we want to investigate if our theory can be extended to prove completeness of static analyses.

8 Conclusion

In this work, we developed a theory for compositional and reusable soundness proofs for static analyses in the blackboard analysis architecture. The blackboard analysis architecture modularizes the implementation of static analyses with analyses composed of independent static modules. We proved that soundness of an analysis follows directly from independent soundness proofs of each module. Furthermore, we extended our theory to enable the reuse of soundness proofs of existing modules across different analyses. We evaluated our approach by implementing four analyses and proving them sound: A pointer, a call-graph, a reflection, an immutability analysis, and a demand-driven reaching definitions analysis.

9 Data Availability

The implementation of the case studies and proofs are provided as an artifact available at `https://doi.org/10.5281/zenodo.10418484`.

Acknowledgments. This research work has been funded by the German Federal Ministry of Education and Research and the Hessian Ministry of Higher Education, Research, Science and the Arts within their joint support of the National Research Center for Applied Cybersecurity ATHENE.

References

1. Afonso, V.M., de Geus, P.L., Bianchi, A., Fratantonio, Y., Kruegel, C., Vigna, G., Doupé, A., Polino, M.: Going native: Using a large-scale analysis of android apps to create a practical native-code sandboxing policy. In: 23rd Annual Network and Distributed System Security Symposium, NDSS 2016, San Diego, California, USA, February 21-24, 2016. The Internet Society (2016)

2. Assaf, M., Naumann, D.A., Signoles, J., Totel, E., Tronel, F.: Hypercollecting semantics and its application to static analysis of information flow. In: Castagna, G., Gordon, A.D. (eds.) Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages, POPL 2017, Paris, France, January 18-20, 2017. pp. 874–887. ACM (2017). `https://doi.org/10.1145/3009837.3009889`
3. Bertot, Y., Castéran, P.: Interactive Theorem Proving and Program Development - Coq'Art: The Calculus of Inductive Constructions. Texts in Theoretical Computer Science. An EATCS Series, Springer (2004). `https://doi.org/10.1007/978-3-662-07964-5`
4. Beyer, D., Henzinger, T.A., Théoduloz, G.: Configurable software verification: Concretizing the convergence of model checking and program analysis. In: Damm, W., Hermanns, H. (eds.) Computer Aided Verification, 19th International Conference, CAV 2007, Berlin, Germany, July 3-7, 2007, Proceedings. Lecture Notes in Computer Science, vol. 4590, pp. 504–518. Springer (2007). `https://doi.org/10.1007/978-3-540-73368-3_51`
5. Bodin, M., Gardner, P., Jensen, T.P., Schmitt, A.: Skeletal semantics and their interpretations. Proc. ACM Program. Lang. **3**(POPL), 44:1–44:31 (2019). `https://doi.org/10.1145/3290357`
6. Bogdanas, D., Rosu, G.: K-java: A complete semantics of java. In: Rajamani, S.K., Walker, D. (eds.) Proceedings of the 42nd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2015, Mumbai, India, January 15-17, 2015. pp. 445–456. ACM (2015). `https://doi.org/10.1145/2676726.2676982`
7. Bravenboer, M., Smaragdakis, Y.: Strictly declarative specification of sophisticated points-to analyses. In: Arora, S., Leavens, G.T. (eds.) Proceedings of the 24th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2009, October 25-29, 2009, Orlando, Florida, USA. pp. 243–262. ACM (2009). `https://doi.org/10.1145/1640089.1640108`
8. Calzavara, S., Grishchenko, I., Maffei, M.: Horndroid: Practical and sound static analysis of android applications by SMT solving. In: IEEE European Symposium on Security and Privacy, EuroS&P 2016, Saarbrücken, Germany, March 21-24, 2016. pp. 47–62. IEEE (2016). `https://doi.org/10.1109/EuroSP.2016.16`
9. Cortesi, A., Costantini, G., Ferrara, P.: A survey on product operators in abstract interpretation. In: Banerjee, A., Danvy, O., Doh, K., Hatcliff, J. (eds.) Semantics, Abstract Interpretation, and Reasoning about Programs: Essays Dedicated to David A. Schmidt on the Occasion of his Sixtieth Birthday, Manhattan, Kansas, USA, 19-20th September 2013. EPTCS, vol. 129, pp. 325–336 (2013). `https://doi.org/10.4204/EPTCS.129.19`
10. Cousot, P., Cousot, R.: Constructive versions of Tarski's fixed point theorems. Pacific Journal of Mathematics **81**(1), 43–57 (1979). `https://doi.org/10.2140/pjm.1979.82.43`
11. Cousot, P.: Syntactic and semantic soundness of structural dataflow analysis. In: Chang, B.E. (ed.) Static Analysis - 26th International Symposium, SAS 2019, Porto, Portugal, October 8-11, 2019, Proceedings. Lecture Notes in Computer Science, vol. 11822, pp. 96–117. Springer (2019). `https://doi.org/10.1007/978-3-030-32304-2_6`
12. Cousot, P., Cousot, R.: Systematic design of program analysis frameworks. In: Aho, A.V., Zilles, S.N., Rosen, B.K. (eds.) Conference Record of the Sixth Annual ACM Symposium on Principles of Programming Languages, San Antonio, Texas,

USA, January 1979. pp. 269–282. ACM Press (1979). `https://doi.org/10.1145/567752.567778`

13. Cousot, P., Cousot, R., Feret, J., Mauborgne, L., Miné, A., Monniaux, D., Rival, X.: The astreé analyzer. In: Sagiv, S. (ed.) Programming Languages and Systems, 14th European Symposium on Programming,ESOP 2005, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2005, Edinburgh, UK, April 4-8, 2005, Proceedings. Lecture Notes in Computer Science, vol. 3444, pp. 21–30. Springer (2005). `https://doi.org/10.1007/978-3-540-31987-0_3`
14. Darais, D., Might, M., Horn, D.V.: Galois transformers and modular abstract interpreters: reusable metatheory for program analysis. In: Aldrich, J., Eugster, P. (eds.) Proceedings of the 2015 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2015, part of SPLASH 2015, Pittsburgh, PA, USA, October 25-30, 2015. pp. 552–571. ACM (2015). `https://doi.org/10.1145/2814270.2814308`
15. Eichberg, M., Hermann, B.: A software product line for static analyses: the OPAL framework. In: Arzt, S., Santelices, R.A. (eds.) Proceedings of the 3rd ACM SIGPLAN International Workshop on the State Of the Art in Java Program analysis, SOAP 2014, Edinburgh, UK, Co-located with PLDI 2014, June 12, 2014. pp. 2:1–2:6. ACM (2014). `https://doi.org/10.1145/2614628.2614630`
16. Gehr, T., Mirman, M., Drachsler-Cohen, D., Tsankov, P., Chaudhuri, S., Vechev, M.T.: AI2: safety and robustness certification of neural networks with abstract interpretation. In: 2018 IEEE Symposium on Security and Privacy, SP 2018, Proceedings, 21-23 May 2018, San Francisco, California, USA. pp. 3–18. IEEE Computer Society (2018). `https://doi.org/10.1109/SP.2018.00058`
17. Glanz, L., Müller, P., Baumgärtner, L., Reif, M., Amann, S., Anthonysamy, P., Mezini, M.: Hidden in plain sight: Obfuscated strings threatening your privacy. In: Sun, H., Shieh, S., Gu, G., Ateniese, G. (eds.) ASIA CCS '20: The 15th ACM Asia Conference on Computer and Communications Security, Taipei, Taiwan, October 5-9, 2020. pp. 694–707. ACM (2020). `https://doi.org/10.1145/3320269.3384745`
18. Haas, A., Rossberg, A., Schuff, D.L., Titzer, B.L., Holman, M., Gohman, D., Wagner, L., Zakai, A., Bastien, J.F.: Bringing the web up to speed with webassembly. In: Cohen, A., Vechev, M.T. (eds.) Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2017, Barcelona, Spain, June 18-23, 2017. pp. 185–200. ACM (2017). `https://doi.org/10.1145/3062341.3062363`
19. Helm, D., Kübler, F., Eichberg, M., Reif, M., Mezini, M.: A unified lattice model and framework for purity analyses. In: Huchard, M., Kästner, C., Fraser, G. (eds.) Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering, ASE 2018, Montpellier, France, September 3-7, 2018. pp. 340–350. ACM (2018). `https://doi.org/10.1145/3238147.3238226`
20. Helm, D., Kübler, F., Kölzer, J.T., Haller, P., Eichberg, M., Salvaneschi, G., Mezini, M.: A programming model for semi-implicit parallelization of static analyses. In: Khurshid, S., Pasareanu, C.S. (eds.) ISSTA '20: 29th ACM SIGSOFT International Symposium on Software Testing and Analysis, Virtual Event, USA, July 18-22, 2020. pp. 428–439. ACM (2020). `https://doi.org/10.1145/3395363.3397367`
21. Helm, D., Kübler, F., Reif, M., Eichberg, M., Mezini, M.: Modular collaborative program analysis in OPAL. In: Devanbu, P., Cohen, M.B., Zimmermann, T. (eds.) ESEC/FSE '20: 28th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering, Virtual Event,

USA, November 8-13, 2020. pp. 184–196. ACM (2020). https://doi.org/10.1145/3368089.3409765

22. Horn, D.V., Might, M.: Abstracting abstract machines. In: Hudak, P., Weirich, S. (eds.) Proceeding of the 15th ACM SIGPLAN international conference on Functional programming, ICFP 2010, Baltimore, Maryland, USA, September 27-29, 2010. pp. 51–62. ACM (2010). https://doi.org/10.1145/1863543.1863553
23. Hughes, J.: Generalising monads to arrows. Sci. Comput. Program. **37**(1-3), 67–111 (2000). https://doi.org/10.1016/S0167-6423(99)00023-4
24. Johnson, N.P., Fix, J., Beard, S.R., Oh, T., Jablin, T.B., August, D.I.: A collaborative dependence analysis framework. In: Reddi, V.J., Smith, A., Tang, L. (eds.) Proceedings of the 2017 International Symposium on Code Generation and Optimization, CGO 2017, Austin, TX, USA, February 4-8, 2017. pp. 148–159. ACM (2017)
25. Jourdan, J.: Verasco: a Formally Verified C Static Analyzer. (Verasco: un analyseur statique pour C formellement vérifié). Ph.D. thesis, Paris Diderot University, France (2016)
26. Jourdan, J., Laporte, V., Blazy, S., Leroy, X., Pichardie, D.: A formally-verified C static analyzer. In: Rajamani, S.K., Walker, D. (eds.) Proceedings of the 42nd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2015, Mumbai, India, January 15-17, 2015. pp. 247–259. ACM (2015). https://doi.org/10.1145/2676726.2676966
27. Keidel, S., Erdweg, S.: Sound and reusable components for abstract interpretation. Proc. ACM Program. Lang. **3**(OOPSLA), 176:1–176:28 (2019). https://doi.org/10.1145/3360602
28. Keidel, S., Poulsen, C.B., Erdweg, S.: Compositional soundness proofs of abstract interpreters. Proc. ACM Program. Lang. **2**(ICFP), 72:1–72:26 (2018). https://doi.org/10.1145/3236767
29. Kester, D., Mwebesa, M., Bradbury, J.S.: How good is static analysis at finding concurrency bugs? In: Tenth IEEE International Working Conference on Source Code Analysis and Manipulation, SCAM 2010, Timisoara, Romania, 12-13 September 2010. pp. 115–124. IEEE Computer Society (2010). https://doi.org/10.1109/SCAM.2010.26
30. Landman, D., Serebrenik, A., Vinju, J.J.: Challenges for static analysis of java reflection: literature review and empirical study. In: Uchitel, S., Orso, A., Robillard, M.P. (eds.) Proceedings of the 39th International Conference on Software Engineering, ICSE 2017, Buenos Aires, Argentina, May 20-28, 2017. pp. 507–518. IEEE / ACM (2017). https://doi.org/10.1109/ICSE.2017.53
31. Lerner, S., Grove, D., Chambers, C.: Composing dataflow analyses and transformations. In: Launchbury, J., Mitchell, J.C. (eds.) Conference Record of POPL 2002: The 29th SIGPLAN-SIGACT Symposium on Principles of Programming Languages, Portland, OR, USA, January 16-18, 2002. pp. 270–282. ACM (2002). https://doi.org/10.1145/503272.503298
32. Leroy, X.: A formally verified compiler back-end. Journal of Automated Reasoning **43**(4), 363–446 (dec 2009). https://doi.org/10.1007/s10817-009-9155-4, https://doi.org/10.1007/s10817-009-9155-4
33. Leroy, X., Blazy, S., Kästner, D., Schommer, B., Pister, M., Ferdinand, C.: CompCert - A Formally Verified Optimizing Compiler. In: ERTS 2016: Embedded Real Time Software and Systems, 8th European Congress. SEE, Toulouse, France (Jan 2016)

34. Livshits, B., Sridharan, M., Smaragdakis, Y., Lhoták, O., Amaral, J.N., Chang, B.E., Guyer, S.Z., Khedker, U.P., Møller, A., Vardoulakis, D.: In defense of soundiness: a manifesto. Commun. ACM **58**(2), 44–46 (2015). https://doi.org/10.1145/2644805
35. Livshits, V.B., Whaley, J., Lam, M.S.: Reflection analysis for java. In: Yi, K. (ed.) Programming Languages and Systems, Third Asian Symposium, APLAS 2005, Tsukuba, Japan, November 2-5, 2005, Proceedings. Lecture Notes in Computer Science, vol. 3780, pp. 139–160. Springer (2005). https://doi.org/10.1007/11575467_11
36. Madsen, M., Lhoták, O.: Safe and sound program analysis with flix. In: Tip, F., Bodden, E. (eds.) Proceedings of the 27th ACM SIGSOFT International Symposium on Software Testing and Analysis, ISSTA 2018, Amsterdam, The Netherlands, July 16-21, 2018. pp. 38–48. ACM (2018). https://doi.org/10.1145/3213846.3213847
37. Madsen, M., Yee, M., Lhoták, O.: From datalog to flix: a declarative language for fixed points on lattices. In: Krintz, C., Berger, E.D. (eds.) Proceedings of the 37th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2016, Santa Barbara, CA, USA, June 13-17, 2016. pp. 194–208. ACM (2016). https://doi.org/10.1145/2908080.2908096
38. Nielson, F., Nielson, H.R., Hankin, C.: Principles of program analysis. Springer (1999). https://doi.org/10.1007/978-3-662-03811-6
39. Nii, H.P.: Blackboard systems, part one: The blackboard model of problem solving and the evolution of blackboard architectures. AI Mag. **7**(2), 38–53 (1986)
40. Reif, M., Kübler, F., Eichberg, M., Helm, D., Mezini, M.: Judge: identifying, understanding, and evaluating sources of unsoundness in call graphs. In: Zhang, D., Møller, A. (eds.) Proceedings of the 28th ACM SIGSOFT International Symposium on Software Testing and Analysis, ISSTA 2019, Beijing, China, July 15-19, 2019. pp. 251–261. ACM (2019). https://doi.org/10.1145/3293882.3330555
41. Roth, T., Helm, D., Reif, M., Mezini, M.: Cifi: Versatile analysis of class and field immutability. In: 36th IEEE/ACM International Conference on Automated Software Engineering, ASE 2021, Melbourne, Australia, November 15-19, 2021. pp. 979–990. IEEE (2021). https://doi.org/10.1109/ASE51524.2021.9678903
42. Schubert, P.D., Leer, R., Hermann, B., Bodden, E.: Into the woods: Experiences from building a dataflow analysis framework for C/C++. In: 21st IEEE International Working Conference on Source Code Analysis and Manipulation, SCAM 2021, Luxembourg, September 27-28, 2021. pp. 18–23. IEEE (2021). https://doi.org/10.1109/SCAM52516.2021.00011
43. Smaragdakis, Y., Kastrinis, G.: Defensive points-to analysis: Effective soundness via laziness. In: Millstein, T.D. (ed.) 32nd European Conference on Object-Oriented Programming, ECOOP 2018, July 16-21, 2018, Amsterdam, The Netherlands. LIPIcs, vol. 109, pp. 23:1–23:28. Schloss Dagstuhl - Leibniz-Zentrum für Informatik (2018). https://doi.org/10.4230/LIPIcs.ECOOP.2018.23
44. Stein, B., Chang, B.E., Sridharan, M.: Demanded abstract interpretation. In: Freund, S.N., Yahav, E. (eds.) PLDI '21: 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation, Virtual Event, Canada, June 20-25, 2021. pp. 282–295. ACM (2021). https://doi.org/10.1145/3453483.3454044
45. Szabó, T., Bergmann, G., Erdweg, S., Voelter, M.: Incrementalizing lattice-based program analyses in datalog. Proc. ACM Program. Lang. **2**(OOPSLA), 139:1–139:29 (2018). https://doi.org/10.1145/3276509

46. Taneja, J., Liu, Z., Regehr, J.: Testing static analyses for precision and soundness. In: CGO '20: 18th ACM/IEEE International Symposium on Code Generation and Optimization, San Diego, CA, USA, February, 2020. pp. 81–93. ACM (2020). https://doi.org/10.1145/3368826.3377927