On Effect Analysis for Programs with Callbacks

Technical Report EPFL-REPORT-180074

Etienne Kneuss Viktor Kuncak Philippe Suter

School of Computer and Communication Sciences (I&C) - Swiss Federal Institute of Technology (EPFL), Lausanne, Switzerland

{firstname.lastname}@epfl.ch

Abstract

We introduce a precise interprocedural effect analysis for programs with mutable state, dynamic object allocation, and dynamic dispatch. Our analysis is precise even in the presence of dynamic dispatch where the context-insensitive estimate on the number of targets is very large. This feature makes our analysis appropriate for programs that manipulate first-class functions (callbacks). We first present a framework in which programs are enriched with special effect statements, and define the semantics of both program and effect statements as relations on states. Our framework defines a program composition operator that is sound with respect to relation composition. Computing the summary of a procedure then consists of composing all its program statements to produce a single effect statement. We propose a strategy for applying the composition operator in a way that balances precision and efficiency.

We instantiate this framework with a domain for tracking read and write effects, where relations on program states are abstracted as graphs. We implemented the analysis as a plugin for the Scala compiler. We analyzed the Scala standard library containing 58K methods and classified them into several categories according to their effects. Our analysis proves that over one half of all methods are pure. We also analyze how context sensitivity and composition operator application strategies impact the analysis precision and performance.

Categories and Subject Descriptors F.3.1 [*Logics and Meaning of Programs*]: Specifying and Verifying and Reasoning about Programs

General Terms Theory, Verification

Keywords Effect analysis, Static analysis, Callbacks, Scala

1. Introduction

An appealing programming style uses predominantly functional computation steps, with higher-order functions and a disciplined use of side effects. An opportunity for parallel execution further increases the potential of this style. Whereas higher-order functions have always been recognized as a pillar of functional programming, they have also become a standard feature of object-oriented languages such as C# (in form of *delegates*), the 2011 standard of C++, and Java 8.[1] Moreover, design patterns in object-oriented programming community also rely on callbacks, especially the the *strategy pattern* and the *visitor pattern* [13].

Precise analysis of side effects is essential for automated as well as manual reasoning about such programs. The combination of callbacks and mutation makes it difficult to design an analysis that is both scalable enough to handle realistic code bases, and precise enough to handle common patterns such as local side effects and initialization, which arise both from manual programming practice and compilation of higher-level concepts. Among key challenges is flow-sensitivity and precise handling of aliases, as well as precise and scalable handling of method calls.

Our aim is to support not only automated program analyses and transformations that rely on effect information, but also program understanding tasks. We therefore seek to generate readable effects that the developers can compare to their intuition of what methods should and should not affect in program heap. Moreover, we expect our results to help in bootstrap annotations for Scala effect type systems [25] as well as lead to the design of more precise versions of such systems.

This paper presents the design, implementation, and evaluation of a new static analysis for method side effects, which is precise and scalable even in the presence of callbacks, including higher-order functions. Key design aspects of our analysis include:

- a framework for relational analysis that can compute higher-order relational summaries of method calls, which are parameterized by the effects of the methods being called;
- a relational analysis domain that computes summaries of code blocks and methods by flow-sensitively tracking side effects and performing strong updates;
- an automated effect classification and presentation of effect abstractions in terms of regular expressions, facilitating their understanding by developers.

Our static analyzer, called Insane, is publicly available[2]. We have evaluated it on the full Scala standard library, which is widely used by all Scala programs, and is also publicly available. Our analysis works on a relatively low-level intermediate representation that is close to Java bytecodes. Despite this low-level representation, we were able to classify most method calls as not having any observational side effects. Moreover, our analysis also detects conditionally pure methods, for which purity is guaranteed provided that a specified set of subcalls (typically to closures) are pure. We also demonstrate the precision of our analysis on a number of examples that use higher-order functions as control structures. We are not aware of any other fully automated static analyzer that achieves this precision while maintaining reasonable performance.

2. Overview of Challenges and Solutions

In this section, we present some of the challenges that arise from analyzing programs written in a higher-order style, and how Insane can tackle them.

[1] See JSR 335 "Project Lambda" `http://www.jcp.org/en/jsr/detail?id=335`.

[2] Insane stands for "INterprocedural Static ANalysis of Effects" and is available at `https://github.com/colder/insane`

Many static analyses compute approximations of sets of states at each program point. However, by definition, an effect analysis needs to describe changes to program state, as opposed to sets of program states. As a result, it needs to be a relational analysis. A key complication is that in languages with mutation and dynamic allocation, there is no simple static way to refer to objects in the heap. Finally, it is widely understood that analysis in the presence of aliasing may give different results depending on the initial aliasing configuration. It is also essential that a method summary is not characterized by too specific precondition information, because this would limit its reusability. For example, if the analysis does not write to certain parts of the heap (if e.g. it accesses only certain bounded part in the middle of a tree or graph structure), it should not be necessary for the analysis to be aware of those parts of the heap. These challenges make precise effect analysis unique compared to most other abstract interpretation techniques, including many shape analyses. Fortunately, they have been addressed to a large extent by Salcianu, Rinard and Whaley [26, 31], whose work this paper extends. The remaining challenge lies in overcoming imprecision arising from highly dynamic method invocations, such as the ones arising from first-class function invocations.

Effect attribution. Specific to higher-order programs is the problem of correctly attributing heap effects. Consider a simple (first-order) function:

def inc(c : Cell) = { c.value = c.value.next() }

Any reasonable analysis for effects would detect that inc potentially alters the heap, as it contains a statement that writes to a field of an allocated object. That effect could informally be summarized as "*inc may modify the field .value of its first argument*". That information could in turn be retrieved whenever inc is used. Consider now the function

def apply(c : Cell, f : Cell⇒Unit) = { f(c) }

where Cell⇒Unit denotes the type of a function that takes a Cell as argument and returns no value. What is the effect of apply on the heap? Surely, apply potentially has all the effects that inc has, since the call apply(c, inc) is equivalent to inc(c). It can also potentially have no effect on the heap at all, e.g. if invoked as apply(c, (cell ⇒ ())). The situation can also be much worse, for instance in the presence of global objects that may be modified by f. In fact, in the absence of a dedicated technique, the only sound approximation of the effect of apply is to state that it can have any effect. This approximation is of course useless, both from the perspective of a programmer, who doesn't gain any insight on the behaviour of apply, and in the context of a broader program analysis, where the effect cannot be reused modularly.

The solution we propose in this paper is, intuitively, to define the effect of apply to be "exactly the effect on calling its second argument with its first as a parameter". To support this, we extend the notion of effect to be expressive enough to represent *control-flow graphs* where edges can themselves be effects (see Section 3).

Compilation artifacts. Any theoretical framework, when faced with the mundane task of analyzing real-world programs, will face unforeseen hurdles. Somewhat ironically, many of these practical challenges arise from the compilation phase, as the compiler needs to transform functional aspects of the code into objects and methods suitable for interpretation by the virtual machine. Typically, a clean model of side-effect free first-class functions can get translated into a score of anonymous classes passing around mutable references representing the captured environment. An effect analysis, to be useful, should then at least revert these compiling artifacts. Other potential sources of misdirections include automatically generated getters and setters, and initialization code (constructors), which may introduce an undesired level of indirection in effects or irrelevant write effects respectively. To tackle these issues, we have designed an analysis that supports *strong updates*, allowing us to reverse many important compilation artifacts (see Section 4). This turned out to be crucial to allow us to analyze the entire Scala standard library, on which we report in Section 6.

Statement	Meaning
v = w	assign w to v
v = o.f	read field o.f into v
o.f = v	update field o.f with v
v = **new** C	allocates an object of class C
v = o.m(a1, ..., an)	calls method m of object o

Figure 1. Program statements considered in the target language.

Making sense of effects. The last challenge we address in this paper is one of presentation: when a function is provably pure, this can be reported straightforwardly to the programmer. When however it can have effects on the heap, the pure/impure dichotomy falls short. Consider a function that updates all (mutable) elements stored in a linked list. That function has an effect, but a summary stating only that it is impure is highly unsatisfactory: crucially, it does not give any indication to the programmer that the list itself is not affected by the writes. As we will see, the internal representation of effects, while suited to a compositional analysis, is impractical for humans, not the least because it is non-textual. We propose to bridge the representation gap by using an additional abstraction of effects in the form of regular expressions that describe sets of fields potentially affected by effects (see Section 5). This abstraction captures less information than the internal representation but can readily represent complex effect scenarios, as our evaluation on Scala collections illustrates (Section 6.3). For the example given above, the regular expression displayed to the programmer could be:

$$\texttt{list}(\texttt{.next})^*\texttt{.elem.value}$$

This indicates that the fields affected are those *reachable* through the list (by following `.next`), but belonging to elements only, thus conveying the desired information.

3. A Compositional Analysis Framework

In this section, we present a framework for interprocedural analysis that allows trading off between precision and efficiency. We start by describing a target language that is expressive enough to encode most features of Scala. (Section 4.3 describes how we handled a few extra features in practice.)

3.1 Target Language

The language we target is a typical object oriented language with dynamic dispatch. A program is made of a set of classes $\mathcal{C}$ which implement methods. We identify methods uniquely by using the method name prefixed with its declaring class: $C.m \in \mathcal{M}$. We assume without loss of generality that the language does not support method overloading –affected methods can always be renamed. We assume that for each method, a standard control-flow graph is available, where edges are labeled with simple program statements. Each of these graphs contains a source node *entry*, and a sink node *exit*. Figure 1 lists the statements we consider along with their meaning.

Because of dynamic dispatch, a call statement can target multiple methods, depending on the runtime value of the receiver. For each method call $o.m()$, we can compute a set of targets $\text{targets}(o.m) \subseteq \mathcal{M} \cup \{?\}$ using the static type of the receiver. If the hierarchy is not bounded as it can be with final classes or methods, we also include the special "?" target to represent the arbitrary methods that could be defined in unknown extensions of the program. Indeed, we do not always assume that we have access to the entire program: this assumption is defined as a parameter of the analysis, and we will see later how it affects it.

Our framework is applicable to abstract domains R that represent relations between program states. We thus have that $\gamma : R \to 2^{S \times S}$. Such abstract domains have for example been shown to elegantly describe memory effects as abstract heap transformers. We assume the existence of a *composition operator* $\diamond : R \times R \to R$ for elements of the abstract domain, with the following property:

$$\forall e, f \in R \,.\, (\gamma(e) \circ \gamma(f)) \subseteq \gamma(e \diamond f)$$

that is

$$\forall s_0, s_1, s_2 \,.\, s_1 \in \gamma(e)(s_0) \wedge s_2 \in \gamma(f)(s_1) \implies s_2 \in \gamma(e \diamond f)(s_0)$$

In other words, $\diamond$ must compose abstract relations in such a way that the result is a valid approximation of the corresponding composition in the concrete domain.

We define $\text{analyze} : CFG \to R$ as a procedure performing the analysis on the provided CFG and returning the facts at the *exit* node once a fix-point is reached.

3.2 Control-Flow Graph Reductions

In order to support reductions, we first augment the language with a special summary statement: $\mathsf{Stmt}_{ext} = \mathsf{Stmt} \cup \{\mathsf{Smr}\}$ that is characterized by an abstract value: $\mathsf{Smr}(e \in \mathsf{R})$. Assuming a transfer function $\mathrm{T_f} : \mathsf{Stmt} \to \mathsf{R} \to \mathsf{R}$ is defined, we extend it for Stmt_{ext} by defining:

$$\mathrm{T_f}_{ext}(s)(p) = \begin{cases} \mathrm{T_f}(s)(p) & \text{if } s \in \mathsf{Stmt} \\ p \diamond e & \text{if } s = \mathrm{Smr}(e) \end{cases}$$

Let c be the CFG of some procedure, and a and b two nodes of c such that a strictly dominates b and b post-dominates a. In such a situation, all paths from entry to b go through a and all paths from a to exit go through b. Let us consider the sub-graph between a and b, which we denote by $a \subset b$. This graph can be viewed as a control-flow graph with a as its source and b as its sink. The reduction operation consists of replacing $a \subset b$ by a single edge labelled with a summary statement. We will refer to this transformation by $\text{reduce} : \mathsf{CFG} \times \mathsf{Node} \times \mathsf{Node} \to \mathsf{CFG}$. The summary statement is obtained by analyzing the CFG $a \subset b$ in isolation.

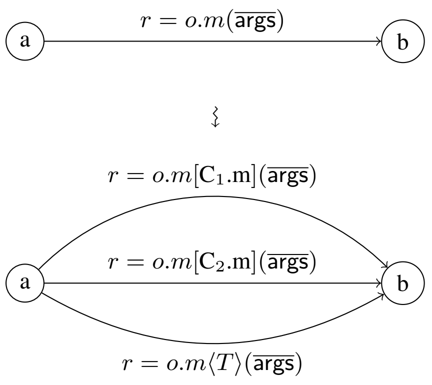
We observe that while $\circ$ over the concrete domain is associative, it is generally not the case for $\diamond$. Moreover, different orders of applications yield incomparable results. In fact, the order in which the reductions are performed plays an important role in the overall result. When possible, left-associativity is preferred as it better encapsulates a forward analysis and can leverage past information.

3.3 Multi-Target Inlining

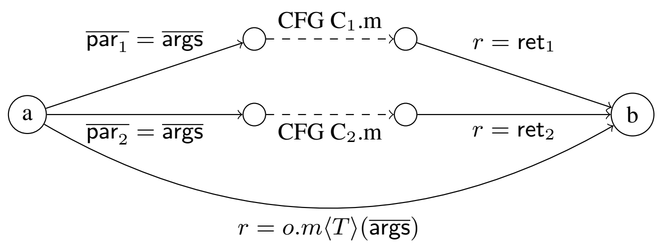
Control-flow graph reductions, presented above, are one of the building blocks of our compositional framework. The other one is a mechanism for replacing method invocations by summaries, or *inlining*, which we present here.

In our framework, inlining consists of replacing a call statement by a summary for the corresponding code. The first step is to identify the possible targets of the method call. Given that this set is potentially unbounded because of unknown descendants in the class hierarchy, one common approach is to assume that the entire program is known (a *closed world* assumption). We leave that assumption for now as a parameter to the analysis and adopt instead an approach that relies on partial inlining: we associate the decision of inlining a method call with a set of targets $T \subseteq \text{targets}(o.m)$ for which summaries are available. The special "?" target, having no implementation, can never be selected for inlining.

To ensure the validity of the partial inlining operation, we first apply a semantic-preserving rewrite: we split the call edge, thus making it explicit which fully determined (non-virtual) calls should be inlined, and record the set of inlined targets in the original edge. This intuitively amounts to annotating call edges with the targets that have already been inlined. For instance, for a set of targets $T = \{C_1.m, C_2.m\}$, we get


$r = o.m(\overline{\mathsf{args}})$
a
b
$r = o.m[\mathrm{C_1.m}](\overline{\mathsf{args}})$
$r = o.m[\mathrm{C_2.m}](\overline{\mathsf{args}})$
a
b
$r = o.m\langle T\rangle(\overline{\mathsf{args}})$

For an annotated call edge $r = o.m\langle T\rangle(\overline{\mathsf{args}})$, we can compute its targets: $\text{targets}(o.m) \setminus T$. The fully determined call edges are here displayed for explanation purposes. In practice, we immediately replace them with their corresponding CFGs, properly versioned and surrounded with appropriate assign statements for the arguments and return values:


$\overline{\mathsf{par}_1} = \overline{\mathsf{args}}$
CFG $\mathrm{C_1}$.m
$r = \mathsf{ret}_1$
a
b
$\overline{\mathsf{par}_2} = \overline{\mathsf{args}}$
CFG $\mathrm{C_2}$.m
$r = \mathsf{ret}_2$
$r = o.m\langle T\rangle(\overline{\mathsf{args}})$

If it is available, we can rely on abstract information on o to refine the number of potential targets. By keeping the call statement and recording inlined targets, we can perform this inlining operation before we reach a fix-point. We argue this by making two observations:

- In the absence of precise information on the runtime type of o, we may have to inline too many targets. Being an over-approximation, it remains sound.
- If during the abstract interpretation the facts at a increase in a way that include new potential targets for o.m, they will be included semantically in the alternative edge and become candidates for inlining. Consequently, the partial inlining does not prevent the discovery of other potential targets at a later stage.

In certain situations, we can conclude that we have inlined all potential targets. In such cases, the alternative call edge becomes infeasible and can be removed. This can happen either because the exact type is known for the receiver, if some methods or classes are known to be final, or we parametrize the analysis to assume a closed world.

3.4 Combining Inlining and Reductions

We distinguish between two main kinds of summaries. A summary that contains unanalyzed method calls is said to be *conditional*.

```
class A {                          // .. continuing class A
  def m1() {
    val o = new A;                   def m3() { }
    this.m2(o)
  }                                  def f() { }
                                   }
  def m2(o: A) {
    this.m3()                      class B extends A {
    o.f()                            override def f() { .. }
  }                                }
```

Figure 2. Example of a chain of method calls.

In contrast, a *definite* summary is fully reduced, thus containing a single edge with a summary statement.

There are in general multiple ways to generate a definite summary from a CFG, depending on the interleaving of reduction and inlining operations. To facilitate the description of such interleavings, we introduce the following notations:

$$\begin{aligned} \lfloor c \rfloor_{\mathsf{stmts}} &:= \text{CFG } c \text{ reduced around statements stmts} \\ \lfloor c \rfloor_{\emptyset} &:= \text{reduce}(c, c.\mathsf{entry}, c.\mathsf{exit}) \\ c_1 \overset{\mathsf{call}}{\prec} \{c_2\} &:= \text{CFG } c_1 \text{ in which call is inlined with } c_2 \end{aligned}$$

The CFG $\lfloor c \rfloor_{\{s_1, s_2\}}$ represents a reduction "down to" the method calls s_1 or s_2. In other words, it represents the result of applying reduction operations on c until the CFG contains only edges with either summary statements or the specific call statements s_1 or s_2. Consequently, we have that $\lfloor c \rfloor_{\emptyset}$ is the CFG with one unique summary statement obtained after completely reducing c. The inlining operator is defined as follows: $c_0 \overset{o.c}{\prec} \{c_1, c_2\}$ represents the CFG c_0 in which the call statement $o.c$ has been inlined using the CFGs c_1 and c_2 (which corresponds to the targets of $o.c$ inferred given c_0).

We now illustrate the flexibility provided by our framework through a simple example. The code for the example is displayed in Figure 2.

Using the notation introduced above, we can describe several interleavings. For instance, one way to generate a summary for $A.m1$ would consist of the following steps: first, we fully reduce $A.m3$, $A.e$ and $B.e$, we inline them in $A.m2$, reduce the result, inline it in $A.m1$ and finally reduce it. This would represent a completely modular approach, where summaries are reused as much as possible. The corresponding inlining/reduction interleaving is:

$$\begin{aligned} t_1 &= \lfloor \mathsf{A.m3} \rfloor_{\emptyset} \\ t_2 &= \lfloor \mathsf{A.f} \rfloor_{\emptyset} \\ t_3 &= \lfloor \mathsf{B.f} \rfloor_{\emptyset} \\ t_4 &= \lfloor (\mathsf{A.m2} \overset{\mathsf{o.f}}{\prec} \{t_2, t_3\}) \overset{\mathsf{this.m3}}{\prec} \{t_1\} \rfloor_{\emptyset} \\ s_1 &= \lfloor \mathsf{A.m1} \overset{\mathsf{this.m2}}{\prec} \{t_4\} \rfloor_{\emptyset} \end{aligned}$$

While being perhaps the most efficient way to compute a summary (since intermediate summaries $t_1 ... t_4$ are small, definite effects) it is also the least precise. Indeed, in this order, we have no precise information on o at the time of inlining o.f() and thus we have to consider every static target, leading to an imprecise summary, depicted here by having to inline both t_2 and t_3. In contrast, we can instead perform the operations in the following order:

$$\begin{aligned} t_1 &= \mathsf{A.m1} \overset{\mathsf{this.m2}}{\prec} \{\mathsf{A.m2}\} \\ t_2 &= t_1 \overset{\mathsf{o.f}}{\prec} \{\mathsf{A.m2}\} \\ s_2 &= \lfloor (t_2 \overset{\mathsf{this.m3}}{\prec} \{\mathsf{A.m3}\}) \overset{\mathsf{o.f}}{\prec} \{\mathsf{A.f}\} \rfloor_{\emptyset} \end{aligned}$$

Here s_2 is a fully precise summary of A.m1: at the time of inlining the o.f() call we already have precise information on o. On the other hand, this order forces us to deal with larger CFGs, which can have an impact on performance. We can imagine a third order, in which A.m2 is partially reduced early on:

$$\begin{aligned} t_1 &= \lfloor \mathsf{A.m3} \rfloor_{\emptyset} \\ t_2 &= \lfloor \mathsf{A.m2} \overset{\mathsf{this.m3}}{\prec} \{t_1\} \rfloor_{\{\mathsf{o.f}\}} \\ t_3 &= \mathsf{A.m1} \overset{\mathsf{this.m2}}{\prec} \{t_2\} \\ t_4 &= \lfloor \mathsf{A.f} \rfloor_{\emptyset} \\ s_3 &= \lfloor t_3 \overset{\mathsf{o.f}}{\prec} \{t_4\} \rfloor_{\emptyset} \end{aligned}$$

Depending on the size of A.m3, s_3 may be much more efficient to compute than s_2 due to the early partial reduction. In this case, the partial reduction does not alter the precision of the resulting effect. We can see from this example that it will often be beneficial to delay the analysis of some method calls, and handle others immediately.

3.5 Heuristic Based Delaying

We have seen through the examples above than choosing when to inline a method summary can be difficult. We refer to the opposite of inlining as *delaying*. The decision to delay or to inline is typically based on a heuristic function h. Formally:

$$\begin{aligned} t_{i+1} &= \begin{cases} t_i \overset{c}{\prec} \text{smr}_c(c) & \text{if } \exists c \,.\, \neg\, \text{h}(c, ...) \\ t_i & \text{otherwise} \end{cases} \\ \text{smr}_c(c) &= \{\text{smr}_m(m) \mid m \in \text{targets}(c)\} \\ \text{smr}_m(m) &= \lfloor t_{fix} \rfloor_{\{c \ \mid \ \text{h}(c,...)\}} \end{aligned}$$

The precision and performance of the analysis are thus parametrized in $\text{h}(\ldots)$. Fixing $\text{h}(\ldots) = \mathsf{true}$ for instance forces the analysis to delay every method call, leading to the analysis of the complete CFG at the entry point. This results in a slow, but fully precise analysis. On the other hand, having $\text{h}(\ldots) = \mathsf{false}$ ensures that every method is analyzed modularly, in a top-down fashion, leading to an efficient but imprecise analysis.

3.6 Handling Recursion

We note however that the overall analysis needs to terminate. Assuming the underlying abstract interpretation-based analysis does terminate, we still need to ensure that the CFG does not keep changing, in other words, t_{fix} must eventually be reached. For this reason, we need to force $\text{h}(\ldots) = \mathsf{false}$ whenever the analysis is within a cycle in the call-graph.

Detecting recursion statically is non-trivial, especially in the presence of callbacks. An attempt using a refined version of a class analysis proved to be overly imprecise, flagging every higher-order functions as recursive. For this reason, Insane discovers recursive methods lazily during the analysis when closing a loop in the progressively constructed call-graph. It then rewinds the analysis until the beginning of the loop in the lasso-shaped call-graph in order to handle the cycle safely.

It is worth noting that while the heuristic h is required to be true *within* the cycle, we are free to decide to delay when located at the boundaries of a cycle. It is in general critical for precision purposes to delay the analysis of the entire cycle as much as possible. When analyzing a recursive cycle, we apply a standard fix-point iterative process by setting all the summaries to $\perp$ and building up effects until convergence.

4. An Abstract Domain for Effect Analysis

In this section, we describe the relation abstract domain with which we instantiate the framework described in the previous section.

We use as a basis the modular, graph-based representation of effects introduced in [26], which we refer to as the WSR analysis. We interpret these graphs as heap transformers, similarly to what was presented in [17]. Adopting this view, these graphs fit the description of an abstract domain that maps to relations in the concrete domain. We refine this model to support strong updates and extend it to be expressive enough to represent unbound local variables.

Graphs are composed of nodes generally representing memory locations. We distinguish three kinds of nodes: *inside* nodes are allocated objects. Since we use the allocation-site abstraction for these, we associate them with a flag indicating whether the node is a singleton or a summary node. *External* or *load* nodes represent unknown fields. The general semantic of load nodes is that they represent accesses to unknown parts of the heap. Finally, we have nodes for unknown *local variables*, similar to the parameter nodes introduced in the WSR analysis, but generalized to arbitrary local variables. We also define two types of edges, both labelled with a field. *Write* edges represented by a plain (solid) edge, and *read* edges represented by dashed edges in the graph. Read edges provide an access path to past or intermediate values of fields, and are used for the resolution of *load* nodes. One of the main improvements over the WSR analysis is that our *write* edges represent *must-write* modifications as opposed to *may-write*. We describe below how it alters the original analysis and argue that it increases precision while remaining sound. Along with the graph, we also keep a mapping from local variables to sets of nodes.

A graph transformer is thus represented by the tuple

$$\langle N_i, N_l, N_v, E_r, E_w, \pi \rangle$$

where $N_i \subseteq \mathsf{V}$ is the set of inside nodes, $N_i \subseteq \mathsf{V}$ is the set of load nodes and $N_v \subseteq \mathsf{V}$ is the set of local variable nodes. We define $N = N_i \cup N_l \cup N_v$ to be the set of all nodes. $E_r \subseteq (N \times \mathsf{Field} \times N)$ is the set of read edges, $E_w \subseteq (N \times \mathsf{Field} \times N)$ is the set of write edges. Similarly, we have $E = E_w \cup E_r$. π encodes the mapping between local variables and their value in terms of nodes: $\pi \in Vars \mapsto 2^N$. We have the property that every local variable is mapped to at least one node in π.

4.1 Abstract Semantics

We now describe how we compute method summaries $[\![\mathsf{C.m}]\!]_a$, and show that the abstract summaries are valid over-approximations of the concrete semantics of the method:

$$\gamma([\![\mathsf{C.m}]\!]_a) \sqsupseteq_c [\![\mathsf{C.m}]\!]_c$$

We start by providing semantics to simple statements of our extended language with summary statements, as depicted in Figure 3. In order to simplify their specification, we define the following auxiliary functions:

Statement	Abstract Semantics
v = w	$\langle N_i, N_l, N_v, E_r, E_w, \pi[n \mapsto \pi(w)] \rangle$
ℓ: v = **new** C	$\text{alloc}(i, v, C, \ell)$
ℓ: v = o.f	$\text{read}(i, v, o, f, \ell)$
ℓ: o.f = v	$\text{write}(i, \pi(o), f, \pi(v), \mathsf{true}, \ell)$
[branch on v1..]	$\text{branch}(i, v1, ..)$
ℓ: v = o.m(a1, ..., an)	$\text{call}(i, v, o, m, (a1, ..., an), \ell)$
ℓ: Smr(e)	$\text{compose}(i, e, \ell)$

Figure 3. Extended program statements and their associated abstract semantics.

$$\begin{aligned}
\text{targ}(n, f, E) &:= \{m \mid n \xrightarrow{f} m \in E\} \\
\text{wTarg}(n, f) &:= \text{targ}(n, f, E_w) \\
\text{rTarg}(n, f) &:= \text{targ}(n, f, E_r) \\
\text{allTarg}(n, f) &:= \text{targ}(n, f, E_w \cup E_r) \\
\text{coalesce}(s_1, s_2) &:= \begin{cases} s_1 & \text{if } s_1 \neq \emptyset \\ s_2 & \text{if } s_1 = \emptyset \end{cases} \\
\text{wTrTarg}(n, f) &:= \text{coalesce}(\text{wTarg}(n, f), \\
&\qquad \text{rTarg}(n, f))
\end{aligned}$$

We provide the abstract semantics of each statement by specifying its transfer function $o = T_f(s)(i)$, where i is the tuple corresponding to the incoming abstract value.

$$i := \langle N_i, N_l, N_v, E_r, E_w, \pi \rangle$$

We provide mostly informal descriptions of the mechanics of each of the transfer functions.

Formal algorithmic descriptions can be found in appendix.

4.1.1 Lattice Join

We start by describing the join ($\sqcup$) operation, used notably when combining the effects of two or more branches. In essence, the join takes the union of all edges and nodes from incoming effects. One notable exception is with respect to writes. In case a write operation is performed only on one of the effects, and the write is made from a node that is accessible in both, the update must become weak. For instance, if the write is performed on an inside node that is only allocated in one branch, the update can remain strong. On the other hand, if the update is made to a load node and not in all branches, the update becomes weak. Load nodes are introduced when needed to represent past values.

Figure 4 shows an example of this classification of different write operations. We join two effects both containing strong updates. The strong update on a.f in G_1 remains strong, since I_1 is only accessible in G_1. On the other hand, Given that L_v may be accessible in both, the update of b.g becomes weak. We also see that a load node is introduced to represent the old value of b.g.

4.1.2 Object Allocation

The transfer function for object allocation creates an inside node and flags it as singleton by default. Depending on whether this node already exists in the graph –an indication that at least two instances from the same allocation site may co-exist, we replace that singleton node with a summary node. We illustrate the necessity of this simple recency abstraction with the following example:

```
     def f(i: Int) {
ℓ1 :   var o = new C;
       while(i < 42) {
```

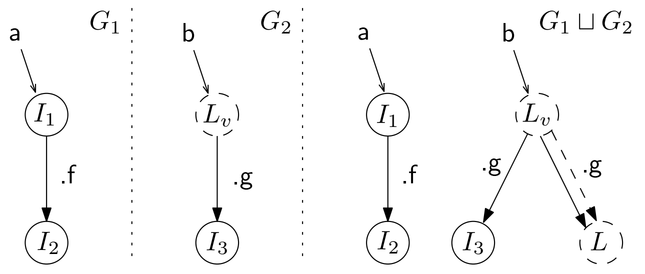


Figure 4. Example of join operation on two effects containing strong updates. Based on the context, strong updates become weak when joining.

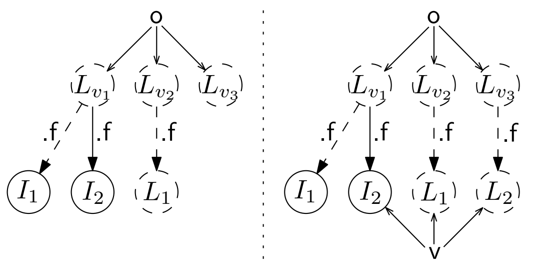


Figure 5. Graph on the right is the result of applying v = o.f to the graph on the left. When possible, reading uses write edges as they represent more recent values. When nothing is available, load nodes are introduced and associated to the label ℓ.

```
ℓ2 :      o = new C;
        }
      }
```

The object allocated at ℓ_1 is never contained in the incoming facts, and thus remains a singleton node. For ℓ_2, it is first created as a singleton node during the first loop iteration. In the second loop iteration, the corresponding node is found in the incoming facts, and is replaced by a summary node. At the end of the function, o either points to the singleton node from ℓ_1 or to the summary node from ℓ_2.

4.1.3 Reading Fields

In order to handle the statement v = o.f, we look for existing targets starting from every node representing the value of o. When looking for targets of a certain field, we privilege write edges over read edges, as they point by construction to more recent values. In case no write edge exists, we look through read edges for a potential value. In the absence of read edges, we introduce a load node to represent this unknown value. Finally, we modify π so that v points to the set of discovered targets. Figure 5 shows an example of the modifications performed to handle a read statement, providing an example of all the access scenarios explained above.

4.1.4 Writing Fields

One of the main improvements of our analysis compared to previous iterations is its optimistic (and sound) handling of strong updates. The key observation here is that an update may often be considered as strong even if the set of objects to which it is applied is unknown at the time. Basically, it considers an update as strong as long as possible until it is forced to be weak. Additionally, even though the updates may end up being weak when applied in a concrete context, they are often locally strong, allowing us to discard intermediate values and simplify effect summaries.

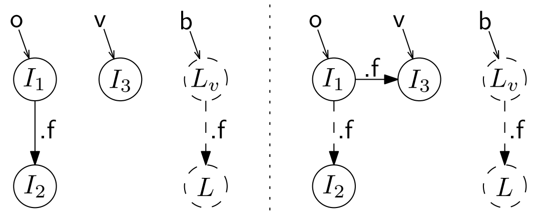


Figure 6. Applying the effect of o.f = v in a context that permits strong updates. However, the old value cannot be entirely discarded since an existing load node might be referencing it, depending on aliasing.

We decompose the transfer function for o.f = v into multiple cases, depending on the environment. First, we check if the update can be strong, this is given by

$$\mathsf{strong} = \pi(o) = \{n\} \wedge \neg\, \mathrm{isSummary}(n)$$

where isSummary() returns true for inside nodes that are not singleton nodes. It is worth noting that we also allow strong updates on load nodes.

Strong updates. In case an update can be strong, we check whether its old value should still be reachable via read edges, this is given by

$$\mathsf{keep} = \exists n \in LN \;.\; n.\mathsf{field} = f$$

In such cases, the intermediate values of a field need to be reachable via read edges in order to ensure that the existing load nodes can be resolved correctly. The following code example illustrates this necessity:

```
      def f(o1: A, o2: A) {
ℓ1 :    o1.f = v1;
ℓ2 :    o1.f = v2;
ℓ3 :    val tmp = o2.f;
ℓ4 :    o1.f = v4;
        // ...
      }
```

In ℓ_2, the old value v2 completely overwrites v1, and v1 is no longer reachable via o.f. However, when updating o1.f in ℓ_4, a load node on the same field exists from ℓ_3, potentially referring to o1.f if o1 and o2 turn out to be aliases. For this reason, the write at ℓ_4 need to make sure that the intermediate value v2 is reachable via o1.f using read edges, allowing the load node to be resolved to that intermediate value later on. It is worth noting though that only v4 is reachable via write edges from o1, indicating that it is the only value that may remain assigned to o1.f after ℓ_4.

An example of a strong update for which the old value must be preserved is illustrated in Figure 6. We can see that the conditions for a strong update are met. Also, we notice that the load node L might alias I_2 depending on aliasing between L_v and I_1. For this reason, I_2 must be still be reachable via I_1. This allows L to be resolved correctly in case L_v and I_1 turn out to be aliases later on. However, the update remain strong. As a result, o.f points exclusively to I_3 after this update.

Weak updates. The handling of weak updates is more complicated, as it needs to keep the old value around even if it is unknown. This is done by introducing a load node when necessary. We first follow write edges and then read edges to find old values. It is worth noticing that, as expected, previous values cannot be overwritten when performing a weak update.

Figure 7 gives an example of a situation where a weak update is inevitable. Load node L is introduced to represent the old value of the field f on L_v.

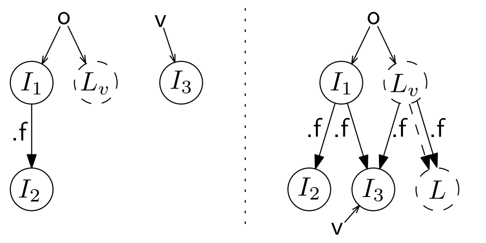


Figure 7. Applying the effect of o.f = v in a context that does not permit strong updates. Load nodes representing the old values are introduced when needed.

4.1.5 Handling Branches

We can handle multiple kinds of branches precisely. This allows us to filter effects depending on the conditions, or even rule out unfeasible paths. The main categories of branches we handle are type checks and reference equalities.

Type checks. Type checks are commonly used by pattern matching, an ubiquitous feature of Scala. It is in practice important to handle type checks precisely, allowing us to rule out impossible effects early. When performing type checks, we not only check whether the path is feasible or not, but also filter the nodes accordingly. The statements we handle here are of the kind: o.isInstanceOf[T]. The transfer function filters $\pi(o)$ and retain only nodes which type is compatible with T, that is

$$r = \{n \mid n \in \pi(o) \wedge type(n) \sqcap T \neq \bot\}$$

where $\sqcap$ is the greatest lower bound of the two types. If $r = \emptyset$, the branch is considered to be impossible, the resulting effect is $\bot$. Otherwise, we update π to $\pi[o \rightarrow r]$.

Reference equalities. Insane also handles equality constraints on branches, such as v1 = v2 or v1 != v2. Here again we check whether, based on the types of the nodes of v1 and v2, the branch is feasible. If all the nodes of v1 are incompatible with all the nodes of v2, the branch becomes impossible, and consequently $\bot$ is returned.

4.1.6 Composing Summaries

Composing effects is one of the central feature of this analysis as it is required both by the interprocedural aspect of the analysis, and to handle summary statements, generated by the reductions in the analysis framework. Effect composition $G_1 \diamond G_2$ is handled by merging the graph G_2 within G_1. We thus identify G_2 as the inner graph, and G_1 as the outer graph.

When merging graph, we also provide an initial map identifying an equivalence relation between nodes from the inner graph and nodes from the outer graph. This map, initially incomplete, expands during the merging process. The procedure can be summarized informally by Algorithm 1.

Algorithm 1 Composing Summaries

```
1: function COMPOSE($i, o, m, \ell$)
2:     $r \leftarrow o$
3:     importInsideNodes()
4:     repeat
5:         resolveLoadNodes()
6:         applyWrites()
7:     until $r$ did not change
8:     return $r$
9: end function
```

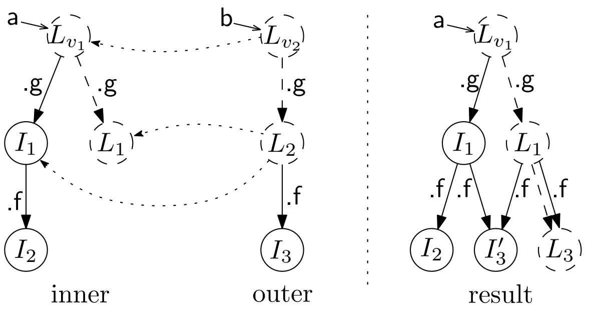


Figure 8. Merging a graph with load nodes and strong updates in a context that does not permit a strong update. Inside nodes are imported after refining their label. As usual, the load nodes representing old values for weak updates are introduced when necessary.

Importing inside nodes. In order to import an inside node node from the inner graph, we first refine the label representing its allocation site. We do this by composing its label with the label representing the point at which we compose the effects. This operator allows the composition of distinct labels in order to distinguish the same node being inlined at distinct places. This allows for instance to precisely handle multiple calls to a factory method.

We define composition operator over labels: $\oplus$. We in fact interpret labels as multisets and define $\oplus$ as $\uplus$ with bounded multiplicity as follows:

$$\ell_1 \oplus \ell_2 = l_p \ s.t. \ \forall x \in \ell_1 \cup \ell_2 \ . \ l_p(x) = \min(\ell_1(x) + \ell_2(x), M)$$

M is given as a parameter of the analysis. Having $M > 1$ allows to refine the analysis of certain loops and recursive methods. It however has exponential cost. Note that when $M = 1$, our labels become simple sets and $\oplus$ becomes $\cup$. When $M = 0$, we define $\oplus$ as

$$\ell_1 \oplus \ell_2 = \ell_1$$

It looks in the outer graph if a similar node already exists, in which case it makes sure that it becomes a summary node. This is similar to what is done in order to handle object allocation.

Once the refined label is determined, we check that we do not import a singleton node in an environment in which it already exists. In such case, the node is imported as a summary node. This mimics the checks performed by the transfer function for object allocation.

Resolving load nodes. Another important operation when merging two graph is the resolution of load nodes from the inner graph to nodes in the outer graph. The procedure works as follows: for each load node we look at all its source nodes, by following read edges in the opposite direction. Note that the source node of a load node might be a load node itself. In this case, we recursively invoke the resolution operation. We then obtain all the nodes in the outer graph corresponding to the source nodes.

The resolution follows by performing a read operation from the corresponding source nodes in the outer graph. As described in Section 4.1.3, reading may create additional load nodes, and returns the set of targets. Once a load node is resolved to a set of nodes in the outer graph, the map is updated to reflect this.

Applying write effects. Given the map obtained by resolving load nodes, we apply write edges found in the inner graph to corresponding edges in the outer graph. Again, we need to make sure that a strong update in the inner graph can remain strong, given the outer graph and the map.

4.1.7 Method Calls

In order to analyze a method call, we need to distinguish whether we are within a cycle in a call-graph, or not. Either way, we start by computing the call context, as described in Section 4.2. This context is then used to specialize the analysis of the potential targets of the call.

Recursive calls. In case recursion is detected, meaning that the current call happens within a cycle in the call graph, we check whether the number of target is not too big. In practice, we consider this upper limit to be 50. We argue that effects would become overly imprecise anyway once we exceeds this many targets for a single call. In such cases, the effect becomes $\top$ immediately. Otherwise, each target is analyzed independently until a fix-point is reached. Note that, by construction, we always obtain a definite summary from $\mathrm{analyze}_{fix}()$.

Non-Recursive calls. In the case of non-recursive call, we query the heuristic to check whether, depending on the current facts, we should delay the analysis of the method call. We can imagine different ways to estimate the benefits from delaying. In practice, we not only use the number of targets, but we also check whether o escapes, indicating that the number of targets might shrink when delaying.

If the heuristic decides to delay the analysis, the effect that flows out of this analyzed method is stripped from anything that might be modified by the call itself. We conservatively assume that the unanalyzed call might modify every mutable fields. The $\mathrm{stableEffectsOf}$ function consequently removes effects on mutable fields, as they might relay information that is invalidated by the call.

In case the analysis is not delayed, we check whether one of the summary is *conditional*. In such cases, we perform the inlining described in Section 3.3 after having properly renamed and versioned the conditional summary. If all summaries are *definite*, we merge them within the current effect using the composition operator. Of course we map variables such as parameters accordingly.

Finally, we join the effects obtained from multiple targets using the lattice join operation described in Algorithm 4.1.1.

Algorithm 2 Calling methods

```
function CALL($i, v, o, m, (a1, ..., an), \ell$)
    ctx = $\mathrm{Ctx}(o, a_1, ..., a_n)$
    targets = targets of $o.m$
    if within recursion then
        if too many targets then
            return $\top$
        else
            smrs = $\{\mathrm{analyze}_{fix}(t, \mathsf{ctx}) \mid t \in \mathsf{targets}\}$
        end if
    else
        if $\mathrm{h}(o.m, \ldots)$ then
            return $\mathrm{stableEffectOf}(in)$
        else
            smrs = $\{\mathrm{analyze}(t, \mathsf{ctx}) \mid t \in \mathsf{targets}\}$
            if $\exists s \in$ smrs . $s$ is conditional then
                Interrupt analysis and inline CFGs of smrs
            end if
        end if
    end if
    return $\bigsqcup\{in \diamond s \mid s \in smrs\}$
end function
```

4.2 Context Sensitivity

Using relational summaries already gives us a powerful form of context sensitivity, but it is not always sufficient in practice. In order to analyze recursive methods precisely, Insane needs to implement an additional form of context-sensitivity, associating different relational summaries based on the call context. We implemented this context-sensitivity by identifying methods by a tuple $(\mathsf{C.m}, \mathsf{ctx})$ where ctx corresponds to the call context. In order to describe our contexts, we first introduce a notion of type signature which is recursively defined by:

$$\begin{aligned} \mathsf{TypeSig} &:= \mathsf{TypeInfo} \times 2^{\mathsf{Fields} \times \mathsf{TypeSig}} \\ \mathsf{TypeInfo} &:= \mathsf{Type} \times \mathsf{Boolean} \end{aligned}$$

Type signatures represent type information about a particular object. They not only give information about the type of the object itself, but can also provide a type signature for some of its fields. Each type information is annotated with a flag, indicating whether subtypes should also be considered in order to distinguish $\_ \sqsubseteq T$ and $\_ = T$. We define the *depth* of a type signature to be the maximum depth until no field information is specified:

$$\mathrm{depth}(s) := 1 + \max\ \{0\} \cup \{\mathrm{depth}(s_f) \mid (f, s_f) \in s.\mathsf{fields}\}$$

We also introduce $\mathsf{TypeSig}_d \subset \mathsf{TypeSig}$ to represent signatures of maximum depth d. The call context of each method belong to the domain $\mathsf{TypeSig}_d^{1+n}$ where n is the number of its arguments. In other words, the calling context is composed of type signatures of maximum depth d for its receiver and each of its arguments.

4.3 Application to the Analysis of Real-World Scala Code

Analyzing real-world Scala code adds practical complications that were not directly embedded into the target language described in Section 3.1 and thus not originally accounted for. We here list several of these complications and explain how we handled them.

Java dependencies. The Scala library depends heavily on the Java library, it is thus crucial for our analysis to handle Java code as well. While the Scala compiler does not compile Java source-code, it provides an utility to read Java byte-code into one of its intermediate representation used later in the pipeline. We were able to convert this stack-based intermediate representation into our CFGs, allowing us to analyze the Java dependencies. However, due to technical limitations in this utility, some Java classes fail to parse and thus not all dependencies are available. In such cases, the method calls to those unknown dependencies get delayed, just like imprecise method calls, and eventually yield conditional summaries.

Native code. Having access to all Java dependencies in the available `jar` files is not be sufficient to completely analyze the Scala library. Indeed, both the Java and Scala library rely on classes that are implemented as native (C) libraries, or by the JVM itself. This is the case for instance for `arrays`. In order to handle them correctly, Insane relies on custom, stub implementations for these native classes and methods: it intercepts the calls to native methods and redirects them to the stub implementations. Arrays for instance are implemented as an instance of a class containing one field acting as a store. Writing to the array becomes a weak update on that store. (We thus do not distinguish between two elements stored at different array indices.) This store field is notably apparent in Figure 12 as `store`. As an illustration, we provide the skeleton of the stub implementation for arrays:

```
class ArrayStub[T](val length: Int) {
  var store: T = _
  def update(i: Int, v: T) = if (?) { store = v } else { }
  def apply(i: Int) = store
}
```

Exceptions. Accounting for exceptional flow is a challenging precision problem as it typically clutters the CFG with several edges to exception handlers. Scala has no checked exception and thus its compiler provides no information on which exceptions a method call might throw. We decided to ignore exceptional flow in this version of our analyzer. We are thus in general unsound in the presence of exceptions. Currently, throwing an exception in one branch results in a bottom effect, indicating an impossible branch. This is consistent with the view that the results should be valid for non-exceptional executions.

Specialization for literal values. It is also worth noting that Insane specializes Integers and Boolean types in order to distinguish between different literal values. Consequently, graphs can contain nodes representing $AnyInt$ as well as $IntVal(n)\ \forall n \in Int$. This specialization is both at the node and type level, we thus have that:

$$\text{type}(\text{BoolVal}(\mathsf{true})) \sqcap \text{type}(\text{BoolVal}(\mathsf{false})) = \bot$$

This is especially useful for boolean values, as it allows us to filter branches protected by unsatisfiable boolean conditions.

5. Producing Readable Effect Summaries

We have demonstrated that summaries based on control-flow graphs are a flexible and expressive representation of heap modifications. They are, however, typically not suitable as such as feedback to provide to programmers: they capture both read and write effects, while users are presumably interested primarily in writes. They can expose memory cells that are allocated in a method and do not participate directly in an effect. Perhaps most importantly of all from a usability point of view, they are not in textual form.

To improve the usefulness of the analysis for program understanding purposes, we aim to describe effect summaries of methods in a concise and textual manner. We found regular expressions to be suitable, as they are a common tool to describe potentially infinite sets of strings. We show below how to construct a regular expression from a *definite* summary, and highlight the main differences in terms of expressive power.

For definite summaries, a graph-based effect is available that summarizes the method. The graph not only describes which fields can been modified, but also to which value they can be assigned. On the other hand, the corresponding regular expression only describes which fields could be written to. The task therefore reduces to generating a conservative set of paths to fields that may be modified. We construct the following non-deterministic finite state automaton $(Q, \Sigma, \delta, q_0, \{q_f\})$ based on a graph effect G:

$$\begin{aligned}
Q &:= G.V \cup \{q_f, q_0\} \\
\Sigma &:= \{f \mid v_1 \xrightarrow{f} v_2 \in G.E\} \\
\delta &:= G.E \cup \{q_0 \xrightarrow{n} n \mid n \in G.V \wedge \text{connecting}(n)\} \\
&\qquad \cup \{v_1 \xrightarrow{f} q_f \mid v_1 \xrightarrow{f} v_2 \in G.IE \wedge \\
&\qquad\qquad v_1 \text{ is not an inside node}\}
\end{aligned}$$

The NFA accepts strings of words where "letters" are names of the method arguments and field accesses. Given an access path, $o.f_1.f_2.\cdots.f_{n-1}.f_n$, the NFA accepts it if f_n might be modified on the set of objects reached via $o.f_1.f_2.\cdots.f_{n-1}$. We exclude writes on inside nodes, as they represent writes that are not be observable from outside, since the node represents objects allocated within the function. From the NFA, we produce a regular expression by first determinizing the automaton, then minimizing the obtained DFA, and finally applying a standard transformation into a regular expression by successive state elimination. We found the passage through DFA and minimization to be very important for the conciseness of the final expression. As an illustration, the regular expression for TreeSet in Figure 12 originated from an effect graph with 14 nodes. The corresponding NFA has 10 nodes, and so does the non-minimal DFA. The minimal DFA has 5 nodes and translates into the compact regular expression as shown in the figure.

For a *conditional* summary, we extract the set of unanalyzed method calls, then compute a (definite) effect assuming that they are all pure, and present the corresponding regular expression along with the set of calls. The natural interpretation is that the regular expression captures all possible writes under the assumption that no function in the set has a side effect.

Section 6.3 and in particular Figure 12 below show some of the regular expressions that were built from our analysis of collections in the standard Scala library.

6. Evaluation

We implemented the analysis described in the previous sections as part of a tool called Insane. Insane is a plugin for the official Scala compiler.

6.1 Overall Results

To evaluate the precision of our analysis, we ran it on the entire Scala library, composed of approximately 58'000 methods at our stage of compilation. We believe this is a relevant benchmark: due to the functional paradigm encouraged in Scala, several methods are of higher-order nature. For instance, collection classes typically define traversal methods that take functions as arguments, such as filter, fold, exists, or foreach [21]. It is worth noting that we assumed a closed-world in order to analyze the library. Indeed, since most classes of the library are fully extensible, analyzing it without this assumption would not yield interesting results. Given that even getters and setters can in general be extended, most of effects would depend on future extensions, resulting in almost no definite summary.

We proceeded as follows: for each method, we analyzed it using its declaration context and classified the resulting summary as a member of one of four categories: if the summary is definite, we look for observable effects. Depending on the presence of observable effects, the method is flagged either as *pure* or *impure*. If the summary is conditional, we check if the effect would be pure under the assumption that every remaining (delayed) method call is pure. In such cases, the effect is said to be *conditionally pure*. Otherwise, the effect is said to be *impure*. Lastly, an effect can be *top* if either the analysis timed out, or if more than 50 targets were to be inlined in a situation where delaying was not available (e.g. recursive methods). We used a timeout of 2 minutes per function. We note that while these parameters are to some extend arbitrary, we estimate that they correspond to reasonable expectations for the analysis to be useful. The different categories of effects form a lattice:

$$\texttt{pure} \sqsubseteq \texttt{conditionnally pure} \sqsubseteq \texttt{impure} \sqsubseteq \top$$

Figure 9 displays the number of summaries per category and per package. We can observe that most methods are either *pure* or *conditionally pure*, which is what one would expect in a library that encourages functional programming.

Overall, the entire library takes short of twenty hours to be fully processed. This is mostly due to the fact that in this scenario, we compute a summary for each method. Thanks to its modularity though, this analysis could be used in an incremental fashion, reanalyzing only modified code and new dependencies while reusing past, unchanged results. Depending on the level of context-sensitivity, past results can be efficiently reused in an incremental fashion and allow the analysis to scale well to large applications.

Package	Methods	Pure	Cond. Pure	Impure	$\top$
`scala`	5721	79%	11%	10%	1%
`scala.annotation`	41	93%	2%	2%	2%
`scala.beans`	25	64%	8%	28%	0%
`scala.collection`	5182	52%	28%	17%	4%
`scala.collection.concurrent`	608	40%	19%	37%	4%
`scala.collection.convert`	1106	62%	23%	13%	1%
`scala.collection.generic`	649	61%	22%	12%	5%
`scala.collection.immutable`	6027	58%	13%	23%	6%
`scala.collection.mutable`	7263	48%	18%	29%	5%
`scala.collection.parallel`	13842	36%	13%	37%	14%
`scala.collection.script`	132	86%	1%	13%	0%
`scala.compat`	9	22%	33%	44%	0%
`scala.io`	546	47%	11%	40%	2%
`scala.math`	1847	67%	28%	5%	0%
`scala.parallel`	39	77%	23%	0%	0%
`scala.ref`	113	58%	3%	39%	0%
`scala.reflect`	5862	50%	9%	40%	1%
`scala.runtime`	1620	61%	25%	14%	1%
`scala.sys`	767	44%	22%	30%	4%
`scala.testing`	44	52%	2%	43%	2%
`scala.text`	115	87%	0%	11%	2%
`scala.util`	1786	51%	11%	32%	6%
`scala.util.parsing`	2206	56%	12%	27%	5%
`scala.xml`	2860	56%	11%	30%	3%
Total:	58410	52%	15%	27%	6%

Figure 9. Decomposition of resulting summaries per package.

6.2 Comparative Analysis

To motivate some of the key features of Insane, we analyzed parts of the Scala library in three different settings: 1) the default configuration and strategy that ship with Insane 2) disabled context-sensitivity 3) delaying and context-sensitivity both disabled. We analyzed a representative subset of the Scala library, namely all mutable and immutable collections, under the three analysis configurations.

Figure 10 displays the decomposition of effects for each of the analysis settings. First of all, we can see that the default configuration used by Insane is more precise than the alternatives. Indeed, it produces the largest number of purity guarantees. This precision has a cost though: it is also the slowest configuration. In the second configuration, Insane without context sensitivity, we can first notice that while it produces a smaller number of definite-pure guarantees, it is also able infer a lot of conditionally pure methods. However, the quality of those conditional summaries is likely to be worse than for the first setting, resulting in assumptions that probably does not hold in practice. By comparing to the default Insane, we even see that some of the conditionally pure are in fact most likely impure, but that additional precision is required to figure this out. The results for the third configuration need to be interpreted with care. In this setting, delaying is not permitted. As a result, methods relying on higher order functions are in general misclassified as impure. In fact, we see that more than 50% of the methods are considered to be impure when running the analysis in this configuration.

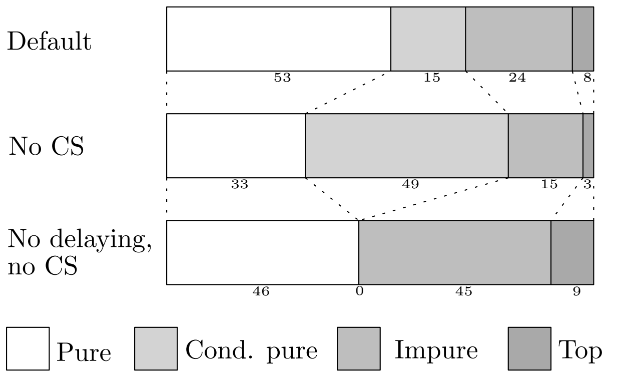


Figure 10. Comparing strategies. Numbers below boxes are percentages. Running times are 166, 123 and 57 minutes respectively.

6.3 Selected Examples

In order to demonstrate the precision of the analysis, we now take a closer look at a few methods relying on the library, for which the pre-computed summaries can be reused in order to efficiently produce precise results. We targeted five collections, two immutable ones: TreeSet and List, and three mutable ones: HashSet, LinkedList and ArrayBuffer. For each of these collections, we performed four standard operations: 1) Generic Traversal: calling foreach with an arbitrary closure 2) Pure Traversal: calling foreach with a pure closure 3) Impure Traversal: calling foreach with a closure modifying the collection elements and 4) calling a "grow" operation that builds a larger collection, either by copying and extending it for immutable ones, or by modifying it in place otherwise. The method used to grow depends on what is available in the public interface of the collection, e.g. add, append or prepend. As an illustration, Figure 11 shows functions corresponding to these four operations when applied to a TreeSet.

The resulting effects are converted into a readable format, as described in Section 5 and displayed in Figure 12. We note that producing these regular expressions takes in each case under 5 seconds. First of all, we can see that all pure traversals are indeed proved pure and have no effect on the internal representation of the collections. Also, we are often able to report that a generic

```
class Elem(val i: Int) {
  var visited = false
}

def genTrav(es: TreeSet[Elem], f: Elem ⇒ Unit) = es.foreach(f)

def pureTrav(es: TreeSet[Elem]) = es.foreach { e ⇒ () }

def impureTrav(es: TreeSet[Elem]) = es.foreach { _.visited = true }

def grow(es: TreeSet[Elem], e: Elem) = es + e
```

Figure 11. Using a TreeSet collection in four different ways.

traversal has no effect on the collection assuming the closure passed is pure. The exceptions are the generic traversals of TreeSet and ArrayBuffer. In these two cases, the computed effect is $\top$, due to the fact that their respective traversal routines are implemented using written using a recursive function. Without the ability to delay, we cannot produce a conditional summary and thus $\top$ is returned. We can see however that thanks to context sensitivity, we are able to obtain precise results when the closure is determined.

In the cases of impure traversals, the effects correctly report that all elements of the collections may have been modified. Additionally, they uncover the underlying implementation structures. For example, we can see that the HashSet class is implemented using a flat hash table (using open addressing) instead of the usual array of chained buckets. It is worth noting that TreeSet is implemented using red-black trees. For mutable collections, growing the collection indeed has an effect on the underlying implementation. Growing immutable collections remains pure since the modifications are applied to the returned copy only.

Overall, we believe such summaries are extremely useful, as they qualify the impurity. In almost all cases, the programmer can rely on the result produced by Insane to conclude that the collection itself will not be affected by changes.

7. Related work

Our goals stand at the crossroad of two long-standing, fundamental, problems: on one hand, effect analysis and the related problem of alias analysis [7, 9, 16, 20, 24, 30] and on the other hand the question of control-flow analysis [28], originally developed for analyzing functional programs. While we have so far focused most of our efforts on ideas adopted from the first category, we hope in the future to be able to incorporate recent insights from the latter [19].

The analysis domain presented in this paper builds on the work of Salcianu, Rinard and Whaley [26, 27, 31], who used graphs to encode method effect summaries independently from aliasing relations. The elements of this abstract domain are best understood as state transformers, rather than sets of heaps. This observation, which is key to the applicability of the generic relational framework described in Section 3, was also made by Madhavan, Ramalingam, and Vaswani [17], who have formalized their analysis and applied it to C# code. The same authors very recently extended their analysis to provide special support for higher-order procedures [18]. An important difference with our work is that [18] summarizes higher-order functions using only CFGs or a particular, fixed, normal form: a loop around the un-analyzed invocations. Because our analysis supports arbitrary conditional summaries, it is a strict generalization in terms of precision of summaries. Another distinctive feature of our analysis is its support for strong updates, which is crucial to obtain a good approximation of many patterns commonly found in Scala code. In fact, the reduction of CFGs to normal form in [18] relies on graph transformers being monotonic, a property that is incompatible with strong updates. Finally, our tool also produces regular expression summaries, delivering results that can be immediately useful to programmers.

The idea of delaying parts of the analysis has been explored before in interprocedural analyses to improve context-sensitivity [10, 32] or to speed up bottom-up whole-program analyses [14]. Our work shows that this approach also brings benefits to the analysis of programs with callbacks, and is in fact critical to its applicability.

Our analysis masks only effects that can be proved to be performed on fresh objects in given procedure call contexts. A more ambitious agenda is to mask effects across method calls of an abstract data types, which resulted in a spectrum of techniques with different flexibility and annotation burden [1, 2, 4–6, 8, 12, 15, 23]. While analysis is fully automated, we expect it could still benefit from the work in the area of encapsulation, information hiding, and representation independence.

Separation logic [3, 11] and implicit dynamic frames [22, 29] are two popular paradigms for controlling modifications to heap regions. We note that effect analysis is a separate analysis, whereas separation logic analyses need to perform shape and effect analyses at the same time. This coupling of shape and effect, through the notion of footprint, makes it harder to deploy separation logic-based analyses as lightweight components that are separate from subsequent analysis phases. Moreover, the state of the art in separation logic analyses is such that primarily linked list structures can be analyzed in a scalable way, whereas our analysis handles general graphs and is less sensitive to aliasing relationships.

The importance of conditional effects expressed as a function of arguments has been identified in an effect system for Scala, which requires some type annotations and is higher-level, but provides more control over encapsulation [25]. The resulting system is fully modular and supports, e.g. separate compilation. In the future, we expect to be able to use Insane as an automated annotation engine for the effect system of Rytz et al., thus alleviating the bootstrapping problems that come with the annotation requirements.

8. Conclusion

We have presented a generic framework for relational effect analyses. Our framework is designed to support various strategies, allowing analysis designers to experiment with trade-offs between precision and time. We have also presented the key features of an abstract domain designed to track read and write effects on the heap. Combining our generic framework with this abstract domain, we have developed an effect analysis for Scala programs and have evaluated it on the entire Scala standard library, thus producing a detailed breakdown of its 58K functions by purity status. Finally, we have proposed a technique to produce human-readable summaries of the effects to make them immediately useful to programmers. We have shown that these summaries can concisely and naturally describe heap regions, thus producing feedback that conveys much more information than a simple pure/impure dichotomy. Insane works on unannotated code and can thus readily be applied to existing code bases.

References

[1] Banerjee, A., Naumann, D.A.: State based ownership, reentrance, and encapsulation. In: ECOOP (2005)

[2] Barnett, M., DeLine, R., Fähndrich, M., Leino, K.R.M., Schulte, W.: Verification of object-oriented programs with invariants. Journal of Object Technology 3(6), 27–56 (2004)

[3] Berdine, J., Cook, B., Ishtiaq, S.: SLAyer: Memory safety for systems-level code. In: CAV. pp. 178–183 (2011)

Collection	Operation	Regex
`immutable.TreeSet`	Generic traversal	ANY
	Pure traversal	PURE
	Impure traversal	`es.tree(.right` | `.left)`*`.key.visited`
	Grow	PURE
`immutable.List`	Generic traversal	conditionally on the closure: PURE
	Pure traversal	PURE
	Impure traversal	`es.tl`*`.hd.visited`
	Grow	PURE
`mutable.HashSet`	Generic traversal	conditionally on the closure: PURE
	Pure traversal	PURE
	Impure traversal	`es.table.store.visited`
	Grow	`es.tableSize` | `es.table.store` | `es.sizemap.store` | `es.sizemap` | `es.table`
`mutable.LinkedList`	Generic traversal	conditionally on the closure: PURE
	Pure traversal	PURE
	Impure traversal	`es.next`*`.elem.visited`
	Grow	`es.next.next`*
`mutable.ArrayBuffer`	Generic traversal	ANY
	Pure traversal	PURE
	Impure traversal	`es.array.store.visited`
	Grow	`es.size0` | `es.array.store` | `es.array`

Figure 12. Regular expressions obtained as effect summaries.

[4] Boyapati, C., Liskov, B., Shrira, L.: Ownership types for object encapsulation. In: Proc. 30th ACM POPL (2003)

[5] Boyland, J.: The interdependence of effects and uniqueness. In: 3rd workshop on Formal Techniques for Java Programs (2001), uRL: http://www.informatik.fernuni-hagen.de/import/pi5/workshops/ecoop2001_papers.html

[6] Cavalcanti, A., Naumann, D.A.: Forward simulation for data refinement of classes. In: Proceedings of Formal Methods Europe FME'2002. LNCS, vol. 2391, pp. 471–490 (2002)

[7] Chase, D.R., Wegman, M.N., Zadeck, F.K.: Analysis of pointers and structures. In: PLDI. pp. 296–310 (1990)

[8] Clarke, D., Drossopoulou, S.: Ownership, encapsulation and the disjointness of type and effect. In: Proceedings of the 17th ACM conference on Object-oriented programming, systems, languages, and applications. pp. 292–310. ACM Press (2002)

[9] Cooper, K.D., Kennedy, K.: Interprocedural side-effect analysis in linear time. In: PLDI. pp. 57–66 (1988)

[10] Cousot, P., Cousot, R.: Modular static program analysis. In: CC. pp. 159–178 (2002)

[11] Dinsdale-Young, T., Dodds, M., Gardner, P., Parkinson, M.J., Vafeiadis, V.: Concurrent abstract predicates. In: ECOOP. pp. 504–528 (2010)

[12] Fähndrich, M., Leino, K.R.M.: Heap monotonic typestates. In: International Workshop on Aliasing, Confinement and Ownership in object-oriented programming (IWACO) (2003)

[13] Gamma, E., Helm, R., Johnson, R., Vlissides, J.: Design Patterns. Elements of Reusable Object-Oriented Software. Addison-Wesley, Reading, Mass. (1994)

[14] Jensen, S.H., Møller, A., Thiemann, P.: Interprocedural analysis with lazy propagation. In: SAS. pp. 320–339 (2010)

[15] Jifeng, H., Hoare, C.A.R., Sanders, J.W.: Data refinement refined. In: ESOP'86. LNCS, vol. 213 (1986)

[16] Jouvelot, P., Gifford, D.K.: Algebraic reconstruction of types and effects. In: POPL. pp. 303–310 (1991)

[17] Madhavan, R., Ramalingam, G., Vaswani, K.: Purity analysis: An abstract interpretation formulation. In: SAS. pp. 7–24 (2011)

[18] Madhavan, R., Ramalingam, G., Vaswani, K.: Modular heap analysis for higher-order programs. In: SAS (2012), to appear.

[19] Might, M., Smaragdakis, Y., Horn, D.V.: Resolving and exploiting the *k*-CFA paradox: illuminating functional vs. object-oriented program analysis. In: PLDI. pp. 305–315 (2010)

[20] Milanova, A., Rountev, A., Ryder, B.G.: Parameterized object sensitivity for points-to and side-effect analyses for java. In: ISSTA. pp. 1–11 (2002)

[21] Odersky, M., Moors, A.: Fighting bit rot with types (experience report: Scala collections). In: FSTTCS. pp. 427–451 (2009)

[22] Parkinson, M.J., Summers, A.J.: The relationship between separation logic and implicit dynamic frames. In: ESOP. pp. 439–458 (2011)

[23] de Roever, W.P., Engelhardt, K.: Data Refinement: Model-oriented proof methods and their comparison. Cambridge University Press (1998)

[24] Rountev, A.: Precise identification of side-effect-free methods in java. In: ICSM. pp. 82–91 (2004)

[25] Rytz, L., Odersky, M., Haller, P.: Lightweight polymorphic effects. In: ECOOP (2012)

[26] Salcianu, A., Rinard, M.C.: Purity and side effect analysis for Java programs. In: VMCAI. pp. 199–215 (2005)

[27] Salcianu, A.D.: Pointer Analysis for Java Programs: Novel Techniques and Applications. Ph.D. thesis, Massachusetts Institute of Technology (2006)

[28] Shivers, O.: Control-flow analysis in scheme. In: PLDI. pp. 164–174 (1988)

[29] Smans, J., Jacobs, B., Piessens, F.: Implicit dynamic frames: Combining dynamic frames and separation logic. In: ECOOP. pp. 148–172 (2009)

[30] Tkachuk, O., Dwyer, M.B.: Adapting side effects analysis for modular program model checking. In: ESEC / SIGSOFT FSE. pp. 188–197 (2003)

[31] Whaley, J., Rinard, M.C.: Compositional pointer and escape analysis for Java programs. In: OOPSLA. pp. 187–206 (1999)

[32] Yorsh, G., Yahav, E., Chandra, S.: Generating precise and concise procedure summaries. In: Necula, G.C., Wadler, P. (eds.) POPL. pp. 221–234. ACM (2008)

A. Additional Algorithms

A.1 Object Allocation

Algorithm 3 Allocation: $\ell : v = newC$

```
1: function ALLOC($\langle N_i, N_l, N_v, E_r, E_w, \pi\rangle, v, C, \ell$)
2:     $n_* = inode(C, \ell, summary)$
3:     $n_1 = inode(C, \ell, singleton)$
4:     if $n_1 \in N_i$ then
5:         $n = n_*$
6:         return $\langle N_i[n_1 \to n_*], N_l, N_v, E_r[n_1 \to n_*], E_w[n_1 \to n_*], \pi[v \to n_*]\rangle$
7:     else if $n_* \in N_i$ then
8:         return $\langle N_i, N_l, N_v, E_r, E_w, \pi[v \to n_*]\rangle$
9:     else
10:        return $\langle N_i \cup \{n_1\}, N_l, N_v, E_r, E_w, \pi[v \to n_1]\rangle$
11:    end if
12: end function
```

A.2 Field Reads

Algorithm 4 Reading: $\ell : v = o.f$

```
1: function READ($\langle N_i, N_l, N_v, E_r, E_w, \pi\rangle, v, o, f, \ell$)
2:     $E_{new} = \emptyset$
3:     $N_{new} = \emptyset$
4:     $targs = \emptyset$
5:     for $n \in \pi(o)$ do
6:         $targ = wTrTarg(n, f)$
7:         if $targ = \emptyset$ then
8:             $n_\ell = loadNode(f, \ell)$
9:             $E_{new} = E_{new} \cup \{n \xrightarrow{f} n_\ell\}$
10:            $N_{new} = N_{new} \cup \{n_\ell\}$
11:            $targ = \{n_\ell\}$
12:        end if
13:        $targs = targs \cup targ$
14:    end for
15:    return $\langle N_i, N_l \cup N_{new}, N_v, E_r \cup E_{new}, E_w, \pi[v \to targs]\rangle$
16: end function
```

A.3 Field Updates

Algorithm 5 Writing Fields: $\ell : o.f = v$

```
1: function WRITE($in, os, f, vs, allowStrong, \ell$)
2:     $\langle N_i, N_l, N_v, E_r, E_w, \pi\rangle = in$
3:     $strong = allowStrong \wedge os = \{n\} \wedge \neg isSummary(n)$
4:     if $strong$ then
5:         $keep = \exists ln \in N_l \,.\, ln.field = f$
6:         $old = wTargets(o, f)$
7:         $writes = \{o \xrightarrow{f} t \mid t \in vs\}$
8:         if $keep$ then
9:             $reads = E_r \cup old$
10:        else
11:            $reads = E_r$
12:        end if
13:        return $\langle N_i, N_l, N_v, reads, (E_w \cup write) \setminus old, \pi\rangle$
14:    else
15:        $E_{r,new} = \emptyset$
16:        $E_{w,new} = \emptyset$
17:        $N_{new} = \emptyset$
18:        for $n \in \pi(o)$ do
19:            $old = wTrTargets(o, f)$
20:            if $old = \emptyset$ then
21:                $n_\ell = loadNode(f, \ell)$
22:                $E_{r,new} = E_{r,new} \cup \{n \xrightarrow{f} n_\ell\}$
23:                $N_{new} = N_{new} \cup \{n_\ell\}$
24:                $old = \{n_\ell\}$
25:            end if
26:            $E_{w,new} = E_{w,new} \cup \{n \xrightarrow{f} x \mid x \in old \cup vs\}$
27:        end for
28:        return $\langle N_i, N_l \cup N_{new}, N_v, E_r \cup E_{r,new}, E_w \cup E_{w,new}, \pi\rangle$
29:    end if
30: end function
```

A.4 Branch Filtering

Algorithm 6 Checking types: $[v <: t]$

```
1: function BRANCH($\langle N_i, N_l, N_v, E_r, E_w, \pi\rangle, v, T$)
2:     $N = \emptyset$
3:     for $n \in \pi(v)$ do
4:         if $(type(n) \sqcap t) \neq \bot$ then
5:             $N = N \cup \{n\}$
6:         end if
7:     end for
8:     if $N = \emptyset$ then
9:         return $\bot$
10:    else
11:        return $\langle N_i, N_l, N_v, E_r, E_w, \pi[v \to N]\rangle$
12:    end if
13: end function
```

Algorithm 7 Checking equalities: $[v_1 = v_2]$

```
function BRANCH($\langle N_i, N_l, N_v, E_r, E_w, \pi\rangle, v_1, v_2$)
    if $\exists n_1 \in \pi(v_1), n_2 \in \pi(v_2) . (type(n_1) \sqcap type(n_2)) \neq \bot$ then
        return $\langle N_i, N_l, N_v, E_r, E_w, \pi\rangle$
    else
        return $\bot$
    end if
end function
```

A.5 Load Nodes Resolution

Algorithm 8 Resolving load nodes

```
function RESOLVE($outer, n, visited, \ell$)
    $f = n.field$
    $innerFrom = \{n_1 \mid n_1 \xrightarrow{f} n \in outer.E_r\}$
    $fromNodes = \emptyset$
    $result = \emptyset$
    for $in \in innerFrom$ do
        if $in$ is load node $\wedge$ $in \notin visited$ then
            $(outer, ons) = resolve(in, visited \cup \{in\})$
        else
            $ons = map(in)$
        end if
        $fromNodes = fromNodes[in \to ons]$
    end for
    for $(in \to ons) \in fromNodes \wedge on \in ons$ do
        $pointed = targ(on, f, outer.E)$
        if $pointed = \emptyset$ then
            $n_\ell = loadNode(f, n.\ell \oplus \ell)$
            $outer = outer[E_r \to outer.E_r \cup \{on \xrightarrow{f} n_\ell\}]$
            $outer = outer[N_l \to outer.N_l \cup \{n_\ell\}]$
            $result = result \cup \{n_\ell\}$
        else
            $result = result \cup pointed$
        end if
    end for
    return $(outer, result)$
end function
```

A.6 Importing Inside Nodes

Algorithm 9 Merging inside nodes

```
function MERGEINODE($\langle N_i, N_l, N_v, E_r, E_w, \pi\rangle, n, \ell$)
    $n_* = inode(n.type, n.\ell \oplus \ell, summary)$
    $n_1 = inode(n.type, n.\ell \oplus \ell, singleton)$
    if $n_1 \in outer.N_i$ then
        return $(\langle N_i[n_1 \to n_*], N_l, N_v, E_r[n_1 \to n_*], E_w[n_1 \to n_*], \pi\rangle, N_*)$
    else if $n_* \in outer.N_i$ then
        return $(\langle N_i, N_l, N_v, E_r, E_w, \pi\rangle, N_*)$
    else if $isSummary(n)$ then
        return $(\langle N_i \cup \{n_*\}, N_l, N_v, E_r, E_w, \pi\rangle, N_*)$
    else
        return $(\langle N_i \cup \{n_1\}, N_l, N_v, E_r, E_w, \pi\rangle, N_1)$
    end if
end function
```

A.7 Composing effects

Algorithm 10 Composing summaries: $Smr(inner)$

```
function COMPOSE($outer, inner, \ell$)
    if $(outer \sqcup inner) = \top$ then
        return $\top$
    else
        $map_1 = \emptyset$
        for $n \in inner.N_v$ do
            $map_1 = map_1[n \to outer.\pi(n.ref)]$
        end for
        $(res, map_2) = merge(inner, outer, map_1, \ell)$
        $\pi_{new} = res.\pi$
        for $(r \to ns) \in inner.\pi$ do
            $\pi_{new} = \pi_{new}[r \to \bigcup\{map_2(n) \mid n \in ns\}]$
        end for
        return $res[\pi \to \pi_{new}]$
    end if
end function
```

Algorithm 11 Merging graphs

```
function MERGE($outer, inner, map, \ell$)
    for $n \in inner.N_i$ do
        $(outer, n_i) = mergeINode(outer, n, \ell)$
        $map = map[n \to \{n_i\}]$
    end for
    repeat
        $old = (outer, map)$
        for $(i_1 \xrightarrow{f} i_2) \in inner.E_r$ do
            if $i_1$ is load node then
                $(outer, o_1s) = resolve(outer, i_1, \{i_1\}, \ell)$
            else
                $o_1s = map(i_1)$
            end if
            if $n_2$ is load node then
                $(outer, o_2s) = resolve(outer, i_2, \{i_2\}, \ell)$
            else
                $o_2s = map(i_2)$
            end if
            $outer = outer[E_r \to outer.E_r \cup \{o_1 \xrightarrow{f} o_2 \mid o_1 \in o_1s \wedge o_2 \in o_2s\}$
        end for
        $wr = \emptyset$
        for $o_1 \xrightarrow{f} o_2 \in inner.E_w \wedge n_1 \in map(o_1) \wedge n_2 \in map(o_2)$ do
            $wr = wr[(n_1, f) \to (wr((n_1, f)) \cup \{(n_2, o_1)\})]$
        end for
        for $((n_1, f) \to tos) \in wr$ do
            $(n_2s, o_1s) = tos.unzip$
            $allowStrong = \forall o \in o_1s . |map(o)| = 1$
            $env = write(env, \{n_1\}, f, n_2s, allowStrong, \ell)$
        end for
    until $old == (outer, map)$
end function
```

A.8 Calling Methods

Algorithm 12 Calling methods

```
1:  function CALL($in, v, o, m, (a1, ..., an), \ell$)
2:      $targets = findTargets(in.\pi(o), m)$
3:      $smrs = \emptyset$
4:      $ctx = Ctx(o, a_1, ..., a_n)$
5:      if $withinRecursion()$ then
6:          if $|targets| > T_{max}$ then
7:              return $\top$
8:          else
9:              $smrs = \{analyze_{fix}(t, ctx) \mid t \in targets\}$
10:         end if
11:     else
12:         if $h(in, o, targets)$ then
13:             Delay the analysis of this call.
14:             return $stableEffectOf(in)$
15:         else
16:             $smrs = \{analyze(t, ctx) \mid t \in targets\}$
17:             if $\exists s \in smrs\ .\ s$ is Conditional then
18:                 Interrupt analysis and inline CFGs of $smrs$
19:             end if
20:         end if
21:     end if
22:     $res = \emptyset$
23:     for $s \in smrs$ do
24:         $map_1 = CreateMap(v, o, (a_1, ...a_n))$
25:         $(eff, map_2) = merge(in, s, map_1, \ell)$
26:         $oRet = \{o \mid i \in s.\pi(s.ret) \land o \in map_2(i)\}$
27:         $eff = eff[\pi \to eff.\pi[v \to oRet]]$
28:         $res = res \cup eff$
29:     end for
30:     return $\bigsqcup res$
31: end function
```