Effect Analysis for Programs with Callbacks

Etienne Kneuss[1], Viktor Kuncak[1*], and Philippe Suter[1,2]

[1] École Polytechnique Fédérale de Lausanne (EPFL), Switzerland
[2] IBM T.J. Watson Research Center, Yorktown Heights, NY, USA
{firstname.lastname}@epfl.ch, psuter@us.ibm.com

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 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 58000 methods and classified them into several categories according to their effects. Our analysis proves that over one half of all methods are pure, identifies a number of conditionally pure methods, and computes summary graphs and regular expressions describing the side effects of non-pure methods.

1 Introduction

An appealing programming style uses predominantly functional computation steps, including higher-order functions, with 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 the form of *delegates*), the 2011 standard of C++, and Java 8.[3] Moreover, design patterns popular in the object-oriented programming community also rely on callbacks, for instance the *strategy pattern* and the *visitor pattern* [13].

[*] This research was supported in part by the European Research Council Project "Implicit Programming"

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

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 are 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 effect summaries that developers can compare to their intuition of what methods should and should not affect in program heap. Such summaries must go beyond a pure/impure dichotomy, and should ideally capture the exact frame condition of the analyzed code fragment — or at least an acceptable over-approximation. We expect our results in this direction will help in bootstrapping annotations for Scala effect type systems [26], 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 relational analysis domain that computes summaries of code blocks and methods by flow-sensitively tracking side effects and performing strong updates;
– a framework for relational analyses to compute higher-order relational summaries of method calls, which are parameterized by the effects of the methods being called;
– an automated effect classification and presentation of effect abstractions in terms of regular expressions, facilitating their understanding by developers.

Our static analyzer, called Insane (INterprocedural Static ANalysis of Effects) is publicly available from

`https://github.com/epfl-lara/insane`

We have evaluated Insane 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 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 when analyzing programs written in a higher-order style, and how Insane can tackle them.

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

```
class Cell(var visited : Boolean)

def toggle(c : Cell) = {
  c.visited = !c.visited
}
```

Any reasonable analysis for effects would detect that toggle 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 "*toggle may modify the .value field of its first argument*". That information could in turn be retrieved whenever toggle 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 toggle has, since the call apply(c, toggle) is equivalent to toggle(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 of 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. In the context of Insane we have applied this idea to a domain designed for tracking heap effects (described in Section 3), although the technique applies to any relational analysis, as we show in Section 4.

Equipped with this extended notion of effects, we can classify methods as *pure*, *impure*, and *conditionally pure*. The apply function falls in this last category: it is pure as long as the methods called from within it are pure as well (in this case, the invocation of f). Notable examples of conditionally pure functions include many of the standard higher-order operations on structures which are used extensively in functional programs (map, fold, foreach, etc.). As an example, a typical implementation of foreach on linked lists is the following:

```
class LinkedList[T](var hd : T, var tl : LinkedList[T]) {
  def foreach(f : T ⇒ Unit) : Unit = {
    var p = this
    do {
      f(p.hd)
      p = p.tl
    } while(p != null)
  }
}
```

Correctly characterizing the effects of such functions is essential to analyzing programs written in a language such as Scala.

Making sense of effects. Another 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:

```
def update(es : LinkedList[Cell]) = {
  es.foreach(c ⇒ c.visited = true)
}
```

Because the closure passed to foreach has an effect, so does the overall function. A summary stating only that it is impure would be highly unsatisfactory, though: crucially, it would not give any indication to the programmer that the structure of the list itself cannot be affected by the writes. As we will see, the precise 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 this 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. For the example given above, the following regular expression is reported to the programmer:

$$\texttt{es}(\texttt{.tl})^*\texttt{.hd.visited}$$

It shows that the fields affected are those *reachable* through the list (by following chains of .tl), but belonging to elements only, thus conveying the desired information. In Section 6.2, we further demonstrate this generation of human-readable effect summaries on a set of examples that use the standard Scala collections library.

3 Effect Analysis for Mutable Shared Structures

The starting point for our analysis is the effect analysis [28, 32]. We here present an adaptation to our setting, with the support for strong updates, which take into account statement ordering for mutable heap operations. In the next section we lift this analysis to the case of programs with callbacks (higher-order programs), for which most existing analyses are imprecise. We thus obtain a unique combination of precision, both for field updates and for higher-order procedure invocations.

We start by describing a target language that is expressive enough to encode most of the intermediate representation of Scala programs that we analyze.

3.1 Intermediate Language Used for the Analysis

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 as in $C.m$ and denote the set of methods in a program $\mathcal{M}$. Our intermediate language has no ad-hoc method overloading because the affected methods can always be renamed after type checking. 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 in our intermediate language, along with their meaning.

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	allocate an object of class C and store the reference to it in v
v = o.m(a1, ..., an)	call method m of object o and store the result in v

Fig. 1. Program statements $\mathcal{P}$ considered in the target language.

Because of dynamic dispatch, a call statement can target multiple methods, depending on the runtime type of the receiver object. For each method call $o.m()$, we can compute a superset of targets targets$(o.m) \subseteq \mathcal{M} \cup \{?\}$ using the static type of the receiver. If the hierarchy is not bounded through **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. Thus, we do not always assume access to the entire program: this assumption is defined as a parameter of the analysis, and we will see later how it affects it.

3.2 Effects as Graph Transformers

We next outline our graph-based representation of compositional effects. Our approach is related to the representation originally used for escape analysis [27, 28]. The meaning of such an effect is a relation on program heaps which over-approximates the behavior of a fragments of code (e.g. methods). Section 4 lifts

```
class List(var elem: Int,
           var nxt: List = null)

def prepend(lst: List, v: Int) {
  lst.nxt = new List(lst.elem, lst.nxt)
  lst.elem = v
}
```

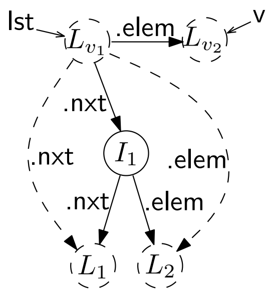

Fig. 2. Example of a graph representing the effects of prepend. Read edges lead to load nodes that represent unknown values, and solid edges represent field assignments.

this representation to a more general, higher-order settings, which gives our final analysis.

Figure 2 shows an example of a simple function and its resulting graph-based effect. In this graph, L_{v_1} and L_{v_2} represent unknown local variables, here the parameters of the function. I_1 is an inside node corresponding to an object allocation. L_1 and L_2 are two load nodes that reference values for fields of L_{v_1} which are unknown at this time. While read (dashed) edges do not strictly represent effects, they are necessary to resolve the meaning of nodes when composing this effect at call-sites.

In general, our effect graphs are composed of nodes representing memory locations. We distinguish three kinds of nodes: *inside* nodes are allocated objects. Because we use the allocation-site abstraction for these, we associate with them a flag indicating whether the node is a singleton or a summary node. *Load* nodes represent unknown fields. Load nodes represent accesses to unknown parts of the heap; supporting them is a crucial requirement for modular effect analyses. Finally, graphs may contain special nodes for unresolved *local variables*, such as parameters.

We also define two types of edges labeled with fields. *Write* edges, represented by a plain (solid) edge in the graphical representation, and *read* edges, represented by dashed edges in the graph. Read edges provide an access paths to past or intermediate values of fields, and are used to resolve *load* nodes. *Write* edges represent *must-write* modifications. Along with the graph, we also keep a mapping from local variables to sets of nodes.

Our analysis directly defines rules to compute the composition of any effect graph with a statement that makes an individual heap modification. It is also possible to represent the meaning of each individual statement as an effect graph itself; the result of executing statement on a current effect graph then corresponds to *composing* two effect graphs. However, the main need for composition arises in modular analysis of function calls.

3.3 Composing Effects

Composition is a key component of most modular analyses. It is typically required for interprocedural reasoning. In our setting, it also plays an important role as a building block in our analysis framework for programs with callbacks, which we describe in Section 4. We now describe how composition applies to effect graphs. This operation is done in a specific direction: we say that an *inner* effect graph is applied to an *outer* effect graph. Merging graphs works by first constructing a map from inner nodes to equivalent outer nodes. This map, initially incomplete, expands during the merging process.

Importing inside nodes. The first step of the merging process is to import inside nodes from the inner graph to the outer graph. We specialize the labels representing their allocation sites to include the label corresponding to the point at which we compose the graphs. This property is crucial for our analysis as case-classes, an ubiquitous feature of Scala, rewrite to factory methods. Once the refined label is determined, we check whether we import a singleton node in an environment in which it already exists. In such case, the node is imported as a summary node.

Resolving load nodes. The next important operation when merging two graphs is the resolution of load nodes from the inner graph to nodes in the outer graph. The procedure works as follows: for each inner 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 which case we recursively invoke the resolution operation. We then compute using the map 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. Once a load node is resolved to a set of nodes in the outer graph, the equivalence 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. We need to make sure that a strong update in the inner graph can remain strong, given the outer graph and the map.

The composition not only executes the last two steps, but repeats them until convergence of the outer graph. Once a fix-point is reached, we have successfully applied full meaning of the inner graph to the outer graph. Such application until fix-point is crucial for correctness in the presence of unknown aliasing and strong updates. We illustrate this merging operation in Figure 3.

4 Compositional Analysis of Higher-Order Code

The composition operator on effect graphs presented in the previous section allows us to analyze programs without dynamic dispatch. Standard approaches to extend it to dynamic dispatch are either imprecise or lose modularity. In

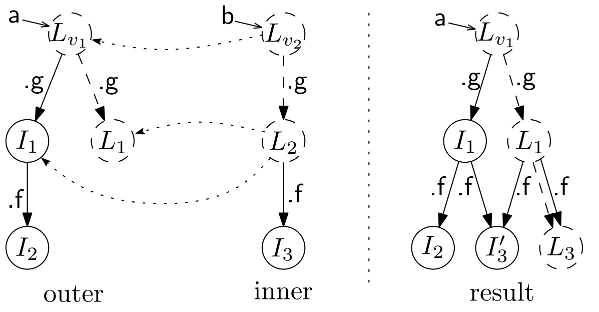

Fig. 3. Merging a graph with load nodes and strong updates in a context that does not permit a strong update. Inside nodes are imported after refining their label.

this section, we therefore extend the basic analysis to support dynamic dispatch (including higher-order functions and callbacks) in both precise and a rather modular way. The methodology by which we extend the core analysis to the higher-order case is independent of the particular domain of effect graphs, so we present it in terms of a framework for precise interprocedural analysis of functions with callbacks.

Our framework works on top of any abstract interpretation-based analysis whose abstract domain R represents relations between program states. The abstract domain described in the previous section matches these requirements. Along with a set of control-flow graphs over statements $\mathcal{P}$ previously discussed, we assume the existence of other usual components of such analyses: a concretization function $\gamma : R \to (2^S)^S$ and a transfer function $\mathrm{T_f} : (\mathcal{P} \times R) \to R$.

We now define 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

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

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.

4.1 Control-Flow Graph Summarization

Summarization consists of replacing a part of the control-flow graph by a statement that over-approximates its effects. Concretely, we first augment the language with a special summary statement, characterized by a single abstract value:

$$\mathcal{P}_{ext} = \mathcal{P} \cup \{\mathsf{Smr}(a \in R)\}$$

Consequently, we define $\mathrm{T_f}_{ext}$ over $\mathcal{P}_{ext}$:

$$\mathrm{T_f}_{ext}(s)(r) = \begin{cases} \mathrm{T_f}(s)(r) & \text{if } s \in \mathcal{P} \\ r \diamond a & \text{if } s = \mathsf{Smr}(a) \end{cases}$$

Let c be the control-flow graph of some procedure over $\mathcal{P}_{ext}$, 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 $a \supset b$. This graph can be viewed as a control-flow graph with a as its source and b as its sink. The summarization consists of replacing $a \supset b$ by a single edge labelled with a summary statement obtained by analyzing the control-flow graph $a \supset b$ in isolation.

We observe that while composition 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 summarizations are performed plays an important role in the overall result. When possible, left-associativity is preferred as it better encapsulates a forward, top-down analysis and can leverage past information.

4.2 Partial Unfolding

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

When faced with a call statement $o.m(\overline{\mathsf{args}})$, the analysis will extract information about o from the data-flow facts and compute the set of its potential static targets $T_{o.m} \subseteq \mathcal{M}$. The control-flow graphs corresponding to the targets are then included after a non-deterministic split. It is worth noting that the set of targets $T_{o.m}$ is generally not complete. Indeed, this process is performed during the fix-point computation, facts about o might still grow in the lattice during future iterations. The original call is therefore kept and annotated to exclude targets already unfolded as pictured in Figure 4. In certain situations, we can conclude that all targets have been covered, rendering the alternative call edge infeasible and thus removable.

4.3 Combining Unfolding and Summarization

We distinguish two main kinds of summaries. A summary that contains unanalyzed method calls is said to be *conditional*. In contrast, a *definite* summary is fully reduced down to a single edge with a summary statement.

We now illustrate the flexibility provided by our framework through a simple example displayed in Figure 5. There are in general multiple ways to generate a definite summary from a control-flow graph, depending on the interleaving of summarization and unfolding operations.

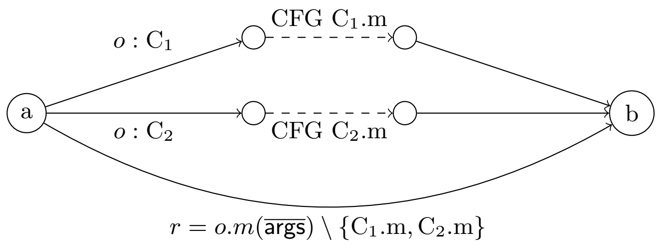
$o : \mathrm{C}_1$
CFG C_1.m
a
b
$o : \mathrm{C}_2$
CFG C_2.m
$r = o.m(\overline{\mathsf{args}}) \setminus \{\mathrm{C}_1.\mathrm{m}, \mathrm{C}_2.\mathrm{m}\}$

Fig. 4. Example of unfolding with $T_{call} = \{\mathrm{C}_1.\mathrm{m}, \mathrm{C}_2.\mathrm{m}\}$.

```
sealed class A {
  def m1() {
    val o = new A;
    this.m2(o)
  }

  def m2(o: A) {
    this.m3()
    o.f()
  }
// .. continuing class A

  def m3() { }

  def f() { }
}

class B extends A {
  override def f() { .. }
}
```

Fig. 5. Example of a chain of method calls.

For instance, one way to generate a summary for $A.m1$ would consist of the following steps: first, we fully summarize $A.m3$, $A.f$ and $B.f$. We unfold their call in $A.m2$, summarize the result, unfold it in $A.m1$ and finally summarize it. This would represent a completely modular approach, where summaries are reused as much as possible. While being perhaps the most efficient way to compute a summary (since intermediate summaries for $A.m2$, $A.m3$, $A.f$ and $B.f$ 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 analyzing o.f() and thus we have to consider every static targets— here $A.f$ and $B.f$, leading to an imprecise summary. We note that this approach, while generally used by traditional compositional analyses, falls short in the presence of callbacks where the number of static targets is typically large (>1'000 for the Scala library). In contrast, we could have waited to analyze o.f() by generating a conditional summary for $A.m2$ where **this**.m3() is unfolded but o.f() remains unanalyzed. We refer to the decision of not analyzing a method call as *delaying*.

4.4 Controlled Delaying

We have seen through the examples above that choosing when to unfold a method call can have a important impact in terms of performance and precision. In our framework, we delegate this decision to a function $\mathrm{D}(call, ctx)$. The precision and performance of the analysis are thus parametrized in D. Fixing D$(\ldots) = \mathsf{false}$ ensures that every method is analyzed modularly, in a top-down fashion, leading to an imprecise analysis. On the other hand, having D$(\ldots) = \mathsf{true}$ forces the analysis to delay every method call, leading to the analysis of the complete control-flow graph at the entry point. While it ensures a precise result, it will produce the largest intermediate graphs, which will slow the analysis considerably. Another problem we can identify is with respect to recursion, which we discuss specifically in the following section.

We also note that the analysis must be able to conservatively reason about delayed method calls in order to proceed past them. A conservative approach is to assume that facts flowing through such method calls get reset to the identity relation.

4.5 Handling Recursion

Assuming the underlying abstract interpretation-based analysis does terminate (which we do ensure for effect graphs), we still need to ensure that the control-flow graph does not keep changing due to unfoldings. For this reason, we need to take special measures for cycles in the call-graph.

Detecting recursion statically is non-trivial, especially in the presence of callbacks. An attempt using a refined version of a standard class analysis proved to be overly imprecise: it would flag every higher-order functions as recursive. Therefore, 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. We handle recursion by ensuring that only definite summaries are generated for methods within the cycle. We in fact enforce termination by requiring that D(c, ctx) returns *false* for any call c within the call-graph cycle.

It is worth noting that $D(\ldots)$ is only constrained for calls within the call-graph 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 set of mutually recursive functions, we start by assuming that all have a definite summary of identity, indicating no effect. The process then uses a standard fix-point iterative process and builds up summaries until convergence.

4.6 Instantiation for Effect Graphs

We now discuss the instantiation of this framework in the context of effect graphs presented in Section 3. We can quickly identify that our abstract domain is relational and thus candidate for use in this framework. The original statements are thus extended with a summary statement characterized by an effect graph:

$$\mathcal{P}_{ext} := \mathcal{P} \cup \{\mathsf{Smr}(G)\}$$

We can also notice that the graph merging operation acts as composition operator $\diamond$:

$$G_1 \diamond G_2 := \text{merge } G_2 \text{ in } G_1$$

For the delaying decision function D, we base our decision on a combination of multiple factors. One important factor is of course the number of targets a method currently has. We also check whether the receiver escapes the current function, indicating that delaying might improve its precision. As expected, experiments indicate that this decision function dictates the trade-off between performance and precision of the overall analysis.

In case the call at hand is recursive, we conservatively prevent its delaying. However, we also check whether the number of targets 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, without the ability to delay. In such cases, the analysis gives up and assigns $\top$ as definite summary to all concerned functions.

Compositional summaries already give us a powerful form of context sensitivity but it is not always sufficient in practice, namely in the presence of recursive methods relying on callbacks. We thus had to introduce another form of context-sensitivity, which specializes the analysis of the same method for multiple call signatures. We compute these signatures combining the type-estimates for each argument.

5 Producing Readable Effect Summaries

We have demonstrated that summaries based on control-flow graphs are a flexible and expressive representation of heap modifications. However, such graph-based summaries are often not directly usable as feedback to programmers, for several reasons. First, they capture both read and write effects, whereas users are likely interested primarily in write effects. Next, they can refer to internal memory cells that are allocated within a method and do not participate directly in an effect. Last, but not the least they are not in textual form and can be difficult to interpret by developers used to textual representations.

To improve the usefulness of the analysis for program understanding purposes, we aim to describe effect summaries of methods in a more concise and textual form. For this purpose we adopt regular expressions because they are a common representation for infinite sets of strings, and can therefore characterize access paths [10]. They also have a notable tradition of use for representing heap effects [17]. We adopt the general idea of representing graphs using sets of paths to generate an approximate textual representation of graph-based summaries for our analysis.

We first show how we construct a regular expression for a *definite* summary. 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:

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

The automaton 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 automaton 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 non-deterministic automaton, we produce a regular expression by first determinizing it, then minimizing the obtained deterministic automaton, and finally applying a standard transformation into a regular expression by successive state elimination. Figure 6 shows the effect graph and the corresponding automata (non-minimized and minimized) for the example from the end of Section 2. In general, we found the passage through determinization and minimization to have a significant positive impact on the conciseness of the final expression.

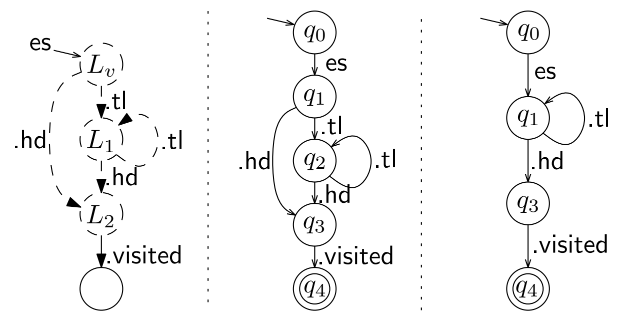

Fig. 6. Transformation steps from an effect graph to a minimized DFA. The graph on the left is the *definite* effect of an impure list traversal. The center graph is the corresponding NFA whose accepting language represents paths to modified fields. The last graph is the minimized DFA to be translated to a regular expression.

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 6.2 and in particular Figure 8 below show some of the regular expressions that were built from our analysis of collections in the standard Scala library.

6 Evaluation on Scala Library

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 58000 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 . 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 unrolled 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 extent arbitrary, we estimate that they correspond to reasonable expectations for the analysis to be useful. The different categories of effects form a lattice:

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

Figure 7 displays the number of summaries per category and per package. 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`	34810	46%	17%	29%	8%
`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%

Fig. 7. Decomposition of resulting summaries per package.

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

Fig. 8. Readable effect descriptions obtained from graph summaries from four operations performed on five kinds of collections.

6.2 Selected Examples

To demonstrate the precision of the analysis, we take a closer look at several 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 analyze code performing four operations, shown in Figure 9. Figure 10 shows functions corresponding to these four operations when applied to the TreeSet collection, and summarizes the general classes of operations.

1. Generic Traversal: call foreach with an arbitrary closure,
2. Pure Traversal: call foreach with a pure closure,
3. Impure Traversal: call foreach with a closure modifying the collection elements,
4. Growing: build a larger collection, by copying and extending it for immutable ones, or modifying it in place for mutable ones. The method used for growing depends on what is available in the public interface of the collection, e.g. add, append or prepend.

Fig. 9. Operations on containers used to evaluate analysis results

```
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
```

Fig. 10. The particular four operations applied on the TreeSet collection

The resulting effects are converted into a readable format, as described in Section 5 and displayed in Figure 8. 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 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 a recursive function with highly dynamic dispatch within its body. We can see however that thanks to context sensitivity, we are able to obtain precise results when the closure is determined. For impure traversal of TreeSet, the analysis had to generate and combine no less than 27 method summaries. The fact that the resulting effect remains precise despite the fundamental complexity of the library shows that the analysis achieves its goal of combining precision and modularity through summaries, even in the case of higher-order programs.

In the cases of impure traversals, the effects correctly report that all elements of the collections may be 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 functions have the intended effects.

7 Related Work

Our goals stand at the intersection of two long-standing fundamental problems:

1. effect and alias analysis for mutable linked structures; [8, 6, 16, 21, 31, 25];
2. control-flow analysis [29] for higher-order functional programs.

Because we have considered the heap analysis to be the first-order challenge, we have focused on adapting the ideas from the first category to higher-order settings. In the future we will also consider the dual methodology, incorporating recent insights from the second line of approaches [20].

The analysis domain presented in this paper builds on the work [27, 28], 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 4, was also made by Madhavan, Ramalingam, and Vaswani [18], 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 [19]. An important difference with our work is that [19] 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 [19] 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 [9, 33] 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 [15, 24, 5, 7, 4, 12, 2, 1]. What differentiates our analysis is that it is fully automated, but we do hope to benefit in the future from user hints expressing encapsulation, information hiding, or representation independence.

Separation logic [11, 3] and implicit dynamic frames [30, 23] are two popular paradigms for controlling modifications to heap regions. Nordio et al. describe an adaptation of dynamic frames [22] for the automated verification of programs with higher-order functions. 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 [26] for Scala, which requires some type annotations and is higher-level, but provides more control over encapsulation and elegantly balances the expressive power with the simplicity of annotations. The resulting system is fully modular and supports, e.g. separate compilation. In the future, we will consider using a system such as Insane as an automated annotation engine for the effect system, alleviating the bootstrapping problems that come with the annotation requirements.

8 Conclusion

Knowing the effects of program procedures is a fundamental activity for any reasoning task involving imperative code. We have presented an algorithm, a tool, and experiments showing that this task is feasible for programs written in Scala, a modern functional and object-oriented language. Our solution involves a general framework for relational effect analyses designed to support different automated reasoning strategies and allowing analysis designers to experiment with trade-offs between precision and time. Building on this framework we have introduced an abstract domain designed to track read and write effects on the heap. Combining the framework with the abstract domain, we have obtained an effect analysis for Scala. We have implemented and evaluated the analysis on the entire Scala standard library, producing a detailed breakdown of its 58000 functions by purity status. Finally, we have developed and implemented 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, facilitating program understanding, as well as subsequent deeper analyses and verification tasks.

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. J. 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)
4. Boyapati, C., Liskov, B., Shrira, L.: Ownership types for object encapsulation. In: POPL. pp. 213–223 (2003)
5. Cavalcanti, A., Naumann, D.A.: Forward simulation for data refinement of classes. In: Proceedings of Formal Methods Europe FME'2002. LNCS, vol. 2391 (2002)
6. Chase, D.R., Wegman, M.N., Zadeck, F.K.: Analysis of pointers and structures. In: PLDI. pp. 296–310 (1990)
7. Clarke, D., Drossopoulou, S.: Ownership, encapsulation and the disjointness of type and effect. In: OOPSLA (2002)
8. Cooper, K.D., Kennedy, K.: Interprocedural side-effect analysis in linear time. In: PLDI. pp. 57–66 (1988)
9. Cousot, P., Cousot, R.: Modular static program analysis. In: CC. LNCS, vol. 2304, pp. 159–178. Springer (2002)
10. Deutsch, A.: A storeless model of aliasing and its abstractions using finite representations of right-regular equivalence relations. In: Proc. Int. Conf. Computer Languages, Oakland, California. pp. 2–13 (1992)
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: 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. LNCS, vol. 6337, pp. 320–339. Springer (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. Larus, J.R., Hilfinger, P.N.: Detecting conflicts between structure accesses. In: Proc. ACM PLDI. Atlanta, GA (Jun 1988)
18. Madhavan, R., Ramalingam, G., Vaswani, K.: Purity analysis: An abstract interpretation formulation. In: SAS. LNCS, vol. 6887, pp. 7–24. Springer (2011)
19. Madhavan, R., Ramalingam, G., Vaswani, K.: Modular heap analysis for higher-order programs. In: SAS. LNCS, vol. 7460, pp. 370–387. Springer (2012)
20. 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)
21. 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)
22. Nordio, M., Calcagno, C., Meyer, B., Müller, P., Tschannen, J.: Reasoning about function objects. In: Vitek, J. (ed.) TOOLS (48). LNCS, vol. 6141, pp. 79–96. Springer (2010)

23. Parkinson, M.J., Summers, A.J.: The relationship between separation logic and implicit dynamic frames. In: ESOP. pp. 439–458 (2011)
24. de Roever, W.P., Engelhardt, K.: Data Refinement: Model-oriented proof methods and their comparison. Cambridge University Press (1998)
25. Rountev, A.: Precise identification of side-effect-free methods in java. In: ICSM. pp. 82–91 (2004)
26. Rytz, L., Odersky, M., Haller, P.: Lightweight polymorphic effects. In: ECOOP. LNCS, vol. 7313, pp. 258–282. Springer (2012)
27. Salcianu, A., Rinard, M.C.: Purity and side effect analysis for Java programs. In: VMCAI. LNCS, vol. 3385, pp. 199–215. Springer (2005)
28. Salcianu, A.D.: Pointer Analysis for Java Programs: Novel Techniques and Applications. Ph.D. thesis, Massachusetts Institute of Technology (2006)
29. Shivers, O.: Control-flow analysis in scheme. In: PLDI. pp. 164–174 (1988)
30. Smans, J., Jacobs, B., Piessens, F.: Implicit dynamic frames: Combining dynamic frames and separation logic. In: ECOOP. pp. 148–172 (2009)
31. Tkachuk, O., Dwyer, M.B.: Adapting side effects analysis for modular program model checking. In: ESEC / SIGSOFT FSE. pp. 188–197 (2003)
32. Whaley, J., Rinard, M.: Compositional pointer and escape analysis for Java programs. In: Proc. 14th Annual ACM Conference on Object-Oriented Programming, Systems, Languages, and Applications. Denver (Nov 1999)
33. Yorsh, G., Yahav, E., Chandra, S.: Generating precise and concise procedure summaries. In: POPL. pp. 221–234. ACM (2008)