Precise and Scalable Analysis of Programs with Callbacks

Etienne Kneuss
LARA, I&C, EPFL

Abstract—Our work is in the area of automated verification of software properties in modern programming languages. We present three papers related to various aspects of our research field: a modular pointer and effect analysis for object oriented programs, a precise and efficient typestate analysis, and a precise, parametric array content analysis. We discuss their contributions as well as their shortcomings. Finally, we present our recent work on analyzing software with complex, dynamic properties, and discuss ideas for future research.

Index Terms—effect analysis, static analysis, callbacks, scala

I. INTRODUCTION

The current world increasingly relies on computer software, and this trend shows no indication of slowing down. At the same time, we as customer have little guarantees on the reliability of the software we explicitly or implicitly rely upon. We can identify at least three causes for this: first of all, software engineering techniques are still in its infancy. For instance, best development practices are still religiously debated, and no clear standard has emerged. Secondly, certifying software nowadays is very difficult. On one hand, doing it manually has enormous cost and obvious limitations. On the other hand, automated techniques have yet to demonstrate that they are applicable and useful for general purpose software companies. The current status-quo seems to be based on trial-and-error which is ultimately best-effort and cannot be translated into meaningful guarantees.

Given that static verification and program understanding tools rely on problems that are generally undecidable, they often either restrict the verification to a limited set of properties for which a decision procedure is known, or rely on over-approximation to ensure (fast) termination. We can also witness that software grows rapidly, and often rely on external, large libraries. Consequently, static analysis techniques need to scale up to large code-bases in order to be useful in practice. Another important aspect of sound analysis techniques is their precision: over-approximating techniques need to be precise enough in order to limit the ratio of false-alarms. Spurious alarms result in human cost which needs to be taken into account in the overall usefulness of the analysis. We put aside unsound analysis techniques since, while they are useful for development purposes, they often fall short of providing meaningful guarantees.

Each of the three papers presented here advanced the state-of-the-art in automated verification or program understanding on different aspects, by either providing precise yet scalable analysis of certain properties of program, or by providing precise modular approaches which are known to scale well.

• The first paper, Purity analysis: An abstract interpretation formulation by Ravichandran Madhavan, Ganesan Ramalingam, and Kapil Vaswani [15] addresses the problem of detecting methods that have no observable side-effects with respect to the memory. This analysis is modular and thus is designed to scale to large code-bases. Additionally, modular analyses have yet to demonstrate that they are applicable and useful for general purpose software companies. They present an analysis based on abstract interpretation that uses graph-based summaries of memory effects.

• The second paper, Effective Typestate Verification in the Presence of Aliasing by Stephen J. Fink, Eran Yahav, Nurit Dor, Ganesan Ramalingam, and Emmanuel Geay[9] presents an interesting approach at precisely and efficiently tracking aliasing relations, a key requirement for typestate analysis. They propose a analysis combining multiple abstract domains to address the problem of weak-updates, and describe how this combined domains can be used to increase the number of destructive updates, providing precision improvement that is critical for typestate verification.

• The last paper, A Parametric Segmentation Functor for
Fully Automatic and Scalable Array Content Analysis by Patrick Cousot, Radhia Cousot, and Francesco Logozzo[6] describes a parametric analysis technique for precise yet scalable analysis of array contents. The analysis is able to fully-automatically partition arrays in consecutive and possibly empty segments. Values within the same segment by a single abstract value. Segments are delimited by bounds, themselves abstracted in a specific abstract domain. They instantiate this parametric analysis with various abstract domains to demonstrate how the parameters affect the performance and precision of the analysis.

In the final section of this report, we describe our recent work in the domain of static analysis of dynamic features of modern programming languages. We also indicate directions and ideas for future work.

II. Survey of Selected Papers

We now present the key ideas presented in the selected papers, their limitations and how we plan to build upon them in future research. The first paper describes recent work on a modular analysis for memory effects. The second paper describes a combination of several analyses of varying precision which are then used for typestate analysis. The two papers describe different approaches for their pointer analysis, a key component of most analyses targeted at object oriented languages, or more generally languages with support for pointers. The last paper describes a technique that allows for tractable and precise analysis of array content. While mostly orthogonal to the first two papers, it outlines the problem of analysing collections both precisely and efficiently and proposes interesting solutions.

A. Modular Effect Analysis

Effect analysis is a technique that statically establishes the side-effects of a procedure. A particular instance of effect analysis is purity analysis which computes the set of procedures having no side-effects. We can identify a variety of effect categories, like exceptional effects or memory effects. Here we only focus on memory effects in a Java-like object-oriented language and thus consider accesses and modifications to the heap via field reads and updates.

In [15], the authors presents an analysis domain for modular purity analysis that builds on the work of Salcianu, Rinard and Whaley [22], [19], [20], who used graphs to encode procedure effect summaries independently from aliasing relations. These graphs use nodes to represent objects, and edges to represent either read or write effects. Some of these nodes represent unresolved objects, corresponding the receiver of the method, its arguments or unknown field values. Handling method calls consists of merging the summary of the target methods into the current effect graph by first resolving nodes when possible.

One of their main contribution is a reformulation of the analysis using abstract interpretation. Originally, the graph-based procedure summaries were interpreted as sets of heaps, requiring an intricate definition of composition which is key to modular analyses. Instead, [15] describes them as state transformers with relational properties which provides natural composition.

Another contribution described in the paper is optimization strategies for the generated graphs. Indeed, they observe that the generated graphs have a tendency to grow rapidly in size causing performance problems. They describe several optimisations done to the structure of the generated graphs and prove them sound. They then argue the benefits of the optimisation by comparing them in the evaluation section.

There exists a close relationship between memory effects and aliasing relations. Indeed, both are mutually dependant: updating fields might update aliasing relations, and aliasing relations affect how a field update might modify the heap. The graph-based representation carries information on both domains. We note that the precision of the described analysis is relatively low in its tracking of destructive updates. Indeed, strong-or destructive updates will only be permitted if the object on which a field is modified is known to be unique under a set of very strict rules, which will in practice prevent destructive updates in most cases. While weak updates may be sufficient in the case of purity analysis, strong-updates become essential for general effect analysis or when used to enable further analyses, such as typestate analysis. Indeed, the precision of tracking the absence of effects does not improve much when increasing the number of strong updates. However, precisely tracking pointers relations and describing the modifications to the heap is very much dependant on the ability to perform destructive updates as often as possible.

Another important limitation is the lack of precision for procedure calls with a large number of potential targets. This case typically occurs in the presence of callbacks or higher-order functions, recognized as a pillar of functional programming but also becoming a standard feature of object-oriented languages such as C# (in form of delegates), the 2011 standard of C++, and Java 8. Moreover, design patterns in object-oriented programming community also rely on callbacks, especially the strategy pattern and the visitor pattern [10]. Their analysis will in fact handle every potential targets, yielding overly imprecise summaries: some higher-order functions present in the Scala library have calls with more than 1’000 potential targets.

We note that they recently extended their analysis with special support for higher-order procedures [16] using mechanisms that delay the analysis of imprecise procedure calls, and is very similar to our current work. The evaluation section presented in [15] gives little insight to the reader on how well the overall analysis is performing. First of all, it only compares the various optimizations implemented in the analysis. Secondly, we don’t know the actual number of pure methods present in these benchmarks, preventing us from establishing its precision. Thirdly, these analysis are very sensitive to hidden assumptions. We note for instance that based on their evaluation results the extended version of their analysis finds less pure methods than the original one on the same benchmarks. This is unexpected given the nature of the extension, and it seems to indicate that some assumptions changed in their analysis between the two runs.

B. Typestate Analysis

Typestate analysis consists of checking whether objects are used according to predetermined usage rules, specified as sequences of method calls. The set of valid usage scenarios are typically encoded using a finite state machine, where transitions represent method calls. Typestate analysis then consists of checking that objects will not reach an error state. Typical examples of usage rules include IO-related objects, for which reading and writing should not occur before the resource has been opened. Another example often seen in the literature is the safe usage of iterators: `next()` should not be called before `hasNext()`.

In this paper[9], the authors present an interesting approach at precisely and efficiently tracking aliasing relations, a key requirement for typestate analysis. Since objects allocated on the heap are abstracted depending on their allocation site, a simple conservative assumption is to only allow weak-updates as this abstracted object may in general represent multiple objects. This however would lead to many spurious warnings, and would preclude the verification of interesting typestate properties. Indeed, no object would have their state definitively updated. The authors note that this would however be sufficient for typestate properties that are omission closed[8].

They propose a staged analysis combining multiple abstract domains to address the problem of weak-updates, and describe how this combined domains can be used to increase the number of destructive updates, providing precision improvement that is critical for typestate verification. The main component enabling strong updates is a simple uniqueness analysis, and could be seen as a simple form of recency abstraction[1]. In order to allow a destructive update, two conditions must be met: 1) the variable on which the update occurs must point to a single abstract object 2) the abstract object must represent a single concrete object. While the first condition can be established easily, the second condition is where their uniqueness analysis comes into play: the main idea is to detect allocation sites (which map directly to abstract objects) that are used in loops, thus potentially yielding more than one concrete object. They use a simple data-flow analysis which tracks allocated objects: if, at an allocation site $\ell$, the corresponding abstract object $o_2$ can be found in the incoming flow, the abstract object is flagged as non-unique. They pair this uniqueness analysis with a liveness analysis, allowing to remove dead objects from the flow.

Their second contribution is the use of a `focus` operator similar to what can be found in shape analysis[18]. The focus operation splits the state according to both outcomes of weak-update and also tracks aliasing conditions for each of them (in terms of must-point-to and must-not-point-to). We illustrate the use of the focus operation on the following code and typestate property displayed as a state machine in Figure 1.

1. $\ell_1: \text{e.lock()}$
2. $\ell_2: \text{e.unlock()}$

Assuming we have an object $o$ in typestate `unlocked` which $e$ may point to, a weak update of the typestate of $o$ would normally take place at $\ell_1$. Instead, `focus` will split the facts about $o$ in two: $o_1$, recording that it must alias $e$ and $o_2$ recording that it must not alias $e$. A strong update on the typestate can then be applied to $o_1$ only, leaving $o_2$ untouched. Similarly, a strong update will be applied at $\ell_2$ on $o_1$ which will set its typestate to `unlocked`. We can see that the focus operation prevents a spurious warning. Indeed, without focus $e.unlock()$ would be applied on $o$ that may be in state `unlocked`, which is invalid.

Implicitly, the focus operation implements some sort of path-sensitivity. As such, it can be seen a specific case of fluid updates[7].

An immediate problem of the focus operation is that while it is precise, the splitting causes exponential blowup of the number of facts kept in scope. They however present a technique that unifies similar states, mitigating this blowup. However, we note that this blow-up still exists in the presence of dynamic dispatch with a large number of potential targets.

C. Array Content Analysis

Arrays have always been problematic to handle in static analyzers: they serve as storage for heterogeneous objects indexed by potentially dynamic keys. Several different approaches have been developed to solve this problem:

One one side of the spectrum, we have the technique of expanding array into individual local variables. Which in terms of abstract interpretation equates to keeping an abstract value for every array element. While very precise, this approach is in general so costly that it makes the overall analysis intractable. The extra cost of this approach is is especially apparent if multiple elements of the array share common properties.

On the other side of the spectrum, we have the idea of collapsing all array element into one abstract value. This approach is very efficient since this approximation is independent from the array size. However, it has a very limited precision and will not allow the analysis to prove most interesting properties. The main precision problem comes from the fact that while this value represents every array elements, typical programs will only update arrays one element at a time, resulting in only weak-updates. This is especially troublesome when updates are iteratively applied to all array elements: despite simple and explicit code patterns, analyses using this technique will fail to establish that all array elements are eventually updated.

We provide in Figure 2 a code example illustrating a case for which both approaches would be suboptimal. We see that array collapsing would give us that the array is filled with either -1 or 1, leading to a spurious warning when updating it at $\ell$. On the other hand, tracking individual elements would eliminate that false-positive but at the same time, the array would be extremely costly to handle as we would not be able
EDIC RESEARCH PROPOSAL

val a = new Array(100)
for (i = 0; i < 50; i++) {
    a[i] = -1;
}
for (i = 50; i < 100; i++) {
    a[i] = +1;
}
val tmp = a[55];
ℓ: a[tmp] = 4;

Fig. 2. Code example using an array illustrating the suboptimal nature of both array expansion and collapsing.

to exploit information on the structure of the array, which is explicit in the code.

In between, we can find a variety of examples, like array partitioning [11]. Several array content analyses are not fully automatic and require user intervention, either by specifying usage templates [12], or by specifying most loop invariants [3], [4].

In their paper [6], they present a parametric analysis based on abstract interpretation that is fully-automatic and allows for precise array analysis, even in the context of non-trivial array accesses in loops. The key idea is to partition the array into consecutive (possibly empty) segments, each abstracted by a single abstract value. Segments are delimited by bounds from a different abstract domain, representing symbolic expressions of a certain shape. The symbolic aspect of the segment bounds allow them to track dynamic accesses and modifications to the array.

The presented technique makes several assumptions about the ways arrays are accessed. Indeed, the abstract domain used to represent symbolic segment bounds is well suited for direct array accesses, where the index is computed using simple expressions. In our experience with the Scala, arrays are discouraged for direct use. Instead various collections implemented in the Scala library are often preferred. However, while it is true that some of these collection make use of arrays internally, there is often no semantic relationship between keys and elements. For instance, hashmaps and hashsets both use arrays as internal storage, but analyzing the datastructure’s content at the array level is bound to fail. Instead, we believe that datastructures should be abstracted at a higher level in order to allow key-value relationships.

III. CURRENT AND FUTURE WORK

In the area of static program analysis, we studied scalable and precise analysis for complex features of modern programming languages.

We developed a static analysis for inference and verification of types in PHP, a language known for its dynamic features and therefore lack of any static guarantees. The analysis reconstructed types of variables based on their assigned values and usage and detected potential conflicts. It is worth noting that PHP is dynamically and weakly typed. On top of that, it provides implicit conversions that can silently coerce values into almost any type. This results in silent type errors, leading to unexpected behaviors. Our analysis adopted a stricter type system and tried to detect invalid type conversions. PHP is a good example of languages that are difficult to analyze. Indeed, it supports many dynamic features, like variable-variables (variables named after the value of another variable), dynamic code inclusion, dynamic function calls, and of course eval (). PHP exposes almost no type information. It ships with a big library of functions (4000+) for which the PHP code source is unavailable, as they are implemented in C. To illustrate, we present in Figure 3 one of the problem that our analysis was able to uncover in Dokuwiki, a widely used wiki application.

We further extended our analysis to leverage runtime information in order to improve the analysis results of mostly unannotated PHP applications, allowing us to bypass the often complex phase of configuration and initialization of PHP applications, hence focusing on potential type errors occurring within the application’s core.

The second part of our recent work is a modular memory-effect analysis for programs with callbacks. 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. We developed a modular effect analysis using graph-based procedure, similar in shape to the graphs presented in [15]. We however reformulated the semantics of the summaries to support aggressive use of destructive updates. We developed a framework based on abstract interpretation that, with the use of summary statements and a combination operator, lifts existing analyses to support functions with callbacks by delaying imprecise calls automatically. We instantiated this framework with our effect analysis. The idea of delaying parts of the analysis has been explored before in interprocedural analyses to improve context-sensitivity [5], [23] or to speed
EDIC RESEARCH PROPOSAL

up bottom-up whole-program analyses [14]. Our current work shows that this approach also brings benefits to the analysis of programs with callbacks, and is in fact critical to its applicability. In essence, the delaying is similar to the modular analysis presented in [16]. One of the notable differences is that they summarizes higher-order functions using only CFGs of a particular, fixed, normal form: a loop around the unanalyzed 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 optimistic 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 [16] relies on graph transformers being monotonic, a property that is incompatible with strong updates.

It is worth noting that we currently use array collapsing to represent array contents: we use a single abstract value to represent elements of an array. This is an obvious source of imprecision. It is however not clear whether approaches discussed in [6] would work here. Indeed, Scala’s use of array in collections is mostly internal to the implementation, and there is several layers of indirection between the initial access to the collection and the resulting access to the array. It is not unlikely that the analysis domains for symbolic segment bounds is sufficient to handle those cases.

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. [17] as well as lead to the design of more precise versions of such systems. Our static analyzer, called Insane, is publicly available. We have evaluated it on the full Scala standard library, which is widely used by all Scala programs, and is also publicly available.

Thorough those projects, we have acquired considerable experience in the design and implementation of precise and scalable static analysis techniques for dynamic languages. Based on those early results, we can identify multiple directions for future work:

a) Modular Typestate Verification: an immediate possibility is to extend the modular effect analysis to support pre-conditions in order to support the modular verification of typestate properties. Given the precision of the effect analysis, we expect the typestate analysis to be relatively precise. However, our current representation of aliasing does not support the focus operation, and this may fail to precisely handle certain code patterns.

b) Inference of Data-structure Encapsulation: an important step towards reasonable effect summaries is a specific handling of encapsulated properties. Indeed, abstracting away internal representations of certain data-structures is key to generating meaningful summaries. In terms of aliasing, recursive data-structures often require conservative assumptions, leading to imprecise results. It would be interesting to explore approaches such as dynamic frames[2], [21] to reduce the size of effect summaries while requiring as little manual annotation as possible. We expect this approach to not only increase the scalability of our effect analysis but also improve the readability of our generated summaries.

c) In-code Transactions: If is often hard to guess in advance whether some piece of code will break. Moreover, recovering from errors occurring in imperative programs is even more difficult. Indeed, part of the state might have been wrongly modified prior to the detection of the problem. For example, given a function with pre and post conditions, we can detect at runtime whether the postcondition does in fact hold. In case it does not, we can abort the entire method, leaving us with the state of the execution before the function call. This feature commonly seen in transactions can be implemented using our precise effect analysis:

```scala
def foo () {
  transaction {
    bar ()
    gee ()
    if (went_bad) abort ();
    plop ()
  }
}
```

Based on the effect summaries of bar() and gee(), we will infer the region of the heap that may have been modified within the transaction block. This would dramatically reduce the size of the necessary state snapshot, and thus allow for an efficient implementation of transactional code. Such optimizations have proved useful in software model checkers such as Java Pathfinder [13], and we expect them to be useful as a general-purpose programming mechanism as well.

REFERENCES


Insane stands for “Interprocedural Static ANalysis of Effects” and is available at https://github.com/colder/insane


