Lattice Based Modularization of Static Analyses

M. Eichberg, F. Kübler, D. Helm, M. Reif, G. Salvaneschi and M. Mezini
Technische Universität Darmstadt
Germany
<eichberg,kuebler,helm,reif,salvaneschi,mezini>@cs.tu-darmstadt.de

Abstract

Static analyses which compute conceptually independent information, e.g., class immutability or method purity are typically developed as standalone, closed analyses. Complementary information that could improve the analyses is either ignored by making a sound over-approximation or it is also computed by the analyses but at a rudimentary level. For example, an immutability analysis requires field mutability information, alias/escape information, and information about the concurrent behavior of methods to correctly classify classes like `java.lang.String` or `java.util.BigDecimal`. As a result, without properly supporting the integration of independently developed, mutually benefiting analysis, many analyses will not correctly classify relevant entities.

We propose to use explicitly reified lattices that encode the information about a source code element's properties (e.g., a method's purity or a class' immutability) as the sole interface between mutually dependent analyses. This enables the composition of multiple analyses. Our case study shows that using such an approach enables highly scalable, lightweight implementations of modularized static analyses.

Keywords Static Analysis Framework, Modularization, Lattice, Abstract Interpretation

1 Introduction

The development of new static analyses is a challenging task supported by many frameworks – in particular in the domain of data-flow analyses, e.g., [18, 20]. A crucial step when designing new analyses is to decide which information will be used; this decision determines the analysis' precision and recall. Basically all analyses benefit from using information not belonging to their primary domain. For example, an immutability analysis can benefit from escape/alias information but also from purity information. Yet, current approaches, e.g., [14], do not support the integration with other state-of-the-art analyses which compute independent properties and, therefore, either ignore such information at all or support some basic cases as part of the primary analysis.

Integrating mutually dependent analyses requires to solve several challenges. First, mutual dependencies lead to fix-point computations. Second, some specific properties may create cyclically-dependent computations. Such cycles – that may span across different analyses – must be detected and solved with a sound and precise resolution strategy. Third, integrating multiple analyses requires to identify the parts for which some information is actually required: computing all information would significantly limit the scalability. I.e., analyses should be demand-driven [1, 21]. Finally, assessing the contribution of each analysis towards the overall goal requires means to (de)activate certain analyses which makes their strict modularization an essential property.

To solve the challenges above, we propose a principled approach based on lattices as the foundation for modeling the properties associated with source code elements. Our solution facilitates a lightweight implementation of strictly modularized analyses that collaboratively compute complex information. That is, the analyses have no direct dependencies and can be developed independently. Further, we enable the use of analyses with different precision/performance trade-offs and, therefore, encouraging a thorough performance assessment of analyses. The analyses are lightweight in the sense that (1) they can compute information for one specific entity at a time without analyzing other entities, e.g., the set of transitively thrown exceptions for a specific method can be computed without having to traverse the call graph, and (2) the analyses do not have to deal with the issues of cyclic dependencies or fix-point computations.

2 Lattice Based Modularization

In our approach, analyses are two functions which, together, compute an information of a specific kind for a specific entity, e.g., a method's purity level, an allocation site's escape information, the exceptions thrown by a method, or the mutability of objects. A finite height lattice for each specific kind of information determines all possible extensions and their relation. We use the lattice's top value to model the best value and the bottom value for the sound over-approximation. For example, the top of the lattice encoding a method's purity [8] is Pure and the bottom is Impure. In the following, we say that *an analysis computes a property belonging to a specific (property) kind*. Furthermore, bottom elements are used as the fallback properties if no analysis, which computes a respective property, is scheduled.

Listing 1 gives the Scala definition of the `analyze` function that collects the information to compute a desired property

ISSTA Companion/ECOOP Companion'18 , July 16–21, 2018, Amsterdam, Netherlands
2018. ACM ISBN 978-1-4503-5939-9/18/07…$15.00
https://doi.org/10.1145/3236454.3236509

for a specific entity; e.g., in case of a purity analysis it collects the called methods' purities and the immutability of parameters.

```
1 type OnUpdateContinuation =
2       (Entity, Property, State) => ComputationResult
3 type Dependees = Traversable[(Entity, PropertyKind)]
4 type ComputationResult =
5       (Entity, Property, Dependees, OnUpdateContinuation)
6 def analyze(e: Entity): ComputationResult
```

Listing 1. Core definitions used by the framework

The `analyze` function returns for a source code element (`Entity`) the first result (`Property`) along with the list of required information (`Dependees`) and the second (callback) function (`OnUpdateContinuation`) that processes the queried properties when they become available or are updated. For example, consider determining if the method `getA` in the Java snippet in Listing 2 is side-effect free and deterministic.

```
1 class X {
2   private int a; // (effectively) final
3   public X(int a) { this.a = a; } /*<=pure*/
4   public int getA() { return a; } /*<=pure*/ }
```

Listing 2. Analyzed Java code

The `analyze` function will identify that it is necessary to know if the field `a` is (effectively) final to decide if the method is deterministic or not. Hence, it returns (1) that information about the field `a`'s mutability is required along with (2) the `OnUpdatedContinuation` function that, given the respective information, will continue computing `getA`'s purity. The latter function returns Pure if the field is (effectively) final. This result is considered final by the framework if `Dependees` is empty; in that case the `OnUpdateContinuation` is ignored.

In our model, the `OnUpdateContinuation` functions, that compute a property using other properties, return the best possible property until a counter example is found. In other words, the results of an analysis w.r.t. a specific entity usually starts with the top value of the lattice and then falls down the lattice. All `OnUpdateContinuation` functions have to satisfy the following requirements:

Monotonicity Whenever the function returns from computing a property for a specific entity, the property must be the same as before or must be *below* the previous result w.r.t. the underlying lattice. This is achieved using the *meet* operation of the lattice on the previous result and the best possible result considering just the current update.

Always Sound Over Approximation The returned result must consider all past and current information that was passed to it and the result must be a sound over-approximation w.r.t. this information. For example, if a method `nd`, called by a method `s`, is not deterministic then the analysis for `s` is no longer allowed to report that `s` is pure. Instead, it must return side-effect free or impure.

In the model, *entities* are either concrete code elements, as in the previous example, or artificial entities such as the call graph, allowing attaching properties to entities for which no source code is available. Requiring that all inter-analysis communication is exclusively done by querying the current state of properties and by returning `ComputationResults` enables strict modularization, i.e., the complete decoupling of the code bases belonging to different analyses.

```
1 class PropertyStore {
2   def apply(Entity, PropertyKind): (Entity,Property,State)
3   def schedule(f: (Entity) => ComputationResult) : Unit
4   def waitOnCompletion() : Unit }
```

Listing 3. Core methods of the PropertyStore

The so-called `PropertyStore` implements the functionality to execute analyses and to provide information about properties. To query a property, the `apply` method (cf. Listing 3) is used. It will return the current property for the given property kind and entity. The return value also specifies if the property is final or may be updated. The latter is crucial to enable clients to also commit final results which – in turn – makes it possible to clean up the state required to notify analyses that depended on some information. This prerequisite is necessary for scalability.

To execute an analysis, it is necessary to `schedule` the functions that compute the properties. When all initially and subsequently scheduled computations have finished, we check if all queried properties were computed by some (scheduled) analysis. If not, the fallback properties are used. If no fallbacks are required, cyclic computations are identified and resolved. The analysis finishes if all scheduled computations have finished, no more fallbacks have to be used, and no more cycles are found. By calling `waitOnCompletion` it is possible to wait until the fix-point is reached.

Cyclic Computations In our approach, computations may form a cyclic dependency and explicit support is provided to detect and resolve those. This is possible because the lattices have to be of finite height and updates are required to be monotonic.

```
1  class C {
2    static final Point origin = new Point(0,0); /*<=final*/
3    static Point getOrigin() { return origin; } /*<=pure*/ }
4  class Point { // Point is immutable
5    private final int x, y; // trivially final
6    private int h; // h is lazily initialized
7    Point(int x, int y) { this.x = x; this.y = y; }
8    public int hashCode() { // pure
9      if(h == 0) h = C.getOrigin().x-x + C.getOrigin().y-y;
10     return h; } }
```

Listing 4. Java example leading to cyclic computations

For example, in Listing 4, to assess whether the method `hashCode` is deterministic, it is necessary to know whether the `Point` returned by `getOrigin` is guaranteed to be structurally equal each time this method is called. As `origin` is declared final, this is true if objects of type `Point` are immutable. While `x` and `y` are primitive final fields, proving that `h` is effectively immutable is not trivial: we have to prove that `h` is lazily initialized to a deterministic value. One way to prove this, is to evaluate if `hashCode`, the only method that writes `h`, is deterministic – the question we started out to assess originally. Thus, the determinism of `hashCode` and `getOrigin`, the mutability of `Point`, and whether `h` is lazily initialized to a deterministic value leads to a cyclic dependency among three different property kinds.

As every analysis is required to return a result that is a sound (over-)approximation of currently available information, cycles can simply be resolved by the `PropertyStore` by committing an arbitrary element belonging to the cycle as final. This will then trigger other elements of the cycle to also be committed as final. For example, in the above case, it does not matter whether we consider `h` as immutable or `hashCode` as deterministic – in both cases, the overall results will be that `h` is found to be immutable and `hashCode` to be deterministic.

Cycles are found by computing the closed strongly connected components, i.e., by computing those sets of computations with two or more elements that have no outgoing dependencies to computations which do not belong to the component. Making this restriction is essential to avoid committing properties prematurely when we reach quiescence.

3 Evaluation

The proposed approach is implemented as part of the OPAL project[1]; a static analysis framework for Java bytecode implemented in Scala. We evaluate our approach with a case study. We have modeled ten different property kinds related to classes, fields, methods, and allocation sites (cf. Figure 1). The property kinds provide information eventually used to compute a method's purity. We aim to answer the following questions:

RQ1 Does the approach enable effective modularization of static analyses such that individual analyses with different performance/precision trade-offs are exchangeable?

RQ2 Does the approach facilitate the assessment of individual analyses w.r.t. their contribution on a higher-level analysis goal?

RQ3 Does the approach lead to analyses that are competitive with the state-of-the-art?

We implemented a lattice for each of the ten property kinds in Figure 1. Additionally, we have implemented 14 analyses each deriving exactly one property kind. For three of the property kinds (*Purity*, *Escape*, and *Field Mutability*) we implemented multiple analyses with different precision/performance trade-offs.

3.1 Property Kinds

3.1.1 Properties of Specific Code Elements

Next, we discuss properties which are related to one specific source element, such as a specific class or allocation site:

Field Mutability Specifies whether a field is Effectively Final or not; i.e., whether clients reading the field's value will always see the same value after initialization of the object has finished. Fields which are explicitly declared final, which are just set in a constructor, or which are lazily initialized are considered effectively final.

Class Immutability Specifies the immutability of objects of a concrete class `C`. This lattice has three notable elements: Mutable, Immutable Container, and Immutable. The first identifies those objects which are mutable. Mutable is also the fallback value. Immutable is used for classes where the transitively reachable state cannot be mutated; e.g., `java.lang.Integer`. An Immutable Container class is not directly mutable; i.e., all fields are effectively final, but the transitively reachable state is potentially mutable; immutable collections are prototypical examples of immutable container classes.

Object Escape Describes the lifetime and the accessibility of newly created objects or parameters to determine in which way the value escapes the current scope [3, 13]. The escape information is associated with the respective allocation site (i.e., the `NEW` or `NEWARRAY` instruction which created the object) or formal parameter. No Escape is used for objects that are only accessible from within their creator method. Escape In Callee is used for objects that are passed to methods that do not change the escape state; i.e. the object's lifetime does not exceed the lifetime of its creating method. For objects that leave the scope of their creating method, e.g. via a return or throw statement, or via an assignment to a field of a parameter, the states Escape Via Return, Escape Via Abnormal Return, and Escape Via Parameter are used. Finally, if another thread can get access to an object, e.g. via a static field write, the escape states Escape Via Static Field, and Escape Via Heap Object are used. In case of an unrestricted escape the fallback value Global Escape is used.

Return Value Freshness Characterizes how reference values returned by a method may have escaped. Fresh Return Value is used for methods that always return a newly created object that does not escape by other means than a normal return. Instance methods returning reference values stored in their receiver object's (typically `private`) local fields (cf. Field Locality) are considered as Getters.

Field Locality Characterizes how values referenced by fields may escape their owning instance. Fields holding fresh objects that do not escape their owning instances are Local

[1] www.opal-project.de

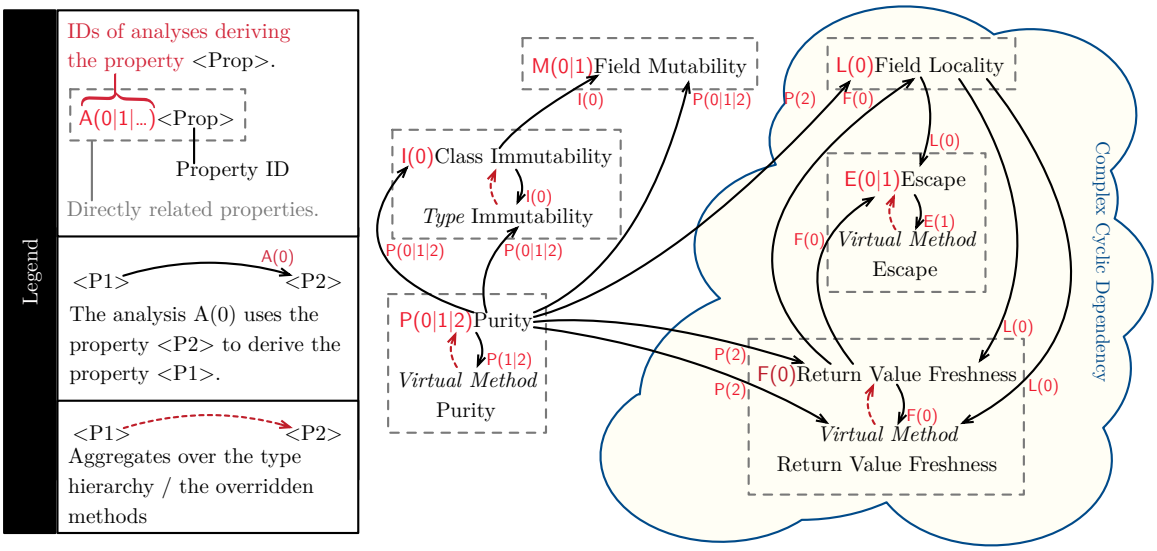

Figure 1. Supported property kinds and their usage by analyses.

Fields [17]. If instances of a subclass leak a field's value, the field is considered an Extensible Local Field. If a field's object only escapes by being returned, the field is a Local Field With Getter. A field with both properties is an Extensible Local Field With Getter.

Purity Specifies whether a method has side-effects and, if not, if it is also deterministic. Deterministic and side-effect free methods are Pure; e.g., `Math.abs(int i)`. If the method is *non-deterministic* but still side-effect free it is Side Effect Free; e.g., `System.currentTimeMillis()`. A Contextually Pure method is pure depending on the calling context, in particular if all parameters are freshly allocated. Additional elements of the *Purity* lattice exist that capture the absence of allocations and for methods that perform user-defined domain-specific actions, such as logging.

3.1.2 Properties Abstracting over Multiple Code Elements

The following property kinds describes properties which abstract over several entities, e.g., a class and its subclasses.

Type Immutability Specifies the immutability for all instances of a specific type; i.e., it abstracts over a specific class as well as all subclasses. For example, while an instance of the class `java.lang.Object` is Immutable, instances of the type `Object`, which abstracts over all types, are Mutable.

Virtual Method Purity Specifies the purity of a method by abstracting over the method itself as well as all methods which override the method. For example, the method `Object.hashCode` is Pure. Yet, some overriding methods perform lazy initializations which are not thread-safe. Therefore the *Virtual Method Purity* is Impure Virtual Method.

Virtual Method Return Value Freshness Specifies the freshness of a return value by abstracting over the method itself as well as all methods which override the method.

Virtual Method Escape Specifies the escape level of a parameter of a method by abstracting over the method itself as well as all methods which override the method.

3.2 Static Analyses

We have implemented 10 basic analyses (cf. Figure 1) in OPAL to compute the different properties (three for the purity of methods: P0,P1,P2; one for the immutability of classes; two for the mutability of fields: M0, M1; two for escape information: E0, E1; one for the freshness of return values: F0; one for field locality: L0. A higher number is used for more precise/more demanding analyses.)

The P0 purity analysis does a linear sweep of a method's code and uses field mutability as well as class/type immutability information. P1 uses the same properties as P0 but performs additional control- and data-flow analyses. P2 uses escape information and also contextual information at call-sites. The immutability analysis (I0) just checks if all fields are effectively immutable. The field mutability analysis (M0) checks if a class' private static fields are only written by the declaring class. The (M1) analysis performs data- and control-flow analyses to also support non-public instance fields. The E0 escape analysis is an intra-procedural escape analysis. E1 is inter-procedural. The F0 analysis determines whether the return value of a method is guaranteed to be freshly allocated and non-escaping. For that it primarily aggregates escape information. For values retrieved from fields, it also uses field locality information. The field locality analysis (L0) uses escape and return value freshness information to determine if all writes to a field write are fresh, non-escaping values and whether no value read from the field escapes.

We have implemented four further analyses which aggregate properties across subclasses / overriding methods. For example, independent of the scheduled purity analysis

Analysis configuration	P2/**E1**/F0/L0/M1/I0	P2/**E0**/F0/L0/M1/I0	P2	P0/M0/I0
Pure	52 628 (20.78%)	52 602 (20.77%)	49 849 (19.68%)	11645 (4.60%)
Side Effect Free	32 951 (13.01%)	32 964 (13.01%)	35 654 (14.08%)	– –
Contextually Pure/Side Effect Free	11 614 (4.59%)	11 459 (4.52%)	11 173 (4.41%)	– –
Impure	156 089 (61.63%)	156 257 (61.69%)	156 606 (61.83%)	241456 (95.40%)
Relative execution time	100%	100%	75%	15%

Table 1. Purity results (*absolute number and proportion of all methods*) for different analysis configurations

(< *None* >,P0,P1,P2), we always used the same aggregating analysis to compute *Virtual Method Purity*. In all cases these aggregating analyses basically *just* implement meet operations over the underlying lattices and are very fast. Therefore, only a single implementation is required.

3.3 Executing Different Analysis Schedules

We executed four different configurations of our purity analyses on the Oracle JDK 8 Update 151 to test if the modularization is effective and enables the trivial exchange of analyses and also the assessment of their contributions to the overall analysis goal. Three configurations execute the *P2* purity analysis. The first one uses the best supporting analyses available. The second one uses the weaker (intra-procedural) *E0* analysis instead of the inter-procedural *E1* analysis. The third one does not use supporting analysis at all; in that case all respective queries just return their fallback values. The final configuration evaluates the *P0* purity analysis with the best supporting analyses it can use.

We scheduled all aggregating analyses to compute the properties which abstract over sets of elements whenever we scheduled a corresponding analysis. The results for determining the methods' purity as well as the relative runtimes when compared to each other are given in Table 1.

3.4 Evaluation Results and Discussion

Modularization of Analyses (RQ1) Based on the case study, we can conclude that the approach supports an effective modularization of analyses, where each analysis computes a single well-defined property kind. The analyses are also lightweight in the sense that each one is implemented such that it analyzes each entity in isolation.

Crucially, the analyses are also easily exchangeable and reusable. Exchanging a more precise analysis for a faster one is a simple configuration matter. This result enables fine-tuning the trade-off between an analysis' precision and performance to specific use cases.

Assessing the Contribution of Individual Supporting Analyses (RQ2) In our study, exchanging the *E1* escape analysis for *E0* results in negligible differences, suggesting that a simple, intra-procedural escape analysis is sufficient to support our purity analysis. However, the performance overhead is basically none for the inter-procedural *E1* escape analysis. Therefore, it is still possible to use it and to get better results.

Program	Batik	Xalan
ReIm		
Side Effect Free methods	6 072 (37.88%)	3 942 (37.95%)
#Analyzed methods	16 029	10 386
OPAL		
Pure methods	4 009 (25.20%)	2 492 (23.15%)
Side Effect Free (incl. Pure) methods	6 780 (42.61%)	4 390 (40.79%)
Contextually Pure/SEF methods	987 (6.20%)	748 (6.95%)
#Analyzed methods	15 911	10 763

Table 2. Purity results for Batik/Xalan

Not executing any supporting analyses leads to 2779 methods (≈ 5.3% of those identified by the best analysis configuration) being just Side Effect Free instead of Pure. The decreased execution time by about 25%, however, suggests that relying on sound fallback values – instead of executing the supporting analyses – may be preferable for use cases that do not require precise identification of deterministic methods. Similar to exchanging the escape analyses, it is possible to evaluate the effect of individual supporting analyses in order to fine-tune the precision/performance trade-off to the specific use case.

The *P0* purity analysis is significantly less precise than any previous configuration. It identifies less than 5% of all methods as Pure compared to ≈ 20%. This analysis also does not identify Side Effect Free or Contextually Pure methods. With an 85% reduced execution time - compared to the most precise configuration - this may still be a viable configuration if the low precision is sufficient.

Based on the results, we conclude that our approach enables assessing the contribution of individual analyses w.r.t. their precision/performance trade-off.

Enabling Competitive Analyses Implementations (RQ3) As a final step, we compared our best configuration (*P2 with best supporting analyses*) to the state-of-the-art in analyses for side-effect free methods, ReIm [10, 11]. Table 2 shows that our analyses outperform ReIm on two medium sized open-source applications: Batik and Xalan. Both, precision (we identify more purity levels than just Side Effect Free) and recall (we identify over 40% of as Side-Effect Free compared to less than 38%) have improved. This result demonstrates that the combination of multiple analyses – enabled by our approach – provides better precision results compared to previous work.

4 Related Work

Lattices are at the core of many formal methods – in particular abstract interpretation [4] – and our approach relies on the respective mathematical foundations regarding the composition of lattices and analyses [16]. The goal of the proposed approach is to provide a practically usable, scalable, and sound framework that enables the modularization and exchange of static analyses such that it is easily possible to reason about their correctness as well precision effects in the presence of mutual dependencies.

Magellan [6] is an open framework for parallelizing the execution of collaborating static analyses. Compared to the proposed approach the dependencies are specified by the data that is processed and provided.

Several frameworks were proposed that rely on DSLs for developing static analyses. E.g., Klint et al. [12] discuss an approach that facilitates writing static analyses that support code rewritings. They do not address the modularization of exchangeable static analyses. Datalog, e.g. used as the foundation of DOOP [2], is successfully used to build frameworks for points-to analyses that enable to build various respective analyses with different precision/performance trade-offs. Compared to our approach, they are highly specialized. Another example of a framework specialized for data-flow analyses is IDE/IFDS [18, 20], discussed, e.g., in [19]. In [5] the use of Prolog is discussed as a foundation for a framework that supports the automatic incrementalization of queries in case of updates to the underlying code base. An alternative approach that relies on Scala is presented in [15]. Supporting incrementalization is not in scope for this approach due to its inherent limitations w.r.t. the scalability which is a primary target of our approach. Attribute grammars can be used to infer program properties in a modular way as used e.g. in JastAdd [7]. They are used for basic compilation tasks,such as name and type analysis, and not for complex static analyses based on abstract interpretation results. Lerner et al. [14] propose a framework for mutually benefiting static analyses that communicates analyses results through code optimizations. Our approach allows use of analysis results even if they can not be used for optimizations directly. In [9] a lattice-based approach for scheduling and executing concurrent tasks is discussed and also applied to static analyses. Compared to them, in our approach dependencies are *just* specified by the analyses and then automatically managed.

5 Conclusion

We propose to use explicitly reified lattices, which represent the information derived by static analyses, as the exclusive interface between modularized analyses used to exchange results. Our evaluation shows that the design of generic lattices is highly effective to achieve fine-grained reusable and composable analyses. Further, our approach enables assessing very precisely the effect of analyses with different precision/run-time trade-offs w.r.t. a certain goal.

Acknowledgments

This work was supported by the DFG as part of CRC 1119 CROSSING, by DFG grant SA 2918/2-1, by the German Federal Ministry of Education and Research (BMBF) as well as by the Hessen State Ministry for Higher Education, Research and the Arts (HMWK) within CRISP.

References

[1] S. Arzt, S. Rasthofer, C. Fritz, E. Bodden, A. Bartel, J. Klein, Y. Le Traon, D. Octeau, and P. McDaniel. 2014. FlowDroid: Precise Context, Flow, Field, Object-sensitive and Lifecycle-aware Taint Analysis for Android Apps *(PLDI)*.

[2] M. Bravenboer and Y. Smaragdakis. 2009. Strictly Declarative Specification of Sophisticated Points-to Analyses *(OOPSLA)*.

[3] J.-D. Choi, M. Gupta, M. Serrano, V. C. Sreedhar, and S. Midkiff. 1999. Escape Analysis for Java. In *OOPSLA*.

[4] P. Cousot and R. Cousot. 2014. Abstract Interpretation: Past, Present and Future *(CSL-LICS)*.

[5] M. Eichberg, M. Kahl, D. Saha, M. Mezini, and K. Ostermann. 2007. Automatic Incrementalization of Prolog Based Static Analyses *(PADL)*.

[6] M. Eichberg, M. Mezini, S. Kloppenburg, K. Ostermann, and B. Rank. 2006. Integrating and Scheduling an Open Set of Static Analyses. *(ASE)*.

[7] Torbjörn Ekman and Görel Hedin. 2007. The Jastadd Extensible Java Compiler *(OOPSLA)*.

[8] M. Finifter, A. Mettler, N. Sastry, and D. Wagner. 2008. Verifiable functional purity in Java *(CCS)*.

[9] P. Haller, S. Geries, M. Eichberg, and G. Salvaneschi. 2016. Reactive Async: expressive deterministic concurrency *(SCALA)*.

[10] W. Huang and A. Milanova. 2012. ReImInfer: Method purity inference for Java *(FSE)*.

[11] W. Huang, A. Milanova, W. Dietl, and M. D. Ernst. 2012. ReIm & ReImInfer: Checking and inference of reference immutability and method purity *(OOPSLA)*.

[12] P. Klint, T. van der Storm, and J. J. Vinju. 2009. RASCAL: A Domain Specific Language for Source Code Analysis and Manipulation *(SCAM)*.

[13] T. Kotzmann and H. Mössenböck. 2005. Escape Analysis in the Context of Dynamic Compilation and Deoptimization *(VEE)*.

[14] Sorin Lerner, David Grove, and Craig Chambers. 2002. Composing dataflow analyses and transformations *(POPL)*.

[15] Ralf M. 2014. *Scalable Automated Incrementalization for Real-Time Static Analyses.* Ph.D. Dissertation. Technische Universität Darmstadt.

[16] F. Nielson, H. Nielson, and C. Hankin. 2005. *Principles of Program Analysis.*

[17] D. Pearce. 2011. JPure: a modular purity system for Java *(CC)*.

[18] T. Reps, S. Horwitz, and M. Sagiv. 1995. Precise Interprocedural Dataflow Analysis via Graph Reachability *(POPL)*.

[19] J. Rodriguez and O. Lhoták. 2011. Actor-based parallel dataflow analysis *(CC)*.

[20] Mooly Sagiv, Thomas Reps, and Susan Horwitz. 1996. Precise interprocedural dataflow analysis with applications to constant propagation. *Theoretical Computer Science* 167 (1996).

[21] J. Späth, L. Nguyen Quang Do, K. Ali, and E. Bodden. 2016. Boomerang: Demand-Driven Flow- and Context-Sensitive Pointer Analysis for Java *(ECOOP)*.