# A Datalog Model of Must-Alias Analysis

George Balatsouras[1] Kostas Ferles[2] George Kastrinis[1] Yannis Smaragdakis[1]
[1]University of Athens, Greece, [2]University of Texas at Austin, USA

## Abstract

We give a declarative model of a rich family of must-alias analyses. Our emphasis is on careful and compact modeling, while exposing the key points where the algorithm can adjust its inference power. The model is executable, in the Datalog language, and forms the basis of a full-fledged must-alias analysis of Java bytecode in the DOOP framework.

***CCS Concepts*** • **Theory of computation → Program analysis**

***Keywords*** Alias analysis, Datalog, must analysis

## 1. Introduction

*Pointer analysis* is the backbone of many realistic static analyses, as it offers a scalable way to model heap behavior. Pointer analysis typically comes in two flavors: *alias analysis*, which computes program expressions that may alias, i.e., refer to the same heap object, and *points-to analysis*, which computes the heap objects that program variables and expressions may refer to.

A *must-alias* (or *definite-alias*) analysis computes alias relationships (between program expressions) that are guaranteed to always hold during program execution. The analysis is typically flow-sensitive, i.e., it computes information per-program-point, respecting the control-flow of the program. A must-alias analysis has several applications:

- it is useful for optimizations—e.g., constant folding, common subexpression elimination, and register allocation;
- it can increase the precision of bug detectors: Nikolić and Spoto [12] report that a must-alias analysis significantly increases the precision of both a null-reference detector (46% fewer warnings) and a non-termination detector (11% fewer warnings). Earlier work has reported similar benefits [9];
- it can be used as an internal component as part of a more complex analysis. For instance, must-alias results may enable an analysis to perform "strong updates" at instructions that modify the heap. Earlier work has used must-alias analysis to similar benefit [4, 7].

*SOAP'17*, June 18, 2017, Barcelona, Spain
ACM. 978-1-4503-5072-3/17/06...$15.00
http://dx.doi.org/10.1145/3088515.3088517

To illustrate must-alias reasoning, consider the code below. In this example, `a2.next` and `a1` form an alias pair after line 8. (Other alias pairs include `a1.next` and `null` after line 7, `a2.next.next` and `a2` after line 11, and more.) Alias pairs are established via direct variable assignments, calls, as well as heap stores and loads. A must-alias analysis has to report aliases only when they are guaranteed to hold, and needs to invalidate them on store instructions or method calls that may change the fields of objects pointed by subexpressions in an alias pair.

```
1  class Node {
2    Node next;
3    Node(Node next) { this.next = next; }
4    void wrap() { next.next = this; }
5  }
6  void main() {
7    Node a1 = new Node(null);
8    Node a2 = new Node(a1);
9    Node a3 = new Node(null);
10   a1.next = a3;
11   a2.wrap();
12 }
```

For instance, line 10 invalidates the alias pair `a1.next` and `null`. However, the analysis is sound (i.e., it remains a must-analysis) if it also invalidates alias pairs for expressions involving `a2.next` or `a3.next`. The base specification of a must-alias analysis has to integrate such soundness safeguards, while interplay with other analyses (e.g., a may-not-alias analysis) can lead to more inferences.

In this work, we present a simple declarative model of a must-alias analysis over access paths (i.e., expressions of the form "*var*(.*fld*)*"). The model underlies the implementation of must-alias analysis in the DOOP framework [2], which expresses several analyses of Java bytecode using Datalog specifications. DOOP employs must-alias analysis as an enhancer of its standard array of may-analyses—e.g., in order to enable "strong updates".

The model is interesting in a few different ways:

- It is an instance of a flow-sensitive analysis in Datalog. As such, it introduces idioms and patterns also used in a multitude of other (current or future) analyses in DOOP.
- The analysis is minimal, yet models the core features of a general must-alias analysis in a handful of declarative rules. In this way, the analysis semantics are easily under-

stood and can be further enhanced. The rules allow configurability and employ several techniques for conciseness and power. The use of context, in particular, is crucial: the analysis introduces context variables, much like in traditional may analyses (e.g., [8, 16]), yet uses the context highly unconventionally. Context is used as "fuel", to guarantee the "must" nature of the analysis: must-alias inferences are propagated inter-procedurally, with context extended for every call. When maximum context depth is reached, inferences cannot propagate any further.

- The analysis gives rise to several observations, concerning the representation of equivalence relations in a Datalog engine, and the need for implicit encodings of aliasing.

## 2. Must-Alias Analysis Model

We next present a minimal Datalog model of an inter-procedural must-alias analysis algorithm on a static-single assignment (SSA) intermediate language.

### 2.1 Intermediate Language / Analysis Schema

The language can be enhanced with features such as arrays, static members and calls, exceptions, etc. to be a full-fledged intermediate language. Indeed, our actual analysis implementation is on the Jimple intermediate language of the Soot framework [18],which models all features of Java bytecode. Yet the core of the analysis is represented well in the minimal language.

Figure 1 shows the domain of the analysis (i.e., the different value sets that constitute the space of our computation) and three different groups of relations.

***Input Relations.*** The input relations correspond to our intermediate language features. They are logically grouped into relations that represent instructions and relations that represent name-and-type information. In particular, the Phi relation captures $\phi$ instructions, for the SSA form of our intermediate language. The Next relation expresses directed edges in the control-flow graph (CFG): Next(*i,j*) means that *i* is a CFG predecessor of *j*.

Similarly, there are relations that encode type system, symbol table, and program environment information—e.g., FormalArg, ActualArg, FormalRet, ThisVar. The input intermediate language program is assumed to be in a single-return form, for each method. LookUp matches a method signature to the actual method definition inside a type. InMethod is a function from instructions to their containing methods. Resolved is a predicate that can be computed by an external call-graph or may-point-to analysis: it holds variables that are determined to only point to objects with a unique dynamic type, so that virtual method calls are resolved. (Note that the form of the predicate is context-insensitive, yet the analysis that computes it may be context-sensitive, for increased precision—the contexts are merely projected out.) Finally, RootMethod is a predicate over methods, used to start must-alias reasoning from a user-selected set of methods.

| | | | |
|---|---|---|---|
| $V$: | program variables | $M$: | method identifiers |
| $S$: | method signatures | $F$: | fields |
| $I$: | instructions | $T$: | types |
| $C$: | contexts | $\mathbb{N}$: | natural numbers |

$A$: access paths of the form $V(.F)*$

| | |
|---|---|
| Move(*i: I, to: V, from: V*) | *# i: to = from* |
| Load(*i: I, to: V, base: V, fld: F*) | *# i: to = base.fld* |
| Store(*i: I, base: V, fld: F, from: V*) | *# i: base.fld = from* |
| Call(*i: I, base: V, sig: S*) | *# i: base.sig(..)* |
| Phi(*i: I, to: V, from1: V, . . .*) | *# i: to =* $\phi$*(from1, . . . )* |
| Next(*i: I, j: I*) | *# j is CFG successor of i* |

FormalArg(*meth: M, n:* $\mathbb{N}$*, arg: V*)
ActualArg(*invo: I, n:* $\mathbb{N}$*, arg: V*)
FormalRet(*instr: I, meth: M, ret: V*)
ThisVar(*meth: M, this: V*)
LookUp(*type: T, sig: S, meth: M*)
InMethod(*instr: I, meth: M*)
<u>Resolved</u>(*var: V, type: T*)
<u>RootMethod</u>(*meth: M*)

MustAlias(*inst: I, ctx: C, ap1: A, ap2: A*)
MustCallGraphEdge(*invo: I, ctx: C, toMth: M, toCtx: C*)
Reachable(*ctx: C, meth: M*)

**AP**(*access path expression*) = *ap: A*
**PrimeAP**(*ap: A*) = *newAp: A*
**UnprimeAP**(*ap: A*) = *newAp: A*
**NewContext**(*invo: I, ctx: C*) = *newCtx: C*

**Figure 1:** Our domain, input relations (Move, ...), computed relations (MustAlias, ...), and constructors (**AP**, ...).

***Computed Relations.*** Figure 1 also shows the computed relations of our must-alias analysis. The first relation, MustAlias, is also the main output of the analysis. The relation is defined on access paths, i.e., expressions of the form "*var*(*.fld*)*". The meaning of MustAlias(*i, ctx, ap1, ap2*) is that access path *ap1* aliases access path *ap2* (i.e., they are guaranteed to point to the same heap object, or to both be `null`) right after program instruction *i*, executed under context *ctx*, provided that the instruction is indeed executed under *ctx* at program run-time. The two access paths are said to form an *alias pair*.

Other computed relations represent intermediate results of the analysis. MustCallGraphEdge holds information for fully-resolved virtual calls: invocation site *invo* will call method *toMth* under the given contexts. Reachable computes which methods and under what context are of interest to the must-alias analysis.

***Constructors.*** We assume a constructor function **AP** that produces access paths. For instance, inside a logic program, "**AP**(*var.fld1.fld2*) = *ap*" means that the access path *ap* has length 3 and its elements are given by the values of bound logical variables *var*, *fld1* and *fld2*. We also manipulate access paths with two functions **PRIMEAP** and **UNPRIMEAP**. **PRIMEAP** takes an access path and returns a new one by "priming" the base variable of the original. **UNPRIMEAP** reverses this mapping. For instance, **PRIMEAP**(*"v.fld"*) = *"v'.fld"*. **UNPRIMEAP** only applies to access paths with primed variables as their base—otherwise the rule fails to match. Priming and unpriming of access paths is done at method call and return sites, to mark access paths that arrive from callers. This is necessary for avoiding confusion of variables in recursive calls.

Similarly, we construct new contexts using function **NEWCONTEXT**. The definition of this constructor serves to configure the analysis for different context settings, as discussed later. If **NEWCONTEXT** does not return a value (e.g., because the maximum context depth has been reached), the current rule employing the constructor will not produce facts. The constant **ALL** is used to signify the initial context.

We shall also use **AP** as a pattern matcher over access paths. For instance, the expression "**AP**(*_fld*) = *ap*" binds the value of logical variable *fld* to the last field of access path *ap*. (_ is an anonymous variable that can match any value.)

Constructors of access paths and contexts are much like other relations. In practical analyses, the space of access paths and contexts is made finite, by bounding their length.

### 2.2 Analysis Model

Figure 2 shows our analysis model, in four groups of rules. For conciseness, we have employed some syntactic sugar:

- In addition to conjunction (signified by the usual "," in a rule body) our rules also employ disjunction (";") and negation ("!"). Negation is stratified: it is only applied to predicates that are either input predicates or whose computation can complete before the current rule's evaluation. We also permit multiple predicates in a rule head, as syntactic sugar for replicating the rule body.
- We use the shorthand P* for the reflexive, symmetric, transitive closure of relation P, which is assumed to be binary. For larger arities, underscore (_) variables are used to distinguish variables of a relation that are affected by the closure rule. Specifically, MUSTALIAS*(*i, ctx, _, _*) denotes the reflexive, symmetric, transitive closure of relation MUSTALIAS with respect to its last two variables.
- We introduce ∀: syntactic sugar that hides a Datalog pattern for enumerating all members of a set and ensuring that a condition holds universally.[1] An expression "∀*i*: P(*i*) ⟶ Q(*i*,...)" is true if Q(*i*,...) holds for all *i* such that P(*i*) holds. Such an expression can be used in a rule body, as a condition for the rule's firing. Multiple variables can be quantified by a ∀. Variables not bound by ∀ remain implicitly existentially quantified, as in conventional Datalog. However, the existential quantifier is interpreted as being outside the universal one. For instance, "∀*i,j*: P(*i,j,k*) ⟶ Q(*i,j,k,l*)" is interpreted as "there exist *k,l* such that for all *i,j* ...".

[1] Emulating universal quantification in Datalog requires ordered domains. An arbitrary ordering relation (e.g., by internal id of facts as assigned by the implementation) can be imposed on all our domains.

***Base Rules.*** The top part of Figure 2 lists six rules: one to initialize interesting analysis contexts and five for must-alias inferences. The former rule employs configuration predicate ROOTMETHOD. This predicate designates methods that are to be analyzed unconditionally: the inference is made under the special context value **ALL**. For a non-root method, aliasing inferences can only be made under a specific context, for which the method has been computed to be reachable.

The next four MUSTALIAS rules handle one instruction kind each: MOVE, PHI, LOAD, and STORE. The MOVE rule merely establishes an aliasing relationship between the two assigned variables, at the point of the move instruction. The PHI rule promotes aliasing relationships that hold for all the right-hand sides of a $\phi$ instruction to its left hand side. The LOAD and STORE rules establish aliases between the loaded/stored expression, *base.fld*, and the local variable used. The last MUSTALIAS rule makes the MUSTALIAS relation symmetrically and transitively closed.

***Inter-Procedural Propagation Rules.*** The second part of Figure 2 presents four rules responsible for the inter-procedural propagation of access path aliasing.

The first rule continues the handling of program instructions with a treatment of CALL. At a CALL instruction, for method signature *sig* over object *base*, if *base* has a unique (resolved) type, then the method is looked up in that type, a MUSTCALLGRAPHEDGE is inferred from the invocation instruction to the target method and the method is also marked as REACHABLE with a callee context computed using constructor **NEWCONTEXT**. Recall that the **NEWCONTEXT** function may fail to return a new context (e.g., because *ctx* has already reached the maximum depth and *toCtx* would exceed it) in which case the rule will not infer new facts.

The other three rules handle aliasing induced at a method invocation site. Despite their rather daunting form, the rules are quite straightforward. The first states that, at the first instruction of a called method, the formal and actual arguments are aliased. In combination with other rules (discussed next, under "Access Path Extension") this is sufficient for transferring all alias pairs from the caller to the callee! The actual argument is "primed" appropriately, to mark that it is received from a caller. For instance, if the analyzed program contains a call "`foo(x)`" to a method defined as "`void foo(Object y)`", the rule will simply infer that `x`' and `y` are aliased. The rule infers the same aliasing for the base variable of the method call and the pseudo-variable `this` inside the receiver method. (Note how the first instruction of the called method is computed as the only instruction in the method that has no CFG predecessors: ∀*k* ⟶ !NEXT(*k, firstInstr*). This convention is assumed to hold for our input intermediate language.)

```
REACHABLE(ctx,m) ← ROOTMETHOD(m), ctx = ALL.

MUSTALIAS(i, ctx, AP(from), AP(to)) ←
  MOVE(i, to, from), INMETHOD(i, m), REACHABLE(ctx,m).

MUSTALIAS(i, ctx, ap, AP(to)) ←
  (∀from: PHI(i, to, ..., from, ... ) →
     MUSTALIAS(i, ctx, AP(from), ap)),
  INMETHOD(i, m), REACHABLE(ctx,m).

MUSTALIAS(i, ctx, AP(to), AP(base.fld)) ←
  LOAD(i, to, base, fld),
  INMETHOD(i, m), REACHABLE(ctx,m).

MUSTALIAS(i, ctx, AP(from), AP(base.fld)) ←
  STORE(i, base, fld, from),
  INMETHOD(i, m), REACHABLE(ctx,m).

MUSTALIAS(i, ctx, _, _) ← MUSTALIAS*(i, ctx, _, _).

MUSTCALLGRAPHEDGE(i, ctx, toMth, toCtx),
REACHABLE(toCtx, toMth) ←
  CALL(i, base, sig),
  INMETHOD(i, m), REACHABLE(ctx,m),
  RESOLVED(base, type), LOOKUP(type, sig, toMth),
  NEWCONTEXT(i,ctx) = toCtx.

MUSTALIAS(firstInstr, toCtx, ap1, ap2) ←
  MUSTCALLGRAPHEDGE(i, ctx, toMth, toCtx),
  INMETHOD(firstInstr, toMth), (∀k → !NEXT(k, firstInstr)),
  ((FORMALARG(toMth, n, toVar), ACTUALARG(i, n, var));
   (THISVAR(toMth, toVar), CALL(i, var, _))),
  PRIMEAP(AP(var)) = ap1, AP(toVar) = ap2.

MUSTALIAS(firstInstr, toCtx, ap1, ap2) ←
  MUSTCALLGRAPHEDGE(i, ctx, toMth, toCtx),
  INMETHOD(firstInstr, toMth), (∀k → !NEXT(k, firstInstr)),
  (∀j: NEXT(j, i) →
     MUSTALIAS(j, ctx, callerAp1, callerAp2)),
  PRIMEAP(callerAp1) = ap1, PRIMEAP(callerAp2) = ap2.

MUSTALIAS(i, ctx, ap1, ap2) ←
  MUSTCALLGRAPHEDGE(i, ctx, toMth, toCtx),
  FORMALRET(reti, toMth, _),
  (MUSTALIAS(reti, toCtx, calleeAp1, calleeAp2);
   MUSTALIAS(reti, ALL, calleeAp1, calleeAp2)),
  UNPRIMEAP(calleeAp1) = ap1,
  UNPRIMEAP(calleeAp2) = ap2.

MUSTALIAS(i, ctx, ap3, ap4) ←
  MUSTALIAS(i, ctx, ap1, ap2),
  AP(ap1.fld) = ap3, AP(ap2.fld) = ap4.

MUSTALIAS(i, ctx, ap1, ap2) ←
  !STORE(i, _, _, _), !CALL(i, _, _),
  (∀j: NEXT(j, i) → MUSTALIAS(j, ctx, ap1, ap2)).
```

**Figure 2:** Datalog rules for a model must-alias analysis.

The third rule (again, in the second part of Figure 2) similarly identifies the first instruction of a called method. It then propagates to it all alias pairs that hold *after all predecessor instructions*, *j*, of the calling instruction, *i*. The base variables of the alias pairs are "primed", as appropriate, to denote that they come from the caller.

The fourth and final rule performs the inverse mapping of access paths from a return instruction to the call site. For alias pairs to propagate back (to the caller, with context *ctx*), they need to hold either in the appropriate context (*toCtx*, which matches *ctx* in the call graph), or unconditionally, i.e., with context **ALL**. Access paths are "unprimed" when propagating to the caller. Note that this implies that local alias pairs (e.g., among local variables of the callee) do not propagate to the caller.

Crucially, the handling of a method return is the *only* point where a context can become stronger. MUSTALIAS facts that were inferred to hold under the more specific *toCtx* are now established, modulo unpriming, under *ctx*.

***Access Path Extension.*** The next-to-last rule group of Figure 2 contains a straightforward, yet essential, rule. This rule allows access path extension: if two access paths alias, extending them by the same field suffix also produces aliases. It is important to note that the constructor **AP** is not used in the head of the rule, thus the extended access paths are not generated but assumed to exist. Therefore, the rule does not spur infinite creation of access paths.

This powerful rule is responsible for much of the simplicity of our must-alias analysis specification. For instance, recall how earlier we handled the mapping of actual to formal method arguments quite simply: we merely added an alias between the (primed) actual argument variable and the formal argument. It is the access path extension rule that takes care of also generalizing this mapping to longer access paths whose base variable is the actual argument of the call.

***Frame Rules: From One Instruction To The Next.*** The rule in the bottom part of Figure 2 determines how must-alias facts can propagate from one instruction to its successors. The rule simply states that all aliases are propagated if the instruction is not a store or a call. (Because of SSA, access paths cannot be invalidated via move instructions.)

***Comments.*** The model we just presented is carefully designed to encompass a minimal, highly-compact but usefully representative must-alias analysis. There are several extensions that can apply, but all of them are analogous to features shown. For instance, we are missing a rule for propagating back to the caller complex access paths (i.e., of length greater than 1) that are based on the formal return variable. Similarly, store or call instructions do not invalidate aliasing between local variables—an extra rule could allow further propagation. Furthermore, it is not always necessary for an alias pair to hold in all predecessors: it could hold in one and others may be dominated by the instruction and not invalidate the alias pair. Our actual implementation contains the handling of such cases, but these complexities do not affect the discussion of our model.

## 3. Discussion

There are several parts of the model and its implementation that are worth emphasizing.

***Context-Sensitivity in Must-Alias.*** The use of context in our must-alias analysis is subtle. Context in a pointer analysis is used to distinguish different dynamic execution flows when analyzing a method. That is, the same method gets analyzed once per each applicable context, under different information. The context effectively encodes different scenarios under which the method gets called, allowing more faithful analysis in the specialized setting of the context.

Our analysis model of Section 2 employs context to transmit alias pairs from a caller to a callee, yet qualify them with the context identifier to which they pertain. This enables producing more alias pairs, however, their validity is conditional on the context used. Generally, the use of a deeper context in a must-analysis can extend its reach, allowing *more inferences*, i.e., a larger result, whereas deeper context in a may-analysis it results in *more precision*, i.e., a smaller result.

What can our context be, however? In typical context-sensitive pointer analyses in the literature, a variety of context creation functions can be employed. There are context flavors such as *call-site sensitivity* [14, 15], *object sensitivity* [10, 11], or *type sensitivity* [17]. Our **NEWCONTEXT** constructor (employed at method calls) could be set appropriately to produce such context variety. However, the current form of our rules restricts our options to call-site sensitivity, with potential extra information adding to, but not replacing, call sites. The signature of constructor **NEWCONTEXT** is **NEWCONTEXT**(*invo: I, ctx: C*) = *newCtx: C*. The assumption is that the new context produced uniquely identifies both invocation site *invo* and *its* context, *ctx*. Effectively, if **NEWCONTEXT** produces a *newCtx* at all, it can do little other than push *invo* onto *ctx* and return the result.

The analysis then propagates MUSTALIAS pairs from (all predecessors of) call site *invo* under context *ctx* to the first instruction of a called method, *toMth*, under context *newCtx*. Thus, *newCtx* should be enough to establish that these inferences *must* hold. There is no room for conflating information from multiple execution paths (i.e., callers and calling contexts).[2]

The requirement that **NEWCONTEXT**(*invo,ctx*) produce contexts that uniquely identify both *invo* and *ctx* means that context can only grow from an original source in our analysis. Consider a set of three methods, `meth1`, `meth2`, and `meth3`, each calling the next. If we allow **NEWCONTEXT** to produce contexts that are stacks of invocation sites, *i*, each starting with **ALL** and growing up to depth 2, then starting from `meth1` we will propagate its aliases to `meth2`, which will propagate the resulting combined aliases to `meth3`. The propagation will stop there, i.e., the aliases of `meth1` cannot influence inferences for callees of `meth3`. However, `meth3` (assuming it is included in the root methods) will itself also be analyzed with a context of **ALL**, allowing its own aliases (independently derived from those of `meth1` or `meth2`) to be a source of a similar propagation.

***Representation of Equivalence Classes.*** MUSTALIAS encodes equivalence classes on access paths. Datalog inherently has no such notion and any attempt to compute a must-alias relation has to explicitly encode all aliasing pairs. E.g., if variable `v1` is an alias for variable `v2`, and `v2` of variable `v3`, we have to explicitly record the following pairs: `v1` and `v2`, `v2` and `v1`, `v2` and `v3`, `v3` and `v2`, `v1` and `v3`, `v3` and `v1`. This effect is exacerbated for longer access paths.

In theory, this redundancy will greatly hinder performance. In practice, it is often affordable because of keeping access paths short and computing must-alias information where needed. The analysis is fully modular and can be applied to any subset of the program code. Still, future work should address this shortcoming in the general setting of Datalog computation of equivalence relations.

## 4. Related Work

There are several approaches in the literature that present must-analyses in the pointer analysis setting or employ them in a may-analysis. Our approach is a must-alias analysis applied to Java bytecode, but conceptually it is distinguished by its minimizing the distance between the implementation and the declarative specification.

Nikolić and Spoto [12] present a must-alias analysis that tracks aliases between program expressions and local variables (or stack locations, since they analyze Java bytecode). The analysis is related to ours both because of its application to Java bytecode and because it is constraint-based: the analysis is a generator of constraints, which are subsequently solved to produce the analysis results.

Hind et al. [6] present a collection of pointer analysis algorithms. Among them, the most relevant to this work is a flow-sensitive interprocedural pointer alias analysis. The authors optimistically produce *must* information for pointers to single non-summary objects.

Emami et al. [4] present an approach that simultaneously calculates both must- and may-point-to information for a C analysis. Their empirical results "show the existence of a substantial number of definite points-to relationships, which forms very valuable information"—much in line with our own experience.

Must- information is often computed in conjunction with a client analysis. One of the best examples is the typestate verification of Fink et al. [5], which demonstrates the value of a must-analysis and the techniques that enable it.

[2] One could imagine doing so under the premise that all such calling contexts agree on the aliases they establish at the beginning of the callee function. However, this is unlikely to arise often in practice.

The analysis of [3] is essentially a flow-sensitive may-point-to analysis that performs strong updates, as it maps *access paths* to *heap objects* (abstracted by their allocation sites). The approach uses a flow-insensitive may-point-to analysis to bootstrap the main analysis. However, it provides no *definite* knowledge of any sort, since the aim is to increase the precision of the may-analysis. For instance, even if an access path points to a single heap object, according to the De and D'Souza analysis, there is no *must* point-to information derived, since this object could be a summary object (i.e., one that abstracts many objects allocated at the same allocation site). To reason about such cases, other approaches, such as the more expensive shape analysis algorithms [13], additionally maintain summary information per heap object. In this way, they allow must point-to edges to exist only if the target is definitely not a summary node.

Generally, must-analyses can vary greatly in sophistication and can be employed in an array of different combinations with may-analyses. The analysis of Balakrishnan and Reps [1], which introduces the *recency abstraction*, distinguishes between the most recently allocated object at an allocation site (a concrete object, allowing strong updates) and earlier-allocated objects (represented as a summary node). The analysis additionally keeps information on the size of the set of objects represented by a summary node. At the extreme, one can find full-blown shape analysis approaches, such as that of Sagiv et al. [13], which explicitly maintains must- and may- information simultaneously, by means of three-valued truth values, in full detail up to predicate abstraction: a relationship can definitely hold ("must"), definitely not hold ("must not", i.e., negation of "may"), or possibly hold ("may"). Summary and concrete nodes are again used to represent knowledge, albeit in full detail, as captured by arbitrary predicates whose value is maintained across program statements, at the cost of a super-exponential worst-case complexity.

## Acknowledgments

We gratefully acknowledge funding by the European Research Council, grant 307334 (SPADE).

## References

[1] G. Balakrishnan and T. W. Reps. Recency-abstraction for heap-allocated storage. In *Proc. of the 14th International Symp. on Static Analysis*, SAS '06, pages 221–239. Springer, 2006.

[2] M. Bravenboer and Y. Smaragdakis. Strictly declarative specification of sophisticated points-to analyses. In *Proc. of the 24th Annual ACM SIGPLAN Conf. on Object Oriented Programming, Systems, Languages, and Applications*, OOPSLA '09, New York, NY, USA, 2009. ACM.

[3] A. De and D. D'Souza. Scalable flow-sensitive pointer analysis for java with strong updates. In *Proceedings of the 26th European Conference on Object-Oriented Programming*, ECOOP'12, pages 665–687, Berlin, Heidelberg, 2012. Springer-Verlag.

[4] M. Emami, R. Ghiya, and L. J. Hendren. Context-sensitive interprocedural points-to analysis in the presence of function pointers. In *Proc. of the 1994 ACM SIGPLAN Conf. on Programming Language Design and Implementation*, PLDI '94, pages 242–256, New York, NY, USA, 1994. ACM.

[5] S. Fink, E. Yahav, N. Dor, G. Ramalingam, and E. Geay. Effective typestate verification in the presence of aliasing. In *International Symposium on Software Testing and Analysis (ISSTA)*, pages 133–144, New York, NY, USA, 2006. ACM.

[6] M. Hind, M. Burke, P. Carini, and J.-D. Choi. Interprocedural pointer alias analysis. *ACM Trans. Program. Lang. Syst.*, 21(4):848–894, July 1999.

[7] S. Jagannathan, P. Thiemann, S. Weeks, and A. Wright. Single and loving it: Must-alias analysis for higher-order languages. In *Proceedings of the 25th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '98, pages 329–341, New York, NY, USA, 1998. ACM.

[8] G. Kastrinis and Y. Smaragdakis. Hybrid context-sensitivity for points-to analysis. In *Proc. of the 2013 ACM SIGPLAN Conf. on Programming Language Design and Implementation*, PLDI '13, New York, NY, USA, 2013. ACM.

[9] X. Ma, J. Wang, and W. Dong. Computing must and may alias to detect null pointer dereference. In *Proc. of the 3rd International Symp. On Leveraging Applications of Formal Methods, Verification and Validation*, volume 17 of *ISoLA '08*, pages 252–261. Springer, 2008.

[10] A. Milanova, A. Rountev, and B. G. Ryder. Parameterized object sensitivity for points-to and side-effect analyses for Java. In *Proc. of the 2002 International Symp. on Software Testing and Analysis*, ISSTA '02, pages 1–11, New York, NY, USA, 2002. ACM.

[11] A. Milanova, A. Rountev, and B. G. Ryder. Parameterized object sensitivity for points-to analysis for Java. *ACM Trans. Softw. Eng. Methodol.*, 14(1):1–41, 2005.

[12] D. Nikolić and F. Spoto. Definite expression aliasing analysis for Java bytecode. In *Proc. of the 9th International Colloquium on Theoretical Aspects of Computing*, volume 7521 of *ICTAC '12*, pages 74–89. Springer, 2012.

[13] M. Sagiv, T. Reps, and R. Wilhelm. Parametric shape analysis via 3-valued logic. *ACM Transactions on Programming Languages and Systems*, 24(3):217–298, May 2002.

[14] M. Sharir and A. Pnueli. Two approaches to interprocedural data flow analysis. In S. S. Muchnick and N. D. Jones, editors, *Program flow analysis: theory and applications*, chapter 7, pages 189–233. Prentice-Hall, Inc., Englewood Cliffs, NJ, 1981.

[15] O. Shivers. *Control-Flow Analysis of Higher-Order Languages*. PhD thesis, Carnegie Mellon University, may 1991.

[16] Y. Smaragdakis and G. Balatsouras. Pointer analysis. *Foundations and Trends in Programming Languages*, 2(1):1–69, 2015.

[17] Y. Smaragdakis, M. Bravenboer, and O. Lhoták. Pick your contexts well: Understanding object-sensitivity. In *Proc. of the 38th ACM SIGPLAN-SIGACT Symp. on Principles of Programming Languages*, POPL '11, pages 17–30, New York, NY, USA, 2011. ACM.

[18] R. Vallée-Rai, P. Co, E. Gagnon, L. J. Hendren, P. Lam, and V. Sundaresan. Soot - a Java bytecode optimization framework. In *Proc. of the 1999 Conf. of the Centre for Advanced Studies on Collaborative research*, CASCON '99, pages 125–135. IBM Press, 1999.