NATIONAL AND KAPODISTRIAN UNIVERSITY OF ATHENS

SCHOOL OF SCIENCE
DEPARTMENT OF INFORMATICS AND TELECOMMUNICATIONS
POSTGRADUATE PROGRAM
COMPUTER SYSTEMS TECHNOLOGY

MASTER THESIS

General Declarative Must-Alias Analysis

Konstantinos Ferles

Supervisor: Yannis Smaragdakis, Dr.

ATHENS

JUNE 2015

ΕΘΝΙΚΟ ΚΑΙ ΚΑΠΟΔΙΣΤΡΙΑΚΟ ΠΑΝΕΠΙΣΤΗΜΙΟ ΑΘΗΝΩΝ

ΣΧΟΛΗ ΘΕΤΙΚΩΝ ΕΠΙΣΤΗΜΩΝ
ΤΜΗΜΑ ΠΛΗΡΟΦΟΡΙΚΗΣ ΚΑΙ ΤΗΛΕΠΙΚΟΙΝΩΝΙΩΝ
ΠΡΟΓΡΑΜΜΑ ΜΕΤΑΠΤΥΧΙΑΚΩΝ ΣΠΟΥΔΩΝ
”ΤΕΧΝΟΛΟΓΙΑ ΣΥΣΤΗΜΑΤΩΝ ΥΠΟΛΟΓΙΣΤΩΝ”

ΔΙΠΛΩΜΑΤΙΚΗ ΕΡΓΑΣΙΑ

Γενική και Δηλωτική Ανάλυση Σίγουρης Συνωνυμίας Δεικτών

Κωνσταντίνος Φερλές

Επιβλέπων: Γιάννης Σμαραγδάκης, Αναπληρωτής Καθηγητής

ΑΘΗΝΑ

ΙΟΥΝΙΟΣ 2015

MASTER THESIS

General Declarative Must-Alias Analysis

Konstantinos Ferles
R.N.: M1261

ATHENS

JUNE 2015

ΔΙΠΛΩΜΑΤΙΚΗ ΕΡΓΑΣΙΑ

Γενική και Δηλωτική Ανάλυση Σίγουρης Συνωνυμίας Δεικτών

Κωνσταντίνος Φερλές
Α.Μ. Μ1261

ΕΠΙΒΛΕΠΩΝ:
Γιάννης Σμαραγδάκης, Αναπληρωτής Καθηγητής

ΕΞΕΤΑΣΤΙΚΗ ΕΠΙΤΡΟΠΗ:
Γιάννης Σμαραγδάκης, Αναπληρωτής Καθηγητής
Παναγιώτης Ροντογιάννης, Αναπληρωτής Καθηγητής

ΑΘΗΝΑ

ΙΟΥΝΙΟΣ 2015

ABSTRACT

Most published pointer analysis algorithms are may-analyses: they over-approximate aliasing or points-to relations. Must-alias analyses are more rarely studied but offer attractive benefits, for optimization and program understanding. In this thesis we give a declarative model of a rich family of must-alias analyses. Although other specifications of must-alias algorithms exist in the literature, our emphasis is on modeling and exposing the key points where the algorithm can adjust its inference power vs. scalability trade-off. Furthermore, we show that our model can be easily extended to also incorporate a must-point-to analysis. Our model is executable, in the Datalog language, and forms the basis for a full-fledged must-alias analysis of Java bytecode. We discuss insights on configuring a must-alias analysis and quantify the impact of design decisions on large Java benchmarks.

SUBJECT AREA: Program Analysis

KEYWORDS: points-to analysis, alias analysis, context-sensitivity, object-sensitivity, type-sensitivity

ΠΕΡΙΛΗΨΗ

Οι περισσότερες δημοσιευμένες αναλύσεις για δείκτες είναι *ίσως-αναλύσεις*: δηλαδή υπερεκτιμούν τη σχέση συνωνυμίας δεικτών ή τη σχέση “δείχνει-σε”. Οι αναλύσεις σίγουρης-συνωνυμίας δεικτών έχουν μελετηθεί λιγότερο αλλά προσφέρουν ελκυστικά πλεονεκτήματα, για τη βελτιστοποίηση και την κατανόηση των προγραμμάτων. Σε αυτήν την εργασία δίνουμε ένα δηλωτικό μοντέλο για μια πλούσια οικογένεια αναλύσεων σίγουρης-συνωνυμίας δεικτών. Αν και υπάρχουν ήδη στη βιβλιογραφία φορμαλισμοί ανλύσεων σίγουρης-συνωνυμίας, δίνουμε έμφαση στη μοντελοποίηση και την ανάδειξη των κύριων σημείων όπου ένας αλγόριθμος μπορεί να προσαρμόσει την ισορροπία μεταξύ της συλλογής πληροφορίας και της απόδοσης της ανάλυσης. Επιπλέον, δείχνουμε ότι το μοντέλο μας μπορεί εύκολα να επεκταθεί για να συμπεριλάβει μια ανάλυση για τη σχέση “σίγουρα-δείχνει-σε”. Το μοντέλο μας είναι εκτελέσιμο, στη γλώσσα Datalog, και αποτελεί τη βάση για μια ολοκληρωμένη ανάλυση σίγουρης-συνωνυμίας δεικτών για κώδικα σε μορφή Java bytecode. Εξετάζουμε σε βάθος πώς μπορεί να παραμετροποιηθεί η ανάλυση και ποσοτικοποιούμε την επίδραση των σχεδιαστικών αποφάσεών σε μεγάλα δοκιμαστικά προγράμματα Java.

ΘΕΜΑΤΙΚΗ ΠΕΡΙΟΧΗ: Ανάλυση Προγραμμάτων

KEYWORDS: “δείχνει-σε” ανάλυση, ανάλυση συνωνύμων, “ευαισθησία”-συμφραζομένων, “ευαισθησία”-αντικειμένων, “ευαισθησία”-τύπων

To my parents, Eleni & Dimitris

ACKNOWLEDGMENTS

First and foremost, I would to thank my supervisor, Prof. Yannis Smaragdakis, for providing me with the opportunity to work with him the last few years. I truly enjoyed working on several research projects under Prof. Smaragdakis' supervision, his expertise and guidance throughout my studies were valuable and helped me to develop as a researcher.

Next, I would like to thank all the members of the PLAST lab, my first academic home. Studying in such a fun environment was extremely motivating and helpful. Special thanks to my labmates, George Balatsouras, Aggelos Biboudis and George Kollias. I am really grateful for their help and guidance in both academic and non-academic matters.

Finally, I am deeply indebted to my parents and my aunt Katerina. Without their support nothing that I have accomplished would be possible.

CONTENTS

LIST OF FIGURES

LIST OF TABLES

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.

The vast majority of pointer analysis techniques that have appeared in the recent research literature (e.g., [3,9,11,15,22,26,27]) are *may-analyses*. That is, the techniques attempt to *over-approximate* the (unattainable) fully precise result. All possible alias pairs are guaranteed to be included in the outcome of a may-alias analysis. All possible abstract objects that may be referenced by a variable are included in the variable's may-point-to set. However, spurious inferences (which will never occur in program execution) may also be included in the analysis output.

In contrast, an *under-approximate*, *must-analysis* is often desirable. A must-analysis computes aliasing or points-to relationships that are guaranteed to always hold during program execution, at the cost of missing some inferences. A must-analysis for pointers is invaluable for automatic optimizations, such as constant folding, common subexpression elimination, and register allocation, as well as for better program understanding: the results of a must-inference are guaranteed facts, of immediate value to the human programmer. Furthermore, must-analyses are ideal for bug detection that traditionally has a high false-warnings rate. For instance, a may-analysis for null pointer dereferences (either in may-alias or may-point-to form) is rarely of engineering value, due to the preponderance of false warnings. In contrast, a must-analysis for the same problem yields very few warnings, but virtually all of them are actionable.

In practice, a must-analysis for pointers is often a must-alias analysis, and not a must-point-to analysis. Points-to facts are harder to establish than aliasing relationships in a conservative must- fashion. The difficulty is dual: First, the creation sites of objects are often far from their use sites, making the establishment of must-point-to relationships unlikely. Second, must-point-to reasoning requires careful modeling of abstract vs. concrete objects. For instance, for techniques such as *strong updates* (i.e., replacing the value of an object field at a store instruction) it is not sufficient to know the abstract object that the base expression must-point-to, since the abstract object may conflate many concrete objects during program execution, and only one of them will have its field updated.

In this work, we present MADOOP: a general must-alias analysis framework, with significant configurability. Our framework is implemented in the Datalog language, as an extension of the DOOP may-point-to analysis framework [5]. The MADOOP framework comprises some-300 logical rules, as well as scaffolding code for the low-level (flow-sensitive) modeling of Java bytecode as logical relations. However, the essence of the framework, for a minimal input language, is well-captured in a small number of rules, which we present in detail in this thesis.

The framework specification is quite modular: extra functionality can be supported as additional rules. Furthermore, the framework can be configured to achieve different tradeoffs of inference power (i.e., more alias pairs established) and scalability. For instance, the framework allows control over context creation, access path creation, and more. An important dimension, which our work explores, is that of context-sensitivity. Context-sensitivity is a concept of diminished value for an alias analysis, since alias pairs already offer a summary of a function's behavior, and this summary is customized at call sites. However, we discuss interesting ways in which context-sensitivity can be profitable in the must-alias analysis setting. Finally, as is common in practice, a must-alias analysis employs a may-analysis as a pre-processing step. Our framework concisely captures all the points at which may- information interfaces with must- information. Furthermore, we analyze and quantify the impact of the choice of may-analysis on the effectiveness of the must-alias analysis.

A major benefit of a must-alias analysis is its *incrementality*. In a well-specified must-alias analysis, soundness is not compromised if only a portion of the program-under-analysis or its libraries are available. This key element is emphasized in our declarative model. We control the program points where the full analysis applies and leverage context-sensitivity to allow analysis of other program points.

In essence, our analysis infers normal must-alias relationships for a user-selected core part of the program, and infers conditional, context-qualified must-alias relationships for other parts that interact with the core program.

Overall, our work:

- models a general, configurable, powerful yet fully declarative must-alias analysis framework, with context-sensitivity used to control analysis applicability;
- applies the framework, with extensions for fully realistic treatment of language features, to Java bytecode;
- discusses insights and trade-offs of must-alias analysis;
- presents experimental results that illustrate interesting configurations of the must-alias analysis.

In the rest of the thesis, we present an example for illustration of must-alias reasoning (Chapter 2), show our analysis algorithm (Chapter 3), discuss its parametric character and configurability options (Chapter 4), analyze tradeoffs experimentally (Chapter 5), detail related work (Chapter 6) and conclude (Chapter 7).

2. BACKGROUND AND EXAMPLE

We illustrate some basic concepts of must-alias and must-point-to reasoning with a small example.

```
 1 class A {
 2   A next;
 3   B member;
 4
 5   A(A next, B member) {
 6     this.next = next;
 7     this.member = member;
 8   }
 9
10   void foo(A a) {
11     member.container = a;
12   }
13 }
14
15 class B {
16   A container;
17   B(A container) {
18     this.container = container;
19   }
20 }
21
22 public class Test {
23   public static void main(String[] args) {
24     B b1 = new B(null);
25     A a1 = new A(null, b1);
26     A a2;
27     if (args != null)
28       a2 = new A(null, b1);
29     else
30       a2 = new A(a1, b1);
31     b1.container = a2;
32     a1.foo(a1);
33   }
34 }
```

Figure 1: Simple illustration of must-alias inferences.

Consider the small Java program in Figure 1. Even at this size, inspecting the program requires human effort. A must-alias analysis can provide useful information to tools and humans alike. The output consists of must-alias pairs: expressions that are guaranteed to point to the same object. (More precise definitions follow in Chapter 3.) For instance, `a1.member` and `b1` form an alias pair for almost the entire body of method `main`. Alias pairs are established by direct variable assignments (which are plentiful in a compiler intermediate language, although less so in original source code), 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. In Figure 1, `b1.container` is an alias for `a2` on (i.e., after) line 31. However, the analysis needs to recognize that line 32 invalidates that alias pair. Line 32 instead establishes an aliasing relationship between `b1.container` (as well as `a1.member.container`) and `a1`. The analysis remains sound (i.e., safely under-approximate) if the earlier `b1.container` $\sim$ `a2` alias pair is invalidated, regardless of whether the new alias pair (`b1.container` $\sim$ `a1`) is established, via inter-procedural reasoning, on line 32. Such invalidation can take place either conservatively, at all method call points, or by use of a may-analysis, which informs the must-analysis that the call to `foo` will result in changes to the `container` field of an object.

Other aliasing relationships hold throughout the program. Establishing them often requires some inter-procedural reasoning—e.g., to see the aliasing effects of the constructor call on lines 25, 28, or 30. Constructors feature prominently in the example, since they are one of the best sources of must-alias information in a typical program.

Similarly, we can infer must-point-to relations between access paths and heap objects, represented by their allocation sites. For instance, access paths `b1` and `a1` must point to the heap allocation on lines 24 and 25 respectively. These must-point-to facts are the easiest to establish, since we only have to keep track of the instructions that allocate a new heap object. However, if we leverage the aforementioned must-alias inferences, we can infer must-point-to facts for more complex access paths. For example, before line 32 we know that `a1` must point to the heap allocated on line 25 (there is no instruction throughout main that can invalidate this fact). If we propagate this information interprocedurally and use the fact that within `foo` `member.container` and `a` are aliased (i.e., they point to the same heap allocation), we can infer that `member.container` also points to the heap allocated on line 25. We can also propagate this fact back to `main` (`foo`'s caller) and infer that `a1.member.container` must point to the same heap allocation. As we mention later, the must-point-to relation holds only if the allocation site refers to the most recently allocated object, as presented in reference [2].

3. MUST-ALIAS ANALYSIS MODEL

We present next our minimal model of a must-alias analysis algorithm. The model is expressed in the Datalog language. Since the early work of Reps [17], Datalog has been repeatedly used to express program analysis algorithms (e.g., [5,13,25,28]). The language ideally captures the usual program analysis notion of monotonic iteration until fixpoint. Datalog rules infer more facts from combinations of previously established facts, with variables implicitly quantified existentially. A rule of the form "$\text{HEAD}(x,y) \leftarrow \text{P}(x,z),\ \text{Q}(y,z).$" means that if values for x,y,z can be found so that $\text{P}(x,z)$ and $\text{Q}(y,z)$ are simultaneously true, then $\text{HEAD}(x,y)$ is inferred. Syntactically, the left arrow symbol ($\leftarrow$) separates the inferred facts (i.e., the *head* of the rule) from the previously established facts (i.e., the *body* of the rule). For our analysis to be expressed in Datalog, we assume that the program-under-analysis is represented as input relations (typically implemented as database tables) that encode its different elements. Such pre-processing is a relatively straightforward one-to-one translation.

3.1 Schema of Analysis Relations

We show our algorithm on a minimal static-single assignment (SSA) intermediate language[1] with a) a "move" instruction for copying between local variables; b) a "phi" instruction for merging of single-assignment variables; c) "store" and "load" instructions for writing to the heap (i.e., to object fields); d) a "virtual method call" instruction that calls the method of the appropriate signature defined in the dynamic class of the receiver object; and e) an "assign heap allocation" instruction that allocates and constructs an object of the given type. Furthermore, 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 [23, 24], which models all features of Java bytecode. Yet the core of the analysis is expressed faithfully in the minimal language.

Figure 2 shows the domain of the analysis (i.e., the different value sets that constitute the space of our computation) and four different groups of relations. These relations are explained in more detail below, but their declarations and type signatures can be handy for reference. The first group (starting with ALLOC) contains relations representing the input language instructions and other program text information, such as types and member lookup. The second group (starting with MUSTALIAS) contains relations computed by our algorithm. The third group (starting with **AP**) contains functions that produce new objects, either contexts or access paths. The fourth group (starting with $\underline{\text{RESOLVED}}$) also lists input

[1]That is, every local variable is assigned exactly once in the input program. For variables with multiple assignments in the original source code, the merging of their values is indicated by a ϕ node. $var = \phi(var1, var2, var3)$ signifies that the value of var can be either one of the three values on the right-hand side. For the purposes of static analyses, like ours, that do not track path conditions, it is not relevant which of the three values is actually picked.

V is a set of program variables	M is a set of method identifiers
S is a set of method signatures (name+types)	F is a set of fields
I is a set of instructions	T is a set of types
C is a set of contexts	A is $V.(F)*$: a set of access paths
H is a set of heap allocations (i.e., allocation sites)	$\mathbb{N}$ is the set of natural numbers

ALLOC(*i: I, to: V, heap: H*)	*# i: to = new ...*
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 = ϕ(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*)	ACTUALRET(*invo: I, var: V*)
THISVAR(*meth: M, this: V*)	LOOKUP(*type: T, sig: S, meth: M*)
INMETHOD(*instr: I, meth: M*)	

MUSTALIAS(*i: I, ctx: C, ap1: A, ap2: A*)
ACCESSPATHMUSTPOINTTO(*i: I, ctx: C, ap: A, heap: H*)
MUSTCALLGRAPHEDGE(*invo: I, ctx: C, toMth: M, toCtx: C*)
REACHABLE(*ctx: C, meth: M*)
REBASEATCALL(*i: I, ctx: C, fromVar: V, toVar: V*)
REBASEATRETURN(*i: I, ctx: C, fromVar: V, toVar: V*)

AP(*access path expression*) = *ap: A*
RebaseAP(*ap: A, fromVar: V, toVar: V*) = *newAp: A*
NewContext(*invo: I, ctx: C*) = *newCtx: C*

RESOLVED(*var: V, type: T*)	MAYALIAS(*var1: V, var2: V*)
CALLMAYSTORETOFIELD(*invo: I, fld: F*)	ROOTMETHOD(*meth: M*)
CALLMAYALLOCATEHEAP(*invo: I, heap: H*)	

Figure 2: Our domain, input relations (Alloc, ...), computed relations (MustAlias, ...), constructors (AP, ...), and configuration predicates (Resolved, ...). The input relations are of two kinds: relations encoding program instructions (the form of the instruction is shown in a comment) and relations encoding type system and other environment information.

relations, but of a different kind. These are expected to either be supplied by the user for analysis configuration purposes, or to be computed by an earlier-run may-analysis.

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. For instance, the MOVE relation represents instructions that assign a local variable to another. There are similar input relations for other instruction types (ALLOC, LOAD, STORE, and CALL, for allocating heap objects, for reading/writing heap object fields, and for virtual calls, respectively). The PHI relation captures ϕ 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. For instance, FormalArg shows which variable is a formal argument of a given method at a certain index (i.e., the n-th argument). The relation is a function from the first two arguments to the third. ActualArg is similar, but at a method invocation site. FormalRet combines information on the return variable of a function with the index of the return instruction. Note that the input intermediate language program is assumed to be in a single-return form, for each method. ActualRet is a function of its first argument (a method invocation site) and returns the local variable at the call site that receives the method call's return value. ThisVar returns the `this` variable of a method. LookUp is a function from its first two arguments to the third. It matches a method signature to the actual method definition inside a type. InMethod is a function from instructions to their containing methods.

Computed Relations. Figure 2 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*. The second relation, AccessPathMustPointTo, has similar meaning but instead of relating two access paths, it relates an access path with a heap allocation. That is, AccessPathMustPointTo(*i, ctx, ap, h*) means that, right after instruction *i*, under context *ctx*, access path *ap* must point to the most recently allocated object represented by allocation site *h* (provided *i* is executed under *ctx*).

The introduction of access paths and contexts raises natural questions: how complex can access paths get? What is a context and what does it mean for it to occur at run-time? We postpone the full treatment of these questions until Chapter 4.

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. The RebaseAtCall and RebaseAtReturn relations hold variable pairs for access-path remapping (between the caller and the callee) at call sites.

Constructors. We assume a constructor function **AP** that produces access paths, and another function **RebaseAP** that takes an access path and returns a new one by changing the base variable of the original. 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*. Similarly, we construct new contexts using function **NewContext**. The above constructor functions also serve to configure the analysis. If a constructor 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 any field of access path *ap*. (_ is an anonymous variable that can match any value in a Datalog program.)

Constructors of access paths and contexts are much like other relations. In practical analyses, the space of access paths is made finite, by bounding their length, and similarly for contexts. Therefore, all possible access paths and contexts could be computed prior to the analysis start and supplied as inputs. However, this is unlikely to be desirable in practice, for efficiency reasons, and is limiting in principle: by separating constructors, our model also allows analyses with unbounded access paths and contexts.

Configuration Predicates. The last four elements of Figure 2 show input predicates that can be used to configure the must-alias analysis. Predicates RESOLVED, MAYALIAS, CALLMAYSTORETOFIELD, and CALLMAYALLOCATEHEAP are expected to be computed by a *may-point-to* analysis, running as a preprocessing step. RESOLVED holds variables that are determined to only point to objects with a unique dynamic type. MAYALIAS reflects whether two variables (in the same method) may point to the same object. CALLMAYSTORETOFIELD does a transitive search of all methods that may be called at an invocation site, *invo*, looking for store instructions to field *fld*. Likewise, CALLMAYALLOCATEHEAP computes the transitive closure of all the abstract heap objects that *invo* may possibly allocate. The power of the must-alias analysis will hinge on the precision of these relations.

Finally, ROOTMETHOD is a predicate over methods, used to limit the applicability of base must-alias reasoning to a user-selected set of methods. As we will see, our analysis algorithm will venture beyond these root methods only to the extent that its context constructor allows.

3.2 Core Analysis Model

Figures 3 and 4 show our must-alias algorithm. To keep the rules concise, we have employed some syntactic sugar, which straightforwardly maps to more complex Datalog rules:

- 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 use the shorthand P^* for the reflexive, 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, $\textsc{MustAlias}^*(i,\ ctx,\ \_,\ \_)$ denotes the reflexive, transitive closure of relation $\textsc{MustAlias}$ with respect to its last two variables.
- We introduce $\forall$: syntactic sugar that hides a Datalog pattern for enumerating all members of a set and ensuring that a condition holds universally.[2] An expression "$\forall i$: $\mathrm{P}(i) \rightarrow \mathrm{Q}(i,...)$" is true if $\mathrm{Q}(i,...)$ holds for all i for which $\mathrm{P}(i)$ 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 $\forall$. If a variable is not bound by a $\forall$, it remains implicitly existentially quantified, as in conventional Datalog. However, the existential quantifier is interpreted as being outside the universal one. For instance, "$\forall i,j$: $\mathrm{P}(i,j,k) \rightarrow \mathrm{Q}(i,j,k,l)$" is interpreted as "there exist k,l such that for all i,j ...".

The rules in Figures 3 and 4 are split into five groups. We discuss them in order, next.

Base Rules. The top part of Figure 3 lists six rules: two to initialize interesting analysis contexts and four for must-alias inferences. The first rule employs configuration predicate $\textsc{RootMethod}$. This predicate designates methods that are to be analyzed unconditionally: the inference is made under the special context value **All**. Additionally, our algorithm analyzes all methods that are fully resolved, i.e., discovered by $\textsc{MustCallGraphEdge}$.

The above mechanism controls the application extent of the analysis. Recall that incrementality is a key benefit of a must-alias analysis. Therefore, it is desirable to be able to apply the algorithm as locally as the user may desire. The context mechanism is then used to explore other code, but only to the extent that such exploration benefits the root methods intended for analysis.

The four $\textsc{MustAlias}$ rules handle one instruction kind each: $\textsc{Move}$, $\textsc{Phi}$, $\textsc{Load}$, and $\textsc{Store}$. The $\textsc{Move}$ rule merely establishes an aliasing relationship between the two assigned variables, at the point of the move instruction. The $\textsc{Phi}$ rule promotes aliasing relationships that hold for all the right-hand sides of a ϕ instruction to its left hand side. The $\textsc{Load}$ and $\textsc{Store}$ rules establish aliases between the loaded/stored expression, *base.fld*, and the local variable used.

Inter-Procedural Propagation Rules. The bottom part of Figure 3 presents five rules responsible for the inter-procedural propagation of access path aliasing.

The first rule continues the handling of program instructions with a treatment of $\textsc{Call}$. At a $\textsc{Call}$ instruction, for method signature *sig* over object *base*, if *base* has a unique (resolved)

[2]Emulating universal quantification in Datalog requires ordered domains. In practice this is not a restriction: an arbitrary ordering relation (e.g., by internal id of facts as assigned by the implementation) can be imposed on all our domains.

```
Reachable(ctx,m) ← RootMethod(m), ctx = All.
Reachable(toCtx,toMth) ← MustCallGraphEdge(_, _, toMth, toCtx).

MustAlias(i, ctx, AP(from), AP(to)) ←
 Move(i, to, from), InMethod(i, m), Reachable(ctx,m).

MustAlias(i, ctx, AP(from), 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).

MustCallGraphEdge(i, ctx, toMth, toCtx) ←
 Call(i, base, sig), InMethod(i, m), Resolved(base, type),
 LookUp(type, sig, toMth), Reachable(ctx,m), NewContext(i,ctx) = toCtx.

RebaseAtCall(i, ctx, var, toVar) ←
 MustCallGraphEdge(i, ctx, toMth, _),
 ((FormalArg(toMth, n, toVar), ActualArg(i, n, var));
  (ThisVar(toMth, toVar), Call(i, var, _))).

RebaseAtReturn(i, ctx, var, toVar) ←
 MustCallGraphEdge(i, ctx, toMth, _),
 ((ActualRet(i, toVar), FormalRet(_, toMth, var));
  (ActualArg(i, n, toVar), FormalArg(toMth, n, var));
  (Call(i, toVar, _), ThisVar(toMth, var))).

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)),
 RebaseAtCall(i, ctx, var1, toVar1), RebaseAP(callerAp1, var1, toVar1) = ap1,
 RebaseAtCall(i, ctx, var2, toVar2), RebaseAP(callerAp2, var2, toVar2) = ap2.

MustAlias(i, ctx, ap1, ap2) ←
 MustCallGraphEdge(i, ctx, toMth, toCtx), FormalRet(ret, toMth, _),
 MustAlias(ret, toCtx, calleeAp1, calleeAp2),
 RebaseAtReturn(i, ctx, var1, toVar1), RebaseAtReturn(i, ctx, var2, toVar2),
 RebaseAP(calleeAp1, var1, toVar1) = ap1,
 RebaseAP(calleeAp2, var2, toVar2) = ap2.
```

Figure 3: Datalog rules for a model must-alias analysis: handling move, load, store, and call instructions.

type, then the method is looked up in that type, and a MustCallGraphEdge is inferred from the invocation instruction to the target method. The callee context is 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 four rules handle access path rebasing, i.e., the mapping of an access path

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

MustAlias(i, ctx, ap1, ap2) ←
MustAlias(i, **All**, ap1, ap2), InMethod(i, meth), Reachable(ctx, meth).

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)).

MustAlias(i, ctx, ap1, ap2) ←
Call(i, _, _), (∀j: Next(j, i) → MustAlias(j, ctx, ap1, ap2)),
AP(var1) = ap1, **AP**(var2) = ap2.

MustAlias(i, ctx, ap1, ap2) ←
Call(i, _, _), (∀j: Next(j, i) → MustAlias(j, ctx, ap1, ap2)),
(∀fld: (**AP**(_.fld._) = ap1; **AP**(_.fld._) = ap2) → !CallMayStoreToField(i, fld)).

MustAlias(i, ctx, ap1, ap2) ←
Store(i, base, fld, _), (∀j: Next(j, i) → MustAlias(j, ctx, ap1, ap2)),
(!**AP**(_.fld._) = ap1; (**AP**(var1.fld) = ap1, !MayAlias(var1, base))),
(!**AP**(_.fld._) = ap2; (**AP**(var2.fld) = ap2, !MayAlias(var2, base))).

Figure 4: Datalog rules for a model must-alias analysis: transitive closure, context weakening, access path extension, and frame rules.

from the local variables of one method to those of another, during calls and returns. The rules establishing RebaseAtCall and RebaseAtReturn are straightforward. The former computes mappings from every actual parameter to its matching formal parameter, as well as from the base variable of the call to `this`. The latter computes inverse mappings (recall that our input is in SSA form, so the values of local variables cannot be reassigned), as well as a mapping from actual return variable to the formal one, inside the caller.

The last two rules in Figure 3 employ these mappings. Alias pairs that hold for every predecessor instruction, *j*, of the calling instruction, *i*, are rebased (using function **RebaseAP**) per the RebaseAtCall mappings and inferred for the first instruction of the called method. (The first instruction of the called method is computed as the only instruction in the method that has no CFG predecessors. This convention is assumed to hold for our input intermediate language.) Alias pairs that hold at the return instruction of a method are rebased, per the RebaseAtReturn mappings, and inferred for the invocation site.

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

Transitive Closure, Reachability, Access Path Extension, and Contexts. The top part of Figure 4 contains straightforward, yet essential, rules. The very top rule makes relation MustAlias symmetrically and transitively closed.

The second rule is a context weakening rule: any must-alias fact that holds for an **All** context also holds for any specific context and method of interest (i.e., in REACHABLE).

The third rule of Figure 4 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. Thus, the rule does not spur infinite creation of access paths. We discuss the issue of the space of access paths and how to populate it efficiently in Chapter 4.

Frame Rules: From One Instruction To The Next. Our last four rules, in the bottom part of Figure 4, determine how must-alias facts can propagate from one instruction to its successors. These rules liberally employ negation. They establish that must-alias facts are propagated if some disabling conditions do *not* hold. Therefore, for a full-fledged analysis, the rules need to be enriched with more preconditions, to cover all different kinds of program instructions that may invalidate access paths.

Each rule body contains a premise that establishes must-alias facts that hold for all predecessors of an instruction. The first rule then simply states that all aliases are propagated if the instruction is not a store or a call. (Because of our SSA-input assumption, access paths cannot be invalidated via move instructions.) The second rule propagates over call instructions alias relationships between access paths that consist of mere variables.

The last two rules are more interesting. The next-to-last rule propagates an alias pair over a call, as long as no field contained in either access path is invalidated by any method transitively reachable from the call site. The latter condition is established by configuration predicate CALLMAYSTORETOFIELD, supplied by a prior may-analysis.

The very last rule propagates alias pairs over a store instruction, as long as, for both access paths, either the stored field is not in the access path or the access path is of the simple form *var.fld* and *var* cannot be aliased to the base of the store. This reasoning again employs the results of a prior may-alias analysis, encoded in configuration predicate MAYALIAS.

3.3 Must-Point-To Modeling

Now we present the additional rules needed in order to include a must-point-to analysis in our core model. Predicate ACCESSPATHMUSTPOINTTO calculates the heap that an acess path points to per program point (under a context ctx). Since we are modeling a “must” analysis, the heap object an access path can point to is unique, therefore ACCESSPATH-MUSTPOINTTO is a function from its three first arguments to the fourth.

Recall that if ACCESSPATHMUSTPOINTTO holds for an access path *ap* and a heap object *h*, this means that *ap* points to the most recently allocated object represented by allocation site *h* [2]. This is the only way to ensure that *h* is not a summary object. This is a necessary

$$\begin{array}{l}
\textsc{AccessPathMustPointTo}(\mathit{i},\ \mathit{ctx},\ \textbf{AP}(\mathit{to}),\ \mathit{heap}) \leftarrow \\
\quad \textsc{Alloc}(\mathit{i},\ \mathit{to},\ \mathit{heap}),\ \textsc{InMethod}(\mathit{i},\ \mathit{m}), \\
\quad \textsc{Reachable}(\mathit{ctx},\mathit{m}).
\end{array}$$

$$\begin{array}{l}
\textsc{AccessPathMustPointTo}(\mathit{firstInstr},\ \mathit{toCtx},\ \mathit{calleeAp},\ \mathit{heap}) \leftarrow \\
\quad \textsc{MustCallGraphEdge}(\mathit{i},\ \mathit{ctx},\ \mathit{toMth},\ \mathit{toCtx}), \\
\quad \textsc{InMethod}(\mathit{firstInstr},\ \mathit{toMth}),\ (\forall k \rightarrow !\textsc{Next}(\mathit{k},\ \mathit{firstInstr})), \\
\quad (\forall j\text{: } \textsc{Next}(\mathit{j},\ \mathit{i}) \rightarrow \textsc{MustAlias}(\mathit{j},\ \mathit{ctx},\ \mathit{callerAp},\ \mathit{heap})), \\
\quad \textsc{RebaseAtCall}(\mathit{i},\ \mathit{ctx},\ \mathit{var},\ \mathit{toVar}),\ \textbf{RebaseAP}(\mathit{callerAp},\ \mathit{var},\ \mathit{toVar}) = \mathit{calleeAp}.
\end{array}$$

$$\begin{array}{l}
\textsc{AccessPathMustPointTo}(\mathit{i},\ \mathit{ctx},\ \mathit{callerAp},\ \mathit{heap}) \leftarrow \\
\quad \textsc{MustCallGraphEdge}(\mathit{i},\ \mathit{ctx},\ \mathit{toMth},\ \mathit{toCtx}),\ \textsc{FormalRet}(\mathit{ret},\ \mathit{toMth},\ \_), \\
\quad \textsc{AccessPathMustPointTo}(\mathit{ret},\ \mathit{toCtx},\ \mathit{calleeAp},\ \mathit{heap}), \\
\quad \textsc{RebaseAtReturn}(\mathit{i},\ \mathit{ctx},\ \mathit{var},\ \mathit{toVar}),\ \textbf{RebaseAP}(\mathit{calleeAp},\ \mathit{var},\ \mathit{toVar}) = \mathit{callerAp}.
\end{array}$$

$$\begin{array}{l}
\textsc{AccessPathMustPointTo}(\mathit{i},\ \mathit{ctx},\ \mathit{ap},\ \mathit{heap}) \leftarrow \\
\quad !\textsc{Store}(\mathit{i},\ \_,\ \_,\ \_),\ !\textsc{Call}(\mathit{i},\ \_,\ \_),\ !\textsc{Alloc}(\mathit{i},\ \_,\ \_), \\
(\forall j\text{: } \textsc{Next}(\mathit{j},\ \mathit{i}) \rightarrow \textsc{AccessPathMustPointTo}(\mathit{j},\ \mathit{ctx},\ \mathit{ap},\ \mathit{heap})).
\end{array}$$

$$\begin{array}{l}
\textsc{AccessPathMustPointTo}(\mathit{i},\ \mathit{ctx},\ \mathit{ap},\ \mathit{heap}) \leftarrow \\
\quad \textsc{Call}(\mathit{i},\ \_,\ \_),\ (\forall j\text{: } \textsc{Next}(\mathit{j},\ \mathit{i}) \rightarrow \textsc{AccessPathMustPointTo}(\mathit{j},\ \mathit{ctx},\ \mathit{ap},\ \mathit{heap})), \\
\quad (\forall \mathit{fld}\text{: } (\textbf{AP}(\_.\mathit{fld}.\_) = \mathit{ap}) \rightarrow !\underline{\textsc{CallMayAllocateHeap}}(\mathit{i},\ \mathit{heap}),\ !\underline{\textsc{CallMayStoreToField}}(\mathit{i},\ \mathit{fld})
\end{array}$$

$$\begin{array}{l}
\textsc{AccessPathMustPointTo}(\mathit{i},\ \mathit{ctx},\ \mathit{ap},\ \mathit{heap}) \leftarrow \\
\quad \textsc{Store}(\mathit{i},\ \mathit{base},\ \mathit{fld},\ \_),\ (\forall j\text{: } \textsc{Next}(\mathit{j},\ \mathit{i}) \rightarrow \textsc{AccessPathMustPointTo}(\mathit{j},\ \mathit{ctx},\ \mathit{ap},\ \mathit{heap})), \\
\quad (!\textbf{AP}(\_.\mathit{fld}.\_) = \mathit{ap};\ (\textbf{AP}(\mathit{var}.\mathit{fld}) = \mathit{ap},\ !\underline{\textsc{MayAlias}}(\mathit{var},\ \mathit{base}))).
\end{array}$$

$$\begin{array}{l}
\textsc{AccessPathMustPointTo}(\mathit{i},\ \mathit{ctx},\ \mathit{ap2},\ \mathit{heap}) \leftarrow \\
\quad \textsc{MustAlias}(\mathit{i},\ \mathit{ctx},\ \mathit{ap1},\ \mathit{ap2}),\ \textsc{AccessPathMustPointTo}(\mathit{i},\ \mathit{ctx},\ \mathit{ap1},\ \mathit{heap}).
\end{array}$$

Figure 5: Datalog rules for a model must-point-to analysis.

requirement in order to perform strong updates.

Figure 5 presents the Datalog rules that populate the AccessPathMustPointTo predicate. As before, we separate the rules into three groups. The first group presents rules related to allocation instructions and interpocedural propagation of AccessPathMustPointTo facts. The second group presents the frame rules for AccessPathMustPointTo, i.e., how we propagate facts from instruction to the next one. Lastly, the third group presents how must-point-to analysis interacts with must-alias. Next we briefly discuss the key points for each group of rules.

Allocations and interpocedural logic. The first rule is the most crutial one, since heap allocations are the source for all must-point-to facts. At each allocation site, we can infer that the target variable must point to the heap allocation. The interpocedural logic is similar to the must-alias algorithm, as presented in figure 3, since it also uses the RebaseAtCall and RebaseAtReturn predicates.

Frame rules. Again the frame rule logic is almost identical to the must-alias analysis. The key point again is to detect instructions that may affect an access path in order to

$\textsc{AccessPathMustPointTo}(\textit{i}, \textbf{All}, \textbf{AP}\textit{(to)}, \textit{heap}) \leftarrow$
$\quad\textsc{Alloc}(\textit{i, to, heap}), \textsc{InMethod}(\textit{i, m}),$
$\quad\textsc{Reachable}(\_,\textit{m}).$

$\textsc{AccessPathMustPointTo}(\textit{i, ctx, ap2, heap}) \leftarrow$
$\quad\textsc{MustAlias}(\textit{i, ctx1, ap1, ap2}), \textsc{AccessPathMustPointTo}(\textit{i, ctx2, ap1, heap}),$
$\quad\sqcap(\textit{ctx1, ctx2}) = \textit{ctx}.$

Figure 6: Optimized rules that avoid materializing redundant facts.

stop propagating AccessPathMustPointTo facts. However, these frame rules must also ensure that every access path points to the most recently allocated object. To establish this we use the CallMayAllocateHeap relation at the second frame rule, the one responsible to propagate AccessPathMustPointTo facts after a call instruction.

Interaction with must-alias. Finally, the last group contains a single rule that combines the two parts of our model, must-alias and must-point-to analyses. Until now we have seen how we establish AccessPathMustPointTo facts only for simple variables. By exploiting our must-alias analysis, which holds for arbitrary access paths, we can infer AccessPathMustPointTo facts for more complex paths. The last rule of Figure 5 captures the aforementioned logic in a straightforward manner. If we know that the MustAlias relation holds for two access paths and that AccessPathMustPointTo holds for either of them, we infer an AccessPathMustPointTo edge for the other.

A crucial optimization. The weakening rule, presented in Figure 4, is the one that glues all the analysis together. This rule is responsible for combining facts that hold unconditionally, i.e., under the special context value **All**, and facts that hold under a specific calling context. However, we avoid materializing these facts since MustAlias is a dense relation. Instead, we employ several optimizations to achieve context weakening. For example, we define operator ⊓ for two contexts as follows:

$$\sqcap(\text{ctx1, ctx2}) = \begin{cases} \text{ctx1}, & \text{ctx2 = } \textbf{All} \\ \text{ctx2}, & \text{ctx1 = } \textbf{All} \\ \text{ctx1}, & \text{ctx1 = ctx2} \\ \bot, & \text{ctx1 != ctx2} \wedge \text{ctx1 != } \textbf{All} \wedge \text{ctx2 != } \textbf{All} \end{cases}$$

So, when two predicates that involve contexts apear in the body of a rule, we bound their contexts to different variables and provide them to the ⊓ operator. If the result is bottom, the rule body is false and the rule produces no facts; otherwise we use the returned context in the head of the rule as the context of the newly inferred facts.

To illustrate the optimization consider the first and last rules in Figure 5. The first one infers an AccessPathMustPointTo fact for every reachable context. Nevertheless, we can safely infer that **AP**`(to)` must-point to *heap* only under the special context **All**, since ALLOC is not affected by the calling context. The same applies for other instructions as well, such as MOVE, LOAD, etc. Now, we also need to adjust the last rule of Figure 5, because it depends on existing MUSTALIAS and ACCESSPATHMUSTPOINTTO facts. The two optimized rules are presented in Figure 6.

4. DISCUSSION: ANALYSIS CONFIGURABILITY

The analysis model of the previous section is configurable in many ways. The creation of access paths and contexts (e.g., their maximum depth), the choice of may-analysis, the applicability to specific parts of the program, are all means to configure the analysis. We discuss some topics in more detail next.

Context-Sensitivity in Must-Alias. The use of context in our must-alias analysis is subtle, with several aspects deserving clarification.

The concept of 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 Chapter 3 employs context (Figure 3) to transmit alias pairs from a caller to a callee, yet qualify such $\textsc{MustAlias}$ facts with the context identifier to which they pertain. This enables producing more alias pairs, however, their validity is conditional on the context used. Still, this conditional information can be used for further inferences, in logical rules not shown in Figures 3 and 4. For instance, $\textsc{MustAlias}$ inferences can be combined with allocation instructions (allocation sites can be viewed as global access paths) in order to determine, when possible, which objects an access path must point to. In turn, this can inform virtual method resolution, which our current rules (Figure 3) only perform via a may-analysis. In this way, specialized alias relations for a given context can result in more inferences (since a method call may now have a known target). These inferences can be propagated back to the caller, where they hold unconditionally. (Recall that the rules handling returns can remove access path assumptions.) Generally, the use of a deeper context in a must-analysis can extend its reach, allowing *more inferences*, i.e., a larger result, whereas 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* [19, 20], *object sensitivity* [14, 15], or *type sensitivity* [21]. 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, only allowing variable depth. The signature of constructor **NewContext** is $\textbf{NewContext}(\mathit{invo}\colon I,\ \mathit{ctx}\colon C) = \mathit{newCtx}\colon C$. 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. This assumption is reflected in our other rules—for instance:

$\textsc{MustAlias}(\mathit{firstInstr},\ \mathit{toCtx},\ \mathit{ap1},\ \mathit{ap2}) \leftarrow$
$\quad\textsc{MustCallGraphEdge}(i,\ \mathit{ctx},\ \mathit{toMth},\ \mathit{toCtx}),$
$\quad\textsc{InMethod}(\mathit{firstInstr},\ \mathit{toMth}),\ (\forall k \rightarrow\ !\textsc{Next}(k,\ \mathit{firstInstr})),$
$\quad(\forall j\colon \textsc{Next}(j,\ i) \rightarrow \textsc{MustAlias}(j,\ \mathit{ctx},\ \mathit{callerAp1},\ \mathit{callerAp2})),$
$\quad\textsc{RebaseAtCall}(i,\ \mathit{ctx},\ \mathit{var1},\ \mathit{toVar1}),\ \textbf{RebaseAP}(\mathit{callerAp1},\ \mathit{var1},\ \mathit{toVar1}) = \mathit{ap1},$
$\quad\textsc{RebaseAtCall}(i,\ \mathit{ctx},\ \mathit{var2},\ \mathit{toVar2}),\ \textbf{RebaseAP}(\mathit{callerAp2},\ \mathit{var2},\ \mathit{toVar2}) = \mathit{ap2}.$

In the above, all $\textsc{MustAlias}$ inferences from (all predecessors of) call site i under context ctx are transmitted to the first instruction of toMth, under context toCtx. Thus, toCtx should be enough to establish that these inferences *must* hold. There is no room for conflating information from multiple execution paths.

We can fix the rules to allow such generality. For the above rule, we get:

$\textsc{MustAlias}(\mathit{firstInstr},\ \mathit{toCtx},\ \mathit{ap1},\ \mathit{ap2}) \leftarrow$
$\quad(\forall i,\ \mathit{ctx}\colon \textsc{MustCallGraphEdge}(i,\ \mathit{ctx},\ \mathit{toMth},\ \mathit{toCtx}) \rightarrow$
$\qquad\textsc{InMethod}(\mathit{firstInstr},\ \mathit{toMth}),\ (\forall k \rightarrow\ !\textsc{Next}(k,\ \mathit{firstInstr})),$
$\qquad(\forall j\colon \textsc{Next}(j,\ i) \rightarrow \textsc{MustAlias}(j,\ \mathit{ctx},\ \mathit{callerAp1},\ \mathit{callerAp2})),$
$\qquad\textsc{RebaseAtCall}(i,\ \mathit{ctx},\ \mathit{var1},\ \mathit{toVar1}),\ \textbf{RebaseAP}(\mathit{callerAp1},\ \mathit{var1},\ \mathit{toVar1}) = \mathit{ap1},$
$\qquad\textsc{RebaseAtCall}(i,\ \mathit{ctx},\ \mathit{var2},\ \mathit{toVar2}),\ \textbf{RebaseAP}(\mathit{callerAp2},\ \mathit{var2},\ \mathit{toVar2}) = \mathit{ap2}).$

That is, the first premise of the rule becomes the guard of a $\forall$, and the entire rest of the body is now under the $\forall$. The meaning of the new rule is that a caller-side alias pair is transmitted to the callee only if *all* call sites and contexts, (i and ctx) that result in calls to toMth under the same toCtx agree that the alias pair is valid.[3]

With such a modification to our rules, it is possible to use arbitrary **NewContext** constructors that conflate or distinguish context information as they see fit. However, in practice, different context flavors are unlikely to be useful: the same alias pairs will rarely hold for different call sites. Since alias pair information is kept on a per-instruction-context basis, call-site sensitivity is quite natural in the domain, and allows weakening the $\forall$ premise to a single call site.

The requirement that **NewContext**(i,ctx) produce contexts that uniquely identify both i and ctx means that context can only grow (and not mutate) 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 the user designates it a root method) will itself also be analyzed with a context of **All**, allowing its own aliases (independently derived from those

[3]An even better version of the rule would check that all call sites that result in the same context agree on access paths *after* rebasing. However, this requires more significant refactoring of our rules and hinders our illustration.

of `meth1` or `meth2`) to be a source of a similar propagation.

Access Path Creation. Our access path constructor, **AP**, hides the details of the space of access paths and their construction. There are several different policies that an analysis can pick. In theory, we could up-front populate the entire combinatorial space of "*var*.(*fld*)*" up to a certain depth. However, the large sizes of the domains of variables and fields make this prohibitive. An efficient way to create access paths lazily (also used in our full implementation) is to initially generate all primitive access paths (variables and variable-single-field combinations) that appear explicitly in the program text, and then close the set of access paths by employing the rule:

$$\mathbf{AP}(\mathit{ap2.fld}) = \mathit{newAp} \longleftarrow \textsc{MustAlias}(\mathit{i,\ ctx,\ ap1,\ ap2}),\ \mathbf{AP}(\mathit{ap1.fld}) = \_.$$

Note that one use of the constructor **AP** is in the head of the rule (thus generating new access paths on the fly) and one in the body (checking that the access path already exists). That is, extended access paths (*base.field*) are generated only if their base access path is found to be aliased with another path, which already exists with the *field* suffix.

Finally, new access paths are also generated on the fly by rebasing (at method calls and returns) per the **RebaseAP** constructor.

5. IMPLEMENTATION AND EXPERIMENTS

The minimal model of earlier sections captures the essence of our full-fledged implementation. Our framework, MaDoop, is built on top of the Doop Datalog framework for may-point-to analysis of Java bytecode [5]. Whereas Doop is a flow-insensitive points-to analysis framework (ignoring the order of instructions in a method), MaDoop adds flow-sensitive modeling of the Java bytecode program (a full control-flow graph over basic blocks). All program structure information is represented as logical tables, which are subsequently processed by about 300 logical rules.

To demonstrate the framework experimentally, we apply it to the DaCapo benchmark programs [4] v.2006-10-MR2 under JDK 1.7.0_55. We use the LogicBlox Datalog engine, v.3.10.14, on a Xeon E5-2667 3.3GHz machine with only one thread running at a time and 256GB of RAM. When a may-analysis is needed, we employ the most precise analysis that scales to large programs in the Doop framework—a 2-object-sensitive analysis with a context-sensitive heap (2objH).

We study five of the DaCapo benchmarks: antlr, chart, luindex, lusearch, pmd. We wanted a small enough set for human inspection of alias pairs at select program points, as well as programs for which a 2objH analysis terminates. The scalability of our own must-alias analysis is not a concern: as discussed earlier, a must-alias analysis is naturally incremental. Hence, scalability to large programs is largely not an issue: we can apply the analysis to just the application classes or to a smaller hand-selected subset of the program code.

Table 1 presents statistics on the sizes of the benchmarks, in terms of application code deemed reachable by the Doop may-point-to analysis (2objH), as well as the running time of this analysis. We also include the execution time for the preparatory computation over the may-analysis results ("must pre-analysis" column)—i.e., the time to compute our required input predicates, Resolved, MayAlias and CallMayStoreToField.

Our must-alias analysis is run with a maximum access path length of 3. Context depth varies per experiment, as discussed below.

Comparison with Intra-Procedural Analysis. As a first measure of the value of our inter-procedural must-alias analysis, we compare it against intra-procedural analyses. Traditional compilers can already compute aliases using intra-procedural data-flow analysis and employ them for optimizations such as common subexpression elimination, constant folding, or register allocation. It is interesting to consider whether there is benefit from considering more precisely the effects of called methods, in order to infer more local aliases.

For this experiment, we ran our analysis algorithm (with a context depth of 1) over the entire application portion of the benchmarks. That is, the size of our **RootMethod** set is the number of methods shown in Table 1.

The first and last settings shown in Table 2 ("intra-procedural" and "inter-procedural", re-

			may analysis time	
Benchmark	methods	classes	core (2objH)	must pre-analysis
antlr	1635	228	4m42s	0m51s
chart	1173	515	9m56s	0m57s
luindex	690	349	2m25s	0m27s
lusearch	1046	349	2m35s	0m29s
pmd	1877	553	3m06s	0m35s

Table 1: Statistics for reachable application code, 2objH may-point-to analysis and must pre-analysis.

Alias pairs						
	intra-procedural			intra-proc.$^{+\text{may}}$		
Bench.	per instruction	per return	time	per instruction	per return	time
antlr	58	34	3m04s	82	44	3m07s
chart	50	40	2m41s	59	44	2m25s
luindex	11	11	0m70s	17	14	0m71s
lusearch	12	11	0m74s	17	13	0m76s
pmd	28	23	3m28s	34	27	3m02s
	inter-proc.$_{-\text{may}}$			inter-procedural		
	per instruction	per return	time	per instruction	per return	time
antlr	58	34	3m41s	580	154	20m56s
chart	51	40	3m22s	76	53	17m19s
luindex	12	11	0m87s	31	26	2m46s
lusearch	12	11	0m94s	27	21	3m03s
pmd	28	23	4m48s	41	32	9m27s

Table 2: Average #alias pairs per program element (divided by 2, i.e., with equivalent symmetric pairs removed), and timings for various settings.

spectively) present a purely intra-procedural analysis and our full inter-procedural algorithm. The intra-procedural analysis does not exploit aliasing inferences from other methods, and conservatively invalidates alias pairs at method calls and heap stores.

The main metric to watch for in this comparison is “alias pairs per instruction”, i.e., the first of the three columns in every grouping of numbers. As can be seen, the full inter-procedural analysis (last of the four settings shown in Table 2) yields significantly richer information than the intra-procedural analysis (first setting). With only intra-procedural information, we can infer at most one half, and often less than one third, of the alias pairs. The alias pairs shown are unconditional (i.e., for context **All**) with symmetric pairs removed. The numbers are over the Jimple intermediate language of the Soot framework and, thus, the alias pairs are more numerous than one would expect from program inspection, due to the introduction of several temporary variables.

	Alias pairs under 1-must-context			*Alias pairs* under 5-must-context		
Benchmark	per instr.	per all-return	time	per instr.	per all-return	time
antlr	10	8	0m51s	19	12	2m48s
chart	25	13	1m25s	30	18	1m21s
luindex	8	6	0m20s	9	7	0m25s
lusearch	6	5	0m22s	9	7	1m12s
pmd	6	4	0m25s	11	9	10m12s

Table 3: Alias pairs (with equivalent symmetric pairs removed, i.e., numbers divided by 2), and timings for different contexts of the must-alias analysis.

Value of May-Analysis for Must- Inference. Table 2 shows two more settings of our analysis: intra-procedural$^{+\text{may}}$ and inter-procedural$_{-\text{may}}$. These help us quantify how exactly may-analysis information contributes to the must-analysis.

The intra-procedural$^{+\text{may}}$ setting does intra-procedural must-alias reasoning but invalidates access paths by employing the inter-procedural may-alias analysis. That is, of the three predicates computed by the may-analysis, the intra-procedural$^{+\text{may}}$ analysis does *not* use predicate $\underline{\text{Resolved}}$ to do virtual method resolution, but *does* use predicates $\underline{\text{MayAlias}}$ and $\underline{\text{CallMayStoreToField}}$, in order to decide when to invalidate access paths at call and store instructions. Conversely, the inter-procedural$_{-\text{may}}$ analysis *does* use predicate $\underline{\text{Resolved}}$ to do virtual method resolution, but does *not* use predicates $\underline{\text{MayAlias}}$ and $\underline{\text{CallMayStoreToField}}$, instead invalidating alias information conservatively at store and call instructions.

These variations of settings give us a picture of the separate benefit from various kinds of inter-procedurality (may vs. must reasoning). As can be seen, inter-procedural reasoning benefits greatly from both kinds of inter-procedural may-analysis information. The inter-procedural$_{-\text{may}}$ setting is significantly less effective than the full inter-procedural analysis, while still typically much better than intra-procedural$^{+\text{may}}$, which, in turn, is better than the purely intra-procedural setting.

Note that all of the above configurations were produced via straightforward configuration of our full implementation. This is testament to the inherent configurability of a modular, declarative framework. This ease of experimentation is testament to the inherent configurability of a modular, declarative framework, as opposed to a monolithic implementation.

Timings. Table 2 also shows the running time for our analysis, as well as for intra-procedural analyses. As can be seen, the must-alias analysis is quite fast. In the case of antlr, the analysis time blows up, but so do the inferred facts. Generally, our timings do not aim to reflect an ideal implementation of the declarative rules. Specifically, our Datalog engine does not use union-find trees for the equivalence classes of reflexively and transitively closed relations, thus incurring significant overhead. On the other hand, our implementation is heavily optimized in terms of rule execution and indexing (for fast combination of relations), to an extent that a manual implementation will have difficulties matching.

Results for Human Inspection. For human inspection of aliases, we have found it useful to produce all alias pairs that hold (unconditionally, i.e., under an **All** context) for *all* return statements of a method. This “forall” intersection of aliasing information over all return sites offers a good summary of the method’s effects, and is more concise than alias information at a single given instruction.

Table 2 shows the sizes of the sets computed in this fashion, as the second of the three columns for every analysis setting. As can be seen, the full inter-procedural must-alias analysis is again significantly more information-rich than any other setting. The sizes of the resulting alias sets (from 10 to 27) are small enough for human inspection and typically quite informative.

Exploring Parts of the Program, Using Context. To explore parts of the program only conditionally (with alias pairs qualified by context), we selected at random 20 methods in each of the benchmarks. We ran the analysis with these methods in the **RootMethod** set, and different maximum context depth values, to allow exploration of code outside that set as much as the context depth permits.

Table 3 shows the result of this experiment, for context depth settings of 1 and 5. The alias pairs shown are computed *at the root methods only*, i.e., they do not count alias pairs in the methods analyzed under a non-**All** context, but they do count the impact of such methods on the alias information at the original root method set.

The execution time demonstrates the incrementality of the must-alias analysis—its running time was often negligible.

For a deeper context, the analysis produces often significantly richer facts. Since the methods of origin are randomly selected, the number of extra alias pairs (when mapped back at the root methods) varies, but is generally consistent.

6. RELATED WORK

There are several approaches in the literature that combine may- and must-analyses in the pointer analysis setting. 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, and by its exposition of configuration points.

Ma et al. [12] present an algorithm for null-pointer dereference detection using a context-insensitive may-alias and a must-alias analysis; the latter is used to increase the precision of the former, by enabling strong updates when possible.

Nikolić and Spoto [16] present a must-alias analysis that tracks aliases between program expressions and local variables (or stack locations, since they analyze Java bytecode, which is a stack-based representation). The analysis itself does not expose any clear configuration points but it 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. Abstractly, this is a relative of our Datalog-based approach, but it is unclear how the two may compare in terms of engineering tradeoffs.

Emami et al. [7] 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. [8], which demonstrates the value of a must-analysis and the techniques that enable it.

The analysis of [6] 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). As in this thesis, it 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 [18], 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.

An approach for integrating *must* point-to reasoning in an analysis is to propagate such information only at instructions where we know that the given heap allocation target still refers to the last object allocated at that site [1]. Thus, an execution path that may create another object at the same site (such as when reaching the end of the loop) would invalidate any previous must-point-to facts (i.e., it will stop them from propagating any further).

Jagannathan et al. [10] present an algorithm for must-alias analysis of functional languages. The algorithm adapts must-alias insights to the setting of captured variables. For instance, must-alias information for non-summary objects permits strong updates, which the authors find to improve analysis precision.

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 [2], 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. [18], which explicitly maintains must- and may- information simultaneously, by means of three-valued truth values, in full detail up to predicate abstraction.

7. CONCLUSIONS

We presented a modular, declarative algorithm model for must-alias analyses and discussed its features and configurability options. The model faithfully reflects MaDoop: a full-fledged implementation of more than 300 Datalog rules to analyze Java bytecode. Our analysis interfaces with the may-point-to analyses of the Doop framework and can leverage their precision, at clearly defined interaction points.

The literature on must-alias analyses is sparse and the distance of specification to implementation is typically large. In our literature survey we have not found a single must-alias analysis publication that concretely refers to another and shows how its approach differs. Thus, the goal of our work is largely to provide concrete exposition and a reference point. We believe that our model is clear yet concrete enough to spur further development and a better understanding of the comparative features of different must-alias analysis algorithms. In practical terms, must-alias analysis is valuable and woefully under-exploited in the literature. Our experiments show concrete value for (human) program understanding and (automatic) optimization.

REFERENCES

[1] Altucher, R.Z., Landi, W.: An extended form of must alias analysis for dynamic allocation. In: Proc. of the 22Nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages. pp. 74–84. POPL '95, ACM, New York, NY, USA (1995)

[2] Balakrishnan, G., Reps, T.W.: Recency-abstraction for heap-allocated storage. In: Proc. of the 14th International Symp. on Static Analysis. pp. 221–239. SAS '06, Springer (2006)

[3] Berndl, M., Lhoták, O., Qian, F., Hendren, L.J., Umanee, N.: Points-to analysis using BDDs. In: Proc. of the 2003 ACM SIGPLAN Conf. on Programming Language Design and Implementation. pp. 103–114. PLDI '03, ACM, New York, NY, USA (2003)

[4] Blackburn, S.M., Garner, R., Hoffmann, C., Khan, A.M., McKinley, K.S., Bentzur, R., Diwan, A., Feinberg, D., Frampton, D., Guyer, S.Z., Hirzel, M., Hosking, A.L., Jump, M., Lee, H.B., Moss, J.E.B., Phansalkar, A., Stefanovic, D., VanDrunen, T., von Dincklage, D., Wiedermann, B.: The DaCapo benchmarks: Java benchmarking development and analysis. In: Proc. of the 21st Annual ACM SIGPLAN Conf. on Object Oriented Programming, Systems, Languages, and Applications. pp. 169–190. OOPSLA '06, ACM, New York, NY, USA (2006)

[5] Bravenboer, M., Smaragdakis, Y.: 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, ACM, New York, NY, USA (2009)

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

[7] Emami, M., Ghiya, R., Hendren, L.J.: 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. pp. 242–256. PLDI '94, ACM, New York, NY, USA (1994)

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

[9] Hardekopf, B., Lin, C.: The ant and the grasshopper: fast and accurate pointer analysis for millions of lines of code. In: Proc. of the 2007 ACM SIGPLAN Conf. on Programming Language Design and Implementation. pp. 290–299. PLDI '07, ACM, New York, NY, USA (2007)

[10] Jagannathan, S., Thiemann, P., Weeks, S., Wright, A.: 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. pp. 329–341. POPL '98, ACM, New York, NY, USA (1998)

[11] Kastrinis, G., Smaragdakis, Y.: Efficient and effective handling of exceptions in Java

points-to analysis. In: Proc. of the 22nd International Conf. on Compiler Construction. pp. 41–60. CC ’13, Springer (2013)

[12] Ma, X., Wang, J., Dong, W.: 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. ISoLA ’08, vol. 17, pp. 252–261. Springer (2008)

[13] Madsen, M., Livshits, B., Fanning, M.: Practical static analysis of JavaScript applications in the presence of frameworks and libraries. In: Proceedings of the ACM SIGSOFT International Symposium on the Foundations of Software Engineering (Aug 2013)

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

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

[16] Nikolić, D., Spoto, F.: Definite expression aliasing analysis for Java bytecode. In: Proc. of the 9th International Colloquium on Theoretical Aspects of Computing. ICTAC ’12, vol. 7521, pp. 74–89. Springer (2012)

[17] Reps, T.W.: Demand interprocedural program analysis using logic databases. In: Ramakrishnan, R. (ed.) Applications of Logic Databases, pp. 163–196. Kluwer Academic Publishers (1994)

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

[19] Sharir, M., Pnueli, A.: Two approaches to interprocedural data flow analysis. In: Muchnick, S.S., Jones, N.D. (eds.) Program flow analysis: theory and applications, chap. 7, pp. 189–233. Prentice-Hall, Inc., Englewood Cliffs, NJ (1981)

[20] Shivers, O.: Control-Flow Analysis of Higher-Order Languages. Ph.D. thesis, Carnegie Mellon University (may 1991)

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

[22] Sridharan, M., Bodík, R.: Refinement-based context-sensitive points-to analysis for Java. In: Proc. of the 2006 ACM SIGPLAN Conf. on Programming Language Design and Implementation. pp. 387–400. PLDI ’06, ACM, New York, NY, USA (2006)

[23] Vallée-Rai, R., Co, P., Gagnon, E., Hendren, L.J., Lam, P., Sundaresan, V.: Soot - a Java bytecode optimization framework. In: Proc. of the 1999 Conf. of the Centre for Advanced Studies on Collaborative research. pp. 125–135. CASCON ’99, IBM Press (1999), `http://dl.acm.org/citation.cfm?id=781995.782008`

[24] Vallée-Rai, R., Gagnon, E., Hendren, L.J., Lam, P., Pominville, P., Sundaresan, V.: Optimizing Java bytecode using the Soot framework: Is it feasible? In: Proc. of the

9th International Conf. on Compiler Construction. pp. 18–34. CC ’00, Springer (2000)

[25] Whaley, J., Avots, D., Carbin, M., Lam, M.S.: Using Datalog with binary decision diagrams for program analysis. In: Proc. of the 3rd Asian Symp. on Programming Languages and Systems. pp. 97–118. APLAS ’05, Springer (2005)

[26] Whaley, J., Lam, M.S.: Cloning-based context-sensitive pointer alias analysis using binary decision diagrams. In: Proc. of the 2004 ACM SIGPLAN Conf. on Programming Language Design and Implementation. pp. 131–144. PLDI ’04, ACM, New York, NY, USA (2004)

[27] Yong, S.H., Horwitz, S., Reps, T.: Pointer analysis for programs with structures and casting. In: PLDI ’99: Proceedings of the ACM SIGPLAN 1999 Conference on Programming Language Design and Implementation. pp. 91–103 (1999)

[28] Zhang, X., Mangal, R., Grigore, R., Naik, M., Yang, H.: On abstraction refinement for program analyses in Datalog. In: Proc. of the 2014 ACM SIGPLAN Conf. on Programming Language Design and Implementation. pp. 239–248. PLDI ’14, ACM, New York, NY, USA (2014)