NATIONAL AND KAPODISTRIAN UNIVERSITY OF ATHENS

SCHOOL OF SCIENCES

DEPARTMENT OF INFORMATICS AND TELECOMMUNICATIONS

UNDERGRADUATE STUDIES PROGRAM

UNDERGRADUATE THESIS

Graph-based data structure for representation of sets of must-alias analysis inferences

Nefeli Prokopaki – Kostopoulou

Supervisors: **Yannis Smaragdakis,** Professor NKUA
George Kastrinis, PhD Student NKUA
George Balatsouras, PhD Student NKUA

ATHENS

NOVEMBER 2016

ΕΘΝΙΚΟ ΚΑΙ ΚΑΠΟΔΙΣΤΡΙΑΚΟ ΠΑΝΕΠΙΣΤΗΜΙΟ ΑΘΗΝΩΝ

ΣΧΟΛΗ ΘΕΤΙΚΩΝ ΕΠΙΣΤΗΜΩΝ

ΤΜΗΜΑ ΠΛΗΡΟΦΟΡΙΚΗΣ ΚΑΙ ΤΗΛΕΠΙΚΟΙΝΩΝΙΩΝ

ΠΡΟΓΡΑΜΜΑ ΠΡΟΠΤΥΧΙΑΚΩΝ ΣΠΟΥΔΩΝ

ΠΤΥΧΙΑΚΗ ΕΡΓΑΣΙΑ

Δομή δεδομένων για αναπαράσταση των αποτελεσμάτων must-alias ανάλυσης

Νεφέλη Προκοπάκη - Κωστοπούλου

Επιβλέποντες: **Γιάννης Σμαραγδάκης,** Καθηγητής ΕΚΠΑ
Γιώργος Καστρίνης, Διδακτορικός φοιτητής ΕΚΠΑ
Γιώργος Μπαλατσούρας, Διδακτορικός φοιτητής ΕΚΠΑ

ΑΘΗΝΑ

ΝΟΕΜΒΡΙΟΣ 2016

UNDERGRADUATE THESIS

Graph-based data structure for representation of must-alias analysis inferences

Nefeli Prokopaki - Kostopoulou

R.N.: 1115201100146

Supervisors: **Yannis Smaragdakis,** Professor NKUA
George Kastrinis, PhD Student NKUA
George Balatsouras, PhD Student NKUA

ΠΤΥΧΙΑΚΗ ΕΡΓΑΣΙΑ

Δομή δεδομένων για αναπαράσταση των αποτελεσμάτων must-alias ανάλυσης

Νεφέλη Προκοπάκη – Κωστοπούλου

Α.Μ.: 1115201100146

Επιβλέποντες: **Γιάννης Σμαραγδάκης,** Καθηγητής ΕΚΠΑ
Γιώργος Καστρίνης, Διδακτορικός φοιτητής ΕΚΠΑ
Γιώργος Μπαλατσούρας, Διδακτορικός φοιτητής ΕΚΠΑ

ΠΕΡΙΛΗΨΗ

Στην παρούσα πτυχιακή εργασία παρουσιάζουμε μια δομή δεδομένων (με τη μορφή γράφου), η οποία αναπαριστά τις κλάσεις ισοδυναμίας δεικτών που βρίσκονται σε κάθε σημείο του προγράμματος, κάτι που είναι χρήσιμο για βελτιστοποιήσεις στη μεταγλώττιση και για την κατανόηση του προγράμματος.

Σκοπός της εργασίας αυτής είναι η επανα-υλοποίηση σε γλώσσα Java, της αρχικής υλοποίησης ενός δηλωτικού μοντέλου της ανάλυσης σίγουρης-ισοδυναμίας δεικτών πάνω σε μονοπάτια πρόσβασης, γραμμένου σε Datalog, που χρησιμοποιείται ήδη από το framework του Doop. Η νέα υλοποίηση κατασκευάζει μια βελτιστοποιημένη δομή δεδομένων η οποία κρατά πολλές σχέσεις ισοδυναμίας δεικτών και μονοπατιών πρόσβασης σε ένα μόνο γράφο και ξεπερνά σε ταχύτητα εκτέλεσης της ανάλυσης την αρχική υλοποιήση.

Ως είσοδο και έξοδο, χρησιμοποιούμε αρχεία, τα οποία περιλαμβάνουν σχέσεις που ανταποκρίνονται στα χαρακτηριστικά της ενδιάμεσης γλώσσας που χρησιμοποιείται από το Doop.

ΘΕΜΑΤΙΚΗ ΠΕΡΙΟΧΗ: Στατική Ανάλυση Προγραμμάτων

ΛΕΞΕΙΣ ΚΛΕΙΔΙΑ: σίγουρη ισοδυναμία, άνεφ συμφραζομένων ανάλυση δεικτών, Java

ABSTRACT

In this thesis we present a graph-based data structure for representing alias pairs that hold in each program point, which is useful for optimizations and program understanding.

The purpose of this project was to re-implement in Java, the original implementation of a declarative model of a must-alias analysis over access paths, written in Datalog and in use in the Doop framework. The new implementation manufactures an optimized data structure that encodes multiple alias sets, and aliasing relations over longer access paths, in a single graph and outperforms in speed the original implementaion.

We use as input and output files with relations that correspond to Doop's intermediate language features.

SUBJECT AREA: Static Programming Analysis

KEYWORDS: must alias, context-insensitive points-to analysis, Java

To my parents

ACKNOWLEDGEMENTS

I would like to thank my supervisor, Prof. Yannis Smaragdakis for providing his expertise and guidance, and for his support and patience throughout the whole time that I have been working on this project.

Also, I would like to thank George Kastrinis for his contribution to this work and together with George Balatsouras for their assistance to overcome the obstacles I encountered during this thesis.

Lastly, I would like to thank my family and friends for supporting me and believing in me.

CONTENTS

LIST OF FIGURES

LIST OF TABLES

PREFACE

This project has been developed since September 2015 in the University of Athens at the department of Informatics and Telecommunications as my undergraduate thesis.

Athens, November 2016

1. INTRODUCTION

Pointer analysis is a static analysis (analysis performed without actually executing programs) that models heap behavior. It is divided in two subsets, *points-to analysis* and *alias analysis*, that are closely related but not actually equivalent.

Points-to analysis, establishes which pointers, or heap references, can refer to which variables, or storage locations.

Alias analysis is used to determine if a storage location may be accessed in more than one way. Note that aliasing refers to the situation in which a data location in memory can be accessed through different symbolic variables in a program. Therefore, two pointers are said to be *aliased* if they point to the same location. To simplify, *points-to analysis* finds the heap objects that variables may point to, and *alias analysis* computes expressions that may alias.

Most pointer analyses are *may-analyses*. That is, they over-approximate the precise result, to be guaranteed that all possible alias pairs or heap objects are found. However, in that way, spurious inferences may be included. Thus, when two memory references are said to have a may-alias relation, their aliasing is not certain that exists, likewise when an abstract object belongs to a *may-point-to* set of a heap reference, their connection is also questionable. But if two variables do not belong in a may-alias or may-point-to set, it is certain that they do not alias, or they do not reference to each other. The *may* information is usually used to prove the absence of bugs in a program.

On the contrary, *must-analysis* under-approximates the accurate result, at the cost of not including all inferences. The facts that are said to have a must-alias or a must-point-to relation are guaranteed to always hold during program execution. The *must-analysis* can be used to prove the existence of bugs in a program. For that reason it is priceless for bug detection (because of the minimized false-warning rate), as well as for automatic optimizations and for better program understanding.

Intuitively, a *may analysis* of a procedure represents a property guaranteed to be true in all executions of that procedure, while a *must analysis* represents witness executions of the procedure that are guaranteed to exist.

The majority of *must analyses* for pointers are *must-alias analyses*, instead of *must-point-to.* The reason is, that alias facts are easier than point-to facts to establish. In this project, we present a graph-based data structure to re-implement a simple declarative model of a *must-alias analysis* over access paths (i.e., expressions of the form "*variable(.field)** "), configured for the Doop framework. The model we introduce compresses a fully-fledged implementation into a few declarative rules written in Datalog, so that the concept of the analysis is easily understandable.

Based on the above model, each instruction maintains a set of alias classes. Namely, it keeps a non-connected directed-graph with variables and access paths that are certain to be aliased at the specific program point.

The operations performed over the graphs are:

- creating a new component (aka alias class) in the graph and adding a single variable or access path;
- removing a variable or an access path from a connected component and adding it to another;

- producing a new graph (aka set of alias classes) by intersecting two others.

We show that our new implementation of the declarative analysis model achieves dramatic speedup compared to the original (mentioned above and written in Datalog).

The rest of the thesis is organized as follows:

- In chapter 2 we give a background of *must-alias analysis* in Datalog.
- In chapter 3 we present the logic behind our graph-based data structure.
- In chapter 4 we perform an evaluation of our implementation in Java by comparing it with the Datalog implementation.
- In chapter 5 we give our conclusions.

2. BACKGROUND

Our data structure is based on an inter-procedural must-alias analysis algorithm used by the Doop framework.

2.1 Example for Must-Alias Reasoning

In Figure 2.1 below, we introduce a small example, written in Java, to illustrate some basic concepts of must-alias reasoning. This example is also used in later chapters to showcase our structure and the algorithms used in it.

```java
class A {
   A field1;
   B field2;

   A(A field1, B field2) {
      this.field1 = field1;
      this.field2 = field2;
   }
}

class B extends A {
   A member1;

   B(B b) {
      this.field1 = b;
   }
}

public class Main {
   public static void main(String[] args) {
      B b1 = new B(null);
      B b2 = b1;
      A a2;
      A a1 = new A(a2, b1);
      if(args == null)
         a2 = new A(a1, b1);
      else
         a2 = new B(a1);
      b1.member1 = a2;
      a1.field2.member1 = a1;
   }
}
```

Fig. 2.1: Simple illustration of must-alias inferences.

As said before, a must-alias analysis is invaluable for providing information in executions that reach a specific program point. For example, in Figure 2.1, the alias pairs `a1.field2 ~ b1` and `a1.field1 ~ a2` established on line 24, hold for almost the entire body of method `main`. On line 26 (i.e., right after line 26), `a2.field1` is an alias for `a1` and `a2.field2` for `b1`, while on line 28, `a2.field1` is aliased to `a1`. We can compute alias pairs by variable assignments (e.g. `b2 ~ b1` on line 22) or by loads and stores, as in the above mentioned aliases.

Due to the fact that the derived must-alias pairs undoubtedly point to the same object, we need to invalidate aliases that no longer hold, on store or method call instructions that may interfere with objects within the alias pair. Thus, line 30 invalidates pair `b1.member1 ~ a2` and validates `b1.member1 ~ a1`, alongside with `a1.field2.member1 ~ a1`. That way the analysis remains accurate.

Some instructions in the program require special treatment. For instance, on lines 24, 26 and 28 we need to use inter-procedural reasoning to handle method calls. On line 29 we have to use intersection of the predecessors (i.e., lines 26 and 28). Both these cases are decomposed on chapter 3.

2.2 Example for SSA and phi

The input we use in this project is on a minimal single-static assignment (SSA) intermediate representation, as well as the intermediate language in which we present our algorithm.

```
    ...
24  A a2, a3, a4;
25  if(args == null)
26     a2 = new A(a1, b1);
27  else
28     a3 = new B(a1);
29  a4 = ϕ(a2, a3);
30  b1.member1 = a4;
    ...
```

Fig. 2.2: Part of Figure 2.1 with SSA.

Figure 2.2 is a part of the `main` method of Figure 2.1 written in SSA form, so that each variable is assigned exactly once. Due to the above, we are introduced to the concept of a ϕ (phi) function, in order to decide which of the newly created variables to use. For instance, the use of ϕ function in Figure 2.2 is to choose which value to give `a4`, between `a2` or `a3`, depending on which path was followed.

3. GRAPH-BASED DATA STRUCTURE

In this chapter we first give the background (i.e., the must-alias analysis model) on which we relied on to develop this project, and second we introduce the main subject of this thesis, the optimized data structure.

3.1 Must-Alias Analysis Model

The model of the inter-procedural must-alias analysis algorithm we use is expressed in Datalog. This language is truly declarative and ideal for iteration until fixpoint. Datalog has rules of the form "`H(x,z) ← P(x,y), Q(z,y).`" which means that if there exist values `x,y,z,w` so that predicates `P(x,y)` and `Q(z,y)` are both (the "`,`" suggests conjunction) true at the same time, then `H(x,z)` is inferred. On the left-hand side of the left arrow (←) is the head of the rule (i.e., the inferred facts) and on the right the body (i.e., the previously established facts).

Some syntactic sugar is also used, to simplify the rules and keep them brief. Multiple predicates are permitted in a rule head, as well as disjunction ("`;`") and negation ("`!`") in the rule body. In addition to the existential quantifier, which is implied in conventional Datalog, the rules also employ universal quantifier (∀), to be able to express the meaning of "for all". Note that the existential quantifier is interpreted as being outside the universal one. For example, the expression "∀`x: P(x,y) → Q(x,y,z).`" is interpreted as "there exist `y,z` such that for all `x` that `P(x,y)` holds then `Q(x,y,z)` also holds". Finally, symbol "`*`" signifies that the relation on which it applies is reflexively, symmetrically and transitively closed.

3.1.1 Analysis Relations

Figure 3.1 shows the domain of the must-alias analysis and three groups of relations. The language used to show the algorithm is a SSA intermediate language.

Input Relations:

As shown in the comments written next to each relation, MOVE represents instructions that assign a local variable to another, LOAD and STORE display instructions with assignments between heap object fields and local variables (i.e., reading/writing), CALL stands for virtual calls and PHI captures ϕ instructions useful for the SSA form. Finally, the NEXT relation gives the control-flow graph (CFG) successor of an instruction.

Other relations are more complex. Both FORMALARG and ACTUALARG show which variable is a formal/actual argument of a certain method/invocation at a certain index.

We assume that for each method the intermediate language program is in a single-return form. So, the FORMALRET encodes which variable is returned from the given method at the given instruction, and ActualRet indicates the return variable at the given invocation site.

ThisVar returns the *this* variable of a function. LookUp shows the method inferred from a specific signature and type. InMethod returns the containing method of a given instruction.

Relation Resolved keeps the variables that only point to an object with a unique dynamic type, for virtual calls to be resolved. It is computed by a may-point-to analysis. The predicate RootMethod holds the user-selected methods to be analyzed first.

```
V is a set of program variables              M is a set of method identifiers
S is a set of method signatures (name+type)  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 a set of access paths: V(.F)*
ℕ is the set of natural numbers
=================================================================================
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:ℕ, arg:V)            ActualArg(invo:I, n: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)                Resolved(var:V, type:T)
RootMethod(meth:M)
=================================================================================
MustAlias(instr:I, ctx:C, ap1:A, ap2:A)
MustCallGraphEdge(invo:I, ctx:C, toMth:M, toCtx:C)
Reachable(ctx:C, meth:M)
RebaseAtCall(instr:I, ctx:C, fromVar:V, toVar:V)
RebaseAtReturn(instr: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
```

Fig. 3.1: The analysis domain, input relations (Move, ...), computed relations (MustAlias, ...) and constructors (**AP**, ...).

Computed Relations:

The main output of the analysis, relation MustAlias, demonstrates that access path *ap1* and access path *ap2* have a must-alias relation (a.k.a. form an alias pair) right after instruction *instr* under context *ctx*.

The predicate MustCallGraphEdge holds the resolved virtual calls: invocation site *invo* under context *ctx* will call method *toMth* under context *toCtx*.

The Reachable relation displays which functions and under what context can be accessed. Eventually, RebaseAtCall and RebaseAtReturn hold the caller and callee variable pair to be remapped at a call site.

Constructors:

The constructor **`AP`** produces access paths and keeps information about them. For instance, " **`AP`**`(a1.field2.member1) = ap` " means that access path `ap` has length 3 and its elements are `a1`, `field2` and `member1`.

`RebaseAP` changes the base variable (in the example above the element `a1`) of the original access path to a new one.

Finally, **`NewContext`** produces contexts, after checking that the maximum context depth has not been reached. If that is not the case, it does not return a value.

3.1.2 Analysis Rules

Figure 3.2, below, shows the Datalog rules that are used for the must-alias analysis algorithm. They are separated into four groups.

Base rules:

The former rule decides the first reachable methods: methods to be analyzed under the special context value **`All`** (i.e., unconditionally).

The next four rules handle MOVE, PHI, LOAD and STORE instructions. The MOVE rule establishes an alias relation between variables `to` and `from`, PHI between variable `to` and the result of ϕ function, and finally LOAD and STORE create alias pairs that consist of the expression `base.fld` and the variable `to` or `from`, respectively.

The latter rule says that MUSTALIAS is reflexively, symmetrically and transitively closed.

Inter-procedural propagation rules:

The first rule infers MUSTCALLGRAPHEDGE predicates, at CALL instructions, from previously established RESOLVED, LOOKUP and REACHABLE relations, should the context depth allow it.

The next four rules are similar to the first. The REBASEATCALL rule computes re-mappings from actual to formal arguments and from the base variable of the call to the variable `this` of the callee. REBASEATRETURN computes the reverse mappings, as well as the mapping between the actual and the formal return value.

The next-to-last rule (that is, the first appearance of the MUSTALIAS predicate) handles the first instruction of each method. All the alias pairs that hold for every predecessor of the calling instruction are re-based, remapped and inferred for the first instruction of the callee. Additionally, the last rule handles the last instruction of each method. All the alias pairs that hold at the return instruction are re-based, remapped and inferred for the invocation site of the caller.

```
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) ←
  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.

REACHABLE(ctx, meth), REACHABLE(toCtx, toMeth) ←
  MUSTCALLGRAPHEDGE(i, ctx, toMeth, toCtx), INMETHOD(i, 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)).
```

Fig. 3.2: Datalog rules for must-alias analysis.

Reachability and access path expansion:

The first rule in this group expands the reachable methods. When the containing method of the invocation site is reachable, then the target function becomes reachable too, if the context depth allows it.

The second rule introduces extended access path aliases. If two variables or access path form an alias pair, then their extended access paths (with the same field suffix), if they exist, form an alias pair too.

From one instruction to the next:

The last rule of Figure 3.2 propagates an alias pair to an instruction, when it is not a store instruction or virtual call, and the alias pair holds for all its predecessors.

3.2 Overview of Data Structure

The purpose of this project is to find the certain aliases over variables and access paths of the program-under-analysis. To do so, we represent the alias information at each instruction as a directed-graph. The nodes of the graph contain the aliased variables at the specific point of the program, and the labeled-edges show the connections between access paths.

In every instruction, we apply the instruction semantics on the graph that holds right before the instruction and we return its altered version. Therefore, each instruction type (e.g. move, load, store) is handled differently.

Below we introduce some of the handling of these instructions. Note that, in each case, we most likely need to move a variable from a node to another. So, the import of a variable to a node, implies its removal from the former container node.

Move Instruction:

If the right-hand side variable already exists in a node, we add the left-hand side variable to the same node. If not, we create a new node with both variables.

Phi Instruction:

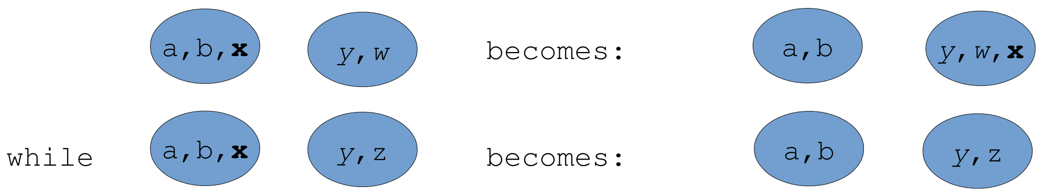

Fig. 3.3: Handle phi instruction `x = y, x = w` of alias graphs.

If the right-hand side variables of the phi instruction are all in the same node, then we also add the left-hand side variable to it. Otherwise, if they are in disjoint nodes, we just remove the left-hand side variable from the node in which it previously belonged, as shown in Figure 3.3 above (e.g. phi instruction `x = y, x = w,` or else $x = \phi(y, w)$).

Load Instruction:

A load instruction has the form *`to = base.field.`*
First, find the node *a* in which the *`base`* variable (right-hand side) exists, or, else, create a new one. Afterwards, find the node shown by an outgoing edge with label *`field`* from node *a*, or create it, and move the left-hand side variable, *`to`*, in it.

Store Instruction:

Store is the reverse action of load (i.e., *`base.field = from`*), but is handled similarly. First, find the node *a* in which the *`base`* variable (left-hand side) exists, or, else, create a new one. Second, find the node *b* in which the right-hand side variable, *`from`*, exists, or, else, create a new one. Last, create or move the existing edge of node *a* with label *`field`* to point to node *b*.

Call Instruction:

It is divided in two categories, the unresolved and the resolved call instructions.

Unresolved: An unresolved call instruction invalidates everything. This means that it inherits no graph from the previous instruction.

Resolved call: A resolved call, however, includes two cases. If the current context depth (i.e., number of nested methods that have been followed) is the maximum allowed, then the call instruction inherits only local aliases and discards heap aliases of the previous instruction's graph.

Else, if the current context depth is smaller that the maximum context depth, the call instruction is handled in two steps. The first step happens at the call site. The first instruction of the callee inherits the graph that holds right before the call instruction. In the latter step, at a return instruction, the call instruction inherits the graph of the last instruction of the callee after all the local variables are removed from the aliases.

A remapping of arguments takes place at call instructions, so that actual and formal arguments refer to the same object in memory. The same happens for return values at return instructions.

Briefly, the graph of a call instruction is inherited from the last instruction of the callee and from the previous instruction in any other case. However, we do not propagate information from predecessors in stores and unresolved calls, because we cannot be certain about the changes they might make to the heap.

3.3 Main Algorithms

Algorithm: all-aliases (ap, k)

Step 1: Find target node that access path *ap* points to, following the edges that match each field of *ap*.
Step 2: Going backwards *k–1* directed edges, find all access paths (of length *k*) that can reach the target node.

For example, in Figure 3.4, which shows the alias graph after line 29 of our example in section 2.1, we can find all aliases of access path `b1.member1` of length 3 by following edge `member1` from node `b1` (and so reaching node `a2`) and then finding all paths with length 2 (that is to go backwards 2 directed edges) that reach the same node (e.g. `a2.field1.field1` and `a1.field2.member1`).

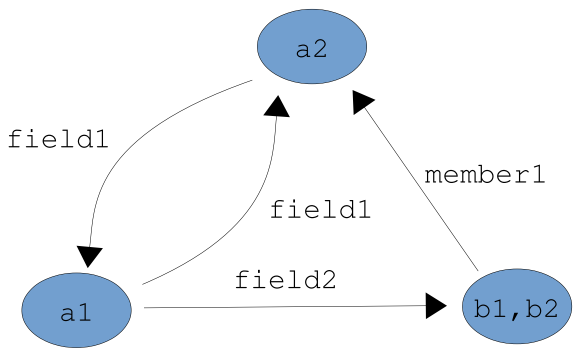

Fig. 3.4: Example of alias graph.

Algorithm: intersect (g1, g2)

Step 1: For every nodes i, j of the original two graphs *g1* and *g2*, we create new node (i,j) whose variables are the intersection of the variables of i and j.
Step 2: For every label f, if *g1* has an edge (i,k) with label f, and *g2* has an edge (j, l), also with label f, the intersection result has an edge ((i,j), (k,l)) with label f.

The algorithm establishes all possible combinations with nodes from both graphs and creates all edges that still hold and connect the emerged nodes.

We use this algorithm to create an alias-graph for instructions that have multiple predecessors (i.e., instruction after an if-then-else or a while clause). First we find the intersection of two graphs, then we intersect the result with a third graph, and so on.

For instance, on line 29 of our example in section 2.1, after the if-then-else clause, we need to intersect the graphs of lines 26 and 28. While all the other alias pairs continue to hold, `a2.field2 ~ b1` is invalidated.

It is common for many produced nodes to have empty variable sets. The example in Figure 3.5 illustrates why.

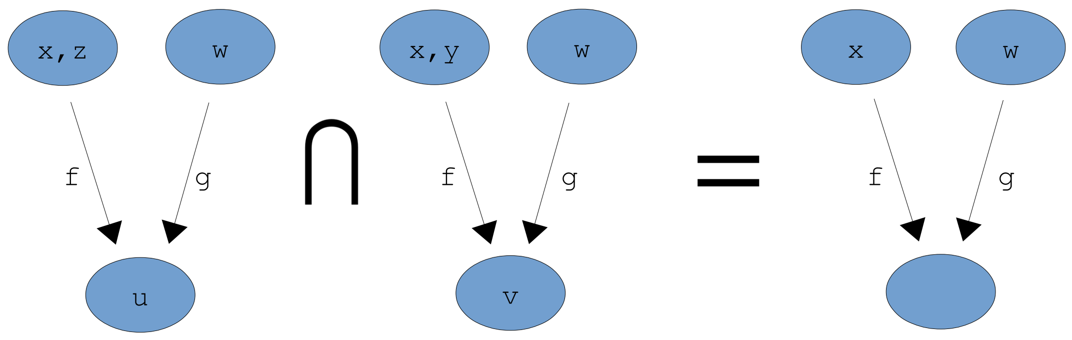

Fig. 3.5: Intersection of alias graphs.

In some cases, like the above, the existence of the empty node is essential, because it encodes that access paths `x.f` and `w.g` are aliased, while in other cases the produced node has no use and will be discarded by the gc algorithm.

Algorithm: gc (g)

Garbage-collection algorithm, eliminates all nodes in graph *g* that either
- contain a single variable and have no incoming or outgoing edges
- contain no variables and have zero incoming edges or have only one incoming and no outgoing edges

We use this algorithm to garbage-collect empty nodes and nodes that no longer denote access path aliasing. Such nodes can arise duo to node intersection. This way we establish maximum efficiency throwing away useless information.

Algorithm: remove-local-variables (g)

Step 1: From every node that contains variables to be remapped (i.e., arguments, return variable, “this”), we remove all variables that will not be remapped.
Step 2: We keep all nodes that connect with a remapping node. That is, they have at least one incoming or outgoing edge to a remapping node or to a node that connects to them.
Step 3: We remove all the remaining nodes.

The algorithm is used for return instructions to get rid of the local-variable-aliases that only stand inside the callee function. We use this to remove from graph useless information about variables that no longer exist.

3.4 Input Files

To express the analysis in Java (same as in Datalog) we represent the program-under-analysis as input relations that encode environment information (i.e., program instructions, CFG information etc).

As said above (section 2.2), we assume that our input is on a minimal SSA intermediate representation, so that each variable is not assigned more than once.

The input relations used in our Java implementation are basically the same with most of the analysis relations presented in section 3.1.1.

3.5 Output Files

The Figure 3.6, below, shows the output relations that our program exports to introduce the alias results, using the domain of the analysis used in our Java implementation (as well as in the Datalog implementation) shown in Figure 3.1.

```
NODES(i:I, nodeId:ℕ, var:V)
EDGES(i:I, nodeId1:ℕ, nodeId2:ℕ, fld:F)
MUSTALIASPAIRS(i:I, var1:V, var2:V)
ACCESSPATHPAIRS(i:I, ap1:A, ap2:A)
```

Fig. 3.6: Output relations that encode CFG information.

The NODES shows all variables that exist in each node of the graph, for every instruction.

The EDGES presents all pairs of nodes and the label of the edge that connect them, for every instruction.

The MUSTALIASPAIRS relation returns all variable pairs that are aliased at a specific instruction. That is, it shows all variables that belong to the same node in the same graph.

Finally, the ACCESSPATHPAIRS presents all aliased access path (AP) pairs of a specific depth that hold at a particular instruction. The desirable depth is given by the user as a configuration parameter.

4. EXPERIMENTAL RESULTS

4.1 Setup

We use a 64-bit machine with an Intel Core i5-5200U 2-core CPU at 2.20GHz. The machine has 8GB of RAM.

We experiment with the DaCapo benchmark programs v.2006-10-MR2 under JDK 1.7.. We use the LogicBlox Datalog engine, v.3.10.14.

4.2 Evaluation

Initial comparison across various benchmarks:

The two must-alias analysis implementations, in Datalog and Java, have only minor differences in experimental results. They are functionally equivalent. Their major difference is that in Datalog the aliased access paths can have a finite length, while in Java there is no such restriction.

Table 4.1: Comparison between Datalog and Java implementation, on speed and number of must-point-to pairs.

Benchmark	**optimized time (sec)**	**original time (sec)**	**speedup (%)**	**#must point-to (Java)**	**#must point-to (Datalog)**
antlr	36	1156	32.1	8646	7430
bloat	25	686	27.4	7536	5132
chart	40	1714	42.9	3082	2587
eclipse	30	692	23.1	6311	5423
fop	25	691	27.6	1284	1103
hsqldb	23	700	30.4	930	727
jython	36	832	23.1	10476	10637
luindex	13	469	36.1	1822	1551
lusearch	14	531	37.9	2180	1940
pmd	30	689	23.0	3361	3169
xalan	28	915	32.7	4876	4642

To compare the initial implementation with the optimized data structure, we show their execution time and the number of must-point-to pairs they infer. In Table 4.1 we run our analyses for context depth 1 and maximum access path length 2. We refer to the Java implementation as “Optimized” and to the Datalog implementation as “Original”.

As can be seen, our Java implementation achieves significant speedups, in average 30.6% (from 23.0% to 42.9%) of the Datalog implementation. Note that, the “optimized” running time increases by the import and the export time, of the previously concluded facts and the results of the analysis, respectively.

Comparison varying access path length:

For further presentation of the level of improvement with our data-structure, we compare the same elements produced with various maximum access path lengths of the original implementation with the optimized implementation. The restriction of the maximum access-path length does not affect the “optimized” implementation, as mentioned before. Its running time and number of must point-to pairs are included just for reference.

Table 4.2 shows the running time for both implementations, as well as the number of must-point-to pairs for maximum path lengths of 1, 2 (same as in Table 4.1) and 3, for the xalan benchmark (for which the Java implementation had near to the average speedup in Table 4.1).

Table 4.2: Comparison between Datalog and Java implementation (on the xalan benchmark) when varying the maximum access-path length.

Access path length	**optimized time (sec)**	**original time (sec)**	**speedup (%)**	**#must point-to (Java)**	**#must point-to (Datalog)**
1	28	339	12.1	4876	4640
2	28	915	32.7	4876	4642
3	28	1700	60.7	4876	4642

We observe that the more the access-path length increases the more the speedup of the optimized data structure grows. The divergence of the number of must point-to pairs between the implementations, on the other hand, seems to hold firm.

Comparison varying context depth:

Similarly, we experiment with the context depth. Table 4.3 demonstrates the speed performance of the two analyses and the number of must-point-to pairs for context depth 0, 1 (same as in Table 4.1) and 2, once more for the xalan benchmark.

Same as in the above analysis results, the speedup rises excessively while the context-depth increases. Once more, the number of must point-to pairs of the Java implementation remains on the same level with the Datalog implementation.

Table 4.3 Comparison between Datalog and Java implementation (on the xalan benchmark) when varying the maximum context depth.

Context depth	optimized time (sec)	original time (sec)	speedup (%)	#must point-to (Java)	#must point-to (Datalog)
0	75	104	1.9	4293	3868
1	28	915	32.7	4876	4642
2	28	1824	65.1	5301	4933

5. CONCLUSIONS

We presented a graph-based data structure that implements a declarative model of a must-alias analysis over access paths, that is already implemented in Datalog and in use in the Doop framework.

Based on our experimental results, this re-implementation improves significantly the speed, while at the same time increases the quantity of the inferences by being able to compute aliases for unlimited access path lengths.

ACRONYMS AND ABBREVIATIONS

Abbreviation	Full Name
SSA	static single assignment
AP	access path
CFG	control flow graph
gc	garbage collect

REFERENCES

[1] Yannis Smaragdakis and George Balatsouras. Pointer Analysis. Foundations and Trends in Programming Languages, vol. 2, no. 1, pp. 1-69, 2015.

[2] S. Dasgupta, C. H. Papadimitriou, and U. V. Vazirani. 2006. Algorithms.

[3] Martin Bravenboer and Yannis Smaragdakis. Strictly declarative specification of sophisticated points-to analyses. OOPSLA, 2009.

[4] Martin Bravenboer and Yannis Smaragdakis. Exception Analysis and Points-to Analysis: Better Together. ISSTA, 2009.

[5] Bjarne Steensgaard. Points-to Analysis in Almost Linear Time. POPL, 1996.

[6] Arnab De and Deepak D' Souza. Scalable Flow-Sensitive Pointer Analysis for Java with Strong Updates. ECOOP, 2012.

[7] George Kastrinis and Yannis Smaragdakis. Efficient and Effective Handling of Exceptions in Java Points-To Analysis. CC, 2013.

[8] John Whaley and Monica S. Lam. Cloning-Based Context-Sensitive Pointer Alias Analysis Using Binary Decision Diagrams. PLDI, 2004.

[9] Stephen J. Fink, Eran Yahav, Nurit Dor, G. Ramalingam, and Emmanuel Geay. Effective typestate verification in the presence of aliasing. TOSEM, 2008.

[10] Michael Hind. Pointer analysis: haven't we solved this problem yet? PASTE, 2001.

[11] Donglin Liang and Mary Jean Harrold. Efficient Points-To Analysis For Whole-Program Analysis. ESEC / SIGSOFT FSE, 1999.

[12] Stephen M Blackburn, Robin Garner, Chris Hoffmann, Asjad M Khan, Kathryn S. McKinley, Rotem Bentzur, Amer Diwan, Daniel Feinberg, Daniel Frampton, Samuel Z. Guyer, Martin Hirzel, Antony Hosking, Maria Jump, Han Lee, J. Eliot B. Moss, Aashish Phansalkar, Darko Stefanovic, Thomas VanDrunen, Daniel von Dincklage, Ben Wiedermann. The DaCapo Benchmarks: Java Benchmarking Development and Analysis. OOPSLA, 2006.

[13] Suan Hsi Yong, Susan Horwitz, and Thomas Reps. Pointer analysis for programs with structures and casting. PLDI, 1999.

[14] Konstantinos Ferles. (2015). *General Declarative Must-Alias Analysis* (Unpublished master's thesis). National and Kapodistrian University of Athens.