Title	Counterexample-guided abstraction refinement for points-to analysis of object-oriented programs
Author(s)	Vu, Quang Vinh
Citation	
Issue Date	2016-09
Type	Thesis or Dissertation
Text version	author
URL	http://hdl.handle.net/10119/13740
Rights	
Description	Supervisor: , ,

Counterexample-guided abstraction refinement for points-to analysis of object-oriented programs

Vu Quang Vinh

School of Information Science
Japan Advanced Institute of Science and Technology
September, 2016

Master's Thesis

Counterexample-guided abstraction refinement for points-to analysis of object-oriented programs

1410213 Vu Quang Vinh

Supervisor : Professor Tachio Terauchi
Main Examiner : Professor Tachio Terauchi
Examiners : Professor Mizuhito Ogawa
Associate professor Nao Hirokawa

School of Information Science
Japan Advanced Institute of Science and Technology

August, 2016

Acknowledgement

Contents

Chapter 1

Introduction

1.1 Points-to analysis

Points-to analysis is a static analysis technique for programs. Its goal is finding out a set O of objects (or storage locations) for every pointer p (variables or heap reference) in a program. It means that each object in O may be pointed to by p while the program is running. See the Java-like program below for an example.

Example 1.1.1 (A simple Java program)**.**

```
1   public class P{
2        P id(P obj){return obj;}
3   }
4   public class C extends P{
5        void donothing(){}
6   }
7   public class M{
8        public static void   main(String[] agrs){
9             P objP, objC1, objC2;
10            objP = new P();
11            objC1 = new C();
12            objC2 = objP.id(objC1);
13            C objC = (C) objC2;
14        }
15  }
```

There are 3 pointers: $objP$, $objC1$ and $objC2$. Their points-to sets are {'10'}, {'11'} and {'11'} respectively. Here '10' and '11' represent for the objects created at line numbers 10 and 11.

In general, the points-to problem is undecidable. It is solved under the soundness property, i.e. finding the set $O \supset O^*$, where O^* is obtained when the program actually runs.

1.2   Problem

Points-to analysis gives the answer for various verification problems such as downcast checking and null-pointer detection. In this document, I focus on the downcast problem. Consider the example program in Section 1.1, class P is a superclass of class C and $objC2$'s points-to set is {'11'}. The statement at line 13, $C\ objC\ =\ (C)\ objC2$;, is a downcast. The downcast problem is finding a set of downcasts of an object-oriented program that will fail when the program runs. Similar to points-to analysis, the downcast problem is also undecidable. The points-to analysis result helps to solve the downcast problem. In this case, the above downcast does not fail.

We formally restate the downcast problem as below.
For a Java program, there are:

- A set of classes C, a variables set V, an object set O

- An inheritance relation set $R = \{(a,b)|\ (a,b) \in (C \times C)\ and\ a \neq b\}$ - $(a,b) \in R$ means that class a inherits from class b

- A variable-typed relation set $T = \{(x,a)|\ x \in V\ and\ a \in C\}$ - $(x,a) \in T$ means that variable x is declared as a pointer of class a

- An object-typed relation set $T' = \{(o,a)|\ o \in O\ and\ a \in C\}$ - $(o,a) \in T'$ means that object o is an instance of class a

- A set of downcast D and each $d_i \in D$ is (x,a) where $\exists b\ (x,b) \in T$ and a is a subclass of b (b is a superclass of a)

In Example 1.1.1, there are:

- $C = \{\text{'}P\text{'}, \text{'}C\text{'}\}$

- $V = \{\text{'}obj\text{'}, \text{'}objP\text{'}, \text{'}objC1\text{'}, \text{'}objC2\text{'}, \text{'}objC\text{'}\}$

- $O = \{\text{'}10\text{'},\ \text{'}11\text{'}\}$

- $R = \{(\text{'}C\text{'}, \text{'}P\text{'})\}$

- $T = \{(\text{'}obj\text{'},\ \text{'}P\text{'}), (\text{'}objP\text{'},\ \text{'}P\text{'}), (\text{'}objC1\text{'},\ \text{'}P\text{'}), (\text{'}objC2\text{'},\ \text{'}P\text{'}), (\text{'}objC\text{'},\ \text{'}C\text{'})\}$

- $T' = \{(\text{'}10\text{'}, \text{'}P\text{'}), (\text{'}11\text{'}, \text{'}C\text{'})\}$

- $D = \{(\text{'}objC2\text{'}, \text{'}C\text{'})\}$

Definition 1.2.1 (Superclass). A class b is a superclass of class a, iff $(a,b) \in R^+$ where R^+ is the transitive closure of R. By contrast, a is a subclass of b, iff b is a superclass of a.

Definition 1.2.2 (Points-to). Points-to information is a set $P \subseteq \{(x,o)|\ x \in V\ and\ o \in O\}$. $(x,o) \in P$ means that variable x points to object o while the Java program is running.

Definition 1.2.3 (Fail downcast)**.** A downcast $d = (x, a)$ is a fail downcast, iff $\exists (x, o) \in P$ and o is neither an instance of class a nor an instance of a's subclass.

The goal of the downcast problem is finding a partition (S, F) of D, where S is the set of downcasts that will not fail and F is the set of downcasts that may fail. With the same set D, a solution (S_0, F_0) is more precise than another solution (S_1, F_1) if and only if $S_0 \supset S_1$. The precise result for Example 1.1.1 is $(S = \{('objC2', 'C')\}, D = \{\})$. Note that the points-to information P is the result from a points-to analysis. P decides which downcast $d_i \in D$ is a fail downcast.

Recall, the points-to problem is undecidable and the downcast problem is also undecidable. Hence, we can only compute the set F of downcasts that may fail, instead of finding the set F^* of downcasts that actually fail. Our analysis minimizes the set F as much as possible.

1.3   Method and experiments

Solving downcast problem with points-to analysis result is straightforward and simple by following the Definition 1.2.3 to check the super class relation of all points-to information which are related with a downcast. So that, our research aims to improve the points-to analysis technique.

Many approaches to points-to analysis have been proposed, such as Andersen's points-to analysis [7], the context-free language (CFL) reachability approach (also called the pushdown approach, see section 4.1). In my master's thesis research, I use the Doop framework [3] which follows the Andersen's approach. The Doop framework also provides context-sensitive analyses [1]. The bound on the method call sequence length is a parameter of Doop, which can be used to set the context sensitivity level (the length of the context chain, see section 2.5). The context-sensitive points-to analysis can be considered as finite state approach. Unfortunately, the number of states grows hugely when the bounding depth increases (in my experiment, 2 is the limit).

Zhang et al. [2] have proposed an *adaptive* pointer analysis method whereby the analysis precision can vary for the different parts of the program. The idea is to analyze the parts that are important to the analysis result with a high precision while analyzing the rest cheaply with a lower precision. I extend their work by using a more fine-grained granularity of program parts. Particularly, our approach uses different precision levels for different method call sites, whereas [2] does not.

Instead of analyzing with an expensive abstraction, we follow the approach of [2], that the precision of an expensive abstraction can be obtained by cheaper abstractions. We use the counterexample-guided abstraction refinement (CEGAR) based algorithm. The CEGAR-based algorithm repeatedly performs an analysis step and a refinement step. In the analysis step, the points-to analysis is executed with an abstraction and counterexample knowledge is accumulated. In the refinement step, a new abstraction is chosen by the guidance of the accumulated counterexample knowledge for the next iteration.

Because I use the Doop framework for experiments, our points-to analysis keeps the same features:

- Context-sensitive analysis, context-sensitive heap, on-the-fly call-graph discovery, precise exception analysis, sophisticated reflection analysis, field-sensitive analyses.

- Flow-insensitive pointer analysis, array-element insensitive analysis.

My analysis handles all features of Java, including the features below.

- Inheritance, exception, reflection, recursion.

I do experiments on the DaCapo benchmark [6] with three programs, antlr, chart and xalan. I compare our adaptive context-sensitive method with the algorithms provided by the Doop framework. Our CEGAR-based algorithm has several hyper-parameters, I try with different values to have a good parameter set. How to appropriately set such hyper-parameters is left for a future work.

1.4 Contributions

- I propose a fine-grained adaptive points-to analysis. It is able to precisely analyze the essential program parts, while save the cost by treating the other parts cheaply.

- We do experiment on the Doop framework with real Java programs of the DaCapo benchmark.

- Using MaxSAT for abstraction choosing is firstly proposed by [2], but it does not work with the Doop framework and big Java programs such as DaCapo's programs. I propose a partitioned (for performance) and repeated (for precision) MaxSAT abstraction picking.

1.5 Outline for later chapters

- Chapter 2 describes the background knowledge of points-to analysis

- Our proposed method is represented in Chapter 3

- Chapter 4 compares our method with the other works

- Details of the implementation and some additional problems are described in Chapter 5

- Experiment data and result are in Chapter 6 and I summarize my work in Chapter 7

Chapter 2

Background

Our points-to analysis follows the Andersen's approach, and it is implemented by the Datalog language [3]. Hence, this chapter only talks about Andersen's points-to analysis under the Datalog syntax.

2.1 Static analysis in Datalog

Datalog is a declarative logic programming language and it is based on predicate logic. Its syntax is described as below.

Definition 2.1.1 (Syntax of Datalog program)**.**

$$
\begin{array}{llll}
(Program) & C := \bar{r} & (rule) & r := l \leftarrow \bar{l}. \\
(Literal) & l := p(\bar{a}) & (argument) & a := v|d
\end{array}
$$

The over line represents a sequence. We interpret a sequence as the set of the elements of the sequence when it is clear from the context. For example, a Datalog program C is a set of rules $\bar{r}$. A rule $r := l \leftarrow \bar{l}$ has a literal as the head and a sequence of literals as the body. However, in the later, there is a rule that has a head with many literal. It simply can be considered as a set of rules that have the same body.

Definition 2.1.2 (Auxiliary definitions and notations)**.**

$$
\begin{array}{llll}
(Predicate-names) & p \in \mathbf{P} = \{p1, p2, ..\} & (variables) & v \in \mathbf{V} = \{x, y, ..\} \\
(Constants) & d \in \mathbf{D} = \{0, 1, .., \text{'c'}, \text{'d'}, ..\} & (tuples) & \in \mathbf{T} = \mathbf{P} \times \mathbf{D}^* \\
(Downcasts) & q \in \mathbf{Q} \subseteq \mathbf{T} & (abstractions) & A \in \mathbf{A} \subseteq \mathcal{P}(\mathbf{T}) \\
(Substitutions) & \sigma \in \Sigma = \mathbf{V} \to \mathbf{D}
\end{array}
$$

A literal (predicate) consists a predicate name p and a set of arguments (free variables and constants). A tuple is an instance of a predicate by applying a substitution σ. Not only our downcast but also the other queries can be easily represented as tuples. The *abstraction* is already defined now but reader can skip it until Section 2.5.

A Datalog program starts with a set of *facts*, where each *fact* is a tuple, and repeatedly apply rules $\bar{r}$ to create more facts until a fixed point reached. In the Datalog language,

there is no duplicated tuple, although the same tuple might be created by applying different rules.

Definition 2.1.3 (Semantics of Datalog)**.**

$$[[C]] \in \mathcal{P}(\mathbf{T}) \qquad\qquad F_C, f_c \in \mathcal{P}(\mathbf{T}) \to \mathcal{P}(\mathbf{T})$$
$$[[C]] = lfp(F_C, T_0) \qquad\qquad F_C(T) = T \cup \bigcup\{f_c(T) \mid c \in C\}$$
$$f_{l_0 \leftarrow l_1, ..., l_n}(T) = \{\sigma(l_0) \mid \sigma(l_k) \in T \text{ for } 1 \leq k \leq n\}$$

where T_0 is the starting facts.

Theorem 2.1.1. A Datalog progam only ends when no more tuple can be produced.

$$[[C]] = F_C([[C]])$$

2.2   Simple points-to analysis program

Example 2.2.1 (A simple points-to analysis program's components [12])**.**
The constants D are:

- Program's pointers P (variables)
- Program's heap allocations H
- Methods M
- Method signatures S

- Program's instructions I
- Object types T
- Integer number N

The input predicates are:

- $Allocation(pointer : P, heap : H, inMethod : M)$
- $Assign(to : P, from : P)$
- $VirtualInvocation(base : P, sig : M, invocation : I, inMethod : M)$
- $DefinedArg(method : M, n : N, arg : P)$
- $DynamicArg(invocation : I, n : N, arg : P)$
- $DefinedReturn(method : M, rePointer : P)$
- $DynamicReturn(invocation : I, rePointer : P)$
- $ObjectType(heap : H, type : T)$
- $LookUp(type : C, sig : S, method : M)$
- $PointerType(pointer, type : T)$
- $InvokedMethod(invo : I, method : M)$
- $Subtype(subtype : C, supertype : T)$
- $Cast(to : P, from : P, type : T)$

The analyzed predicates are:

- $VarPointsTo(pointer : P, heap : H)$
- $CallGraph(invo : I, method : M)$
- $InterMethAssign(to : P, from : P)$
- $Reachable(method : M)$

The variable set is all the above predicate parameters.

The input predicates encode analyzed program's instructions. For example, *Allocation(var, heap, inMeth)* represents object allocation instructions. A new heap object, *heap*, is assigned to the variable, *var*, in the method, *inMeth. Assign, Load, Store, VirtualInvocation* respectively represents assigning, object-field loading, object-field storing and method invocation. *DefinedArg* encodes the local parameters of the method *meth* and *DynamicArg* encodes the variables passed to method *meth* by an invocation *invo*. Similarly, *DefinedReturn* and *DynamicReturn* encode the local return variable and the return object of the invocation *invo*. The *Subtype* denotes superclass relation from Definition 1.2.1. Note that our points-to analysis supports on-the-fly call graph computation so that a method invocation described by *VirtualInvocation* is a virtual call.

The analyzed facts (instances of the analyzed predicates) are the intermediate tuples and the output of the points-to analysis. For example, *VarPointsTo* tuples are the main output (the set P in the definition 1.2.2). Those analyzed facts are created by repeatedly applying analysis rules from the input facts. Consider the example 1.1.1, the input facts

Table 2.1: The input facts for the example 1.1.1

Line number	Instructions	Facts
2	P id(P obj){return obj;}	DefinedArg('P:id:P', 0, 'obj'), DefinedReturn('P:id:P', 'obj')
9	P objP, objC1, objC2;	PointerType('objP','P'), PointerType('objC1','P'), PointerType('objC2','P')
10	objP = new P( ) ;	ObjectType('10','P'), Allocation('objP', '10', 'main')
11	objC1 = new C( ) ;	ObjectType('11','C'), Allocation('objC1', '11', 'main')
12	objC2 = objP.id(objC1);	VirtualInvocation('objP','id:P', '12:objC2 = objP.id(objC1)'), DynamicArg('12:objC2 = objP.id(objC1)', 0, 'objC1'), DynamicReturn('12:objC2 = objP.id(objC1)', 'ObjC2'), ThisVar('P:id:P', 'objP')
13	C objC = (C) objC2 ;	PointerType('objC','C'), Cast('objC', 'objC2', 'C')
	others	Subtype('C','P'), LookUp('P', 'id:P', 'P:id:P'), InvokedMethod('12:objC2 = objP.id(objC1)','P:id:P')

are shown in Table 2.1. '$P : id : P$' is the method signature for the function *id* of class P with one parameter which has type P. '$12 : objC2 = objP.id(objC1)$' is a method invocation. An instruction can be simply represented by its line number. Hence, the heap allocations are denoted as line numbers, {'10', '11'}. After analyzing, the analyzed facts are listed below.

$VarPointsTo(`objP', `10')$ $VarPointsTo(`objC1', `11')$
$VarPointsTo(`objC2', `11')$ $VarPointsTo(`objC', `11')$
$VarPointsTo(`obj', `11')$ $CallGraph(`12{:}objC2 = objP.id(objC1)', `P:id:P')$
$InterMethAssign(`obj', `objC1')$ $InterMethAssign(`objC2', obj)$
$Reachable(`main')$       $Reachable(`P:id:P')$

The simple points-to analysis rules are in Example 2.2.2. I used it to demonstrate how actually the analysis runs.

Example 2.2.2 (The simple points-to analysis rules).
(1) $VarPointsTo(var, heap) \leftarrow Allocation(var, heap, inMeth).$
(2) $VarPointsTo(to, heap) \leftarrow Assign(to, from), VarPointsTo(from, heap).$
(3) $VarPointsTo(to, heap) \leftarrow InterMethAssign(to, from), VarPointsTo(from, heap).$
(4) $VarPointsTo(to, heap) \leftarrow Cast(to, from, type), VarPointsTo(from, heap).$
(5) $CallGraph(invo, toMeth) \leftarrow VirtualInvocation(base, sig, invo, inMeth),$
                                  $VarPointsTo(base, heap),$
                                  $ObjectType(heap, heapT), LookUp(heapT, sig, toMeth).$
(6) $InterMethAssign(to, from) \leftarrow DefinedArg(meth, n, to), DynamicArg(invo, n, from),$
                                    $CallGraph(invo, meth).$
(7) $InterMethAssign(to, from) \leftarrow DefinedReturn(meth, from), DynamicReturn(invo, from),$
                                    $CallGraph(invo, meth).$
(8) $Reachable(`main') \leftarrow .$
(9) $Reachable(meth) \leftarrow Reachable(inMeth), CallGraph(invo, inMeth), InvokedMethod(inMeth, meth$

Two first VarPointsTo tuples are created from the Alloc tuples.

$$VarPointsTo(`objP', `10') \leftarrow Allocation(`objP', `10', `main').$$

$$VarPointsTo(`objC1', 111') \leftarrow Allocation(`objC1', `11', `main').$$

After applying rules (5), (6) and (7), two $InterMethAssign$ tuples are derived, $InterMethAssign(`obj', `objC1')$ and $InterMethAssign(`objC2', `obj')$. Then, rule (3) is applied twice.

$VarPointsTo(`obj', `11') \leftarrow InterMethAssign(`obj', `objC1'), VarPointsTo(`objC1', `11').$

$VarPointsTo(`objC2', `11') \leftarrow InterMethAssign(`objC2', `obj'), VarPointsTo(`obj', `11').$

Finally, the last $VarPointsTo$ tuple comes from rule (4).

$VarPointsTo(`objC', `11') \leftarrow Cast(`objC', `objC2', `C'), VarPointsTo(`objC2', `11').$

In addition, $Cast(`objC', `objC2', `C')$ is a downcast because of $PointerType(`objC2', `P')$ and $Subtype(`C', `P')$. Moreover, this downcast is not a fail downcast because there is only $VarPointsTo(`objC2', `11')$.

8

2.3 Context-sensitive points-to analysis

This section talks about how to deal with method call sequence, looping method call or especially recursive. Consider the simple Java program in Example 2.3.1 below.

Example 2.3.1 (A simple Java program with method calls)**.**

```
1   public class P{
2        P id(P obj){return obj;}
3   }
4   public class C extends P{
5        void donothing(){}
6   }
7   public class M{
8        public static void  main(String[] agrs){
9             P objP1, objP2, objC1, objC2;
10            objP1 = new P();
11            objC1 = new C();
12            objC2 = objP1.id(objC1);
13            objP2 = objP1.id(objP1);
14            C objC = (C) objC2;
15            C objP = (C) objP2;
16        }
17  }
```

By applying the simple points-to analysis rules in Example 2.2.1, it is easy to compute

$$InterMethAssign(\text{'}obj\text{'},\text{'}objC1\text{'}), \ InterMethAssign(\text{'}objC2\text{'},\text{'}obj\text{'}),$$

$$InterMethAssign(\text{'}obj\text{'},\text{'}objP1\text{'}), \ and \ InterMethAssign(\text{'}objP2\text{'},\text{'}obj\text{'}).$$

Then,

$$VarPointsTo(\text{'}objC2\text{'},\text{'}10\text{'}), \ VarPointsTo(\text{'}objC2\text{'},\text{'}11\text{'}),$$

$$VarPointsTo(\text{'}objP2\text{'},\text{'}10\text{'}), \ andVarPointsTo(\text{'}objP2\text{'},\text{'}11\text{'})$$

are derived. But, it is clearly seen that variable $objC2$ only points to '11' and variable $objP2$ only points to '10'. This failure is made because two method invocation at line 12 and 13 are treated under the same context. The tuples in the two pairs below

$$(InterMethAssign(\text{'}obj\text{'},\text{'}objC1\text{'}), \ InterMethAssign(\text{'}objP2\text{'},\text{'}obj\text{'}),)$$

and

$$(InterMethAssign(\text{'}obj\text{'},\text{'}objP1\text{'}), \ InterMethAssign(\text{'}objC2\text{'},\text{'}obj'\text{'}),)$$

do not coexist. In order to encode this kind of information, straightforwardly a *context* term is added to the predicates. Basically, each method invocation must have a distinguished *context*. These *contexts* encode the scope of pointers (local variables and object fields). An instance of the pointer (variable) is cloned for each *context*. For example, there should be two different instances of *obj* for two method invocations at line 12 and 13. Additionally, there are two kinds of *context*, local-variable *context* (shortly *context*) and *heap context*. The *context* and *heap context* are respectively used for local variables and object fields.

Definition 2.3.1. *Context* and *heap context* (*HContext*) in context-sensitive analysis are defined as follows:
$$Context = HContext = Lab^*$$

where *Lab* is the set of line numbers.

Example 2.3.2 (A simple context-sensitive points-to analysis program's components [12]).
Two following constants set are added.

- C is a set of (local-variable) contexts

- HC is a set of heap contexts

The input predicates are similar to Example 2.2.1.
The analyzed predicates are:

- $VarPointsTo(var : V, ctx : C, heap : H, hctx : HC)$

- $CallGraph(invo : I, callerCtx : C, meth : M, calleeCtx : C)$

- $InterMethAssign(to : V, toCtx : C, from : V, fromCtx : C)$

- $Reachable(meth : M, ctx : C)$

Additionally, there are two functions:

- $Record(heap : H, ctx : C) = newHCtx : HC$

- $Merge(heap : H, hctx : HC, invo : I, ctx : C) = newCtx : C$

The atoms of a context-sensitive points-to analysis are shown in Example 2.3.2. In comparison to the Example 2.2.1, two *context* sets, C and HC, are added. The input predicates are the same, it means that we have no change in the input tuples. All analyzed predicates become context-sensitive and they are inserted by *contexts*. The $InterMethAssign$ predicate becomes $InterMethAssign(to, toContext, from, fromContext)$ where *toContext* and *fromContext* are the contexts related to variables *to* and *from* respectively. For the output predicates, $VarPointsTo(var, ctx, heap, hctx)$, the context *ctx* and the *var* identify a cloned variable for the variable *var* in the context *ctx*. Similarly, $(heap, hctx)$ pair denotes a cloned heap of the object created at *heap* in the heap context *hctx*.

Definition 2.3.2. *Merge* and *Record* functions are defined as follows:

$$Record \in \textbf{Record} : Lab \times Context \rightarrow HContext$$

$$Merge \in \textbf{Merge} : Lab \times HContext \times Lab \times Context \rightarrow HContext$$

where *Lab* is the set of line numbers.

The constants are mostly created at the parsing phase (i.e. the constants in Example 2.2.1 are produced in the parsing phase), but contexts instances are generated while the analysis rules are applied (inference). The **Merge** function constructs *contexts* and the **Record** function constructs *heap contexts* (the basic Datalog language does not have the function but some variant have). They are in Definition 2.3.2. Note that *heap* and *invo* in Example 2.3.2 are the line numbers. A *heap context* is created when a new object is allocated with a heap allocation. A method invocation requires a scope identifier for the local variables, then a *context* is constructed by the **Merge** function. See Example 2.3.3.

Example 2.3.3 (The simple context-sensitive points-to analysis rules)**.**

(1) $Record(heap, ctx) = hctx,$
 $VarPointsTo(var, ctx, heap, hctx) \leftarrow Allocation(var, heap, inMeth).$

(2) $VarPointsTo(to, ctx, heap, hctx)$
 $\leftarrow Assign(to, from), VarPointsTo(from, ctx, heap, ctx).$

(3) $VarPointsTo(to, toCtx, heap, hctx)$
 $\leftarrow InterMethAssign(to, toCtx, from, fromCtx),$
 $VarPointsTo(from, fromCtx, heap, hctx).$

(4) $VarPointsTo(to, ctx, heap, hctx)$
 $\leftarrow Cast(to, from, type), VarPointsTo(from, ctx, heap, hctx).$

(5) $Merge(heap, hctx, invo, callerCtx) = calleeCtx,$
 $CallGraph(invo, callerCtx, toMeth, calleeCtx)$
 $\leftarrow VirtualInvocation(base, sig, invo, inMeth),$
 $VarPointsTo(base, callerCtx, heap, hctx),$
 $ObjectType(heap, heapT), LookUp(heapT, sig, toMeth).$

(6) $InterMethAssign(to, calleeCtx, from, callerCtx)$
 $\leftarrow DefinedArg(meth, n, to), DynamicArg(invo, n, from),$
 $CallGraph(invo, callerCtx, meth, calleeCtx).$

(7) $InterMethAssign(to, callerCtx, from, calleeCtx)$
 $\leftarrow DefinedReturn(meth, from), DynamicReturn(invo, from),$
 $CallGraph(invo, callerCtx, meth, calleeCtx).$

Finally, I give an example of the *Merge* and *Record* functions as below.

$$record(heap, ctx) = ctx$$

$$merge(heap, hctx, invo, ctx) = ctx \oplus invo$$

where $\oplus$ is the concatenating operator. Now, we run the context-sensitive analysis program above on the simple Java program in Example 2.3.1. The contexts are three call-graph paths '0' (for the main method execution), '0, 12' (for the first *id* function invocation at line 12) and '0, 13' (for the second one). I use line number 0 to demonstrate the *main* function call. Then, there are

$(InterMethAssign(`obj', `0, 13', `objP1', `0'), InterMethAssign(`objP2', `0', `obj', `0, 13'),)$

and

$(InterMethAssign(`obj', `0, 12', `objC1', `0'), InterMethAssign(`objC2', `0', `obj', `0, 12'),).$

So that, two data flows, $objP1 \rightarrow obj \rightarrow objP2$ and $objC1 \rightarrow obj \rightarrow objC2$, have been separated. Additionally, the simple points-to analysis in Example 2.2.2 is a context-insensitive analysis.

2.4 Call-site sensitivity, object sensitivity and type sensitivity

Our context-sensitive analysis is cloning approach in which the analyzed program's components (variable, objects) are cloned for each *context*. Many kinds of *context* are defined, such as call-site, object, type [1]. Besides the methods that use a single kind of *context*, a hyper approach have been proposed [10] by a combination.

2.4.1 Call-site sensitivity

By changing the definition of *Context*, *HContext* and two manipulating functions (*Record* and *Merge*), a type of context-sensitive analysis is established. In Section 2.3, I defined a call-site-sensitive analysis. *Context* and *HContext* are sequences of *label*s, *label* $\in$ *Lab*, where the *label*s are line numbers of method call instructions (12 and 13 in Example 2.3.1). *Record(heap, ctx) = ctx*. A *heap context* of an allocated object is the call-site (*context*) of the caller. The *Merge* function concatenates the caller's call site with the current call-site (for example, *Merge*('10', '0', '12', '0') = '0, 12'). I remark that the *context*s in the call-site sensitive analysis are sequences of call-sites and they also are the call paths in call-graph. Also, *hcontext*s are sequences of call-site line numbers.

2.4.2 Object sensitivity and type sensitivity

The call-site-sensitive points-to analysis divides *context*s by calling chains. Besides, the object sensitivity has been proposed [1] as an another way to divide contexts. The object-sensitive analysis is suggested to work well for object-oriented programs.

Definition 2.4.1. *Context* and *heap context* (*HContext*) in object-sensitive analysis are as follows:
$$Context = HContext = Lab^*$$
where *Lab* is the set of line numbers of heap allocation instructions.

Example 2.4.1. *Record* and *Merge* in object-sensitive analysis are defined as follows:
$$Record(heap, ctx) = ctx$$

$$Merge(heap, hctx, invo, ctx) = hctx \oplus heap$$

A *context* now is a sequence of heap allocation line numbers. It starts from '0', which denotes the heap allocation of the object of *Main* class. On the other way, variables are cloned based on base object's creation. For example, in this instruction *base.meth(args)*,

the *context* inside *meth* method invocation is identified by the allocation of the *base* object. For understanding, see the following example.

Example 2.4.2 (A simple Java program with method calls)**.**

```
1  public class P{
2       P id(P obj){return obj;}
3  }
4  public class C extends P{
5       void donothing(){}
6  }
7  public class M{
8       public static void  main(String[] agrs){
9            P objP1, objP2, objC1, objC2, objC3;
10           objP1 = new P();
11           objC1 = new C();
12           objC2 = objC1.id(objC1);
13           objP2 = objP1.id(objP1);
14           objC3 = objC1.id(objC1);
15           C objC = (C) objC2;
16           objC = (C) objC3;
17       }
18 }
```

Three method invocations at lines 12, 13 and 14 are only considered under two different contexts. 12 and 13 will have the same *context*; because line 12 and line 14 have the same base object created at line 11 ($objC1 = newC();$). In this case, the object-sensitive analysis is still precise enough to conclude that two downcasts at line 15 and 16 are safe. However, if line 14 is modified to become $objC3 = objP1.id(objC1);$, the analysis will fail to prove the safety of the second downcast. At this point, it seems that object sensitivity is always worse than call-site sensitivity. But it not true. The next section shows that there are cases where an object-sensitive points-to analysis works better than a call-sensitive one. In general, the object-sensitive points-to analysis is more scalable than the call-sensitive analysis.

In 2011, a type-sensitive analysis was proposed by Yannis et. al. [1]. It also takes the object-oriented concept, likes object-sensitive analysis. Type-sensitive analysis has higher scalability than object-sensitive analysis but is less precise.

Definition 2.4.2. *Context* and *heap context* (*HContext*) in type-sensitive analysis are as follows:

$$Context = HContext = Type^*$$

where $Type$ is the set of class types. The type-sensitive contexts are defined above. The manipulating functions only keep the type information instead of heap allocation.

Example 2.4.3.

$$Record(heap, ctx) = ctx$$

$$Merge(heap, hctx, invo, ctx) = hctx \oplus TypeOf(heap)$$

where $TypeOf(\cdot)$ returns type of the created object.

2.5 Abstraction

In order to precisely handle (recursive) call sequence, the *context* should simply be full call path. Hence, the *context* is defined as the sequence of line number in Definition 2.3.1. However, the *context*s will become infinite and our analysis will be undecidable. To this end, k-context-sensitive analysis [2] *abstracts* contexts to be finite as follows.

Definition 2.5.1. *Context* and *HContext* in k-context- h-heap context- sensitive analysis are as follows:

$$Context = Lab^k, HContext = Lab^h$$

.

A bounded version of *contexts* is described in Definition 2.5.1. Often, only (small) bounded length contexts are sufficient. Two manipulating functions, *Record* and *Merge* are also changed.

Example 2.5.1. Two manipulating functions of a k-context- h-heap context- sensitive analysis are as follows:

$$Record(heap, ctx) = last_h(ctx)$$

$$Merge(heap, hctx, invo, ctx) = last_k(ctx \oplus invo)$$

where the $last_n(\text{s})$ function returns the n last elements of the sequence s.

Consider Example 2.3.1, a 1-context-sensitive points-to analysis is precise enough to distinguish two invocations of the *id* method. First, there are

$Record(`10', `0') = `0', VarPointsTo(`objP1', `0', `10', `0') \leftarrow Allocation(`objP1', `10', `main').,$
$Record(`11', `0') = `0', VarPointsTo(`objC1', `0', `11', `0') \leftarrow Allocation(`objC1', `11', `main').,$
$Merge(`10', `0', `12', `0') = `12', CallGraph(`12', `0', `P : id : P', `12')$
$\quad \leftarrow VirtualInvocation(`objP1', `0', `12', `main'),$
$\quad\quad VarPointsTo(`objP1', `0', `10', `0'),$
$\quad\quad ObjectType(`10', `P'), LookUp(`P', `id : P', `P : id : P').$

and

$Merge(`11', `0', `13', `0') = `13', CallGraph(`13', `0', `P : id : P', `13')$
$\quad \leftarrow VirtualInvocation(`objP1', `0', `13', `main'),$
$\quad\quad VarPointsTo(`objP1', `0', `10', `0'),$
$\quad\quad ObjectType(`10', `P'), LookUp(`P', `id : P', `P : id : P').$

Then, there are

$(InterMethAssign(`obj', `13', `objP1', `0'), InterMethAssign(`objP2', `0', `obj', `13'),)$

and

$(InterMethAssign(`obj', `12', `objC1', `0'), InterMethAssign(`objC2', `0', `obj', `12'),).$

Chapter 3

Proposed method

3.1 Adaptive points-to analysis

Consider the following Java program.

Example 3.1.1 (A simple Java program for selective points-to analysis)**.**

```
1  public class P{}
2  public class C extends P{}
3  public class M{
4      public static void  main(String [] agrs){
5          P objP, objC, objP1, objP2, objP3;
6          objP = new P();
7          objC = new C();
8          objP1 = fun1(objP);
9          objP2 = fun1(objC);
10         objP3 = fun2(objP);
11         objP3 = fun2(objC);
12         (C) objP1; // downcast 'd1'
13         (C) objP2; // downcast 'd2'
14         (C) objP3; // downcast 'd3'
15     }
16     public static P fun1(P obj1){ return id(obj1);}
17     public static P fun2(P obj2){
18                             C temp = new C();
19                             return id(temp);}
20     public static P id(P obj){return obj;}
21 }
```

Let us analyze it with the 2-call-site sensitive points-to analysis mentioned in Chapter 2. The result contains

$VarPointsTo('objP', '0', '6', '0')$, $VarPointsTo('objP2', '0', '7', '0')$,

$VarPointsTo('objC', '0', '7', '0')$, $VarPointsTo('objP3', '0', '18', '0, 10')$,

$VarPointsTo('objP1', '0', '6', '0')$, $VarPointsTo('objP3', '0', '18', '0, 11')$.

Therefore, the first downcast at line 12 fails and two remaining are safe. These are the precise result. On the other hand, if we analyze the above Java program with a 1-call-site

sensitive points-to analysis, the analyzed tuples are quite different.

$VarPointsTo$('$objP$', '0', '6', '0'),   **VarPointsTo**('**objP2**', '**0**', '**6**', '**0**'),
$VarPointsTo$('$objC$', '0', '7', '0'), $VarPointsTo$('$objP2$', '0', '7', '0'),
$VarPointsTo$('$objP1$', '0', '6', '0'), $VarPointsTo$('$objP3$', '0', '18', '**10**'),
VarPointsTo('**objP1**', '**0**', '**7**', '**0**'), $VarPointsTo$('$objP3$', '0', '18', '**11**').

The differences make a failure when proving the safety of the second downcast. For this Java program, a level 1 abstraction is precise enough to analyze the behavior of $fun2$, but the method invocations of the $fun1$ need level 2. This leads to the idea that we should handle different parts of the program with different abstraction levels. This section introduces an adaptive context-sensitive points-to analysis, which is able to handle the parts of the program selectively.

Firstly, I define the *context*s in Definition 3.1.1. *Context* and *HContext* are sets of the following *context* and *hcontext* respectively.

Definition 3.1.1 (*context* and *heap context* in adaptive k-context- h-heap context- sensitive analysis)**.**

$$context = lab_1, ..., lab_\alpha \qquad hcontext = lab_1, ..., lab_\beta$$

where $0 \leq \alpha \leq k$, $0 \leq \beta \leq h$ and $lab \in Lab$ (set of line numbers).

Definition 3.1.2. *Merge* and *Record* functions in adaptive k-context- h-heap context-sensitive analysis are as follows:

$$Record \in \textbf{Record} : Lab \times Context \times HAbs \rightarrow HContext$$

$$Merge \in \textbf{Merge} : Lab \times HContext \times Lab \times Context \times Abs \rightarrow HContext$$

where $Abs = [z...k]$, $HAbs = [z...h]$, and z is the smallest abstraction level, e.g. $z = 1$.

The abstraction levels are added into two manipulating functions. Then the abstraction levels no longer are two numbers, k and h. From now, *abstraction* describes the abstraction levels for parts of the analyzed program.

Definition 3.1.3. *abstraction* and *heap abstraction* in adaptive k-context- h-heap context-sensitive analysis are as follows:

$$abstraction = abs_1, ..., abs_n$$

$$heapAbstraction = habs_1, ..., habs_m$$

where $abs \in Abs$, $habs \in HAbs$, and n and m are numbers of the analyzed program's parts.

In detail, our method divides the program by instruction. Hence, n and m are the numbers of method invocations and heap allocations respectively. An *abstraction* for the program in Example 3.1.1 is '1,1,1,1,2,1' (corresponding with the method invocations

16

at lines 8, 9, 10, 11, 16, 18). With it, the program is analyzed precisely. In addition, '1,1,1,1,2,1' is also the least precise abstraction that can prove all downcasts in the above program. In the rest of this document, only *context* and *abstraction* is mentioned in the examples. The *hcontext* and *heapAbstraction* are hidden for simplifying the examples.

Definition 3.1.4 (*Abstraction* comparison)**.** An *abstraction* A is more precise than an abstraction B, if and only if $A_i < B_i$ for any pair $(A_i \in A, B_i \in B)$

3.2 Problem restatement

In fact, an adaptive k-context- h-heap context- sensitive analysis is not more precise than a k-context- h-heap context- sensitive analysis. The equality happens with highest *abstraction*s ('$k, k, .., k$' and '$h, h, ..., h$'). In my experiment, 3-context- 3-heap context-sensitive analysis is not achievable because of state explosion. Therefore, I propose obtaining the precision of a high-abstracted analysis by running the adaptive analysis with lower abstractions. The downcast problem is simply restated as below.

Definition 3.2.1 (The downcast problem)**.** Given a Java program P, a set D of downcasts and a valid abstraction set $\mathbf{A}$.
Compute a partition (T, F) of D, where T contains down casts that can be proved by some valid abstraction $A \in \mathbf{A}$ and F is a set of down casts that may fail.

3.3 CEGAR-based algorithm

In order to solve the downcast problem, a CEGAR-based algorithm has been proposed by Zhang et. al [2]. The CEGAR-based algorithm is shown in Algorithm 1. CEGAR iteration starts by running the adaptive analysis with the lowest abstraction. After every analyzing step, the counterexample knowledge is accumulated. It is used to choose an abstraction for the next iteration.

The key points are the way to represent the counterexample knowledge ϕ and how it can guide the abstraction refinement (the *choose* function). I define ϕ and *choose* later. To give an early taste, I describe how the CEGAR-based algorithm works on Example 3.1.1 with an adaptive 2-call-site sensitive points-to analysis ($h = k = 2$). First, The analysis starts with the lowest abstraction, $a_1 = $ '$1, 1, 1, 1, 1, 1$'. There are

$VarPointsTo($'$objP$', '0', '6', '0'), $\qquad\qquad VarPointsTo($'$objP2$', '$0$', '$6$', '$0$'),$
$VarPointsTo($'$objC$', '0', '7', '0'), $\qquad\qquad VarPointsTo($'$objP2$', '$0$', '$7$', '$0$'),$
$VarPointsTo($'$objP1$', '0', '6', '0'), $\qquad\qquad VarPointsTo($'$objP3$', '$0$', '$18$', '$10$'),$
$VarPointsTo($'$objP1$', '0', '7', '0'), $\qquad\qquad VarPointsTo($'$objP3$', '$0$', '$18$', '$11$');$

and fail downcasts:

$FailDownCast($'$objP1$', '', 'C'), $\qquad\qquad FailDownCast($'$objP2$', '', '$C$').$

The lowest abstraction means that there is a single context,'0', for everywhere. The counterexample knowledge is a chain of inferences. For the first downcast,

---
Algorithm 1 CEGAR-based algorithm

Input: Program P and downcasts D

Output: A partition (T, F) of D, where T contains the down-castings that will not fail and F is set of the down-casting that may fail.

  $a := \bot$ // as initial abstraction (lowest-precision)
  $\phi := \{\}$ // ϕ is accumulated counter-example and initiated as empty set
  $T := \{\}$ and $F := D$
  loop
    $\phi, (T', F') = Analyze(F, a)$ // invoke the analysis
    $T := T \cup T'$
    $F := F'$
    $a := choose(\phi, F)$
  end loop

$FailDownCast(\text{`}objP1\text{'}, \text{`'}, \text{`}C\text{'})$
    $\leftarrow VarPointsTo(\text{`}objP1\text{'}, \text{`}0\text{'}, \text{`}6\text{'}, \text{`}0\text{'}), Cast(\text{`}objP1\text{'}, \text{`'}, \text{`}C\text{'}).$
$VarPointsTo(\text{`}objP1\text{'}, \text{`}0\text{'}, \text{`}6\text{'}, \text{`}0\text{'})$
    $\leftarrow InterMethAssign(\text{`}objP1\text{'}, \text{`}0\text{'}, \text{`}id(obj1)\text{'}, \text{`}8\text{'}), VarPointsTo(\text{`}id(obj1)\text{'}, \text{`}8\text{'}, \text{`}6\text{'}, \text{`}0\text{'}).$
$VarPointsTo(\text{`}id(obj1)\text{'}, \text{`}8\text{'}, \text{`}6\text{'}, \text{`}0\text{'})$
    $\leftarrow InterMethAssign(\text{`}id(obj1)\text{'}, \text{`}8\text{'}, \text{`}obj\text{'}, \text{`}16\text{'}), VarPointsTo(\text{`}obj\text{'}, \text{`}16\text{'}, \text{`}6\text{'}, \text{`}0\text{'}).$
$VarPointsTo(\text{`}obj\text{'}, \text{`}16\text{'}, \text{`}6\text{'}, \text{`}0\text{'})$
    $\leftarrow InterMethAssign(\text{`}obj\text{'}, \text{`}16\text{'}, \text{`}obj1\text{'}, \text{`}8\text{'}), VarPointsTo(\text{`}obj1\text{'}, \text{`}8\text{'}, \text{`}6\text{'}, \text{`}0\text{'}).$
$VarPointsTo(\text{`}obj1\text{'}, \text{`}8\text{'}, \text{`}6\text{'}, \text{`}0\text{'})$
    $\leftarrow InterMethAssign(\text{`}obj1\text{'}, \text{`}8\text{'}, \text{`}objP\text{'}, \text{`}0\text{'}), VarPointsTo(\text{`}objP\text{'}, \text{`}0\text{'}, \text{`}6\text{'}, \text{`}0\text{'}).$

and

$InterMethAssign(\text{`}objP1\text{'}, \text{`}0\text{'}, \text{`}id(obj1)\text{'}, \text{`}8\text{'})$ and $InterMethAssign(\text{`}obj1\text{'}, \text{`}8\text{'}, \text{`}objP\text{'}, \text{`}0\text{'})$ are consequences of $Abs(\text{`}8\text{'}, 1)$. $Abs(\text{`}n\text{'}, abs)$ means that the method invocation at line n is abstracted with level abs. $InterMethAssign(\text{`}id(obj1)\text{'}, \text{`}8\text{'}, \text{`}obj\text{'}, \text{`}16\text{'})$ and $InterMethAssign(\text{`}obj\text{'}, \text{`}16\text{'}, \text{`}obj1\text{'}, \text{`}8\text{'})$ are consequences of $Abs(\text{`}16\text{'}, 1)$. The similar thing happens with the second downcast. Finally, there are two rules below.

$$FailDownCast(\text{`}objP1\text{'}, \text{`'}, \text{`}C\text{'}) \leftarrow ..., Abs(\text{`}8\text{'}, 1), Abs(\text{`}16\text{'}, 1).$$

$$FailDownCast(\text{`}objP2\text{'}, \text{`'}, \text{`}C\text{'}) \leftarrow ..., Abs(\text{`}9\text{'}, 1), Abs(\text{`}16\text{'}, 1).$$

The *choose* function takes ϕ and F as the inputs. At this point, F includes two first downcasts. The above inference chains say that

- $FailDownCast(\text{`}objP1\text{'}, \text{`'}, \text{`}C\text{'})$ may not exist if $Abs(\text{`}8\text{'}, 1)$ or $Abs(\text{`}16\text{'}, 1)$ dose not exist.

- $FailDownCast(\text{`}objP2\text{'}, \text{`'}, \text{`}C\text{'})$ may not exist if $Abs(\text{`}9\text{'}, 1)$ or $Abs(\text{`}16\text{'}, 1)$ dose not exist.

For example, the *choose* function returns $a_2 = $ '1, 1, 1, 1, 2, 1' as the abstraction for the second iteration. Then, only $FailDownCast('objP1', '', 'C')$ remains. The counterexample knowledge now says

- $FailDownCast('objP1', '', 'C')$ is produced for any abstraction $b = $ '$b_1, b_2 = 1, b_3, b_4, b_5, b_6$'.

- $FailDownCast('objP1', '', 'C')$ is also produced by any abstraction $b = $ '$b_1, b_2 = 1, b_3, b_4, b_5 = 1, b_6$'.

The second iteration returns abstraction $a_3 = $ '1, 2, 1, 1, 1, 1' and the third returns $a_4 = $ '1, 2, 1, 1, 2, 1'. However, $FailDownCast('objP1', '', 'C')$ still exists. Recall that the maximum abstraction level is 2. Hence the CEGAR-based algorithms stops after the fourth iteration. The final result is $(T = \{'d2', 'd3'\}, F = \{'d1'\})$.

3.3.1 Choosing abstraction with by maximum satisfiability (MaxSAT)

In this subsection, the *choose* function is defined as a partial weighted MaxSAT problem.

Definition 3.3.1 (MaxSAT). MaxSAT is SAT with soft constraints (pairs of clause and weight). Its goal is to find a model that maximizes the sum of satisfied soft constraints' weight.

Note that, our points-to analysis is implemented in Datalog language, and it is in predicate logic. However, MaxSAT is a binary (propositional) logic problem. Therefore, any tuple needs to be converted to a variable. For example, the abstraction $a = $ '1, 1, 1, 1, 1, 1'$ is encoded as $[a_1 = 1] \wedge [a_2 = 1] \wedge [a_3 = 1] \wedge [a_4 = 1] \wedge [a_5 = 1] \wedge [a_6 = 1]$ where $[a_i = 1]$ is a boolean variable. In the previous CEGAR-based algorithm explanation, the counterexample knowledge ϕ after the second iteration is

$$([a_2 = 1] \rightarrow [FailDownCast('objP1', '', 'C')])$$

or

$$(\neg[a_2 = 1] \vee [FailDownCast('objP1', '', 'C')]).$$

The *choose* function is formulated as below.

Definition 3.3.2 (The *choose(ϕ, F)* function).

$$choose(\phi, F) = MaxSAT(\mathbf{A} \wedge \alpha(F) \wedge \phi, \gamma_a \wedge W_F)$$

where $\mathbf{A} \wedge \alpha(F) \wedge \phi$ is the hard constraint and $\gamma_a \wedge W_F$ is the soft constraint.

In details,

- $\mathbf{A}$ represents the valid abstraction. $\mathbf{A} = \bigwedge\{[a_i = z] \vee [a_i = z+1]) \vee ... \vee [a_i = k]| 1 \leq i \leq n\}$; where z is the minimum abstraction level, k is the maximum abstraction level and n is the number of parts of the analyzed program.

- ϕ encodes the accumulated counter-example knowledge. See Section 3.4.

- The $\alpha(F) = \bigwedge\{\neg[d]|\ d \in F\}$ constraint says that there has to be at least one downcast to be proved by the chosen abstraction.

- γ_a denotes the complexity order of the abstraction levels. γ_a is a set of $([a_i = level], w)$ tuples. The higher abstraction level has the lower weight.

- $W_F = \{([d], maxw)|\ d \in F\}$ where $maxw > sum(\{w|\ ([a_i = k], w) \in \gamma_a\})$ and k is the maximum abstraction level. The W_F constraint enforces that the solution actually proves at least one downcast in F.

Definition 3.3.3 (Valid abstraction). $A = $ '$a_0, a_1, ..., a_n$' is a valid abstraction of a program P with an adaptive k-context-sensitive points-to analysis, iff $z < a_i < k$ for $0 < i < n$.

Theorem 3.3.1. If $MaxSAT(\mathbf{A} \wedge \alpha(F) \wedge \phi, \gamma_a \wedge W_F)$ returns no solution; then there is no valid abstraction which can prove a downcast in F.

Consider Theorem 3.3.1, if a MaxSAT solver gives no solution, it means that there is no model which can satisfy $\mathbf{A} \wedge \alpha(F) \wedge \phi$. Hence, no downcast in F can be proved by a valid abstraction. However, some downcasts in F might be still safe, and it can be proved by some abstractions that are more precise than the highest abstraction ('$k, k, ..., k$'). The CEGAR-based algorithm will terminate if the choose function returns no valid abstraction.

Theorem 3.3.2. The CEGAR-based algorithm with an adaptive k-context-sensitive points-to analysis solves the downcast problem and gives the same precise results as a k-context-sensitive points-to analysis does.

3.4 Provenances vs Datalog runs

In the previous section, the counterexample knowledge ϕ is mentioned as inference chains. More precisely, ϕ is a set of analysis rule applications. According to the notations in the definitions in Section 2.1, a rule is $l_0 \leftarrow l_1, .., l_n$; then a rule application is $\sigma(l_0) \leftarrow \sigma(l_1), .., \sigma(l_n)$ where $\sigma(l_0), \sigma(l_0), ..., \sigma(l_n) \in T$ and T is the produced tuples set.
The counterexample knowledge was defined as a set of *provenances* in [2]; but in our method, I propose to use the set of *Datalog runs* as the counterexample knowledge.

Definition 3.4.1 (Datalog runs). The set of Datalog runs produced by a Datalog program C is

$$[C]_T = \bigwedge\{\sigma(l_0) \leftarrow \sigma(l_1), .., \sigma(l_n)|\ (l_0 \leftarrow l_1, .., l_n) \in C \ and \ \sigma(l_0), ..., \sigma(l_n) \in T\}$$

Definition 3.4.2 (Provenance). The set of provenances $|C|_T$ is the biggest subset of $[C]_T$ that satisfies

$$\forall \sigma(r) \in |C|_T\ \nexists \sigma'(r') \in |C|_T\ \ \sigma(l_0) = \sigma'(l'_0)$$

where $r, r' \in C$, $\sigma(r) = \sigma(l_0) \leftarrow \sigma(l_1), .., \sigma(l_n)$ and $\sigma'(r') = \sigma'(l'_0) \leftarrow \sigma'(l'_1), .., \sigma'(l'_m)$.

The same tuple can be produced by applying two or more rules. Provenances only keep the information of the first-producing rule application, while it skips the applications that create any existing tuple. Differently, Datalog runs save all the producing relations. Therefore, the Datalog-run counterexample has a higher precision than the provenance counterexample, and obviously, pays a higher cost.

3.5 Cheap abstraction for refinement

It is not necessary to run the refinement step on the same abstraction with the analysis step. It is mostly because of the computation cost. The counterexample knowledge can be modeled with a lower abstraction.

Definition 3.5.1. Datalog runs with lower level abstraction is defined as follows:

$$[C]'_T = \bigwedge \{f(\sigma(l_0)) \leftarrow f(\sigma(l_1)), .., f(\sigma(l_n)). | \ (l_0 \leftarrow l_1, .., l_n.) \in C \ and \ \sigma(l_0), ..., \sigma(l_n) \in T\}$$

where f is a context lowering mapping function.

For example, the Datalog run below is produced by 2-call-site-sensitive analysis.

$VarPointsTo(\text{`}objP1\text{'}, \text{`}0\text{'}, \text{`}6\text{'}, \text{`}0\text{'})$
   $\leftarrow InterMethAssign(\text{`}objP1\text{'}, \text{`}0\text{'}, \text{`}id(obj1)\text{'}, \textbf{`0,8'}), VarPointsTo(\text{`}id(obj1)\text{'}, \textbf{`0,8'}, \text{`}6\text{'}, \text{`}0\text{'}).$

Then context constants are mapped to 1-length context.

$VarPointsTo(\text{`}objP1\text{'}, \text{`}0\text{'}, \text{`}6\text{'}, \text{`}0\text{'})$
   $\leftarrow InterMethAssign(\text{`}objP1\text{'}, \text{`}0\text{'}, \text{`}id(obj1)\text{'}, \textbf{`8'}), VarPointsTo(\text{`}id(obj1)\text{'}, \textbf{`8'}, \text{`}6\text{'}, \text{`}0\text{'}).$

Lower context-level counterexample knowledge means that the refinement step is processed with a lower abstraction. It may lead to the false positive results, but at least it still remains sound. As with the abstract semantic, there is a trade-off between precision and analysis cost. Note that a higher level abstraction for refinement makes no sense.

3.6 Partitioned abstraction search

In my experiment, it is too costly to use the MaxSAT solver directly for choosing the abstraction. Hence, I propose a partitioned abstraction search method with a new $choose'$ function below.

Definition 3.6.1. With a partition $(D_0, ..., D_n)$ of F,

$$choose'(\phi, F) = \bigsqcup \{choose(\phi, D_i) | i = 1, ..., n\}$$

where D_i is a set of downcasts and $\cup$ is abstraction joining operator.

Definition 3.6.2 (The abstraction joining operator).
With two abstractions, $A = A_1, A_2, ..., A_n$ and $B = B_1, B_2, ..., B_n$,

$$A \sqcup B = C = C_1, C_2, ..., C_n$$

where $C_i = \max(A_i, B_i)$.

The operation, $A \sqcup B$, returns the least abstraction C that is more precise than A and B. Our method applies the MaxSAT solver on smaller sub-problems. The original MaxSAT problem is solved approximately.

Theorem 3.6.1. If $choose'(\phi, F) = A$ then A is a model of $\mathbf{A} \wedge \alpha(F) \wedge \phi$ where

$$choose(\phi, F) = MaxSAT(\mathbf{A} \wedge \alpha(F) \wedge \phi, \gamma_a \wedge W_F).$$

3.7 Repeated abstraction search

Consider the example in Section 3.3, after two iterations, the counterexample knowledge contains this information, $\phi =$

$$([a_2 = 1] \rightarrow [FailDownCast(`objP1`, ``, `C`)]) \wedge$$

$$([a_2 = 1] \wedge [a_5 = 1] \rightarrow [FailDownCast(`objP1`, ``, `C`)]).$$

Then it requires two more iterations to know that $[FailDownCast(`objP1`, ``, `C`)]$ can not be disproved (run the analysis with two abstractions '$1, 2, 1, 1, 1, 1$' and '$1, 2, 1, 1, 2, 1$' respectively). At the end of the second iteration, the hard constraint is $\mathbf{A} \wedge \alpha(F) \wedge \phi$,

$$\mathbf{A} = ([a_2 = 1] \vee [a_2 = 2]) \wedge ([a_5 = 1] \vee [a_5 = 2]) \wedge ...$$

$$\alpha(F) = \neg[FailDownCast(`objP1`, ``, `C`)]$$

and the soft constraint is $\gamma_a \wedge W_F$,

$$\gamma_a = ([a_2 = 1], 1) \wedge ([a_2 = 2], 0) \wedge ([a_5 = 1], 1) \wedge ([a_5 = 2], 0) \wedge ...,$$

$$W_F = (\neg[FailDownCast(`objP1`, ``, `C`)], 7).$$

Then $MaxSAT(\mathbf{A} \wedge \alpha(F) \wedge \phi, \gamma_a \wedge W_F)$ returns $[a_1 = 1] \wedge [a_2 = 2] \wedge [a_3 = 1] \wedge [a_4 = 1] \wedge [a_5 = 1] \wedge [a_6 = 1]$ (the abstraction '$1, 2, 1, 1, 1, 1$'). However, if it returns '$1, 2, 1, 1, 2, 1$' instead, the CEGAR-based algorithm could end faster. To overcome the above scenario, we propose a repeated abstraction search; a choose function is defined below.

Definition 3.7.1 (*choose* in repeated abstraction search)**.**

$$choose(\phi, D_i) = choose_j(\phi, D_i)$$

where

$$A_{j+1} = choose_{j+1}(\phi, D_i) = A_j \sqcup choose_j(\phi \wedge \beta(A_j), D_i)$$

$$\text{for } \beta(A_j) = \bigwedge_p a_j^p \neq A_j(p) \text{ and } \beta_0 = true$$

and a_j^p is a element of abstraction A_j

Our repeated abstraction choosing method goes forward abstractions. At first, $choose_0(\phi, D_i)$ is the original MaxSAT problem. Then it assumes that the A_0 does not prove any downcast by adding $\beta(A_{j-1}, A)$ into the hard constraint. A new alternative abstraction is found. Finally, the chosen abstraction is $\bigsqcup_{i=1,...,j} A_i$. In the above example, at the end of the second iteration with $j = 1$, '1, 2, 1, 1, 2, 1' is returned instead of '1, 2, 1, 1, 1, 1'. Thus, the CEGAR-base algorithm runs faster. Now the choosing abstraction statement in the Algorithm 1 becomes

$$a := choose'(\phi, F) = \bigsqcup\{choose_j(\phi, D_i)|i = 1, ..., n\}$$

Theorem 3.7.1. For $0 \leq j$, if $choose_j(\phi, D_i) = A_j$ and $choose_{j+1}(\phi, D_i) = A_{j+1}$ then A_{j+1} is not less precise than A_j (better or equal),

Chapter 4

Related works

4.1 Pushdown system

Our points-to analysis is an instance of the reachability problem. In which input tuples encode initial states, analyzing rules are transition and output tuples at the end of inference describe reachable states. In addition, our transition system becomes finite-state by an abstraction in Section 2.5. Also, recursive method invocations are handled by approximating the infinite set of unbounded-length contexts using the finite set of bounded contexts.

Another approach for handling recursive functions is the pushdown approach [8]. A pushdown system is a state transition system with an unbound stack and the pushdown static analysis can precisely handle recursive calls.

The pushdown static analysis is also called context-free language (CFL) reachability [8]. It starts with initial states and then the reachable states are gradually reached. In 2001, the pushdown points-to analysis was introduced by Rehof and Fahndrich [9]. During the analyzing time, the proposed system obtains and accumulates every function behaviors. These behaviors are then encoded as mapping relations between sets of input states and sets of possible output states. In the pushdown approach, the infinite states are abstracted to a finite number of state subsets. In the points-to analysis, the maximum number of state subsets is $s = p \times n$ where p is the number of pointers and n is the number of objects. Furthermore, the mapping function is $M = S \times S$ where S is a set of state subsets. We also have that the number of functions is finite. Therefore, the pushdown static analysis is decidable.

All in all, the points-to analysis with the pushdown approach is decidable. It is hard to compare the precision of this approach and our approach. However, from the performance perspective, our approach has an advantage as being more scalable [13]. In addition, our method works with *antlr* and *chart* while [13] dose not. Besides these advantages, our approach has a key drawback that is how to pick a good abstraction. The program should be analyzed with small enough number of states but still sufficient to prove the goals (downcasts). This thesis focuses on solving these problems.

4.2   Abstraction Refinement

In the previous section, we stated that the points-to analysis can be considered as a reachability problem which is decidable if the states are finite. Beside the pushdown system, there is another method that map the infinite states of the program semantic to a set of finite abstracted states by defining an abstract semantic. Our points-to analysis is influenced by this approach.

Recall that the key problem with this approach is how to pick a good abstraction. There is one prominent work by Zhang et. al. [2]. They try to address this issue. In 2014, they proposed the CEGAR-based abstraction refinement [2] which models the "picking good abstraction" as a MaxSAT problem, see Subsection 3.3.1. They also proposed the usage of provenances as counter example knowledge to generalize the failure of abstractions. The cheapest refined abstraction is chosen based on these provenances.

In this work, we extend [2]'s work by using a more fine-grained granularity of program parts. Namely, our approach uses different precision levels for different method call sites, whereas [2] does not. We also describes some problems we found while extending [2] in Chapter 3. Therefore, we propose the use of Datalog runs instead of provenances to gain higher precision. Finally, we suggest partition abstraction search for feasibility and repeated search for abstraction forwarding.

Chapter 5

Implementation

Our analysis system is described in Figure 5.1. Doop is a points-to analysis framework for Java programs. The Doop framework precisely handles many Java features (such as inheritance, reflection, exception). Doop includes many context-sensitive analyses (call-site-sensitive, object-sensitive and type-sensitive analyses) and it is easy to define a new analysis. Doop uses the Datalog language and Logicblox (a Datalog solver).

In Doop, an input Java program is parsed into input tuples that can be used as data in a Java program by using the Soot library [4]. Logicblox, a Datalog solver, starts the analysis with those input tuples, abstraction tuples and points-to analysis rules. After the analysis, counterexample knowledge is extracted in the form of predicate tuples. There is a gap between the predicate logic (our points-to analysis) and the binary logic (MaxSAT). I use the Redis database [14] to bind them. The MaxSAT problem in Chapter 3 is solved by MiFuMax [5]. If the MaxSAT solver returns no solution (abstraction) then the CEGAR-based algorithm ends. In the following, I discuss the some relevant details of the implementation, such as optimizations that were not described in the earlier part of the thesis.

5.1 Eliminate non-context predicates from counterexample knowledge

In order to decrease the cost of the MaxSAT problem, some redundant tuples should be removed from the counterexample knowledge. Hence, the non-context predicates are considered.

Definition 5.1.1 (Non-context predicate)**.** Non-context predicate is the predicate that does not include any context variable ($Context$ or $HContext$).

The analysis inference produces analyzed tuples from input tuples. There are rules that create non-context analyzed tuples. For example,
$$SupertypeOf(?t, ?s) \leftarrow ComponentType[?s] = ?sc,$$
$$ComponentType[?t] = ?tc, ReferenceType(?sc),$$
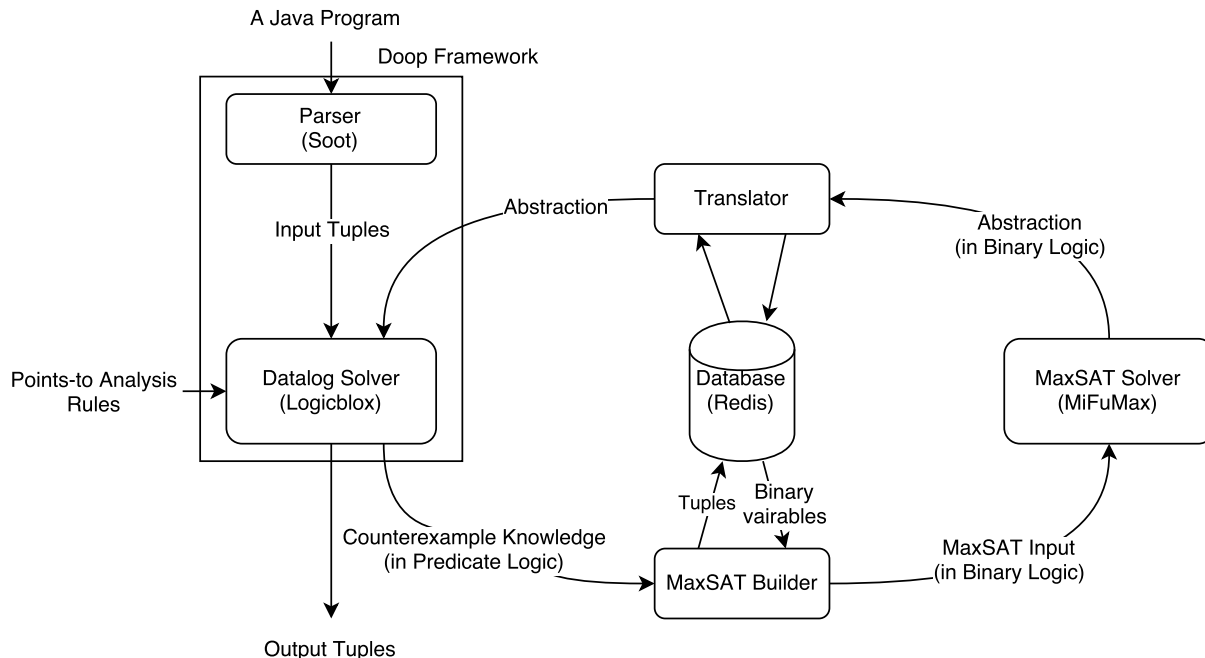$$ReferenceType(?tc), SubtypeOf(?sc, ?tc).$$

Figure 5.1: System architecture

In fact, all input tuples are non-context predicates. Moreover, there is no rule such that a non-context tuple is created from context tuples. Therefore, it is simple to get the following theorem.

Theorem 5.1.1. Let F_{C_1} and F_{C_2} be the points-to analysis rules with different abstractions for a same program. Let $T_i = lfp(F_{C_i}, T_0)$ for each $i \in 1, 2$ where T_0 is the starting facts for the program. Let $T_i' \subseteq T_i$ be the set of non-context tuples of T_i (for each $i \in 1, 2$). Then, $T_1' = T_2'$.

As Theorem 5.1.1, the non-context predicates are always the same, regardless of what the abstraction is. Therefore, the non-context predicates do not help to pick a good abstraction. Hence, in our implementation, the counterexample knowledge dose not include any non-context predicates, except for the abstraction predicates.

5.2   Trace tuples from may fail downcasts

Consider the CERGAR-based algorithm in Algorithm 1, F is the remaining may-fail downcast. This set is deceasing over time. Because of computation cost, the counterexample knowledge ϕ ignores any downcast which does not belong to F. I manually wrote the tracing rules to extract all inferences which lead to a $FailDownCast$ tuple creation. The following is an example.

$FVarPointsTo(?heap, ?from) \leftarrow$
$\qquad ApplicationClass(?class), MethodSignature : DeclaringType[?inmethod] = ?class,$

$Reachable(?inmethod), Cast(?type, ?from, ?to, ?inmethod),$
$VarPointsTo(\_, ?heap, \_, ?from), HeapAllocation : Type[?heap] = ?heaptype,$
$!SupertypeOf(?type, ?heaptype).$

The above is the tracing rule of the rule below.

$FailDownCast(?from, ?to, ?type) \leftarrow$
$\quad ApplicationClass(?class), MethodSignature : DeclaringType[?inmethod] = ?class,$
$\quad Reachable(?inmethod), Cast(?type, ?from, ?to, ?inmethod),$
$\quad VarPointsTo(\_, ?heap, \_, ?from), HeapAllocation : Type[?heap] = ?heaptype,$
$\quad !SupertypeOf(?type, ?heaptype).$

We use the tracing rules instead of recording because Logicblox does not support recording.

5.3   Rewrite points-to analysis rules

In the Doop framework, there are some intermediate rules. They makes the analysis rules easy to read and understand. For example, there are:

$VarPointsTo(?hctx, ?heap, ?toCtx, ?to) \leftarrow$
$\quad VarPointsTo(?hctx, ?heap, ?fromCtx, ?from),$
$\quad Assign(?type, ?toCtx, ?to, ?fromCtx, ?from),$
$\quad HeapAllocation : Type[?heap] = ?heaptype, SupertypeOf(?type, ?heaptype).$

and

$Assign(?type, ?ctx, ?to, ?ctx, ?from) \leftarrow$
$\quad AssignCast(?type, ?from, ?to, ?inmethod), ReachableContext(?ctx, ?inmethod).$

They can be combined to become as follows:

$VarPointsTo(?hctx, ?heap, ?ctx, ?to) \leftarrow$
$\quad VarPointsTo(?hctx, ?heap, ?ctx, ?from),$
$\quad AssignCast(?type, ?from, ?to, ?inmethod), //replaceAssign$
$\quad ReachableContext(?ctx, ?inmethod), //replaceAssign$
$\quad HeapAllocation : Type[?heap] = ?heaptype, SupertypeOf(?type, ?heaptype).$

This is correct because *Assign* predicates only appears once in the rules. Rewriting rules as showns above reduces the input size of the MaxSAT problem.

5.4   Integrate Doop and MiFuMax MaxSAT solver

I use the MiFuMax MaxSAT solver to implement the abstraction searching function. MiFuMaX is an open-source weighted (and unweighted) MaxSAT solver. Its input is in conjunctive normal form (CNF). An example of the weighted partial Max-SAT formula is:

p wcnf 4 5 16
16   1   −2   4   0

```
8  −2  −4  0
4  −3  2  0
```

The first line is configuration parameters. Each next line starts with a weight number and the later part is clauses (disjunction of binary literals). The minus character denotes the negation.

As mentioned, we need to convert the counterexample knowledge from predicate logic to binary logic. Besides, each tuple is considered as a string (a predicate-name and a sequence of constants). We build mapping functions from string to binary variable id and vice versa. In my implementation, I use Redis database to build these mapping functions. Redis is an in-memory key-value data structure store [14]. I choose it because it is fast and easy to use.

Chapter 6

Experiments

Our experiments are executed on Ubuntu 14.04 machine with Intel Core i5-6200U processor (2.30 GHz × 4), 8 GB memory and 128 GB SSD hard disk. The implementation is done in Java 1.7.0, using Logicblox version 3.10, Redis 3.2.0 and MiFuMax.

The experiments are on the DaCapo 2006 benchmark suite. The Dacapo benchmarks are represented in Table 6.1. I analyze the benchmarks together with the Java Runtime Environment (JRE) libraries used by the benchmarks. We use JRE 1.7, which is about 80.6 MB in size.

The DaCapo benchmark includes programs, their input data, their benchmark-specific harness class and configuration files. The harness class invokes the analyzed program with its input data . In addition, DaCapo uses a non-trivial class-loading mechanism and the reflection to specify harness class and analyzed program's main class. A static program analysis has to:

- obtain runtime loaded classes

- access all reflective calls

- handle those refections

Fortunately, Soot includes TamiFlex [11] which provides custom class loader and collects the reflective calls by a reflection log file. Moreover, the Doop framework provides sophisticated reflection analysis.

The experiments are done in call-site-sensitive analysis. There are two kinds of comparison:

- The comparison between our CEGAR-based context-sensitive analysis and the analysis which have been already provided by the Doop framework

- The comparison among our CEGAR-based context-sensitive analyses with several partitioned repeated parameters

The basic criterions are computation time, the number of reachable downcasts and the number of remaining may-fail downcasts. In our experiment, the analysis has 4-day timeout and there is a 5-minutes timeout for the MaxSAT solving step.

Table 6.1: The DaCapo 2006 bench mark programs

Benchmark	Description	Size (MB)
antlr	parses grammar files, then a parser and lexical analyzer is generated for each.	0.619
chart	uses JFreeChart to plot a number of complex line graphs and renders them as PDF.	2.1
xalan	transforms XML documents into HTML	1.2

6.1 Adaptivity vs. non-adaptivity

Firstly, Table 6.2 shows the results of the non-adaptive points-to analysis. Secondly, Table 6.3 shows the results of our adaptive points-to analysis. Here, the first column is the program name. The second column shows the runtime of the analysis. The column Reachble downcasts is the number of the reached downcasts. The fourth column is the number of downcast that may fail. In Table 6.2, there is only one column for the 3-call-site sensitive points-to analysis because this analysis is out of memory with all of the programs.

In Table 6.3, the column Surely-fail downcasts is the number of downcasts that fail even if we run the analysis with the highest abstraction. In addition, we can prove whether a downcast surely fail at the highest abstraction by using the counterexample knowledge. Recall, our analysis uses partitioned repeated abstraction search. The "5-partitioned 1-repeated" means the remaining downcasts are divided into group of five and the abstraction search is solved twice (one more time) in a single CEGAR iteration. In Table 6.3, for antlr, the number of surely fail downcasts is less than the number of may-fail downcasts because of the timeout of the MaxSAT solving step.

Table 6.2 and Table 6.3 show that the number of reachable downcasts are the same for both the non analysis and our adaptive analysis. We remark that, in fact, the reachable downcasts coincide in the two analyses (i.e., not just the numbers). The tables show the followings:

- Our adaptive 3-call-site-sensitive analysis do not do worse (equal or better) than the non-adaptive 2-call-site-sensitive analysis and of course it takes more time.

- The non-adaptive 3-call-site-sensitive analysis does not finish.

- The surely-fail downcast column shows that our method reaches the non-adaptive 3-call-site-sensitive analysis precision, except for antlr.

6.2 Partition and repetition

I compare between the 0-repeated 5-partitioned abstraction search and the 1-repeated 5-partitioned search in Figure 6.1. I also compare between the 0-repeated 5-partitioned

Table 6.2: The results of the non-adaptive call-site-sensitive analysis

Benchmark	2-call-site-sensitive analysis			3-call-site-sensitive analysis
	Time (s)	Reachable downcasts	Reachable may-fail downcasts	
antlr	21420	341	149	Out of memory
chart	24593	396	168	Out of memory
xalan	11994	561	313	Out of memory

Table 6.3: The results of the adaptive (5-partitioned 1-repeated) points-to analysis

Benchmark	Adaptive 3-call-site-sensitive analysis			
	Time (s)	Reachable downcast	Reachable may-fail downcast	Surely-fail downcasts
antlr	78939	341	149	141
chart	112647	396	165	165
xalan	276417	561	309	309

method and the 0-repeated 10-partitioned search in Figure 6.2. The graphs plot the number of downcasts that remain may-fail and detected to be surely-fail over the analysis run time. Both comparisons are on antlr. *MF* stands for may-fail downcasts and *SF* stands for surely-fail downcasts. The postfixes *5-0* and *5-1* respectively denote the 0-repetition and the 1-repetition with the 5-partition. Similarly, *10-0* stands for the 10-partitioned 0-repeated abstraction search. Basically, the 0-repetition is the analysis without the repeated abstraction search. Figure 6.1 shows that our repetition method is better than the non-repeated abstraction search with the antlr benchmark. The number of *MF* are decreased faster, while the *SF* are also proved quicker.

In my experiment, the non-partitioned points-to analysis ends with timeout. Hence, our partitioned analysis is better than the non-partitioned analysis. In addition, the partitioned parameter k of the k-partitioned analysis is chosen by trial and error. Figure 6.2 shows that on the antlr benchmark the 5-partitioned analysis is better than the 10-partitioned one in proving sure-fail downcasts.
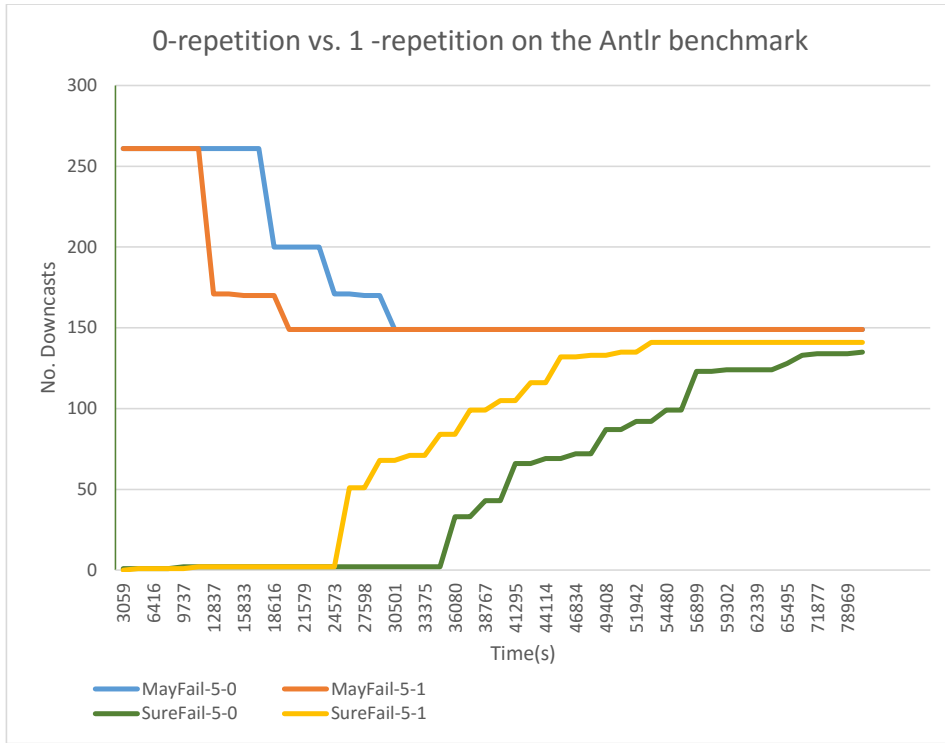
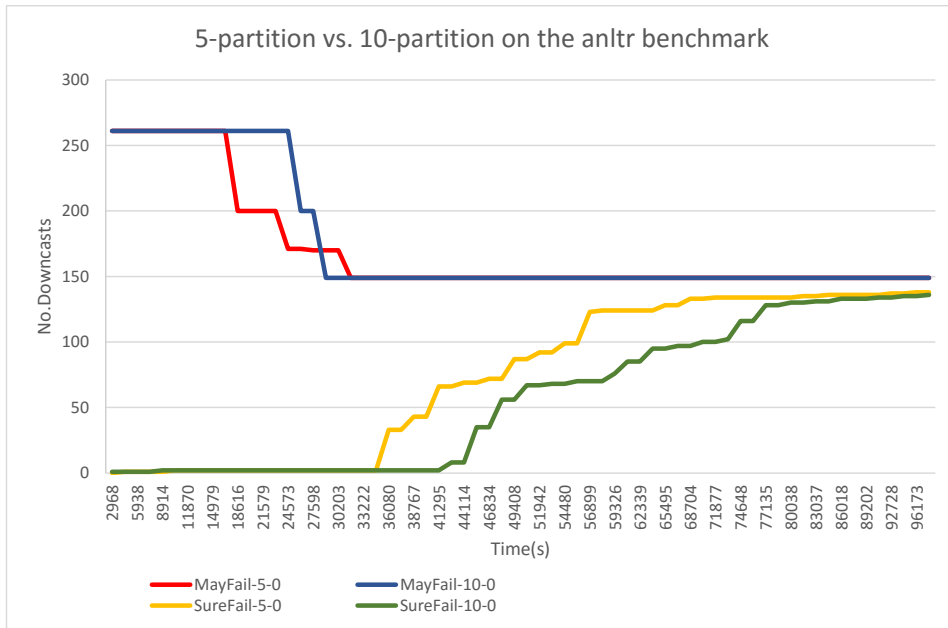Figure 6.1: 0-repetition vs. 1-repetition the antlr benchmark.

Figure 6.2: 5-partition vs. 10-partition on the antlr benchmark.

Chapter 7

Conclusion

We designed a CEGAR-based context-sensitive points-to analysis system for Java programs. First, I proposed a fine-grained adaptive points-to analysis, which extends [2] to handle finer-grained adaptive analysis. Then a refinement method is proposed in order to pick good abstractions. In our counterexample based refinement, the knowledge is modeled by Datalog runs (with a full encoding) which is more precise than provenances. Then the abstraction picking is feasibly solved as a partitioned problem and enhanced with repetition. Also, for scalability, my method runs the refinement step with a lower abstraction than the analysis step.

Finally, I have successfully implemented our analysis on the Doop framework and done the experiment on real Java programs in the DaCapo benchmark. Our CEGAR-based analysis obtained better results compared to non-adaptive context-sensitive analysis.

Future works

There are several issues with the current approach.

- Partitioning in the partitioned abstraction search is done arbitrarily.

- The number of repetitions in the repeated abstraction search are found by tries and errors.

I leave as future work to solve the above issues.

Bibliography

[1] Smaragdakis, Yannis, Martin Bravenboer, and Ondrej Lhot ak. *"Pick your contexts well: understanding object-sensitivity."* POPL 2011.

[2] Xin Zhang, Ravi Mangal, Radu Grigore, Mayur Naik, and Hongseok Yang. *"On abstraction refinement for program analyses in Datalog."* PLDI 2014.

[3] Smaragdakis, Yannis, and Martin Bravenboer. *"Using Datalog for fast and easy program analysis."* DR 2011.

[4] Vall ee-Rai, Raja, Phong Co, Etienne Gagnon, Laurie Hendren, Patrick Lam, and Vijay Sundaresan. *"Soot - a Java bytecode optimization framework."* CASCON 1999.

[5] Manquinho, Vasco, Joao Marques-Silva, and Jordi Planes. *"Algorithms for weighted boolean optimization."* SAT 2009.

[6] Blackburn, Stephen M., et al. *"The DaCapo benchmarks: Java benchmarking development and analysis."* OOPSLA 2006.

[7] Andersen, Lars Ole. *"Program analysis and specialization for the C programming language."* PhD thesis, University of Cophenhagen, 1994.

[8] Thomas Reps, Susan Horwitz and Mooly Sagiv. *"Precise interprocedural dataflow analysis via graph reachability."* POPL 1995.

[9] Jakob Rehof and Manuel Fahndrich. *"Type-base flow analysis: from polymorphic subtyping to CFL-reachability."* POPL 2001.

[10] George Kastrinis and Yannis Smaragdakis. *"Hybrid Context-Sensitivity for Points-To Analysis."* PLDI 2013.

[11] Eric Bodden, Andreas Sewe, Jan Sinschek, Hela Oueslati and Mira Mezini. *"Taming Reflection: Aiding Static Analysis in the Presence of Reflection and Custom Class Loaders."* ICSE 2011.

[12] Yannis Smaragdakis and George Balatsouras (2015), *"Pointer Analysis."* Foundations and Trends in Programming Languages 2015.

[13] Li, Xin, and Mizuhito Ogawa. *"Stacking-based context-sensitive points-to analysis for Java."* Haifa Verification Conference 2009.

[14] Redis. *http://redis.io/.*