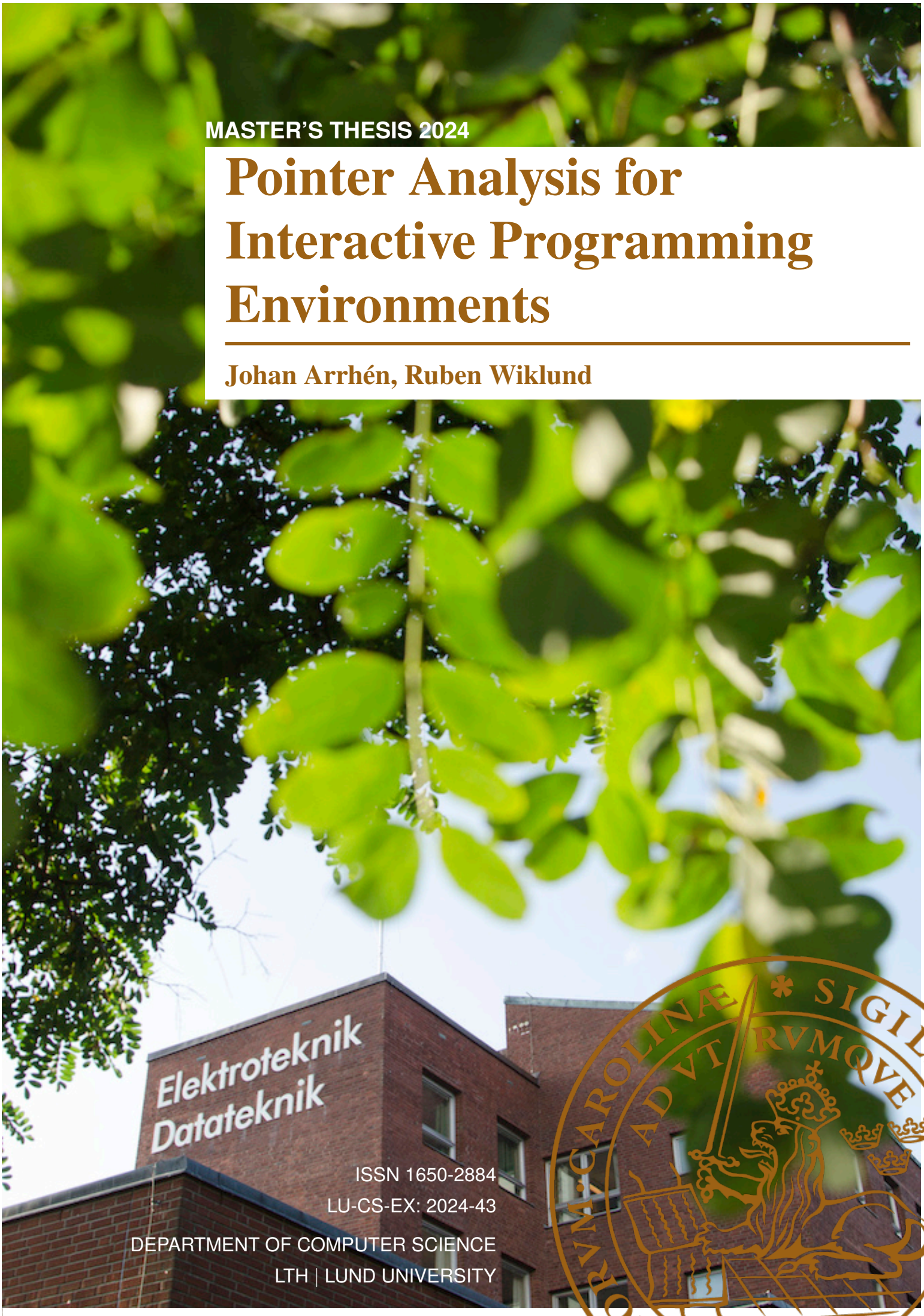


MASTER'S THESIS 2024

Pointer Analysis for Interactive Programming Environments

Johan Arrhén, Ruben Wiklund

ISSN 1650-2884
LU-CS-EX: 2024-43

DEPARTMENT OF COMPUTER SCIENCE
LTH | LUND UNIVERSITY

EXAMENSARBETE
Datavetenskap

LU-CS-EX: 2024-43

Pointer Analysis for Interactive Programming Environments

Pekaranalys för interaktiva programmeringsmiljöer

Johan Arrhén, Ruben Wiklund

Pointer Analysis for Interactive Programming Environments

Johan Arrhén
jo2065ar-s@student.lu.se

Ruben Wiklund
ru6707wi-s@student.lu.se

June 13, 2024

Master's thesis work carried out at
the Department of Computer Science, Lund University.

Supervisors: Idriss Riouak, idriss.riouak@cs.lth.se
Anton Risberg Alaküla, anton.risberg_alakula@cs.lth.se

Examiner: Flavius Gruian, flavius.gruian@cs.lth.se

Abstract

Static program analysis is the practice of analyzing a program's properties and behavior without executing it. It is a fundamental part of the compiler, enabling optimizations and error detection. It is also used in programming tools, such as IDEs, where it can be used for autocompletion and code navigation. Points-to analysis is a static program analysis that finds the set of objects that a pointer may point to during execution of the program. Many other analyses benefit from the result of a points-to analysis when analyzing programs with pointers, but the long execution times limit the use in interactive environments. Many algorithms for points-to analysis exist. In our thesis, we investigate how to speed up the points-to analysis *Andersen's Analysis*. We introduce Pecka, a Reference Attribute Grammar based tool. It speeds up the execution time by only analyzing code which can be reached within k steps in the program's call graph from a selected method.

We found that when limiting the distance to three steps in the call graph from a method, Pecka could find 56% of the results with a 5x speedup compared to analyzing the whole program. This enables running points-to analysis in an interactive environment. A limitation of this approach is that information can be missed in excluded methods, preventing use in, e.g., optimizing compilers.

Keywords: Points-to analysis, Andersen's analysis, Program analysis, Reference Attribute Grammars, Demand-driven evaluation

Acknowledgements

We would like to thank our supervisors, Idriss Riouak and Anton Risberg Alaküla, for their enthusiasm and valuable insights throughout the duration of this thesis.

Contents

1 Introduction **7**
- 1.1 Research Questions . . . 8
- 1.2 Scientific Contribution . . . 8
- 1.3 Individual Contributions . . . 9
- 1.4 Related Work . . . 9

2 Background **11**
- 2.1 Static Program Analysis . . . 11
- 2.2 Points-to Analysis . . . 12
- 2.3 Abstract Interpretation . . . 13
- 2.4 Heap Modeling . . . 14
- 2.5 Andersen's Analysis . . . 15
  - 2.5.1 Constraint Collection . . . 15
  - 2.5.2 Solving the Constraints . . . 16
  - 2.5.3 Analysis Sensitivity . . . 18
- 2.6 Call Graph . . . 21
- 2.7 Source Code Representation . . . 22

3 Reference Attribute Grammars **25**
- 3.1 Attribute Grammars . . . 25
- 3.2 Reference Attribute Grammars . . . 25
- 3.3 JastAdd . . . 26
- 3.4 ExtendJ . . . 27

4 Approach **29**
- 4.1 Design . . . 29
  - 4.1.1 Allocation Sites and Nodes . . . 29
  - 4.1.2 Constraint Generation . . . 30
  - 4.1.3 Limiting Search Distance . . . 30
  - 4.1.4 Solver . . . 31

4.2 Language Features . . . 34
4.2.1 Methods . . . 34
4.2.2 Object Keywords: this and super . . . 34
4.2.3 Dot . . . 35
4.2.4 Arrays . . . 35
4.2.5 List and Map . . . 35
4.2.6 Fields . . . 36
4.3 Limitations . . . 36
4.3.1 Bytecode Analysis . . . 36
4.3.2 Reflection . . . 37
4.3.3 Additional Limitations . . . 37

5 Evaluation **39**
5.1 Evaluating an Analysis . . . 39
5.1.1 Precision and Recall . . . 39
5.1.2 Soundness . . . 40
5.2 Experimental Setup . . . 41
5.2.1 Manual Ground Truths . . . 41
5.2.2 Generated Ground Truths for ANTLR . . . 42
5.2.3 Benchmark Programs . . . 43
5.3 Results . . . 44
5.3.1 Manual Ground Truths . . . 44
5.3.2 Generated Ground Truths for ANTLR . . . 44
5.3.3 Benchmark Programs . . . 46

6 Discussion **51**
6.1 PECKA in Interactive Environments . . . 51
6.2 Implementation . . . 54
6.3 Future Work . . . 55

7 Conclusions **57**

References **59**

Appendix A Supplemental Data and Results **67**

Appendix B Source Code **71**

Chapter 1
Introduction

Program analysis establishes facts about a computer program. Some examples of these facts are which values a variable can have during the program's execution, or which execution paths can be taken through the program [29]. We can distinguish between two kinds of program analysis: dynamic program analysis and static program analysis. A dynamic program analysis executes the program and uses the observed behavior to establish some facts. A static program analysis does not execute the program and instead analyzes some representation of the source code. The result of a static program analysis generally applies for all possible executions of a program, but a dynamic program analysis only produces results for the examined executions. One advantage of dynamic program analyses is that if it can be seen that, e.g., a variable has the value 5 at some point in the program, then you can be completely sure of that fact. In contrast, a static program analysis often produces *false positives*, a fact that does not hold for any execution of the program. The results of a program analysis can be used in several ways. It can be used by a compiler to make optimizations [24] or find bugs in a program, as well as in various analysis tools aiding program development [22].

Points-to analysis is a form of program analysis that determines which objects a pointer can point to during program execution. Many other program analyses can benefit from the results of points-to analysis, such as null pointer analysis (identifying whether a dereferenced pointer can point to `null`) [29].

A program analysis needs to be precise and reasonably efficient for it to have practical use. A points-to analysis that includes too many objects that a pointer cannot point to during a run of the program makes the result less useful. Furthermore, an analysis that requires a considerable amount of memory or takes a long to perform may deter possible users. For instance, the time requirements have led production compilers to only use imprecise points-to analyses that are faster but less precise, which results in missing some optimizations [26]. For interactive environments such as an IDE, the time requirements are even shorter compared to the compiler. In such cases, the result is needed in seconds, making advanced analysis challenging [10, 30].

There are multiple algorithms and approaches that can be used to perform a points-to analysis. In this thesis we implement a type of points-to analysis called Andersen's analysis [3], which uses subset constraints to model the relations between pointers and objects. The time complexity for this algorithm is $\mathcal{O}(n^3)$, where n is the number of statements in the program [29].

Various tools exist to aid the development of a program analysis. One of these tools is the JastAdd [16] system, an implementation of the Reference Attribute Grammar [15] formalism. Reference Attribute Grammars allow for adding attributes to the abstract syntax tree, which is a tree representation of a program. An attribute is defined by an equation and evaluates to some value. Reference Attribute Grammars are an extension of Attribute Grammars [23] that allows for attributes to also evaluate to references to nodes in the abstract syntax tree. In JastAdd, attributes are only evaluated once accessed, which can benefit performance.

Performing points-to analysis on-demand involves determining the points-to set for a specific pointer or set of pointers, rather than computing the points-to sets for all pointers. This can reduce unnecessary computation if the user of a points-to result does not require the full result.

1.1 Research Questions

In this thesis, we aim to make the results of Andersen's analysis fast enough to make it usable in interactive environments. We therefore investigate how the analysis can be implemented with Reference Attribute Grammars using JastAdd, leveraging its intrinsic on-demand capabilities. The following research questions will be explored:

- **RQ1:** How can we speed up the results from Andersen's analysis to enable usage in an interactive environment?
- **RQ2:** How well-suited are Reference Attribute Grammars for implementation of points-to analysis?

1.2 Scientific Contribution

The contributions of this report are the following:

- A context-insensitive, field-sensitive implementation of Andersen's analysis for Java 4 programs called Pecka (Points-to Evaluation via Call-graph K-step Analysis), implemented using JastAdd, with options to run a full analysis on the whole program, or an analysis on a single method on-demand of a filtered set of constraints.
- A new method to limit the scope of the analysis using the call-graph distance from a requested method, yielding an unsound but usable result.

1.3 Individual Contributions

Both authors have performed an equal amount of work, with most tasks have been worked on collaboratively. Ruben primarily focused on the analysis implementation, while Johan focused on the non-analysis aspects, such as the architecture for the tool and the evaluation. The writing has also been highly collaborative, although still maintaining a similar focus as for the work.

1.4 Related Work

Andersen's analysis has been implemented in various papers in multiple languages, including Java [3, 6, 28, 46]. Andersen's analysis includes solving a constraint system. One example of how the constraint system can be solved is shown by Tian Tan [45], which we adapted for use in our implementation. As we implemented the analysis in Java, we could make use of type filtering described by Sridharan et al., where the object types can be used to exclude them from certain points-to sets [42].

Points-to analysis on-demand has also been handled previously in the literature [41, 43, 47]. Späth et al. [41] have implemented a demand-driven flow- and context-sensitive pointer analysis, including outputting points-to results. They have built their analysis on top of the data flow analysis framework *IFDS*, limiting the search space by only using parts of the graph necessary for the calling context. This speeds up the possibility to obtain flow-sensitive points-to information. Their analysis requires constructing the control flow graph for the entire analyzed program, whereas our analysis instead constructs the call graph. In addition, they have created a micro-benchmark suite called PointerBench, containing code with embedded expected results used for benchmarking [41]. We used this benchmark suite in our evaluation (see Section 5.2.1).

Sridharan et al. [43] formulate Andersen's analysis as a CFL-reachability problem, achieving a speedup of 16x for a query compared to computing the whole program. However, they note that some queries can still take several seconds, which might be too long for interactive environments, e.g., IDEs [31]. Their solution to this problem is to over-approximate the result after a certain time by including every allocation in the points-to set. Our tool does not support limiting the search time, but it is possible to limit the search space using a parameter.

Yan et al. [47] handle on-demand computation by first constructing *symbolic points-to graphs*, which is a type of graph with intraprocedural points-to relations where symbolic nodes represent objects created outside of the method. The intraprocedural graphs are then connected to form an interprocedural symbolic points-to graph. They can then find an answer to the query by traversing the graph and only computing the necessary information. Our tool performs on-demand analysis by limiting the search space using the call graph. The analysis receives a query to find points-to information for pointers in some method. The points-to information is then computed using only the information available within a specified distance in the call graph. This is to the best of our knowledge a new way of performing on-demand points-to analysis.

Chapter 2
Background

In this chapter we cover static program analysis, points-to analysis, and other concepts that are useful for understanding the rest of the report.

2.1 Static Program Analysis

Static program analysis establishes facts about a program without executing the program [29]. In contrast to dynamic program analysis where the program *is* executed, a static program analysis generally produces a result which applies for all possible executions of the program. The drawback is that some parts of the result might not apply for any exection. For example, consider the program in Listing 2.1. A static analysis might determine that the possible values of `x` are 2, 3 and 5, but 5 can never be assigned to the variable as `x` will never be greater than 3. A dynamic analysis could fail to find the value 3 and claim that the only possible value of `x` is 2 if the program is never given the input "ABC".

```
1 int x = 2;
2 if (args[0].equals("ABC"))
3     x = 3;
4 if (x > 3)
5     x = 5;
```

Listing 2.1: A program where the possible values of `x` are 2 and 3.

Two examples of static program analyses are escape analysis and live variable analysis [29]. Both of these analyses, along with many others, benefit from the result of points-to analysis, which is a program analysis that finds the set of objects that a pointer may point to. They are then said to be a *client* to the analysis.

Escape analysis determines whether a pointer has "escaped" from a function meaning whether it can be accessed from outside the function or not [29]. In the case that it cannot be accessed from outside the function, then the pointed-to memory location cannot be accessed once the function has been returned from and the pointed-to object can safely be placed on the stack instead of on the heap. In languages like

C, it can also be used to find if a pointer pointing to a stack-allocated variable is returned which could result in undefined behavior.

Live variable analysisis an analysis that determines how long a variable is "live" for, i.e., determining where in the program its value is last used [29] . If a variable is not live at a certain point then its memory location can be used to store another value. When analyzing a program that uses pointers, it must also be made sure that a variable's memory location also cannot be accessed through a pointer before it is safe to reuse for something else.

A non-exhaustive list of uses of static program analysis is shown in Table 2.1.

Use case	Example
Optimizing compilers	If a variable always has the same value at a point in the program, replace it with that value [2].
Bug detection tools	If a dereferenced pointer can potentially point to `null`, alert the programmer [35].
Software verification	If it can be guaranteed that all array accesses are valid, declare the program free of index out of bound errors [21].
Program comprehension	Find all classes that extend the class in the current file and enable the programmer to easily navigate to those.

Table 2.1: Some uses of static program analysis.

2.2 Points-to Analysis

Points-to analysis is the task of finding the set of objects that a pointer may point to during program execution. There are many different algorithms and approaches for performing this task, including:

Andersen's analysis Inclusion-based, uses constraints that *include* one points-to set in another. The time complexity is $\mathcal{O}(n^3)$, where n is the number of statements in the program [29].

Steensgaard's analysis Unification-based, uses constraints that *unifies* points-to sets. The time complexity is $\mathcal{O}(n\alpha(n, n))$, where α is the inverse Ackermann function and n is the number of statements in the program [44].

Das' analysis Extends Steensgaards' algorithm to not merge certain points-to sets which achieves a precision close to that of Andersen's with a time complexity of $\mathcal{O}(n^2)$, where n is the number of statements in the program [8].

Shapiro-Horwitz's analysis Parameterized algorithm that unifies pointers of the same category with a total of k categories. Equal to Steensgaard's when $k = 1$, equal to Andersen's when $k = n$. The time complexity is $\mathcal{O}(k^2 n\alpha(k^2 n, k^2 n))$, where α is the inverse Ackermann function, n the number of statements in the program and k the number of categories [36].

Demand-driven analyses Tries to find the points-to set for a single pointer instead of for all pointers. This often involves finding the points-to set of another pointer, but such information is only computed as needed. Examples include [17] and [41].

It is impossible to determine most interesting properties concerning a program's semantics [32]. Consider the well-known undecidable problem of determining whether a program halts or not for any given pair of program and input. If it were possible to make a precise points-to analysis for any program, then the halting problem would also be decidable.

```
1 String result;
2 if (the program P given the input I halts) {
3     result = "This program halts!";
4 } else {
5     result = "This program does not halt.";
6 }
```

Listing 2.2: An attempt at solving the halting problem using points-to analysis.

Clearly, if it were possible to perform precise points-to analysis on the program in Listing 2.2, it would also solve the halting problem for any program `P` and input `I` since the pointer `result` would point to different strings at the end of the program depending on whether the program `P` would halt or not.

The result of precise points-to analysis being undecidable does however not make it impossible to perform points-to analysis and gain valuable information. One solution is to overapproximate and return a result which also includes objects in the points-to set which the pointer cannot possibly point to. For the program in Listing 2.2, returning the result that the pointer may point to either of the strings at the end of the program is sound, meaning that the result includes all possible objects that the pointer can point to, but imprecise which means that it also includes objects that the pointer cannot point to. It is trivial to produce a sound result by stating that any pointer may point to any object, but such a result would not be of any use.

For our and other points-to analyses, this means that it is impossible to obtain a result without any overapproximation. Overapproximation should be minimized as much as possible. This is because it makes the result less useful, as it includes objects which a pointer cannot actually point to. For this reason, all points-to analysis algorithms produce an overapproximated result. We chose to implement Andersen's analysis as its inclusion-based approach does this to a lesser extent compared to, e.g., Steensgaard's analysis.

2.3 Abstract Interpretation

The overapproximation in the previous section can be described more formally using the abstract interpretation framework. Abstract interpretation was first introduced by Patrick and Radhia Cousot in 1977 [7]. An abstract interpretation of a program approximates the actual behavior of the program with an abstract model of the program. The main idea is that the abstract representation of the program gives information about the semantics of the concrete program, while being easier to reason about.

An example of abstract interpretation given in the original paper is determining the sign of an expression. It is possible to determine the resulting sign of the expression $-17 * 23$ by performing the multiplication, $-17 * 23 = -391$, and seeing that the result is negative. An easier way is to only reason about the signs. The original expression can then be represented as $(-)*(+)$. Using the fact multiplying a positive and negative number always results in a negative number, it can be deduced that the original expression will also result in a negative number. This means that information about the behavior of the original expression has been gained by only reasoning about its abstract model.

However, if the examined program instead features addition, then the information of the abstract model is not sufficient to determine whether the result is positive, negative or zero since adding a negative and a positive number can result in a number with any sign. It must then be assumed that the result might have any sign.

While it is not necessary to apply this level of abstraction to determine what sign the result of an expression will have, the same principle can be used for more complex problems. A halting problem analyzer that uses abstract interpretation could for some program-input pairs return a definite answer that the program will halt or that it will not halt, but for other program-input pairs simply return that the program possibly will halt. It is thus always possible to obtain a correct result by essentially relaxing the problem to also allow the answer "maybe".

Through the use of abstract interpretation, it is possible to obtain the overapproximated result described in the previous section. Our analysis attempts to utilize abstract interpretation to obtain the result. However, in its current state it does not strictly adhere to the constraints of abstract interpretation as the current analysis is not sound. Implementation of solutions to the limitations listed in Section 4.3 could solve this.

2.4 Heap Modeling

The heap is a portion of memory that is usually used for storing data that is needed for a longer amount of time. In Java, all objects are generally stored on the heap and can be referred to using a reference. A reference is similar to a pointer in that it refers to a memory location but does not support pointer arithmetic. From this point, the term pointer will be used instead of reference even when discussing Java as the discussed concepts apply to any language with pointers. Objects that have been created inside a method can still persist even after the method has been returned from. Local variables are stored on the stack and can be removed once the method is returned from.

```
1 Node node = null;
2 for (int i = 0; i < 100; i++) {
3     Node newNode = new Node();
4     newNode.next = node;
5     node = newNode;
6 }
```

Listing 2.3: Storing 100 objects on the heap.

The size of the heap is conceptually unbounded, but limited by available memory in practice. Due to the unbounded size, an abstract model of the heap is needed, as it would not

be feasible to compute something for a very large or infinite heap model. Another challenge is that there is no simple way of referring to a specific allocation since they, unlike variables, do not have a name [20].

In the code snippet in Listing 2.3, 100 objects of the type `Node` are created. One way of referring to the different objects are by using access paths, which means differentiating between the object `node.next` and `node.next.next`. This way of representing the heap can be limited to a finite set with k-limiting, which means paths with more than k accesses are grouped together. When $k = 0$, only the base variable (`node` in this case) is represented and all accesses are grouped together (`node.*`). With $k = 2$, the objects that could be referred to in the abstract model of the heap from Listing 2.3 would be {`node, node.next, node.next.next, node.next.next.*`}. The first three access paths would each represent one object in the actual heap and the last access path, `node.next.next.*`, represents an infinite number of objects.

Another way is to represent objects by their allocation site, which means that all objects created at the same point in the program are grouped together. In the program in Listing 2.3, all objects created on line 3 would be treated as one. This loses some information about the actual heap structure since many objects can be grouped into one allocation site. This could matter when trying to determine if two pointers point to the same memory location and one points to `node.next` and the other `node.next.next`. If representing objects with allocation sites, it would falsely conclude that the pointers point to the same memory location since the objects were created at the same point in the program. This is the heap representation used in our implementation.

2.5 Andersen's Analysis

Andersen's analysis is an inclusion-based type of points-to analysis introduced by Lars Ole Andersen in his 1994 PhD thesis [3]. An inclusion-based points-to analysis means that the analysis uses subset relations between points-to sets ($x \subseteq y$, the set y includes the set x) while a unification-based analysis unifies points-to sets with the equality relation ($x = y$). Andersen's analysis generally gives a more precise result (it includes fewer objects in points-to sets that the associated pointer cannot point to) than a unfication-based analysis like Steensgaard's analysis since it captures the directionality of assignments, the assignment `x = y` can only change the value of the variable `x`. However, Andersen's analysis is slower since Steensgaard's analysis can be done in almost linear time with regard to program size compared to the cubic time complexity of Andersen's analysis [29, 44].

The analysis is performed in two stages: constraint collection and solving.

2.5.1 Constraint Collection

When analyzing a Java program, four types of constraints need to be considered: Allocation, Assignment, Field Store and Field Load. We are using the same names for the constraints as in [25]. The constraints are shown in Table 2.2 along with an example statement that would generate a constraint of that type. We use the notation $pts(\mathtt{x})$ for the points-to set of `x`, i.e., the set of all objects that `x` may point to. We sometimes write that a constraint is created from one pointer to another. This means that the points-to set

elements flow in that same direction, e.g., an Assignment constraint from `x` to `y` means $pts(\texttt{x}) \subseteq pts(\texttt{y})$.

Name	Example	Imposed constraint
Allocation	`x = new A();`	$o_i \in pts(\texttt{x})$
Assignment	`x = y;`	$pts(\texttt{y}) \subseteq pts(\texttt{x})$
Field Store	`x.f = y;`	$pts(\texttt{y}) \subseteq pts(\texttt{o}_\texttt{i}\texttt{.f}), \forall o_i \in pts(\texttt{x})$
Field Load	`x = y.f;`	$pts(\texttt{o}_\texttt{i}\texttt{.f}) \subseteq pts(\texttt{x}), \forall o_i \in pts(\texttt{y})$

Table 2.2: Types of constraints used in Andersen's analysis.

Since all statements do not conform to the forms in Table 2.2, e.g., `x = y.f.g;` or `x.f = new A();`, programs must first be normalized by introducing temporary variables. After normalizing the program and collecting all constraints, the system of constraints can be solved to find the points-to set of each pointer.

2.5.2 Solving the Constraints

A solution to the constraint system is a set of pairs consisting of a pointer and its points-to set that satisfy all constraints. One way to solve the system of constraints is to create a pointer flow graph and propagate the points-to sets of pointers along the graph. The graph is initialized with pointers as nodes. The Assignment constraints introduce edges from the pointer on the right side to the pointer on the left side. Each pointer has a points-to set associated with it. The points-to sets are initialized for each pointer with the objects created in the Allocation constraints.

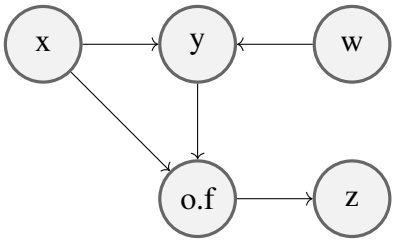
The Field Store constraint introduces field nodes of the form `o`$_i$`.f` to the pointer flow graph. These nodes represent the fields of objects. A Field Store constraint adds an edge from the pointer on the right hand side of the assignment to the new field node and a Field Load constraint creates edges from field nodes to the pointer on the left side.

It is possible to obtain a solution by iterating over the constraints and adding objects to points-to sets so that the constraint is satisfied, and repeating this until all constraints are satisfied. For example, a Field Store constraint generated by the statement `x.f = y;` can be satisfied by adding all objects in $pts(\texttt{y})$ to the points-to sets $o_i.f$ where o_i is all objects in $pts(\texttt{x})$. However, since the points-to sets $pts(\texttt{y})$ and $pts(\texttt{x})$ can both change when processing other constraints, a once satisfied constraint can cease to be satisfied after satisfying some other constraint. It is therefore necessary to iterate over the constraints repeatedly until all constraints are simultaneously satisfied.

A more efficient way to find a solution is to use a worklist. A worklist contains work that needs to be done when solving the constraints. During solving, work is added to and removed from the worklist and once there is no more work to process in the worklist, the constraints have been solved. The use of a worklist eliminates the need to propagate information that has already been propagated along the pointer flow graph. A worklist-based solver is described in Section 4.1.4. An example of a program and its pointer flow graph is shown in Figure 2.1. The steps for creating the pointer flow graph and finding the points-to sets for the pointers in the program are shown in Figure 2.2, where points-to sets are shown next to their respective pointers. Note that while it may appear in the example

```
w = new A();
y = x;
y = w;
w.f = y;
z = y.f;
z.f = x;
```

(a) Program



(b) Pointer flow graph

Figure 2.1: A short program and its resulting pointer flow graph. o represents the object created on the first line of the program.

that the problem can be solved in linear time as each constraint is only processed once and at most one edge is added at each step, in the worst case cubic time is required [29].

Our implementation uses a worklist-based solver and is described in Section 4.1.4. Since the points-to result is obtained by first collecting the constraints and then solving them, it is necessary to be able to solve the constraints in some way.

2.5.3 Analysis Sensitivity

The sensitivity of the analysis describes how much and how granular the information extracted from the source code is. Knowing the different sensitivities gives the user of the analysis a good idea of its possibilities and limitations.

There is a relation between sensitivity, precision, and performance. A more sensitive analysis is generally slower and gives a more precise result, but this is not always the case. For example, a call-site sensitive analysis needs to handle each call to a method separately, but the average size of the points-to sets will be smaller as the points-to set for several call sites are not merged. Since handling large points-to sets can be detrimental to performance during solving, it can sometimes be faster to instead handle multiple smaller sets.

It can also be the case that a more sensitive analysis is faster than a less sensitive analysis for a particular program, but slower for another one. Therefore, it is important to evaluate program analyses on a large set of programs since an analysis could benefit from some characteristic of a program but be worse than other analyses in the general case.

Flow-Sensitivity

Flow-sensitivity concerns the order of the statements in the program. Consider the Java program in Listing 2.4. A flow-sensitive analysis would conclude that `secondAnimal` points to `Cat@line2` after executing Line 2, but to `Dog@line1` after Line 3. A flow-insensitive analysis would conclude that `secondAnimal` can point to either, having the points-to set `{Dog@line1, Cat@line2}`. Andersen's analysis is flow-insensitive, but since the constraints are directional, some dataflow is still modelled [29]. For example, the variable `firstAnimal` on Line 1 will still only have `Dog@line1` in its points-to set, despite the assignment on Line 3.

```
1 Animal firstAnimal = new Dog();
2 Animal secondAnimal = new Cat();
3 secondAnimal = firstAnimal;
```

Listing 2.4: Example used to illustrate flow-sensitivity.

```
1 public void main(...) {
2     process(new Dog());
3     process(new Cat());
4 }
5
6 public void process(Animal a) {
7  System.out.println(a);
8  ...
9 }
```

Listing 2.5: Example used to illustrate context sensitivity.

Context-Sensitivity

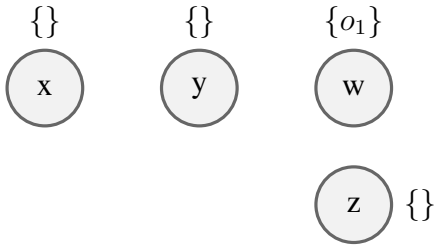
When the analysis is not only for a single method but spans across multiple methods, it is called an *interprocedural analysis*. In such cases, there are two main approaches to consider regarding how to model a method call. The *context-sensitive* approach distinguishes between separate calls to a function and takes the context of where the function were called into account [29]. In Listing 2.5, a context-sensitive analysis could analyze

the method `process` both with `Dog@line2` as a parameter and with `Cat@line3` separately. A *context-insensitive* analysis would instead analyze the method with both inputs at the same time. This has the drawback of making the analysis more imprecise, but the benefit of decreasing the problem size. Our implementation of Andersen's analysis is context-insensitive.

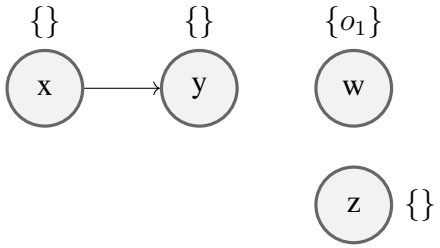
Field-Sensitivity

Field sensitivity regards the way fields of an object are handled, i.e., how the creation of field nodes like $o.f$ should be represented in Figure 2.1b. One way is to treat all fields for an object as the same node. This makes the analysis *field-insensitive* [29]. When the analysis instead creates a unique node for each field for an object, the analysis is *field sensitive*. This is the case for our implementation. This makes the analysis more precise but can increase the solving time as each field needs to be handled separately.

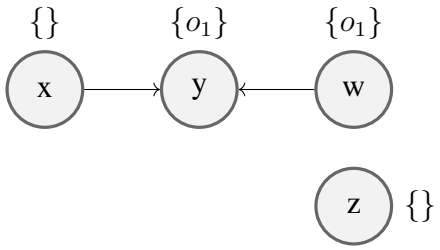
Statement	`w = new A();`
Constraint	Allocation
Description	Add o_1 to $pts(\texttt{w})$.



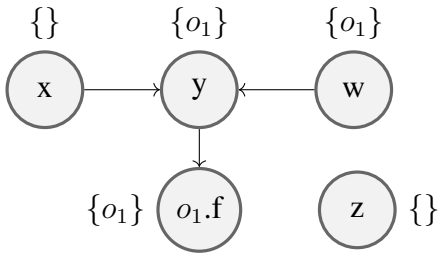
Statement	`y = x;`
Constraint	Assignment
Description	Create an edge from `x` to `y`



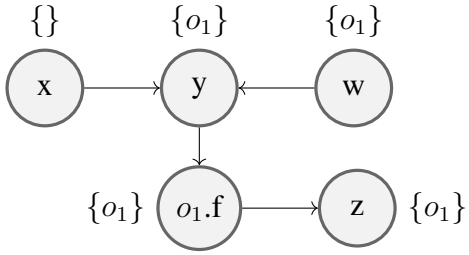
Statement	`y = w;`
Constraint	Assignment
Description	Create an edge from `w` to `y`



Statement	`w.f = y;`
Constraint	Field Store
Description	Create an edge from `y` to $\texttt{o}_i\texttt{.f}\ \forall o_i \in pts(\texttt{w})$



Statement	`z = y.f;`
Constraint	Field Load
Description	Create an edge from $\texttt{o}_i\texttt{.f}$ to `z` $\forall o_i \in pts(\texttt{y})$



Statement	`z.f = x;`
Constraint	Field Store
Description	Create an edge from `x` to $\texttt{o}_i\texttt{.f}\ \forall o_i \in pts(\texttt{z})$

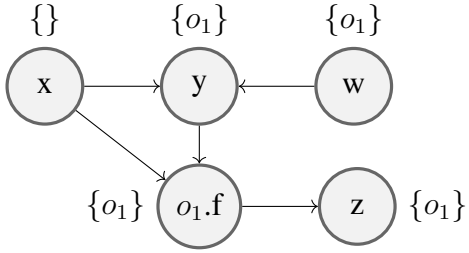


Figure 2.2: Computing the points-to sets for the program in Figure 2.1a.

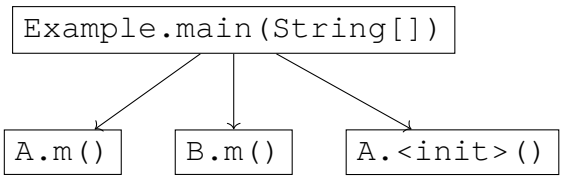
2.6 Call Graph

A call graph is a graph that describes how different functions in a program may call one another [2]. It is commonly drawn with the functions as nodes and if a call to the function `g` can be made from inside the function `f`, then an edge is drawn from `f` to `g`.

Languages that support dynamic dispatch complicates the construction of a call graph. This is because a call to a method may call different functions depending on the class of the object with the accessed method. A possible solution to this problem is to assume that a call to a method may call any matching method in a subtype of the declared type of the object. This is however pessimistic, since the possible types of the object during execution might only be a subset of all subtypes of the declared type. Figure 2.3b shows an overapproximated call graph for the program in Figure 2.3a. It is overapproximated because there is an edge from `Example.main()` to `B.m()`, but that method will never be called from the main method. The graph shown in the figure is what would be generated when using class hierarchy analysis (CHA) [9]. CHA constructs a hierarchy of the classes so that `A` and `B` are children of `I` and determines that calls to methods that are accessed from objects with the declared type `I` can call matching methods in any descendant class. One way to improve the result is to instead use rapid type analysis (RTA) [5]. RTA improves upon CHA by only including classes that are instantiated in some point in the program, so the call to `B.m()` would be correctly omitted if RTA was used instead. An even more precise result can be obtained using points-to analysis by checking the type of the objects in the points-to sets for pointers with accessed methods. In our implementation, we utilized the call graph to limit part of the program to analyze (see Section 4.1.3).

```
1  interface I { void m(); }
2  class A implements I {
3      public void m() {}
4  }
5  class B implements I {
6      public void m() {}
7  }
8  class Example {
9      public static void
           main(String[] args) {
10         I obj = new A();
11         obj.m();
12     }
13 }
```

(a) A program with a virtual method call.



(b) An overapproximated call graph.

Figure 2.3: A call graph overapproximation for a program.

2.7 Source Code Representation

To perform static analysis, a model of the source code is required. A common way that was used in this report is to use the same tools used in compiler construction, using the initial steps excluding machine code generation. Compilations steps are often grouped into front-end, middle-end and back-end [2]. The front-end converts the source code into an intermediate representation. The middle-end performs various optimizations on the intermediate representation. Finally, the back-end outputs code which is executable by the target architecture. The front-end consists of the following steps:

Scanning The source code is transformed into tokens by a scanner. Tokens could for example be a keyword like `public` or the name of an identifier. How to tokenize a program is often defined using regular expressions, which can be used to generate a scanner. This step is also known as lexical analysis.

Parsing Using the tokens produced in the previous step, an abstract syntax tree is constructed by a parser. The tree structure describes the program structure. For instance, a while statement node might have one child that is the loop condition and another child containing the statement that is executed while the condition is true. The parser constructs the tree by processing the tokens and building the tree according to rules created from a language grammar.

Semantic analysis The semantics of the program are analyzed in order to ensure that a syntactically valid program also is valid with regard to other rules of the language. This can include type checking to make sure that variables of a certain type are only used in valid contexts, and confirming that all non-void functions contain a return statement.

Intermediate Representation From the abstract syntax tree, an intermediate representation of the source code is created from the abstract syntax tree. This representation can be used for optimizations, and can be transformed into machine code or executed by the JVM, such as in Java.

A visual representation of the scanning and parsing stages along with their input and output is shown in Figure 2.4. Our analysis implementation works on the abstract syntax tree level, but it is common for Java analyses to work on some other transformed version of the source code, such as bytecode [14, 41].

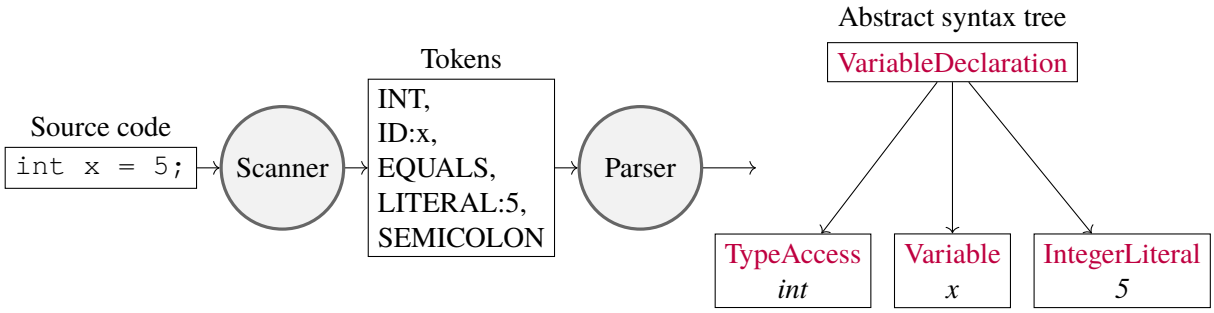


Figure 2.4: Steps of a compiler front-end.

Chapter 3
Reference Attribute Grammars

In this chapter, we describe Attribute Grammars, Reference Attribute Grammars as well as give an introduction to the meta-compilation system JastAdd and the Java compiler ExtendJ. JastAdd is an implementation of the Reference Attribute Grammar formalism which ExtendJ is built on. In this thesis, one of the questions we wanted to answer was how well-suited Reference Attribute Grammars are to implement a points-to analysis. We therefore built the analysis as an extension to ExtendJ.

3.1 Attribute Grammars

Attribute Grammars were first introduced by Knuth in 1968 [23]. Using Attribute Grammars, it is possible to add attributes to nodes in the abstract syntax tree. An attribute is a value that is defined by an equation. The equation can include other attributes in the node itself or in other nodes.

A synthesized attribute is an attribute where the attribute value is defined using attributes of descendant nodes. When using synthesized attributes, information is propagated upward in the abstract syntax tree.

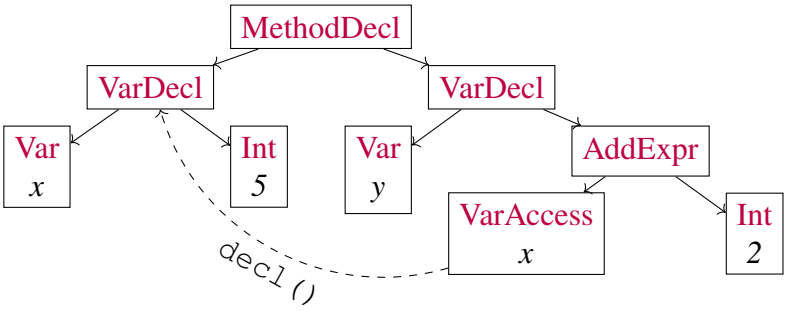
The value of an inherited attribute is defined using attributes of ancestor nodes. An inherited attribute propagates information downward in the abstract syntax tree.

3.2 Reference Attribute Grammars

Reference Attribute Grammars extend Knuth's Attribute Grammars to allow attributes to evaluate to a reference to an AST node [15]. This enables easier expression of non-local dependencies, i.e., equations that use attributes in nodes that can be located anywhere in the tree and not just ancestors or descendants. Figure 3.1 shows a reference attribute that evaluates to the AST node in which the variable was declared. This can be useful to, e.g.,

access the declaration AST node of a variable, check the declared type and report a type error if the variable is used in an invalid context.

```
1  void m() {
2      int x = 5;
3      int y = x + 2;
4  }
```



(a) Code example. **(b)** Abstract syntax tree.

Figure 3.1: A simplified abstract syntax tree showing a reference attribute called `decl()`.

3.3 JastAdd

JastAdd is an meta-compilation system that implements an extension of Attribute Grammars called Reference Attribute Grammars [16]. JastAdd supports the full creation of a compiler, as well as other related program analysis tools. It also supports the concept of modules using *aspects*. Attributes can be added to the same node in multiple aspects, making it easy to extend a language with new features without modifying the original code.

Attributes in JastAdd are evaluated *on-demand*, i.e., they are only evaluated once accessed. This means that execution time is not negatively affected by attributes that are not accessed. Additionally, since attributes should not contain any side effects according to the JastAdd specification, attributes can be cached by using the keyword `lazy`. This can be useful to avoid recomputation of attributes that are expensive to compute.

An example of how JastAdd could be used to evaluate boolean expressions in a language consisting of "true" and "false" literals and the "and" and "or" operators using synthesized attributes is shown in Listing 3.1. The value of the attribute in True and False only depend on the nodes themselves and the value of the attribute in And and Or depend on descendants' attributes. Figure 3.2 shows an abstract syntax tree for the program `true && false || true` along with the value of the `isTrue()` attribute of each node.

```
1  syn boolean True.isTrue()  = true;
2  syn boolean False.isTrue() = false;
3  syn boolean And.isTrue() = getLeft().isTrue() &&
       getRight().isTrue();
4  syn boolean Or.isTrue()  = getLeft().isTrue() ||
       getRight().isTrue();
```

Listing 3.1: Evaluating boolean expressions using synthesized attributes.

Listing 3.2 illustrates how inherited attributes can be used to define an attribute that consists of information of the node's surrounding context. Program is the root node of the abstract syntax tree. When evaluating the `inConstructor()` attribute, the AST is

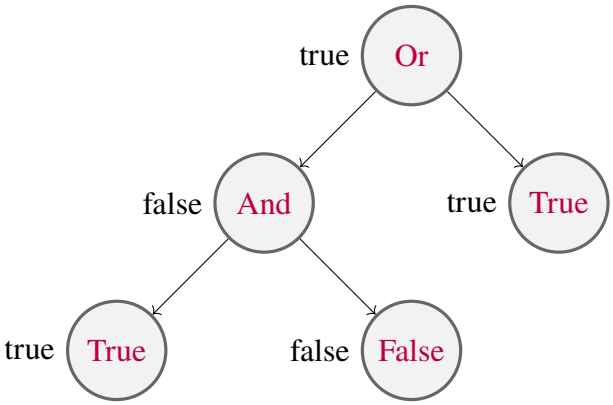


Figure 3.2: An abstract syntax tree for the program `true && false || true`. The value of the `isTrue()` attribute is shown next to each node.

traversed upward until a definition is encountered, which will either be in a constructor node or in the AST root.

```
1 inh boolean SuperConstructorAccess.inConstructor();
2 eq ConstructorDecl.getChild().inConstructor() = true;
3 eq Program.getChild().inConstructor() = false;
```

Listing 3.2: Determining whether a call to `super()` is inside a constructor using inherited attributes.

JastAdd also supports collection attributes, which is a type of attribute that allows for contributions from any node. As the name implies, they evaluate to a collection of values contributed by other nodes. A common usage of collection attributes is to collect errors in the program. An example of how this can be done is shown in Listing 3.3. The collection is initialized with the value inside of the square brackets. If a call to `super()` is made outside of a constructor (which is not allowed in Java), a string describing the problem is contributed to the `errors()` collection in the AST root node Program. It is also possible to specify which node's collection a contribution should be made to, but it is not needed to specify this for the root node.

```
1 coll Set<String> Program.errors() [new HashSet<>()];
2 SuperConstructorAccess contributes "super() called outside of
    constructor" when !inConstructor() to Program.errors();
```

Listing 3.3: Using collection attributes to collect semantic errors in the program.

3.4 ExtendJ

ExtendJ is a Java compiler implemented with JastAdd [48]. It supports Java versions from Java 4 up to Java 11. Due to the modularity of JastAdd, it is easy to extend this implementation with an analysis, making use of the attributes already defined for the language itself [4, 49]. Static program analyses implemented as extensions to ExtendJ include IntraJ [35], a control-flow analysis framework; JFeature [34], a tool for collecting

information about used syntax and language features in a set of Java programs and SinfoJ [39], an information flow analysis.

Chapter 4
Approach

This chapter contains a description of our implementation of Pecka. We cover the overall design, how different language features are modeled and finally limitations of the analysis.

4.1 Design

In this section the design of Pecka is covered. We describe the relations between AST nodes, the pointer flow graph and the created constraints; how a subset of the code to analyze can be selected as well as how the constraint solver works.

4.1.1 Allocation Sites and Nodes

We use the allocation site abstraction described in Section 2.4 to represent objects. This means that all objects that can be created at one point in the program are grouped together into one allocation site. We chose this heap model as the allocation sites can naturally be represented by AST nodes which create an object, such as NewExpr.

AST nodes that are relevant for points-to analysis implement the interfaces `Node` or `AllocationSite`. A `Node` is an AST node that is a pointer or can evaluate to a pointer. Examples are VariableDeclarator which is a pointer and MethodAccess which evaluates to a pointer if the return type is not a primitive type. They are represented as nodes in the pointer flow graph. AST nodes that create an object implement `AllocationSite`. They are elements in the points-to sets which are propagated along the pointer flow graph. Examples of common allocation sites are the NewExpr and StringLiteral nodes.

`AllocationSites` also implement the `Node` interface. The `AllocationSites` create an Allocation constraint with itself as both the object and the pointer. This allows for handling `AllocationSites` and `Nodes` in the same way, not having to differentiate between fields, pointers, and objects when generating constraints. This reduces the number of cases that need to be handled at the expense of an additional constraint. An example

```
A x = new A();
```

(a) Declaration statement.

$o_1 \in pts(\texttt{x})$

(b) Optimal constraints.

$o_1 \in pts(\texttt{NewExpr})$
$pts(\texttt{x}) \subseteq pts(\texttt{NewExpr})$

(c) Actual constraints.

Figure 4.1: Creating additional constraints to reduce number of cases to handle.

is shown in Figure 4.1. With the constraint in Figure 4.1b, it is immediately obvious that the created object is part of $pts(\texttt{x})$, but for the constraints in 4.1c the solver needs to propagate the object to $pts(\texttt{x})$. Since a NewExpr is not a pointer, it cannot actually point to anything, but it is convenient implementation-wise to let it have a points-to set anyway. The advantage of this solution is that no matter what the right hand side is, a Assignment constraint can be created. This also applies to situations other than assigments.

4.1.2 Constraint Generation

Some AST nodes create constraints, such as the AssignExpr AST node which creates a Assignment constraint. Constraints that are generated inside a method are contributed to a collection attribute in that method's MethodDecl node. Field initializers are not inside methods and instead contribute their constraints to all ConstructorDecl nodes of that class.

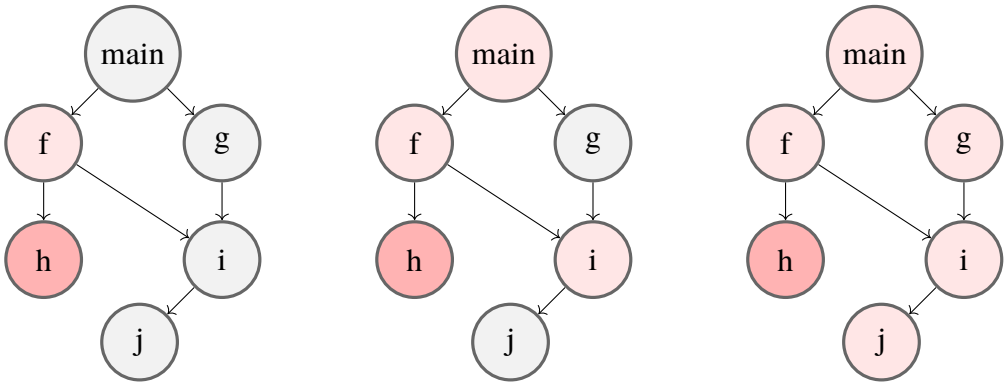
After selecting which methods should be analyzed (all methods when analyzing the entire program), constraints from those methods are collected and passed to the solver, which computes the points-to set for all pointers in the selected methods.

4.1.3 Limiting Search Distance

We include an option to only collect constraints from methods within a certain distance from a specified method. The purpose of this is to try to find the points-to set of pointers in a method without performing a whole program analysis. We use the call graph generated by CAT [33] and start from the selected method and include all methods which can be reached within k steps when traversing the call graph in either direction. Setting k to 0 only includes the selected method, while setting k to 1 includes all methods that either calls or can be called by the current method. The program then computes the points-to sets using the constraints from the methods within distance k. Figure 4.2 shows a call graph where highlighted functions are the ones constraints would be collected from when computing a result for all pointers in `h` with different values for k.
A points-to set can be affected by information in other methods in three different ways:

1. Reading a field that has been written to in another method.
2. Reading a value has been passed as a parameter.
3. Reading a value that has been returned from another method.

This also applies transitively, e.g., if the pointer of interest reads from another pointer which in turn reads a return value, then the function that returned that value must be included. The first and second scenario make it so that predecessors in the call graph must



(a) $k = 1$ **(b)** $k = 2$ **(c)** $k = 3$

Figure 4.2: A call graph where methods reachable from `h` within different values of k are highlighted.
The pink coloured nodes show the reachable methods.

be included and the first and third scenario necessitate the inclusion of successors in the call graph.

Limiting search distance can cause the analysis to miss objects to include in a points-to set. One example is if a pointer can point to an object that was created in a method that is minimum three steps in the call graph away from the current method, then this object will not be found when the search distance is limited to two steps. Limiting search distance makes the analysis unsound: all possible objects that a pointer can point to are not guaranteed to be in its points-to set. However, due to limitations listed in Section 4.3 and the fact that we do not model all Java language features our results cannot be guaranteed to be sound even with unlimited search distance. We discuss what possible values of k can be used and where an unsound result may fit in Section 6.1.

4.1.4 Solver

We use a worklist-based solver for solving the constraints. We chose to use a worklist-based solver since it is faster than simply iterating over the constraints until the solution is reached [25]. Other approaches such as using a Datalog engine to solve the constraints [37] are also possible but not explored due to the time constraints of writing this thesis. The psuedocode is shown in Algorithm 1, where $<:$ is used as the subtyping relation. It is based on a presentation by Tian Tan [45] on how to implement points-to analysis. The solver maintains a worklist consisting of pairs where the first element is a pointer flow graph node and the second is a set of objects.

On lines 15-24, the algorithm propagates objects which have not already been propagated along the pointer flow graph. It is also possible to not calculate the set difference on Line 15 and instead propagate the entire set *pts*, but since this information is already available in succeeding nodes there is no need to do so.

On lines 25-34, the Field Load and Field Store constraints are handled. It is not possible to only process these constraints once as for the Allocation and Assignment constraints as they depend on the points-to sets which change during solving.

Type filtering for non-field pointers is performed on Line 17. It can also be done for

field pointers but the structure of our implementation makes it more difficult. Without performing type filtering, it is possible for the points-to sets to include objects which the pointer cannot point to. Pointers in Java can only point to objects which subtype the pointer, so other objects can safely be removed. Listing 4.1 shows a situation where type filtering is useful. Without type filtering, the allocation sites `A@line3` and `B@line4` would both be included in $pts(\texttt{b})$ (since the analysis is flow-insensitive), but `b` cannot point to objects of type `A` so `A@line3` can be removed.

```
1 class Example {
2     public static void main(String[] args) {
3         Object obj = new A();
4         obj = new B();
5         B b = (B) obj;
6     }
7 }
8 class A {}
9 class B {}
```

Listing 4.1: A program illustrating the use of type filtering.

It is also possible to perform type filtering after the constraint system has been solved, but we found this to have lower performance, since it appears to be faster to perform type filtering during solving than to propagate larger, unfiltered sets.

Algorithm 1 Solver

```
1:  procedure SOLVE
2:      for each pfg node n do
3:          n.pts ← ∅
4:          n.succ ← ∅
5:      end for
6:      worklist ← ∅
7:      for each Allocation constraint p = o do
8:          worklist ← worklist ∪ {⟨p, {o}⟩}
9:      end for
10:     for each Assignment constraint p = q do
11:         q.succ ← q.succ ∪ {p}
12:     end for
13:     while worklist is not empty do
14:         pfgNode, objs ← pop(worklist)
15:         deltaPts ← objs \ pfgNode.pts          ▷ Only propagate new information
16:         if pfgNode is not a field node then
17:             deltaPts ← {n ∈ deltaPts | n <: pfgNode}          ▷ Perform type filtering
18:         end if
19:         if deltaPts is not empty then
20:             pfgNode.pts ← pfgNode.pts ∪ deltaPts
21:             for each node s ∈ pfgNode.succ do
22:                 worklist ← worklist ∪ {⟨s, deltaPts⟩}
23:             end for
24:         end if
25:         if pfgNode is not a field node then
26:             for each object o ∈ deltaPts do
27:                 for each Field Load constraint p = q.f where q = pfgNode do
28:                     addEdge(o.f, p, worklist)
29:                 end for
30:                 for each Field Store constraint p.f = q where p = pfgNode do
31:                     addEdge(q, o.f, worklist)
32:                 end for
33:             end for
34:         end if
35:     end while
36: end procedure
37: procedure addEdge(p, q, worklist)
38:     if q ∉ p.succ then
39:         p.succ ← p.succ ∪ {q}
40:         if p.pts is not empty then
41:             worklist ← worklist ∪ {⟨q, p.pts⟩}
42:         end if
43:     end if
44: end procedure
```

4.2 Language Features

In this section, we describe how some of the supported of language features are handled and which constraints they generate. The selection consists of features which we consider interesting, excluding language features which are straightforward to handle such as variable assignments.

4.2.1 Methods

```
1  interface A {
2      A m(String str);
3  }
4  class A1 implements A {
5      public A m(String str) { return str; }
6  }
7  class A2 implements A {
8      public A m(String str) { return null; }
9  }
10 class Example {
11     public static void main(String[] args) {
12         A a = new A1();
13         String s1 = a.m("ABC");
14         String s2 = a.m("DEF");
15     }
16 }
```

Listing 4.2: A program with two method calls.

To model methods we use the `allDecls` attribute provided by CAT. The attribute evaluates to the set of methods that a method call may refer to. In the example in Listing 4.2, `allDecls` returns method `m` in both `A1` and `A2`. Although in this case it is easy to see that only the method in `A1` can be called, the general case requires pointer analysis. CAT only looks at the declared type of the variable `a` and returns matching methods in all subtypes.

The MethodAccess node on Line 13 contributes Assignment constraints from `"ABC"` to the ParameterDeclaration node `str` in both methods.

Inside the methods, all ReturnExpr nodes contribute an Assignment constraint from the return value (`str` and `null` in this case) to the MethodDecl nodes. The MethodAccess nodes contribute Assignment constraints from the matching MethodDecls to themselves so that they "evaluate" to the return values. This way of handling methods is call-site insensitive which means that different calls to a method are not treated separately, so the analysis would return the result $pts(\texttt{s1}) = pts(\texttt{s2}) = \{\texttt{"ABC"}, \texttt{"DEF"}, \texttt{null}\}$.

4.2.2 Object Keywords: this and super

The `this` expression is handled conservatively. When a `this` expression is used inside a class, Assignment constraints are created from every allocation site of objects that either have the same type or a subtype of that class. A drawback of this approach is that it collects

constraints from the entire program instead of only the selected methods if using a distance limited analysis. It does however not affect the resulting points-to sets as no Allocation constraints are created for the allocations sites outside of the selected methods and thus they are never included in any points-to set. The effect on performance is also almost certainly very minor as the constraint is only processed once during solving. A more precise approach would be to let `this` "evaluate" to the object which the method has been accessed from.

The `super` expression is handled in almost the same way as the this expression, but it instead creates Assignment constraints from all allocation site of objects that have a type which is a supertype of the class that the expression is inside.

4.2.3 Dot

To handle chains of accesses, such as `o.f.m()[20].g`, we introduce "temporary" variables. This is done by adding a higher order attribute to the Dot AST node. A higher-order attribute is an attribute that evaluates to an AST node, but unlike other AST nodes, it is created by an attribute instead of the parser. First an Assignment constraint is added from the declaration of the base variable (`o` in the previous example) to the leftmost dot's temporary variable. If a field access is on the right side of a dot, we create two constraints: a Field Store constraint and a Field Load constraint. This results in that the temporary variable is equal to the accessed field as $pts(\texttt{t}) \subseteq pts(\texttt{x.f}) \wedge pts(\texttt{x.f}) \subseteq pts(\texttt{t}) \leftrightarrow pts(\texttt{x.f}) = pts(\texttt{t})$. Array accesses are handled in the same way, but instead use a unique name that does not overlap with normal field names.

Method accesses do not need to be handled in this way as our analysis is context insensitive. It does not matter which object the accessed method belongs, so it is sufficient to simply create Assignment constraints from the method declarations found by CAT to the temporary variable.

The final access creates constraints to a temporary variable which represents the entire access chain. For example, when assigning the value of an access chain to a variable (e.g., `x = o.f.g;`), an Assignment constraint is created from this temporary variable to the assignment target.

4.2.4 Arrays

Our model is array-insensitive, i.e., all writes and reads to an array are treated equally no matter what index is used. Differentiating between writes and reads to different locations are difficult when the expression used to index the array is not an integer literal. Determining the possible values of that expression would require a data flow analysis. To avoid this we chose to group all locations in the array together. The ArrayInit and ArrayCreationExpr AST nodes are allocation sites where new array objects are created.

4.2.5 List and Map

Since we cannot analyze bytecode (discussed in Section 4.3.1), we cannot analyze classes in the Java standard library. It is possible to manually implement support for methods

that are part of the standard library by letting calls to them generate suitable constraints that match their behavior. We have added support for the `put` and `add` methods in the interfaces `Map` and `List` respectively as well as the `get` method for both interfaces. We chose these methods as they are commonly used and not supporting them would make the analysis result worse by not including reads from calls to `get` in the points-to sets. Writes with `put` or `add` create a Field Store constraint and `get` creates a Field Load constraint.

4.2.6 Fields

Field Store constraints are created for field declarations that include an initializer. The constraints are created from the initializer to each object that is a subtype of the class that include the field.

Static fields are somewhat more complex to handle since they introduce a relation between a method and a class that is not represented in the call graph. Since our analysis uses the call graph to select which methods to collect constraints from, it can happen that a static field is accessed in some method but the constraints for assigning a value to the field is not included. Currently, static fields work as expected if either the class' constructor is included in the methods to collect constraints from or if the static field's initializer does not contain method calls.

4.3 Limitations

Unsoundness means that the analysis has missed a points-to relationship which is undesirable. In this section, we list sources of unsoundness which are not intentional design decisions as for the distance strategy described in Section 4.1.3.

These sources of unsoundness are due to the properties of the systems we built our analysis with, or language features that are difficult to model and were not implemented due to time constraints.

4.3.1 Bytecode Analysis

Java compilers such as javac or ExtendJ generate bytecode which is in turn executed by a Java virtual machine. ExtendJ primarily works on source code, but is also able to read bytecode to some extent. It can read bytecode to gain information about, e.g., method signatures and class hierarchies. However, already compiled code is not included in the abstract syntax tree produced by ExtendJ. Due to this limitation, our analysis cannot analyze classes in the Java standard library or other already compiled code without manually adding support for specific methods (such as in Section 4.2.5).

```
1 class Example {
2     public static void main(String[] args) {
3         List<String> list = List.of("ABC");
4         String s = list.get(0);
5     }
6 }
```

Listing 4.3: A program with two calls to methods in the Java standard library.

While it is easy to see that `s` in Listing 4.3 will point to `"ABC"`, it is not possible to determine without either having access to the source code or having implemented support for the `of` and `get` methods. One common way to produce a sound result in certain situations is to include all objects in the points-to set. However, this is not possible in this situation since new objects could be created inside of the bytecode and these would not be visible to the analysis. Instead it would be necessary to return a more abstract result that says that this pointer might point to anything, which may or may not be satisfactory depending on what the analysis result should be used for.

A bytecode reader that enabled ExtendJ to construct an AST from bytecode would solve this problem and enable the analysis to analyze previously unanalyzable code. We are not aware of any development of a bytecode reader for ExtendJ at this moment, and developing one is not in the scope of this thesis.

4.3.2 Reflection

Reflection would be difficult to handle even with the ability to analyze bytecode. Using reflection it is possible to instantiate objects or invoke methods by passing the name of a class or method as an argument. In Java, an object of class `A` can be created with `Class.forName("A").newInstance()`. When the argument to `forName` is not a string literal it becomes difficult to determine what objects can be created at that point in the program. Many program analyses do not handle reflection in a sound way as assuming that any object can be created by a call to `forName` could be bad for both perfomance and precision [27].

One way to handle reflection is to dynamically record which objects are created and which methods are invoked by using reflection and then using the information to perform the static analysis. Qilin, which we compare our analysis to in Chapter 5, handles reflection in this way. One thing to note is that the result will only be sound if all objects that can be created with reflection are created in some recorded execution. This applies similarly for method invocations.

4.3.3 Additional Limitations

The following is a list of unsupported language features. Since they are not handled, they are also possible sources of unsoundness. These features are not impossible to implement, but due to time constraints they were considered out of scope for this thesis.

Exceptions When an exception is thrown, a new Exception object is created and when caught, the pointer in the catch clause points to the caught exception.

Accessing an outer class with `this` An outer class can be accessed from the inner class using `<OuterClass>.this`.

Accessing classes with `class` E.g., `int.class`.

Method calls in static field initializers E.g., `static A a = m();` only works if performing a whole program analysis or if the analyzed methods contain a instantiation of the class that the field is contained in.

Features added in Java 5 or later The analysis handles the enhanced for loop which was introduced in Java 5 but does not handle any other language features introduced in Java 5 or later. These include enumerations, lambda expressions, switch expressions and more.

Main method arguments The `String[]` parameter in the main method is not modeled as an array of strings, which it should since pointers can be made to point to the array and its contents.

JNI Java Native Interface allows for calling code written in other languages, such as C. To analyze this would require implementing support for other languages which is out of scope.

Chapter 5
Evaluation

This chapter concerns the evaluation of Pecka. We start by describing the different concepts and techniques used for evaluating a program analysis in Section 5.1. Then, a description of the tests and benchmarks used for our analysis is presented in Section 5.2, with the results in the preceding Section 5.3.

5.1 Evaluating an Analysis

In this section, we give definitions of terminology often used when describing program analyses and describe why it can be difficult to evaluate how good a points-to result is.

5.1.1 Precision and Recall

Our analysis answers the question: *"What objects can the requested pointer point to at runtime?"* In some cases, we instead answer what *types* a pointer points to, simply by looking at the type of each object in the points-to set and removing identical types. A correctly generated fact is called a true positive (TP), while an incorrectly generated fact will be false positive (FP). If the analysis fails to report a fact that should be reported, it will be a false negative (FN). A true negative is similarly when the analysis correctly does not report a fact. All combinations can be visualized in a confusion matrix, as shown in Table 5.1.

Precision is a measurement of how many of the generated facts are true. For points-to analysis, precision describes the correctness of the reported points-to set compared to the actual points-to set. It can be calculated using the following equation:

$$\text{Precision} = \frac{\#TP}{\#TP + \#FP} = \frac{\text{correct reports}}{\text{number of reports}}$$

		Predicted	
		Yes	No
Actual	Yes	TP	FN
	No	FP	TN

Table 5.1: Confusion matrix for an analysis describing how a result from an analysis is compared to the actual values, generating combinations of False/True Positives/Negatives.

Precision is measured in the range $[0, 1]$, where 0 represents the lowest and 1 the highest achievable precision.

Recall describes how many of all true facts are generated for some query. In a points-to analysis context, recall describes the proportion of actual objects a pointer could point that the analysis also correctly identified. It can be calculated with the equation:

$$\text{Recall} = \frac{\#TP}{\#TP + \#FN} = \frac{\text{correct reports}}{\text{number of ground truths}}$$

Recall is measured in the range $[0, 1]$, where 0 represents the lowest and 1 the highest achievable recall.

Consider an example where we report what types the variable a can point to. The reported points-to set was `{Cat, Dog, Sheep}`, while the actual true set should be `{Cat, Dog, Cow, Horse}`. This analysis has the two true positives `Cat` and `Dog`, one false positive `Sheep`, and two false negatives `Cow` and `Horse`. The precision of the analysis is therefore $\#TP/(\#TP + \#FP) = 2/3 = 0.67$. The recall for the example would be $\#TP/(\#TP + \#FN) = 2/4 = 0.5$. Note that full recall is obtainable simply by answering that the pointer can point to every object used in the program. Similarly, perfect precision can be achieved by under-approximating the result and only returning completely certain results. The optimal scenario is to gain full precision for a fully sound analysis. This is however not possible as described in Section 2.2.The specific goal of our analysis is to find a balance between precision, recall, and time to compute the result.

5.1.2 Soundness

A program analysis can be either a *may* or a *must* analysis. A may analysis produces facts that may hold, e.g., that a specific pointer may point to a specific object during some execution of the program. A must analysis produces facts that always must hold, e.g., that the variable x always has the same value as the variable y.

A may analysis is *sound* if it includes all possibilities, i.e., it has perfect recall. For a points-to analysis, this would be that no objects that a pointer could point to are excluded from the points-to set [38]. A must analysis is sound if no facts that sometimes do not hold are produced, i.e., it has perfect precision.

Soundness is often an important property for program analyses to have, since it enables more drastic conclusions to be drawn from the result. A compiler making optimizations to a program using unsound program analysis results could result in the program not behaving

as expected, e.g., the program crashes due to some code being optimized away under the false assumption that it cannot ever be executed.

An alternative to soundness is soundiness [27]. A soundy result is a result that is not quite sound for some reason, e.g., it does not model some language feature in a sound way. A notoriously hard to handle language feature is reflection. Using reflection, it is possible to instantiate an object by using the class name in the form of a string. Since it is sometimes impossible to determine what the string will be, e.g., if it is given as input to the program, it must be assumed that an object of any type can be created if a sound result is desired. Conservatively modeling a program in this manner can result in a result that is so imprecise that it is no longer useful or negatively impacts the performance of the analysis. A soundy result can still be useful in contexts where soundness is not essential, such as when giving auto-completion suggestions in an IDE.

5.2 Experimental Setup

Due to the challenges in determining whether the analysis produces the correct results, several evaluation methods were employed for our analysis. The main idea was first to evaluate the analysis for correctness, and when we with some certainty could determine that the analysis was correct, we expanded the evaluation to include larger programs without a ground truth. All benchmarking were executed an octa-core Intel i7-11700K 3.6 GHz CPU with 128 GiB DDR4-3200 RAM, running Ubuntu 20.04.3 with Linux 5.15.0-106-generic and the OpenJDK Runtime Environment version “1.8.0_302”. The evaluations used will be described in the upcoming sections, with the result presented in Section 5.3.

5.2.1 Manual Ground Truths

Benchmarking a points-to analysis is difficult because, to the best of our knowledge, precise information about what a pointer may point to does not exist for any real-world programs. Determining whether two pointers alias, i.e., they may point to the same object at some point in the program, is NP-hard when only considering the flow-insensitive result [18]. Since this can easily be computed by checking whether the intersection of the pointers’ points-to sets is empty or not, precise flow-insensitive points-to analysis is also NP-hard. This means that it is not possible to compute precise flow-insensitive points-to information in polynomial time (unless P = NP). While it might still be possible to implement an analysis that can produce a precise flow-insensitive result, we are not aware of any such analyses or results produced by one. Attempting to implement such an analysis is out of scope for this thesis. Having precise information available would be useful since it could be used to assess the quality of the results produced by our analysis.

Many articles tackle this problem by assuming that their analysis implementation produces a sound result after comparing it against other analyses that are considered state-of-the-art obtaining similar results. When receiving an similar result, they can consider smaller points-to set as more precise yet still sound [14, 25, 46]. This is not an option in this thesis, since our implementation does not produce a sound result, making it harder to reason about the cause of smaller points-to sets. The reason for the unsoundness is discussed in Section 4.3.

Another attempt to address this fact is PointerBench [41]. PointerBench is a manually created benchmark suite consisting of small Java programs along with points-to and alias information to compare against [40]. It tests things that can often be problematic for points-to analyses but excludes many language features and the programs only consists of a few methods. Although it is impossible to write a set of benchmarks that exhaustively covers all possible scenarios for a points-to analysis, PointerBench still serves as a useful start for asserting the precision and recall of a points-to analysis. For this reason, we used PointerBench as one of the factors evaluating the analysis.

PointerBench consists of several Java files that use static test functions to describe the test case and its result. The functions themselves are defined as empty functions and exist to be identified when parsing the program to find the test case and the expected result. Listing 5.1 provides an example of one of the benchmarks. The statement `Benchmark.alloc(1)` marks the succeeding statement as an allocation with an `allocId` of 1. By looking at the string parameters for `Benchmark.test` we can see that the test is for the local variable `b`, with the expected result of the allocation with `allocId` 1. It also contains more information related to alias information, but in our case, we were only interested in comparing the expected `allocId`'s to the findings of our analysis. To cover more Java 5 features, we manually augmented the suite with more examples. In those tests, we omitted the alias information, as we do not use them for our evaluation.

```
1 Benchmark.alloc(1);
2 A a = new A();
3
4 A b = a;
5 Benchmark.test("b",
6         "{allocId:1, mayAlias:[a,b], notMayAlias:[],
             mustAlias:[a,b], notMustAlias:[]}");
```

Listing 5.1: Excerpt from the benchmark *SimpleAlias1* in PointerBench. The test queries the varaible b for all alloactions it can point to. The allocations are marked with `Benchmark.alloc`.

To use these benchmarks, JastAdd code was created to extract the information inside the parameter lists. Each allocation site was assigned an attribute of the specified id if it were a preceding statement of `Benchmark.alloc`, or `0` otherwise. These ids were later compared to the `allocId`'s in the `Benchmark.test` method. The result of this comparison shown in Section 5.3.1.

5.2.2 Generated Ground Truths for Antlr

To effectively validate the implementation of our k-distance option for our algorithm (see Section 4.1.3), a more intricate test set was required. The programs in PointerBench were not representative of a real world program, thus not making the call graph large enough to validate the advantage of this strategy. To our knowledge, there exists no large test set with points-to information. This prompted us to generate a dataset containing the runtime types of the variables in a program. This was achieved by programmatically adding print statements containing each method's parame-

ter's `param.getClass().getSimpleName()`. The testset was limited to only include the parameters of the methods for simplicity, coupled with the fact the tool we were to compare against run on a jar file, making local variables obfuscated. The parameters were found using regular expressions, modifying the source code in a Python script. The program was executed multiple times to cover as many program paths as possible. The program selected for this was the parser generator Antlr [1] with version 2.7.2. The dataset was then generated by executing the program using all different parser options, generating a CSV and JSON file containing the result. The dataset and the code used to produce it is available in the repository in the directory `comparison`.

An example of a modified method is shown in Listing 5.2. The code on line 2 has been automatically inserted and each time the method is executed, the parameter's declared type, name, the file it is in, the method signature and its dynamic type are printed.

```
1 public void refTreeSpecifier(Token treeSpec) {
2     if (treeSpec != null) System.out.println("Token;treeSpec;antlr/
          MakeGrammar.java;refTreeSpecifier(Token treeSpec);" +
          treeSpec.getClass().getSimpleName());
3     context().currentAlt().treeSpecifier = treeSpec;
4 }
```

Listing 5.2: Automatically inserted `println` statement.

Using this dataset, we performed the benchmark with different values of k, measuring recall and precision. In addition, we also ran it using Qilin, which is the current state of the art for pointer analysis [14]. Qilin facilitates many versions of pointer analysis, we selected the analysis called insens (Andersen's context-insensitive analysis), since it was the implementation that most closely matched ours. The result is shown in Section 5.3.2

5.2.3 Benchmark Programs

Additional programs were used for benchmarking Pecka in terms of speed, memory usage, as well as the types and allocation sites reported. We evaluated nine programs, with method counts ranging from 548 to 8057, covering a wide variety of program sizes and coding styles. The following procedure was performed for every project:

- Repeat 3 times:
    - For $k = 0$ to 8:

      Measure the time to retrieve the points-to set for 100 random methods, with measuring starting from a parsed state.

All methods in the program were considered for random selections, resulting in that large methods took a long time, while others were faster.

To make the measurement reflect steady state, the 100 random methods were computed 2 times before taking the actual measurement. Our testing indicates that this should be enough warm up runs to reach steady state, which we identified by plotting the time for 50 warm up runs on some project and visually observing where the time taken stagnated. We tested values of k from 0 to 8, as 8 was observed to be the point where no additional result

could be found for all projects, indicating that all reachable methods in the call graph were included at that point.

The memory usage was obtained by limiting the heap space for the JVM with the java argument `-Xmx`. For instance, to limit the heap space to 8GB, the jar is executed with `java -Xmx8G -jar program.jar`. The benchmark to find the points-to set for 100 random methods was executed until the program suffered a `OutOfMemoryError`, where the lowest value without an error was considered the required memory consumption. These values were found using binary search, looking at the required memory in 50 MB steps between 0 and 5 GB.

Our analysis cannot influence the speed of parsing the program, but since it is required before performing the analysis, we include the measurements for completeness. To obtain steady-state measurements for the benchmark projects, we averaged the last values from 50 runs, each run parsing the benchmark program 10 times.

5.3 Results

This section shows the results from the evaluation of Pecka, using the tests and benchmarks described in the previous section.

5.3.1 Manual Ground Truths

Table 5.2 shows the result of the PointerBench benchmarks. For a full breakdown of each individual test case, see Table A.2 in Appendix A. The analysis found all expected results, yielding a recall of 1.0. The precision of the analysis was 0.72. It is worth mentioning, however, that the precision in this test is closely related to how the test code is written. A test case with more assignments on the requested variable would make the precision go down since the analysis is flow-insensitive, and the test looks for the flow-sensitive result. For instance, replacing the line `A b = a;` on line 4 in Listing 5.1 with `A b = new A(); b = a;` would decrease the precision due to an additional false positive.

Metric	TP	FP	FN	Precision	Recall
Value	55	21	0	0.72	1.00

Table 5.2: Results from the extended PointerBench Benchmarks.

5.3.2 Generated Ground Truths for Antlr

The result for the benchmark on the generated ground truths, compared against the current state of the art, can be seen in Figure 5.1, with the complete counts included in Table A.1 in Appendix A. As described in Section 5.2.2, the truths used to calculate recall and precision have been obtained by running the program multiple times with different configurations, logging the runtime types of the parameters for methods in the parser generator Antlr. As the distance increases, the recall also increases up to a distance of 5, where it performs

almost equal to our analysis using an infinite distance, indicating that this is also close to the average steps needed to reach all reachable methods from an average method in the program. At this point, the recall has reached 0.95. Looking at the precision, we notice that it start high with a gradual decrease as the distance increases. This shows that the false positives that are produced by the analysis occur to a greater extent farther away in the call graph. We found the best tradeoff between time, precision, and recall to occur at the distance 3. It performs the analysis in 0.85 seconds, with a recall of 0.71 and the precision 0.78. In other words, using this distance enables us to find 71% of the types, with 78% of the types we reported being correct, in approximately 1/5 of the time compared to using an infinite distance.

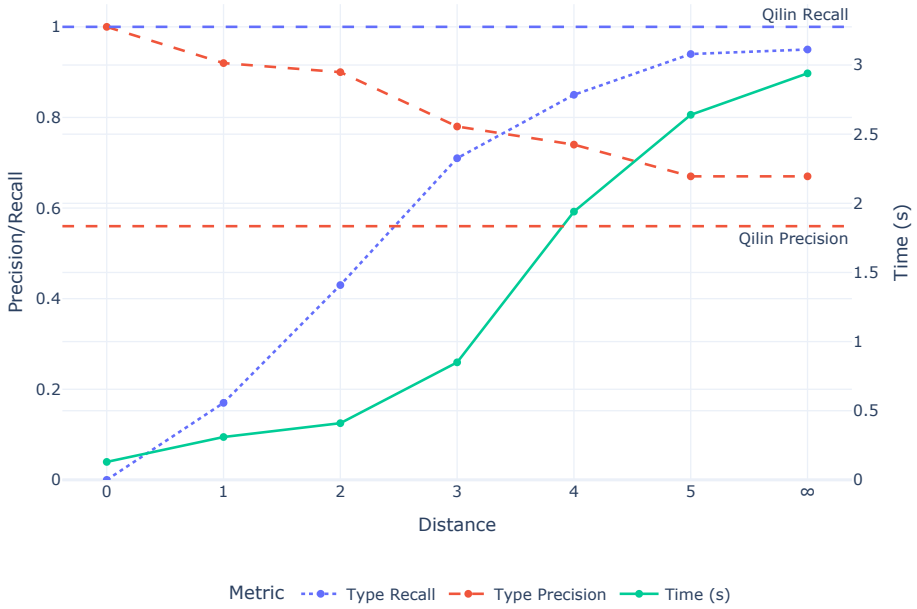


Figure 5.1: Recall and precision for Pecka compared to the types reported from the generated Antlr dataset, using different values for k using our analysis. The dotted lines show the precision and recall for Qilin on the full program. The time for running the full analysis on Qilin were 8.52 s. For the full result in table format, see Table A.1 in Appendix A.

It is worth mentioning how we interpret the results reported by Qilin. Although Figure 5.1 display the recall 1 with a precision of 0.56 from this dataset, there are some factors preventing a too harsh conclusion to be drawn when comparing the tools. The first reason is that since the absolute truths we have been generated by executing the program, all program paths have not been taken, which might cause some possible types to be excluded. If Qilin has reported any of these types while Pecka did not, Qilin would incorrectly get a false positive resulting in a lower precision. The second reason that makes a comparison unjust, is that Qilin handles reflection dynamically, meaning that it had to execute the program before producing the result. What we can interpret from the comparison is that at the higher values of k, our analysis performs similarly to the current state of the art, which indicates that our analysis likely behaves as expected. The full analysis could be executed on Qilin in 8.52 seconds on the same benchmark machine we used. This is not including

the time taken to produce the JAR file from source code. Although this time is higher than Pecka with infinite distance, we can not draw any conclusions other than noting that the times reported by our analysis is reasonable.

5.3.3 Benchmark Programs

The time measurements to calculate the points-to set for all pointers in one method can be seen in Figure 5.2. Comparing the allocation sites found in Figure 5.3 with the types found in Figure 5.4, we notice that all the types are found faster than all the allocations for a method. We can also see that the majority of the types have been found when a distance of 5 is used. This is more visible in Figure 5.6, where the average percentage of found types compared to the total is plotted together with the average percentage of the total time for the projects on each distance. This figure also shows that 56% of the total types found by Pecka can be found in 22% of the time using the distance 3. Although these programs do not have access to what the ground truth, the results seem to align well with the results found on Antlr in Section 5.3.2. This indicates that the results found are generalizable to other programs too.

The memory measurements can be seen in Figure 5.5, with the data separated for each project in Figure A.1 in Appendix A. We can see that the memory usage differs across the projects, but generally increase together with the distance. At the distance 4, there seem to be a point where an increase in memory is required.

The parsing times together with the mean times from Figure 5.2 for the projects can be seen in Table 5.3. The times for the projects ranged between 0.31 and 1.70 seconds.

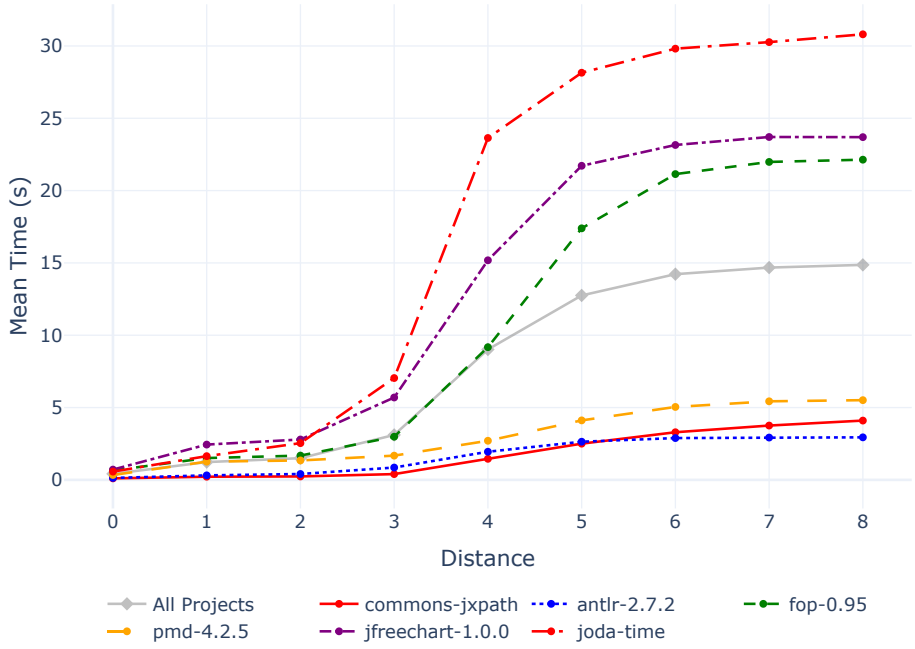


Figure 5.2: Mean time for calculating the points-to information for a method across the benchmarks for different values of k. The gray line shows the mean for all projects. The numbers together with parse data can be seen in Table 5.3.

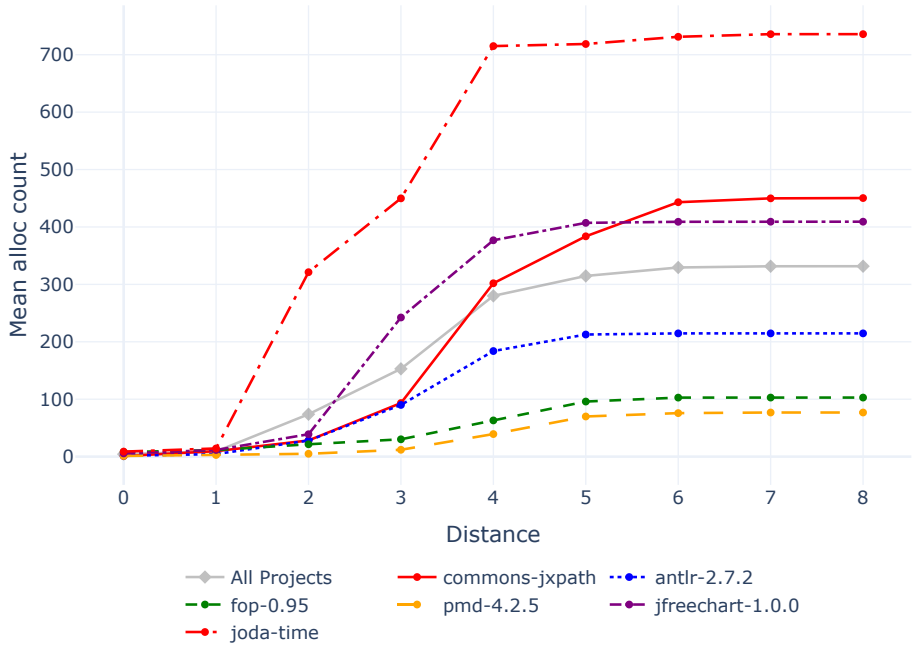


Figure 5.3: Mean allocation count for a method across the benchmarks for different values of k. The gray line shows the mean for all projects.

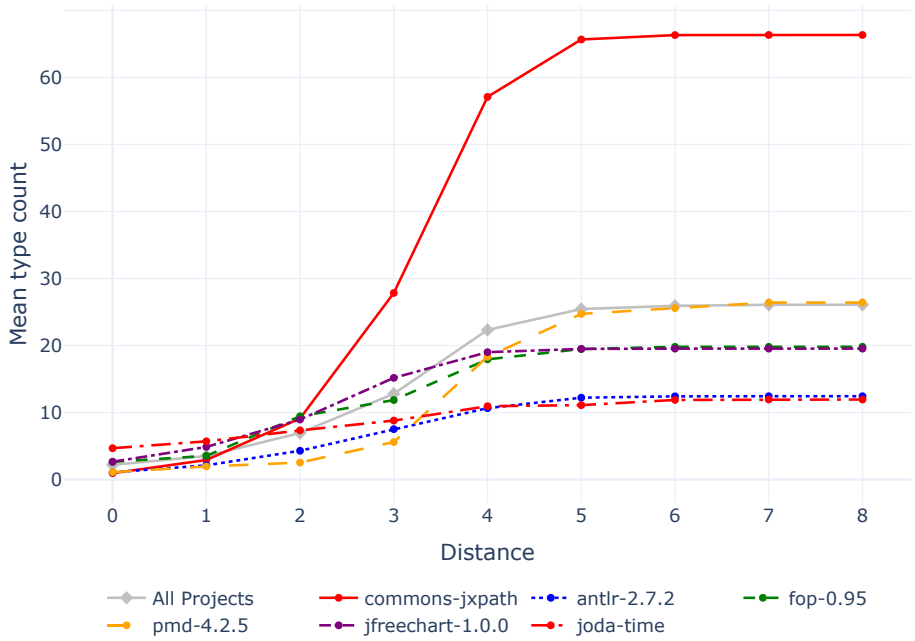


Figure 5.4: Mean type count for a method across the benchmarks for different values of k. They gray line shows the mean for all projects.

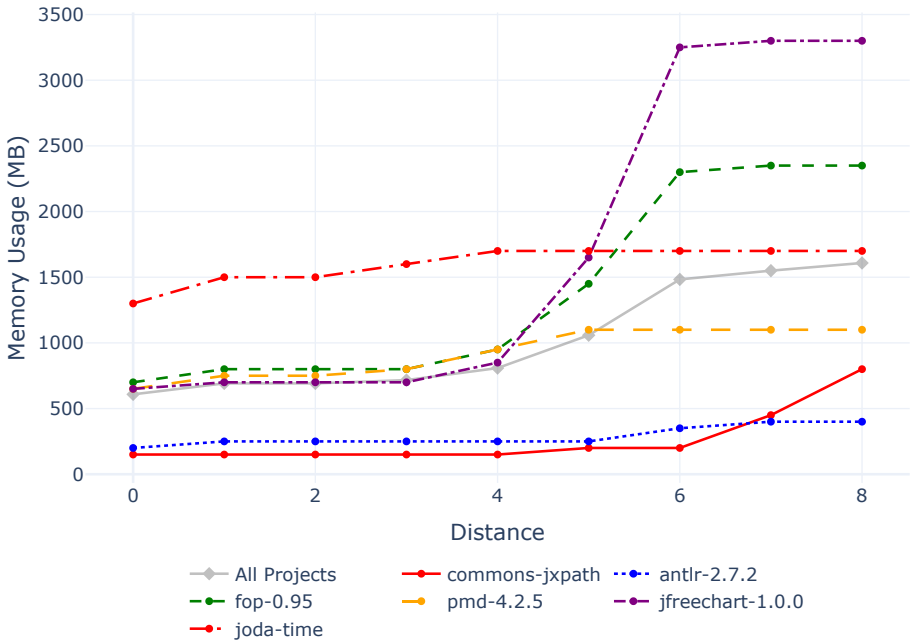


Figure 5.5: Mean memory usage for calculating the points-to information for a method across the benchmarks for different values of k. The gray line shows the mean for all projects.

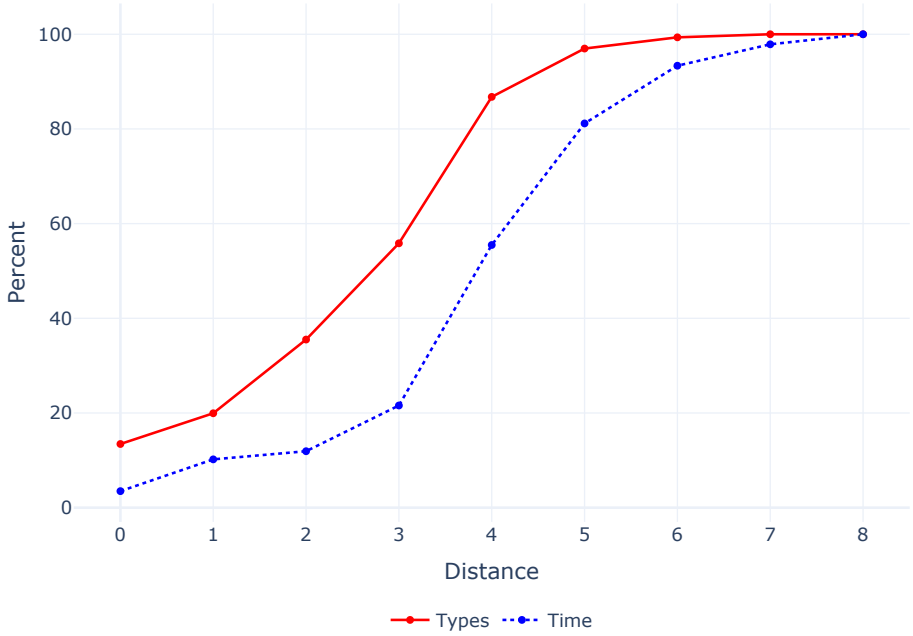


Figure 5.6: The percentage of types found and the percentage of time taken for a given distance, compared to the result at distance 8. These values are calculated from the average of all projects as plotted in Figure 5.4.

		Mean times for different values of k (s)								
Project	Parse time (s)	$k = 0$	$k = 1$	$k = 2$	$k = 3$	$k = 4$	$k = 5$	$k = 6$	$k = 7$	$k = 8$
commons-jxpath	0.31	0.10	0.22	0.24	0.40	1.46	2.50	3.29	3.75	4.10
antlr-2.7.2	0.45	0.13	0.31	0.41	0.85	1.94	2.64	2.89	2.93	2.94
fop-0.95	1.60	0.65	1.51	1.68	2.98	9.17	17.39	21.14	21.98	22.14
pmd-4.2.5	1.09	0.34	1.26	1.34	1.68	2.70	4.11	5.04	5.43	5.51
jfreechart-1.0.0	1.54	0.71	2.44	2.79	5.69	15.18	21.72	23.15	23.71	23.70
joda-time	1.70	0.57	1.64	2.54	7.04	23.64	28.15	29.81	30.27	30.81

Table 5.3: Parse times and time to retreive the points-to set for an average method as plotted in Figure 5.2 for the projects used in the benchmarks programs.

Chapter 6
Discussion

This chapter discusses the result and possible applications that an analysis with the characteristics of Pecka can be used for. It follows with a discussion of the advantages and drawbacks to using Reference Attribute Grammars to implement a points-to analysis, concluding with suggestions for future work.

6.1 Pecka in Interactive Environments

Our analysis enables users to obtain all points-to results related to a single method on request. By specifying the distance k, the analysis can be limited to only include methods within k steps in the call graph originating from the requested method, as described in Section 4.1.3. As shown in Section 5.3, this approach increases the speed of the analysis, but may result in missing some points-to relationships. Pecka can also be used to run a full analysis on the entire program, providing all points-to results for all methods.

The primary requirement to enable the use of a points-to result in an interactive environment, such as an IDE, is to provide the result in a reasonable time. What the reasonable time *is*, we suspect varies depending on the requirements of the client that is using the result of our analysis. Similarly, we also believe that the soundness requirements will differ across different clients. We have identified the following possible client's that could make use of our analysis:

Find allocations Using the points-to result, a more strict version of the LSP language feature *Find References* can be implemented to show only allocations. Limiting the value of k might yield results with higher precision by only returning results close to the location of the request, as shown in Figure 5.1.

Improve autocompletion Points-to information could enable better completions for untyped or gradually typed languages. A part of why programmers prefer TypeScript to JavaScript is due to that the autocompletion is better for TypeScript [11], which

indicates a need for improvement in this area. Andersen's analysis has been implemented for JavaScript [13], where a reduction in execution time might make interactive use more feasible. For such analysis, a low distance could possibly be used, as a match could simply increase the ranking of a completion.

Code Navigation Many editors such as IntelliJ IDEA and Eclipse currently offer many ways to navigate a code base. The tools *Call Hierarchy* and *Go to Implementation(s)* are both used to navigate from the start of a method or a method call. Figure 6.1 shows the usage of these tools on the example code in Listing 6.1. It shows that all implementations from subtypes are included, ignoring the fact that even if there never is an instance of `Cow` created, it will still show up in the list of results. We believe that with the help of the result of a points-to analysis such as ours, these lists could be improved, either by sorting the list having the used methods first, or completely removing unused implementations. For the former alternative, a low distance value could suffice, which would gain the improvement with a lower time and memory cost. If results instead would be completely removed from the results list, a higher distance value might be preferred.

Bug detection Several bug detection tools, such as FindBugs [19], require the use of points-to information. Using the same techniques as in Pecka might make it possible to speed up the results with adequate accuracy, thereby increasing the user experience.

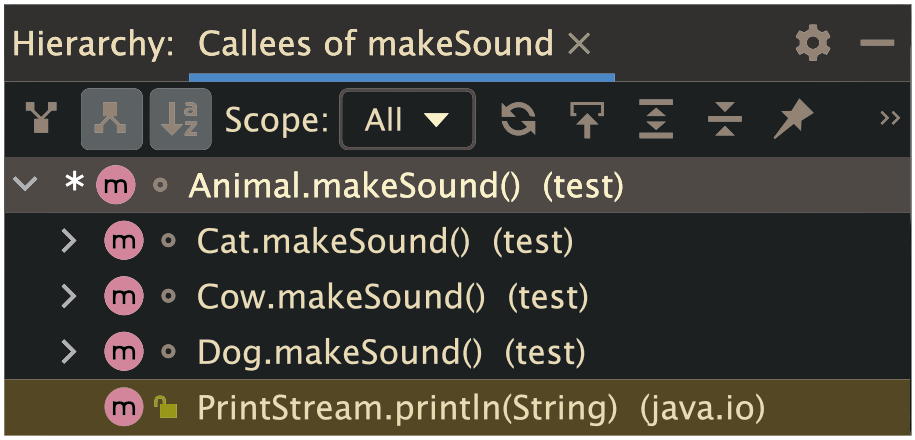
```
1  public class Example {
2    public static void
         main(String[] args) {
3      Animal a1 = new Dog();
4      Animal a2 = new Cat();
5      processAnimal(a1);
6      processAnimal(a2);
7    }
8
9    public static void
         processAnimal(Animal a) {
10     a.makeSound();
11   }
12 }
```

Listing 6.1: Code example processesing animals. The points-to set for a in `processAnimal` is `{Dog, Cat}`.

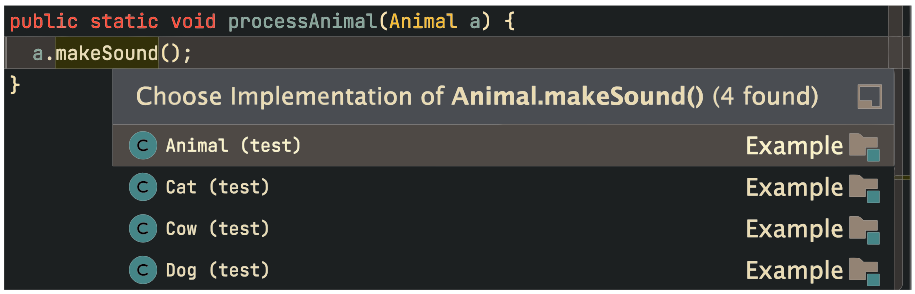
```
class Animal {
    void makeSound() {
        System.out.println("Some
            animal sound");
    }
}

class Cat extends Animal {
    @Override
    void makeSound() {
        System.out.println("Cat
            meows");
    }
}

class Dog extends Animal {
    @Override
    void makeSound() {
        System.out.println("Dog
            barks");
    }
}

class Cow extends Animal {
    @Override
    void makeSound() {
        System.out.println("Cow
            moos");
    }
}
```

Listing 6.2: Animal class definitions.

One advantage of having our tool implemented in JastAdd as an extension of ExtendJ is that we have our result at the source code level, which is not the case for other frameworks that operate on an intermediate representation. We believe this also gives an advantage for



(a) The tool *Call Hierarchy* displaying the callees of the function `makeSound` in Listing 6.1.



(b) The tool Go to Implementation(s). The image shows the usage of this tool on line 10 in Listing 6.1.

Figure 6.1: Code navigation tools in IntelliJ IDEA that could be improved by using more precise points-to information. As can be seen in the source code in Listing 6.1, the type `Cow` is never provided to the function and could therefore be removed or put last in the list of results.

usage in interactive environments, as the mapping between the code and points-to result is supported automatically. This could make it easier for clients to use the results from Pecka. Another advantage of working on the source code level is that our analysis does not need to spend time generating bytecode.

Pecka is not suitable for clients that cannot tolerate false negatives, i.e., where an object that should be included in the points-to set is not. The false negatives produced by our analysis are due to a combination of the distance strategy, together with the features we do not cover in the implementation. Such clients might include those that remove code based on the result, for example compiler optimizations and refactoring tools. In addition, analyses that run on the full program, would likely not benefit from our distance strategy, which in that case would require as many requests as there are methods in the program. In these cases, a full analysis of the whole program, solving the constraints at once would be more efficient.

It is not an easy task to answer if our analysis is fast and precise enough to be useful in interactive environments, as different clients and users may have different requirements and expectations of a result. While we have shown that the time taken for one method using the distance 3 ranges from 0.40 to 7.04 seconds, there exists programs that are larger than the 8057 methods, which might increase the time to get a result. The concept of distance is however not limited to our tool or to JastAdd itself, and we believe that the idea could be utilized in other points-to analyses too. The only requirement for an analysis that uses the distance method is to have the call graph of the program and a way to only analyze a part of the program. This could enhance other tools such as Qilin [14], to benefit from both the speedup of the distance strategy together with the extensive optimizations already implemented for their algorithms.

6.2 Implementation

ExtendJ and JastAdd are effective tools for implementing program analyses. It would have been considerably more challenging to implement the analysis without some type of framework, requiring everything to be implemented from scratch. The implementation of the constraint collection and solving was done using 1233 lines of code (not counting whitespace or comments).

Being able to define the attributes declaratively, by adding equations to the nodes, was beneficial in multiple ways. This approach enhanced the clarity of the code due to the mapping between the node and what it represents (for example, MethodDecl representing a method declaration). Furthermore, it enabled the reuse of attributes already defined on the node, originating from ExtendJ and CAT. The use of collection attributes greatly supported the implementation, making it straightforward for the nodes to contribute their constraints. It eliminated the need to handle all variations in a large loop to prevent traversing the AST multiple times, as this process is efficiently abstracted by JastAdd.

The fact that JastAdd evaluates attributes on-demand is useful when implementing analyses that work on a subset of the program to avoid execution time scaling with the size of the entire program, including parts which are not included in the analysis.

6.3 Future Work

The analysis implementation could be improved in many ways and there are many other approaches for points-to analysis that are worth exploring, including other algorithms, such as Steensgaard's analysis (see Section 2.2). It would be interesting to compare speed and precision to see whether the gained precision from using Andersen's analysis makes it worth having to limit the distance for a quick result. Additionally, it would be interesting to also try distance limiting the other algorithms and seeing how the performance and precision compare to Andersen's. The following list presents a selection of ways that Pecka can be improved:

- Add support for unsupported language features as well as those introduced in newer versions of Java, as described in Section 4.3.3.
- Improve performance by implementing various performance optimizations. Ways that performance could be improved in include collapsing cycles in the pointer flow graph [12] and using a more efficient set implementation [25].
- Add options for different types of sensitivities, in order to increase the precision of the results. Many other points-to analysis implementations allow for choosing the type of context sensitivity to use.
- Explore other ways of selecting methods, e.g., not taking as many forward steps as backward or basing the selection on a heuristic like the method's return type or on the result of some simple analysis, like checking for field writes.

Chapter 7
Conclusions

In this chapter we conclude the thesis by answering the research questions listed in Chapter 1.

RQ1 *How can we speed up the results from Andersen's analysis to enable usage in an interactive environment?*

We were able to speed up the analysis through only analyzing a subset of the code by selecting methods at a limited distance in the call-graph from the pointer we wished to analyze. Using the distance 3, we found a tradeoff between speed and recall, where most of the results could be found in a significantly shorter time. Although this results in an unsound analysis, it can still be of use in certain situations, such as in interactive environments where soundness is not essential. The analysis also benefits from that JastAdd attributes are evaluated on-demand.

RQ2 *How well-suited are Reference Attribute Grammars for implementation of points-to analysis?*

We found Reference Attribute Grammars to work well for implementing a points-to analysis. It was easy to generate constraints by specifying different rules for different types of AST nodes.

We found that some problems do not feel as natural as others to solve using RAGs. These include problems that do not depend on the structure of the AST, such as the constraint solving. While it is possible to solve these problems using RAGs, it can be difficult to adapt existing algorithms so that they would benefit from RAGs compared to an imperative implementation.

References

[1] ANTLR, https://github.com/secure-software-engineering/PointerBench, Accessed June 2024.

[2] Alfred V. Aho, Monica S. Lam, Ravi Sethi, and Jeffrey D. Ullman. *Compilers: Principles, Techniques, and Tools (2nd Edition)*. Addison-Wesley Longman Publishing Co., Inc., USA, July 2006.

[3] Lars Ole Andersen. Program Analysis and Specialization for the C Programming Language. *Citeseer*, 1994.

[4] Johannes Aronsson and David Björk. Extending the ExtendJ Java Compiler. *LU-CS-EX*, 2023.

[5] David F. Bacon and Peter F. Sweeney. Fast static analysis of C++ virtual function calls. In *Proceedings of the 11th ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, OOPSLA '96, pages 324–341, New York, NY, USA, October 1996. Association for Computing Machinery.

[6] Sam Blackshear, Bor-Yuh Chang, Sriram Sankaranarayanan, and Manu Sridharan. The Flow-Insensitive Precision of Andersen's Analysis in Practice (Extended Version). In *Static Analysis - 18th International Symposium, {SAS} 2011, Venice, Italy, September 14-16, 2011. Proceedings*, volume 6887, pages 60–76, September 2011.

[7] Patrick Cousot and Radhia Cousot. Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Proceedings of the 4th ACM SIGACT-SIGPLAN symposium on Principles of programming languages*, POPL '77, pages 238–252, New York, NY, USA, January 1977. Association for Computing Machinery.

[8] Manuvir Das. Unification-based pointer analysis with directional assignments. In *Proceedings of the ACM SIGPLAN 2000 conference on Programming language design and implementation*, PLDI '00, pages 35–46, New York, NY, USA, May 2000. Association for Computing Machinery.

[9] Jeffrey Dean, David Grove, and Craig Chambers. Optimization of Object-Oriented Programs Using Static Class Hierarchy Analysis. In Mario Tokoro and Remo Pareschi, editors, *ECOOP'95 — Object-Oriented Programming, 9th European Conference, Åarhus, Denmark, August 7–11, 1995*, pages 77–101, Berlin, Heidelberg, 1995. Springer.

[10] Lisa Nguyen Quang Do, James R. Wright, and Karim Ali. Why Do Software Developers Use Static Analysis Tools? A User-Centered Study of Developer Needs and Motivations. *IEEE Transactions on Software Engineering*, 48(3):835–847, March 2022.

[11] Lars Fischer and Stefan Hanenberg. An empirical investigation of the effects of type systems and code completion on API usability using TypeScript and JavaScript in MS visual studio. *ACM SIGPLAN Notices*, 51(2):154–167, October 2015.

[12] Manuel Fähndrich, Jeffrey S. Foster, Zhendong Su, and Alexander Aiken. Partial online cycle elimination in inclusion constraint graphs. *ACM SIGPLAN Notices*, 33(5):85–96, May 1998.

[13] Salvatore Guarnieri and Benjamin Livshits. GATEKEEPER: Mostly Static Enforcement of Security and Reliability Policies for JavaScript Code. *USENIX Association*, 2009.

[14] Dongjie He, Jingbo Lu, and Jingling Xue. Qilin: A New Framework For Supporting Fine-Grained Context-Sensitivity in Java Pointer Analysis. In *DROPS-IDN/v2/document/10.4230/LIPIcs.ECOOP.2022.30*. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2022.

[15] Gorel Hedin. Reference attributed grammars. *Informatica (Ljubljana)*, 24(3):301–317, 2000.

[16] Görel Hedin and Eva Magnusson. JastAdd—an aspect-oriented compiler construction system. *Science of Computer Programming*, 47(1):37–58, April 2003.

[17] Nevin Heintze and Olivier Tardieu. Demand-driven pointer analysis. *ACM SIGPLAN Notices*, 36(5):24–34, May 2001.

[18] Susan Horwitz. Precise flow-insensitive may-alias analysis is NP-hard. *ACM Transactions on Programming Languages and Systems*, 19(1):1–6, January 1997.

[19] David Hovemeyer. FindBugs™ - Find Bugs in Java Programs.

[20] Vini Kanvar and Uday P. Khedker. Heap Abstractions for Static Analysis. *ACM Computing Surveys*, 49(2):1–47, June 2017.

[21] Martin Kellogg, Vlastimil Dort, Suzanne Millstein, and Michael D. Ernst. Lightweight verification of array indexing. In *Proceedings of the 27th ACM SIGSOFT International Symposium on Software Testing and Analysis*, ISSTA 2018, pages 3–14, New York, NY, USA, July 2018. Association for Computing Machinery.

[22] Shubhangi Khare, Sandeep Saraswat, and Shrawan Kumar. Static program analysis of large embedded code base: an experience. In *Proceedings of the 4th India Software Engineering Conference*, ISEC '11, pages 99–102, New York, NY, USA, February 2011. Association for Computing Machinery.

[23] Donald E. Knuth. Semantics of context-free languages. *Mathematical systems theory*, 2(2):127–145, June 1968.

[24] C. Lattner and V. Adve. LLVM: a compilation framework for lifelong program analysis & transformation. In *International Symposium on Code Generation and Optimization, 2004. CGO 2004.*, pages 75–86, March 2004.

[25] Ondřej Lhoták and Laurie Hendren. Scaling Java Points-to Analysis Using Spark. In Gerhard Goos, Juris Hartmanis, Jan Van Leeuwen, and Görel Hedin, editors, *Compiler Construction*, volume 2622, pages 153–169. Springer Berlin Heidelberg, Berlin, Heidelberg, 2003. Series Title: Lecture Notes in Computer Science.

[26] Bozhen Liu, Jeff Huang, and Lawrence Rauchwerger. Rethinking Incremental and Parallel Pointer Analysis. *ACM Transactions on Programming Languages and Systems*, 41(1):6:1–6:31, March 2019.

[27] Benjamin Livshits, Manu Sridharan, Yannis Smaragdakis, Ondřej Lhoták, J. Nelson Amaral, Bor-Yuh Evan Chang, Samuel Z. Guyer, Uday P. Khedker, Anders Møller, and Dimitrios Vardoulakis. In defense of soundiness: a manifesto. *Communications of the ACM*, 58(2):44–46, January 2015.

[28] Anders Alnor Mathiasen and Andreas Pavlogiannis. The fine-grained and parallel complexity of andersen's pointer analysis. *Proceedings of the ACM on Programming Languages*, 5(POPL):1–29, January 2021.

[29] Anders Møller and Michael I. Schwartzbach. *Static program analysis*. Aarhus University, Denmark, October 2018.

[30] Jakob Nielsen. Response Time Limits: Article by Jakob Nielsen, 1993.

[31] Goran Piskachev, Stefan Dziwok, Thorsten Koch, Sven Merschjohan, and Eric Bodden. How far are German companies in improving security through static program analysis tools?, August 2022. arXiv:2208.06136 [cs].

[32] H. G. Rice. Classes of Recursively Enumerable Sets and Their Decision Problems. *Transactions of the American Mathematical Society*, 74(2):358–366, 1953. Publisher: American Mathematical Society.

[33] Idriss Riouak. Cat: Java Class Hierarchy Analysis, https://github.com/idrissRio/cat, Accessed June 2024.

[34] Idriss Riouak, Görel Hedin, Christoph Reichenbach, and Niklas Fors. JFeature: Know Your Corpus. In *2022 IEEE 22nd International Working Conference on Source Code Analysis and Manipulation (SCAM)*, pages 236–241, October 2022. ISSN: 2470-6892.

[35] Idriss Riouak, Christoph Reichenbach, Görel Hedin, and Niklas Fors. A Precise Framework for Source-Level Control-Flow Analysis. In *2021 IEEE 21st International Working Conference on Source Code Analysis and Manipulation (SCAM)*, pages 1–11, September 2021.

[36] Marc Shapiro and Susan Horwitz. Fast and accurate flow-insensitive points-to analysis. In *Proceedings of the 24th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, POPL '97, pages 1–14, New York, NY, USA, January 1997. Association for Computing Machinery.

[37] Yannis Smaragdakis and Martin Bravenboer. Using Datalog for Fast and Easy Program Analysis. In Oege De Moor, Georg Gottlob, Tim Furche, and Andrew Sellers, editors, *Datalog Reloaded*, volume 6702, pages 245–251. Springer Berlin Heidelberg, Berlin, Heidelberg, 2011. Series Title: Lecture Notes in Computer Science.

[38] Yannis Smaragdakis and George Kastrinis. Defensive Points-To Analysis: Effective Soundness via Laziness. In *DROPS-IDN/v2/document/10.4230/LIPIcs.ECOOP.2018.23*. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2018.

[39] Max Soller. SinfoJ: A simple Information Flow Analysis with Reference Attribute Grammars. *Lund University*, 2024.

[40] Johannes Späth. PointerBench, https://github.com/secure-software-engineering/PointerBench, Accessed June 2024.

[41] Johannes Späth, Lisa Nguyen Quang Do, Karim Ali, and Eric Bodden. Boomerang: Demand-Driven Flow- and Context-Sensitive Pointer Analysis for Java. In *DROPS-IDN/v2/document/10.4230/LIPIcs.ECOOP.2016.22*. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2016.

[42] Manu Sridharan, Satish Chandra, Julian Dolby, Stephen J. Fink, and Eran Yahav. Alias Analysis for Object-Oriented Programs. In Dave Clarke, James Noble, and Tobias Wrigstad, editors, *Aliasing in Object-Oriented Programming. Types, Analysis and Verification*, Lecture Notes in Computer Science, pages 196–232. Springer, Berlin, Heidelberg, 2013.

[43] Manu Sridharan, Denis Gopan, Lexin Shan, and Rastislav Bodík. Demand-driven points-to analysis for Java. *ACM SIGPLAN Notices*, 40(10):59–76, October 2005.

[44] Bjarne Steensgaard. Points-to analysis in almost linear time. In *Proceedings of the 23rd ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, POPL '96, pages 32–41, New York, NY, USA, January 1996. Association for Computing Machinery.

[45] Tan Tian. Pointer Analysis: Foundations, https://cs.nju.edu.cn/tiantan/software-analysis/PTA-FD.pdf, Accessed June 2024.

[46] John Whaley and Monica S. Lam. An Efficient Inclusion-Based Points-To Analysis for Strictly-Typed Languages. In Manuel V. Hermenegildo and Germán Puebla,

editors, *Static Analysis*, Lecture Notes in Computer Science, pages 180–195, Berlin, Heidelberg, 2002. Springer.

[47] Dacong Yan, Guoqing Xu, and Atanas Rountev. Demand-driven context-sensitive alias analysis for Java. In *Proceedings of the 2011 International Symposium on Software Testing and Analysis*, ISSTA '11, pages 155–165, New York, NY, USA, July 2011. Association for Computing Machinery.

[48] Jesper Öqvist. ExtendJ: extensible Java compiler. In *Conference Companion of the 2nd International Conference on Art, Science, and Engineering of Programming*, pages 234–235, Nice France, April 2018. ACM.

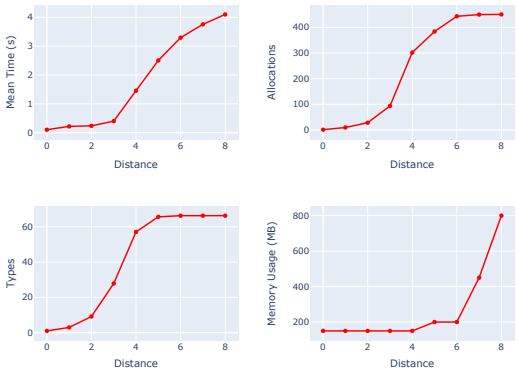
[49] Jesper Öqvist and Görel Hedin. Extending the JastAdd extensible Java compiler to Java 7. In *Proceedings of the 2013 International Conference on Principles and Practices of Programming on the Java Platform: Virtual Machines, Languages, and Tools*, PPPJ '13, pages 147–152, New York, NY, USA, September 2013. Association for Computing Machinery.

Appendices

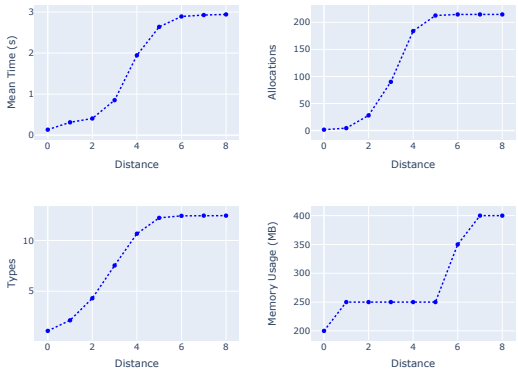
Appendix A
Supplemental Data and Results

Tool	Correct Types	Overapproximated Types	Precision	Recall	Time
Pecka k=0	2	0	1.00	0.00	0.13
Pecka k=1	108	10	0.92	0.17	0.31
Pecka k=2	271	31	0.90	0.43	0.41
Pecka k=3	450	125	0.78	0.71	0.85
Pecka k=4	537	190	0.74	0.85	1.94
Pecka k=5	599	294	0.67	0.94	2.64
Pecka k=∞	603	301	0.67	0.95	2.98
Qilin	631	489	0.56	1.00	8.52

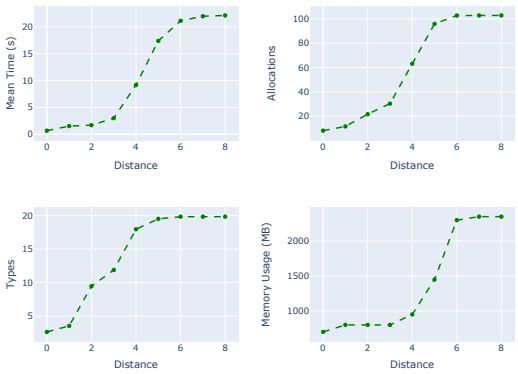
Table A.1: Type recall for the parameter benchmark for ANTLR version 2.7.2. The graph representation of this result is seen in Figure 5.1.



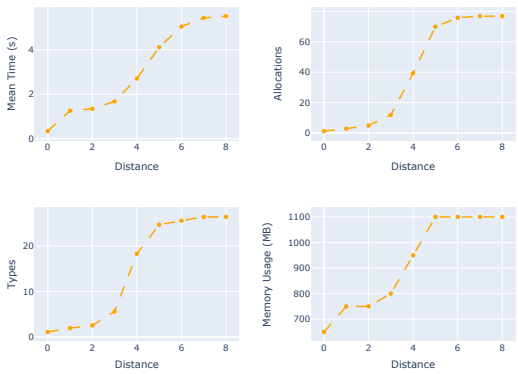
(a) commons-jxpath



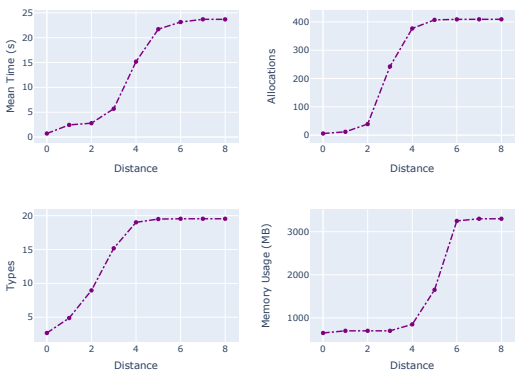
(b) antlr-2.7.2



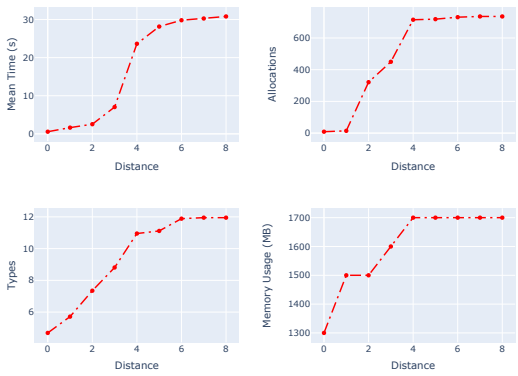
(c) fop-0.95



(d) pmd-4.2.5



(e) jfreechart-1.0.0



(f) joda-time

Figure A.1: The metrics measured for each dataset. This is the same graphs as shown in Figure 5.2-5.5, but plotted separately for each benchmark project.

Test	TP	FP	FN	Precision	Recall
cornerCases/FlowSensitivity1.java	1	1	0	0.50	1.00
cornerCases/FieldSensitivity2.java	1	1	0	0.50	1.00
cornerCases/StrongUpdate1.java	1	1	0	0.50	1.00
cornerCases/ContextSensitivity2.java	2	0	0	1.00	1.00
cornerCases/ContextSensitivity3.java	2	0	0	1.00	1.00
cornerCases/ObjectSensitivity2.java	1	1	0	0.50	1.00
cornerCases/ObjectSensitivity1.java	1	1	0	0.50	1.00
cornerCases/StrongUpdate2.java	1	1	0	0.50	1.00
cornerCases/ContextSensitivity1.java	2	0	0	1.00	1.00
cornerCases/FieldSensitivity1.java	1	1	0	0.50	1.00
basic/ReturnValue2.java	1	0	0	1.00	1.00
basic/Parameter2.java	1	0	0	1.00	1.00
basic/ReturnValue3.java	1	1	0	0.50	1.00
basic/Interprocedural1.java	1	1	0	0.50	1.00
basic/SimpleAlias1.java	1	0	0	1.00	1.00
basic/Loops1.java	1	1	0	0.50	1.00
basic/Branching1.java	2	0	0	1.00	1.00
basic/Loops2.java	2	1	0	0.67	1.00
basic/Recursion1.java	1	1	0	0.50	1.00
basic/Interprocedural2.java	1	1	0	0.50	1.00
basic/Parameter1.java	1	0	0	1.00	1.00
basic/ReturnValue1.java	1	0	0	1.00	1.00
new/FilterArrayCast.java	1	0	0	1.00	1.00
new/FieldList.java	1	0	0	1.00	1.00
new/FilterObject.java	1	0	0	1.00	1.00
new/SimpleMap.java	1	0	0	1.00	1.00
new/EnhancedFor.java	1	1	0	0.50	1.00
new/FilterListCast.java	1	0	0	1.00	1.00
new/ChainedFieldAccesses.java	1	1	0	0.50	1.00
new/FilterCast.java	1	0	0	1.00	1.00
new/FieldMap.java	1	0	0	1.00	1.00
new/FilterRawGenerics.java	1	0	0	1.00	1.00
new/StaticFieldVarAccess.java	1	0	0	1.00	1.00
new/FilterGenericInterface.java	1	0	0	1.00	1.00
new/SimpleList.java	1	0	0	1.00	1.00
new/MethodAccessAfterParExpr.java	1	0	0	1.00	1.00
new/ChainedMixedAccesses.java	1	0	0	1.00	1.00
new/AssignExpression.java	1	1	0	0.50	1.00
new/EnhancedForArray.java	1	1	0	0.50	1.00
collections/List1.java	1	0	0	1.00	1.00
collections/List2.java	1	0	0	1.00	1.00
collections/Array1.java	1	1	0	0.50	1.00
generalJava/Null1.java	1	0	0	1.00	1.00
generalJava/Interface1.java	1	0	0	1.00	1.00
generalJava/Exception2.java	1	0	0	1.00	1.00
generalJava/Exception1.java	1	1	0	0.50	1.00
generalJava/Null2.java	1	0	0	1.00	1.00
generalJava/OuterClass1.java	1	1	0	0.50	1.00
generalJava/StaticVariables1.java	1	0	0	1.00	1.00
generalJava/SuperClasses1.java	1	1	0	0.50	1.00
Total	55	21	0	0.72	1.00

Table A.2: PointerBench test cases and the result for each test. The tests in the directory `new` were added to cover more features in this report.

Appendix B
Source Code

The source code for Pecka is available at `https://github.com/JoArrhen/pecka`.

INSTITUTIONEN FÖR DATAVETENSKAP | LUNDS TEKNISKA HÖGSKOLA | PRESENTERAD 2024-06-13

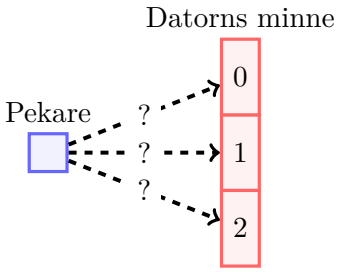
EXAMENSARBETE Pointer Analysis for Interactive Programming Environments
STUDENTER Johan Arrhen, Ruben Wiklund
HANDLEDARE Idriss Riouak (LTH), Anton Risberg Alaküla (LTH)
EXAMINATOR Flavius Gruian (LTH)

Snabb pekaranalys för textredigerare

POPULÄRVETENSKAPLIG SAMMANFATTNING **Johan Arrhen, Ruben Wiklund**

En pekare är en variabel som kan "peka" till platser i datorns minne. En pekaranalys kan ta reda på vilka platser det kan vara. Vi har gjort en pekaranalys som går att köra snabbt nog för att den ska kunna användas medan en programmerare skriver kod.

Datorprogram innehåller variabler som kan läsas och skrivas till. En variabel kan innehålla till exempel ett tal eller ett ord. Datorer håller reda på vilket värde en variabel har genom att spara det i minnet. En speciell typ av variabler kallas för pekare. En pekare är en variabel som "pekar" till en plats i datorns minne. Pekare kan vara användbara när man behandlar data som behövs under lång tid.



Vilka platser i minnet pekar pekaren till? Det kan en pekaranalys ta reda på!

Ibland kan det vara användbart att veta vilka platser i minnet en pekare kan peka till, utan att behöva köra programmet. Den informationen kan man ta reda på genom att utföra en pekaranalys på källkoden. Resultatet från en pekaranalys kan användas till att visa information som underlättar för en programmerare att skriva ett program. Till exempel så skulle datorn kunna varna om ett fel kan inträffa på en viss plats i programmet eller så kan datorn ge förslag på vad programmeraren kan skriva på ett visst ställe. I sådana situationer är det viktigt att det går snabbt att köra analysen. Det skulle vara väldigt irriterande för programmeraren om hen var tvungen att vänta i en hel minut på att få ett förslag i sin textredigerare.

Många av de pekaranalyser som finns är antingen långsamma eller ger ett resultat där det står att en pekare kan peka till många platser som den i själva verket inte kan. I vårt examensarbete testade vi ett nytt sätt att göra pekaranalys på. Vår förhoppning var att det skulle både gå snabbt och ge ett bra resultat.

Vi prövade att göra en analys där man bara analyserar en viss del av koden istället för hela programmet. Hur stor del av programmet som analyseras kan justeras med en parameter k. Om man inte analyserar hela programmet riskerar man att missa viktig information i de delar som man inte analyserar, men genom att välja koden som analyseras på ett särskilt sätt så hoppades vi att det mesta av det som är relevant tas med. Så visades det sig också vara! Det gick att hitta 56% av resultatet på en femtedel av tiden som behövs för att köra analysen på hela programmet.

Eftersom analysen kan missa vissa saker så kan den inte användas till allt. Däremot passar den bra när man vill ha ett snabbt svar, vilket kan vara en fördel exempelvis i en textredigerare.