Feedback-driven Points-to Analysis
Abstract

Points-to analysis is a static program analysis that extracts reference information from a given input program. Its accuracy is limited due to abstractions that any such analysis needs to make. Further, the exact analysis results are unknown, i.e., no so-called Gold Standard exists for points-to analysis. This hinders the assessment of new ideas to points-to analysis, as results can be compared only relative to results obtained by other inaccurate analyses.

In this paper, we present feedback-driven points-to analysis. We suggest performing (any classical) points-to analysis with the points-to results at certain program points guarded by a-priori upper bounds. Such upper bounds can come from other points-to analyses – this is of interest when different approaches are not strictly ordered in terms of accuracy – and from human insight, i.e., manual proofs that certain points-to relations are infeasible for every program run. This gives us a tool at hand to compute very accurate points-to analysis and, ultimately, to manually create a Gold Standard.

1 Introduction

Points-to analysis extracts reference information from a program, e.g., possible targets of a call and possible objects referenced by a field. Such information is essential input to many client applications in optimizing compilers and software engineering tools. Different analysis methods vary in efficiency and accuracy. The efficiency of a method is defined as the memory and execution time required for analyzing a software system of a certain size. A method is completely accurate if it finds all (abstract) references in a program and nothing else. Comparability of analysis methods (or tools) with respect to accuracy is interesting in itself. It is also important that the accuracy of each detection method is approximately on the same level before comparing efficiency.

However, little can be said about the absolute accuracy of points-to analysis [1], as there is no Gold Standard, i.e., no suite of benchmark programs annotated with accurate points-to information. Unfortunately, a Gold Standard for points-to analysis is impossible to compute automatically for realistically large benchmark programs, as the problem is intractable. If determined by human insight, such a task is very time consuming and error prone.

On the other hand, Sim et al. observe that the development of benchmarks in computer science disciplines is often accompanied with technical progress and community building [11]. The lack of such benchmarks, in turn, makes it hard to further develop a field by adopting the successful and avoiding the less promising approaches.

Thus, tool support for the task of building a Gold Standard for points-to analysis is required in order to ease the necessary manual work. Such tools ought to bring automatically calculated over- and under-approximations of the Gold Standard closer together and allow human annotations. The contributions of this paper go into this direction:

(i) We propose feedback-driven points-to analysis, an approach to adding a-priori knowledge, e.g., human expert knowledge, to classical points-to analysis. (ii) We show the benefits of feeding a classic points-to analysis with “human insight”. (iii) As a side-effect of our experiments, we deduce two techniques for static points-to analysis that automatically improve accuracy.

After basic definitions in Section 2, we describe the general approach to feedback-driven points-to analysis in Section 3. In a kickoff case-study, described in Section 4, we then examine a project for sources of inaccuracy, and provide both manual proofs and general optimizations to improve the accuracy of static analysis. In the following
experiments, we apply our findings to additional projects and investigate how automated and manual feedback improve the accuracy for these projects, cf. Section 5. Finally, we discuss related work in Section 6 and conclude our findings in Section 7, where we also outline future work.

2 Background

Any static program analysis needs to abstract from the values which expressions may take during a real application run, as it is impossible to model the exact program state at any time of any possible run of a program. An abstract object \( o \in O \) is an analysis abstraction that represents a set of runtime objects. In this paper, we use an abstraction where all objects created at the same syntactic creation point \( s \) correspond to a unique abstract object \( o_s \). Variables (including, e.g., method arguments) and expressions are associated with analysis values in \( 2^O \), the power set of \( O \). Many points-to analyses (including the one that we use in this paper) work as follows: A program is represented by a program graph where nodes correspond to program points and edges correspond to control and data dependencies among them. Analysis values for each node are computed iteratively by merging values from predecessor nodes and by applying transfer functions representing the abstract program behavior at these nodes. For instance, Allocation nodes create abstract objects; Load and Store operations read and write to a common abstract heap. At control flow confluence points, analysis values are merged, as the points-to-analysis must assume that any control flow path might be possibly taken. The analysis stops once a fixed point is reached, which is guaranteed to happen with the above abstractions and if additionally no strong updates are performed, i.e., no analysis values are ever overwritten but only merged.

Context-Sensitive Points-to Analysis   Precision of points-to analysis is often increased through context sensitivity. In a context-insensitive program analysis, analysis values of different calls may get propagated to the same method and get mixed there. The analysis value is then the merger of all calls targeting that method. A context-sensitive analysis addresses this problem by distinguishing between different calling contexts of a method. It analyzes a method separately for each calling context [9]. A calling context is a static abstraction of all call stacks possibly occurring during any program execution. Approaches differ in their abstraction functions.

We use three different approaches to context sensitivity: CallSite, ThisSens, and ObjSens. All are well-known in the literature, e.g. [4–6]. For the understanding of this paper, it is sufficient to know that experiments show that ObjSens is the most accurate, yet also by far most expensive context-sensitive approach, while CallSite is, in practice, the least accurate, and that, from a theoretical point of view, none of the three techniques is strictly more accurate than the others. We use these three context sensitivities on top of our Points-to SSA based Simulated Execution [5] for the experiments described in this paper.

Assessing Points-to Analysis   The results of points-to analysis are usually very low-level, so we use client analyses for accuracy assessment. For sake of simplicity, we stick to rather simple and easy-to-derive client analyses, as defined in the following. Some of these are still low-level and should thus be capable of distinguishing small differences in analysis accuracy.

For a given program, let \( M \) be the set of its methods and \( O \) the set of all its syntactic creation points. The exact object call graph then consists of nodes and edges \( [o_i, m_j] \rightarrow \)
\[ o_k \times m_j \in O \times M \times O \times M \] where there exists a concrete execution of the program such that (1) \( m_j \) is called on an instance object \([i_v]\), (2) during the execution of \( i_v, m_j \) a call occurs to \( m_j \) on instance object \( i_w \), and (3) \( i_v, i_w \) were created at the syntactic creation points \( o_i \) and \( o_k \), respectively. The exact call graph is the projection of the exact object call graph so that nodes and edges \([o_i, m_j] \rightarrow [o_k, m_l] \) are projected to nodes and edges \( m_j \rightarrow m_l \).

For a given program, let \( F \) be the set of all fields and \( O \) be the set of all its syntactic creation points. The exact abstract heap then consists of all the relations \([o_i, f_s] \leftarrow o_j \in O \times F \times O \) where there exists a concrete execution of the program so that (1) an instance object \( i_w \) is stored into the field \( f_s \) of an instance object \( i_v \), and (2) \( i_v, i_w \) were created at the syntactic creation points \( o_i \) and \( o_j \), respectively.

The exact object call graph, exact call graph, and exact abstract heap can be approximated by both dynamic and static analysis. We then speak of the client analyses object call graph, call graph, and abstract heap, respectively.

The metrics reachable methods (in short, \( N \) for nodes), call graph edges (\( E \)), object call graph nodes (\( ON \)), object call graph edges (\( OE \)), and heap size (\( H \)) are the number of nodes and edges, respectively, in the (object) call graph clients, and the number of relations in the heap client.

3 Feedback-driven Analysis

There are two sources of predefined upper bounds for points-to analysis: complete results obtained from conservative points-to analysis, which allow for automated feedback, and manual proofs, i.e., human insight that certain points-to information is infeasible. We look at the two in Sections 3.1 and 3.2, respectively.

3.1 Automated Feedback

Automated feedback-driven points-to analysis is desirable when different context-sensitive approaches are not strictly ordered in terms of accuracy. This is, for example, the case for all context sensitivities used in this paper. While it is of course possible to define a more precise combined context sensitivity, e.g., a combination of ObjSens and CallSite, this would increase the number of contexts dramatically and can, as preliminary tests suggest, lead to memory problems.

Our approach is now to set upper bounds for the operands to two kinds of operations: Store-operations that write values to memory slots on the heap, and Call-operations.

Each Store-operation \( S, f \) has, next to the (fixed) field \( f \) that is being accessed, two operands that depend on intermediate analysis results: the address \( A \subseteq O \) where to store, and the value \( V \subseteq O \) what to store. Each of them gets assigned an upper bound dependent on previously computed results. These usually stem from a context-sensitive approach, i.e., there is a pair \((A_{ctx}, V_{ctx})\) computed for \( S, f \) for each context \( ctx \) the method containing \( S, f \) is analyzed in.

For example, assume that the previously computed results for \( S, f \) are computed under two different contexts and that \( A_1 = \{o_1\}, V_1 = \{o_2\}, A_2 = \{o_3\}, V_2 = \{o_4\} \). The heap then contains the relations \( (f, o_1) \leftarrow o_2 \) and \( (f, o_3) \leftarrow o_4 \).

When a new analysis with a different context definition updates \( S, f \), we use the set of previously computed pairs as upper bounds. In the new analysis, we can identify the same Store-operation \( S, f \), the same field \( f \in F \), and the abstract objects \( O \) used as addresses or values (provided the new analysis uses the same object abstraction). The different contexts of the previous analysis, however, are not known any longer. But, regardless of the context definition, each new pair \((A_{ctx'}, V_{ctx'})\) analyzed as addresses and values by
the new analysis needs to be consistent with the previously computed upper bound. This means that each address value element \((a,v) \in A_{ctx'} \times V_{ctx'}\) must be contained in at least one pair \((A_{ctx}, V_{ctx})\) of the upper bound, i.e., \((a,v) \in A_{ctx} \times V_{ctx}\) or it is filtered otherwise. Then, the less precise analysis has -- thanks to the upper bounds -- at least as good precision as the previous analysis.

Continuing on our example, if a new analysis updates \(S.f\) with the input pair \(A_{ctx'} = \{o_1,o_3\}, V_{ctx'} = \{o_2,o_4\}\), then, without filtering, the heap is updated with four relations \((f,o_1) \leftarrow o_2, (f,o_1) \leftarrow o_4, (f,o_3) \leftarrow o_2,\) and \((f,o_3) \leftarrow o_4\). This is larger than the previously computed upper bound \(\{(A_1,V_1),(A_2,V_2)\} = \{((o_1,o_2)),((o_3,o_4))\}\) that does not contain the pairs \((o_1,o_4)\) and \((o_3,o_2)\), which can be filtered in the new analysis.

For \textit{Call}-operations, the same could be done for the tuples of this-values \(T\) and corresponding arguments \(A^t\). More precisely, for each context \(ctx\), we could capture and distinguish the tuples \((T_{ctx}, A^1_{ctx}, \ldots, A^n_{ctx})\) as bound but drop the actual context definitions \(ctx\). For a new analysis and a tuple \((T_{ctx'}, A^1_{ctx'}, \ldots, A^n_{ctx'})\) analyzed for the same call operation, each element \((t,a^1,\ldots,a^n) \in T_{ctx} \times A^1_{ctx} \times \ldots \times A^n_{ctx}\) must be contained in at least one tuple of the upper bound or would be filtered otherwise.

For a \textit{Store}-operation, we just perform an abstract heap update for all (non-filtered) tuples. For a \textit{Call}-operation, we would have to analyze the target method for each (non-filtered) tuple. This would just lead to mimicking another, very fine-grained context sensitivity, i.e., a different context-sensitive approach with many distinguished contexts (one for each non-filtered element). As mentioned above, this might not be desirable due to performance issues, and early tests that we performed confirm this. Thus, we merge upper bounds for those tuples and accept the loss of precision.

Assume, for example, \(T_1 = \{o_1\}, A^1_{ctx} = \{o_2\}, T_2 = \{o_3\}, A^2_{ctx} = \{o_4\}\) as the bound computed in one analysis and \(T_{ctx'} = \{o_1,o_3\}, A^1_{ctx'} = \{o_2,o_4\}\) the unfiltered update for a second analysis. In this case, we only check the relaxed bounds \(T_{ctx'} \subseteq T_1 \cup T_2\) and \(A^1_{ctx'} \subseteq A^1_{ctx} \cup A^2_{ctx}\) and accept the less precise update.

Note that this limitation does not have an impact on the set of reachable methods, as invalid target objects are still filtered completely. It indeed seems that, in practice, only the number of edges in object call graphs is affected by this.

### 3.2 Manual Feedback

Another source of upper bounds comes from manual proofs. In addition to the upper bounds from automated feedback, we allow even coarse grained hints which are more accessible to human insight, i.e., (1) which code can be considered dead, (2) what arguments to or return values of methods are infeasible, (3) what call relations can not be taken. Figure 3.1 shows a short example code of where such manual feedback is trivial but which many points-to analysis algorithms cannot deduct: (1) The user is able to prove that method \(f()\) never returns \(-1\), so that the code at line 6 can be considered dead. (2) Human insight can determine that the call \(x.h(y)\) at line 9 must return \(o_2\), so that the use of the local variable \(x\) can be restricted to \(Pt(x) = \{o_2\}\) in the following.

For dead code (1) and impossible return values (2), human insight refines the upper bounds as provided by automated feedback; then both kinds of feedback are treated just alike. All operations that stem from dead code are annotated with empty upper bounds. Impossible return values are removed from the upper bounds of \textit{Call}-operation results. Finally, for a call relation excluded at a \textit{Call}-operation (3), the corresponding target method is just skipped from the analysis.

Manual feedback can be combined with automated feedback, which comes in very handy when performing manual proofs: In a first step, results for the most precise (but
usually also most expensive) available analysis are computed. When iteratively adding and evaluating new manual proofs, a faster analysis is used with those results as upper bounds. This way, faster iteration steps are possible while still having the more precise results as a reference.

Note that the human expert stands for the correctness of his manual feedback. If, after adding some feedback, the dynamic points-to information is no longer a subset of the static points-to information, the feedback is obviously not sound. On the other hand, if this is not the case, then this is still not a proof of the correctness of the manual feedback. However, additional test cases that are provided later may uncover a given manual feedback as invalid.

4 Kickoff case-study

In order to improve the precision of static points-to analysis, we first needed to understand where imprecision originates. We thus ran dynamic and static points-to analysis on a given program, compared the sets of reachable methods, and investigated why these differ. We then attempted to either trigger the execution of methods not covered by dynamic analysis by creating proper input to the program, or prove that the methods are indeed not reachable. As an item to study for this kickoff case-study, we selected the Java 5 to 4 backporting tool that is part of the distribution of recoder (http://recoder.sf.net), a source code analysis and transformation framework for Java. The authors of this paper are well familiar with the implementation of recoder, which is complex enough to imply a challenge to our research.

4.1 Patterns of Imprecision

Bad coverage for the dynamic analysis is an obvious source of imprecision. In the beginning, we fed the transformation tool with only recoder itself, which triggered the execution of 3298 methods. Over the course of the study, we increased the dynamic coverage to 3876 methods, which we achieved partly by transforming more complete projects, and partly by creating specific code that triggers certain aspects (e.g., error-handling) in the transformation tool.
The initial static analysis identified 4715 methods as reachable. We identified four major patterns that contribute to its imprecision, which we discuss in the following.

4.1.1 Dead Code

If an application is built on top of a more general framework, unused parts of the framework are often still deployed with the application, be it for convenience or inaccessibility of the source code of the framework, or because instantiation options are delayed until runtime by means of dynamic binding or other means of configuring the framework at runtime. The latter is usually not recognizable by points-to analysis. Concretely, the tool which we analyze does not change many configuration options provided by the recoder framework, so that we can mark code as dead if it is triggered only when the default values of these options are changed.

4.1.2 Polymorphic Calls in Branching Conditions

When a branching condition compares the result of a polymorphic call with a given constant value, some possible target methods of the polymorphic call may provably never fulfill the given condition. This can be the case if such a method returns a constant value (but other related methods do not). Then, the value of the receiver object can be restricted in the then-block of the condition: abstract objects of the types that never fulfill the condition can be filtered. We found examples for this in some central places of recoder: Conditions of branch statements look like in the following source code excerpt, where ClassType is an interface common to many concrete classes, and getA() returns a list.

```java
recoder.abstraction.ClassType x = ...;
if (x.getA().size() > 0) { use of x ... }
```

Some of the classes that implement the interface ClassType always return an empty list, so that instances of those classes can never reach the body of the if-statement. We manually told our analysis to remove instances of such classes from method arguments, field accesses etc. within the then-block. For future work, it should be possible to perform this optimization automatically by inserting special filter-operations (cf. [2]) into the analysis’ program representation.

4.1.3 Collections and Maps

Array objects are handled like other abstract objects in our points-to analysis implementation, as they are containers that can be passed around on the heap. This means that values stored in collection classes – which are often backed by arrays – are stored in the same abstract array objects. Thus, values stored in distinct collection instances (distinct in the sense of distinct abstract objects) are merged through these array objects, which poses a source of imprecision.

In the case of Java’s collections framework, we know that the backing arrays are never passed to the outside (implementations of toArray() create new array objects). Thus, we implemented a transformation on the intermediate representation where all classes that implement one of the interfaces java.util.Collection and java.util.Map are replaced by a specialized implementation: A regular (non-array) field is used for storage; this is possible because values are never overwritten (no strong updates are performed). This takes care of mixing values from different collections through, e.g., add() and get() methods. Additionally, methods like iterator() and elements() (in the case of java.util.Vector) are replaced by methods that simply return “this”, and collection/map classes implement the
```java
class ArrayList extends AbstractList
    implements Iterator {
    private Object @elems; // artificial field
    Object get(int i) { return @elems; }
    void add(Object o) { @elems = o; }
    // ...
    Iterator iterator() { return this; }
    // java.util.Iterator methods:
    boolean hasNext() { return true; }
    Object next() { return @elems; }
}
```

Figure 4.1: Layout of ArrayList replacement

interfaces Iterator, Map.Entry, and Enumeration, where required. Figure 4.1 illustrates how this looks like for the class java.util.ArrayList.1

4.1.4 Shared Caches

A program may have general, untyped general-purpose object caches that are shared by different parts of the program. If it is guaranteed that each part of the program retrieves only those objects that it puts into a cache itself, the caches can be considered being logically partitioned. However, points-to analysis cannot, in general, recognize this, and thus assumes that all objects stored in the cache may be read by any part of the program, which introduces a source of imprecision.

Recoder uses a single HashMap instance to cache mappings from type-, variable- etc. references to the resolved program elements. Through the cache, resolving the different kinds of references gets mixed in the points-to analysis. We solved this problem by removing the caches from the analysis: Since abstract objects retrieved from the cache (and instantly returned) by the methods in question are computed anyway within the very same methods, this does not pose a threat to the analysis being conservative.

4.2 Summary

During the course of the case-study, we increased the dynamic coverage in terms of methods by 17.5% and decreased the number of statically reachable methods to 4594 (-2.6%). It looks like focusing on increasing the dynamic coverage is more worthwhile than improving the static analysis. However, increasing the dynamic coverage has become more and more difficult over time, so that upcoming work will have to focus on improving the static analysis.

5 Evaluation

We performed experiments with six different benchmark programs (listed in Table 5.1) for which our points-to analysis is conservative, i.e., programs that do not make use of dynamic class loading and reflection. We analyzed each program with all context-sensitive approaches described in Section 2, combined the results by computing the intersection

---

1Note that this of course means that points-to information for the collection framework is not conservative any more. For our purposes this is acceptable, as we are interested only in the reachable methods of the benchmark program (recoder).
Table 5.1: Benchmark Programs

<table>
<thead>
<tr>
<th>program</th>
<th>autom. feedback (4.1)</th>
<th>dead code (4.1.1)</th>
<th>pol. calls (4.1.2)</th>
<th>collections and maps (4.1.3)</th>
<th>caches (4.1.4)</th>
<th>reachable methods initial (dynamic/static)</th>
<th>reachable methods final (dynamic/static)</th>
</tr>
</thead>
<tbody>
<tr>
<td>bloat 1.0</td>
<td>x</td>
<td>x/a</td>
<td></td>
<td></td>
<td></td>
<td>1489/2446</td>
<td>1525/2446</td>
</tr>
<tr>
<td>javac 1.3.1</td>
<td>x</td>
<td></td>
<td>x/a</td>
<td></td>
<td></td>
<td>1184/1482</td>
<td>1264/1416</td>
</tr>
<tr>
<td>javacc 4.2</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>770/1033</td>
<td>871/1030</td>
</tr>
<tr>
<td>jlayer 1.0.1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>161/241</td>
<td>224</td>
</tr>
<tr>
<td>recoder 0.94c</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>3298/4715</td>
<td>3876/4594</td>
</tr>
<tr>
<td>sablecc 3.2</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>1314/1796</td>
<td>1317/1796</td>
</tr>
</tbody>
</table>

Note that, while many programs come from the compiler construction area, their respective developers have made fundamentally different implementation decisions, so that the programs behave quite differently under points-to analysis.
5.1 Results

Figures 5.1(a) to 5.1(e) show lower bounds of the precision of the static analyses for the five metrics $N$, $E$, $ON$, $OE$, and $H$, respectively, for the four remaining benchmark programs. The actual precision is unknown as it requires the Gold Standard, but a lower bound can be computed as $P^- = \frac{|A_{opt}|}{|A_{cons}|}$ with $A_{opt}$ being the result obtained by dynamic analysis, and $A_{cons}$ being the result obtained by conservative static analysis [1]. For each program, the initial $P^-$, $P^-$ with only increased dynamic coverage, and the final $P^-$ are shown. In the following, we discuss the results of each project in detail.

javac is the only benchmark besides recoder that is affected by automated feedback. This shows that automated feedback is not only a theoretical construct but also might have benefit in practice. However, the best improvements we observed was for the metric OE, which improved by 0.7%. A large part of the improvement through our manual feedback is because javac contains code for serializing its AST, e.g., for writing source code back to the file system. This can be triggered by setting a certain command line flag – however, the main() method does not allow the flag to be set. Thus, we can ensure that this is dead code. Our collections and maps replacement has no effect as javac comes with its own implementations for collections. While the precision of the metrics $N$ (0.89) and $E$ (0.69) are rather good, the more fine-grained metrics $OE$ (0.09) and $H$ (0.16) improved only on a low level.

For javacc, the collections and maps replacement shows improvements; for instance, three fewer methods are identified as reachable. The other metrics improve even stronger than this. We could not find any point where we could apply manual feedback; this is because most methods that are identified as reachable by static analysis (but not covered by dynamic test cases) are part of javacc’s template parser, for which we lack expert knowledge.

For jlayer, $P^-$ for metric N was initially 0.67, but we succeeded in computing the exact set of reachable methods by increasing dynamic coverage and identifying dead code. This also had good impact on the other metrics. For instance, we now have rather good $P^-$ even for the fine-grained metrics, namely 0.85 for $H$ and 0.91 for $OE$. For $H$, the result set is rather small anyway: only nine heap relations need to be further investigated for determining the exact result. This shows that at least for small projects it is feasible to
create the Gold Standard even for those client analyses.

We have discussed recoder in detail in the previous section, where we looked at the number of reachable methods. The other metrics improve at approximately the same pace; like for javac, the overall $P^-\text{ is rather low }(<0.2)$ for the more fine-grained metrics ON, OE, and H, despite a rather good precision of the metrics N (0.84) and ON (0.65). Automated feedback also improves the precision of all metrics slightly; most noteworthy is that one more method is identified as not reachable after some iterations.

5.2 Summary

Manual feedback given to points-to analysis improves its precision with the help of human insight. Such manual feedback allowed us to compute the exact set of reachable methods for the benchmark program jlayer. Another important task is to increase the dynamic coverage: For all the considered projects, we could add more methods to the set of dynamically reachable methods than we could remove methods from the set of statically reachable methods. Both tasks, thus, need to go hand-in-hand in order to create a Gold Standard.

The collections and maps replacement shows good positive effects for two benchmark programs, while it is – in its current state – not applicable to further two. The remaining two benchmark programs do not use the collections framework from the Java runtime library at all or not extensively and thus do not benefit from it.

Automated feedback shows positive effects only for two of the six benchmark programs in our evaluation. Its use thus seems to rather lie in speeding up iterations of manual feedback, as discussed in Section 3.2.

6 Related Work

In earlier work, we combined different analyses by intersecting their result sets [1]; the automated feedback-driven points-to analysis can be seen as a direct extension, as we combine different analyses while running an analysis.

Demand-driven and refinement-based Points-to Analysis (e.g., [12, 13]) are techniques where a baseline points-to analysis is performed, and on request the precision is increased for a certain variable’s points-to set. This allows to have very precise points-to analysis where required, while still being highly scalable. Unlike our approach, clients (in our case, clients are limited to human experts) specify where to increase precision, not how.

Rountev et al. discuss the imprecision of static analysis [7] in general. They also define upper and lower bounds of the exact solutions and propose, for the better understanding of imprecision of static analysis, to manually investigate the results of static analysis compared to dynamic analysis, i.e., to create a Gold Standard. The same authors make an empirical study of a Java program where they perform static, dynamic, and manual investigation of feasible call chains [8].

Lhotář presented a tool for comparing call graphs [3]. Its focus is on identifying root causes for differences, e.g., between dynamic and static analysis. Such a tool can be very helpful in the future for creating a Gold Standard for more fine-grained client analyses than reachable methods.

7 Conclusion and Future Work

In this paper, we have presented feedback-driven points-to analysis, where operations in classical points-to analysis are guarded with upper bounds. Upper bounds are provided
either by other points-to analysis runs, or manual feedback provided by human insight. In a kickoff case-study, we examined a benchmark program for sources of imprecision, and provided ideas on how to overcome those imprecisions, either manually or automatically. The ideas could be applied to other projects as well. For example, we computed the exact set of reachable methods for project jlayer.

As a side-effect, we deduced two fully automated optimizations to points-to analysis: automated feedback and collections and maps replacement. While the former shows only little precision improvements for two out of six analyzed projects, it still has value as it speeds up iterative steps of providing manual feedback. The latter shows considerable improvements for two of the benchmark programs, and is thus worth being examined more thoroughly in future work.

In Section 4.1.2, we have discussed a source of imprecision due to polymorphic calls in branching conditions. So far, we have overcome this source of imprecision manually, but it should be possible to generalize it to an automated optimization in the future.

A main focus of our future work will be in developing and evaluating specialized tools that aid in improving the dynamic coverage as well as creating manual feedback. Dynamic coverage can be improved by dynamic test case generation (also called concolic testing). The idea is to execute a program with a given input, and then automatically vary the input so that different control flow paths are taken. This approach is successfully applied in the area of bug-finding, e.g., [10] and might be adaptable to our needs as well.

There are two general ideas for tools that help in creating manual feedback that we intend to follow: First, tools that guide us in finding worthwhile places to look at, e.g., like the tool for call graph comparisons provided by Lhoták, but generalized to information closer to low-level points-to information. Such tools can prevent us from wasting time on finding proofs that only have minimal impact. Second, tools that ease performing manual proofs. For this, the applicability of theorem provers for our needs should be investigated.

References


