Superimposing a points-to analysis on the AST

Anton Skorup
D20, Lund University, Sweden
an7247sk-s@student.lu.se

Abstract

This paper presents the implementation of a combined points-to analysis and rapid type analysis (RTA) for the Java programming language, using the ExtendJ Java compiler and JastAdd. The goal is to superimpose the points-to analysis on the abstract syntax tree (AST) while interleaving it with the RTA. The analysis aims to determine which memory locations a pointer can refer to, which can be used to identify potential null pointer references. The implementation leverages JastAdd's reference attribute grammar to automate the interleaving of analyses, reducing manual work. The proof-of-concept demonstrates the feasibility of this approach, although it currently faces limitations with array dereferences and generic types. Future work will focus on refining the RTA and evaluating the scalability of the analysis on larger code bases.

Keywords: Points-to-analysis, Rapid type analysis, JastAdd, Reference Attribute Grammars

1 Introduction

A frequent problem in creating computer programs is null pointer dereferencing, which could lead to unexpected crashes. A points-to analysis, also called a pointer analysis, could help in finding potential null-pointer dereferences. A points-to analysis is static program analysis that aims to determine which memory locations a pointer may be referencing [10]. The result from the points-to analysis can help developers find if a pointer dereference may result in a null pointer dereference.

Lhoták and Hendren [6] and Milanova, Rountev and Ryder [9] proposed that the points-to analysis constructed based on the result of a rapid type analysis (RTA). The combination of a points-to analysis and RTA for the Java programming language resulted in a sound over-approximation with high precision [6, 9]. The result of the RTA, which is information about what Java classes different pointers could be an instance of, is used to determine which methods and fields are being referenced. This especially useful for virtual, or abstract, methods where the RTA information could be used to determine which virtual methods may be invoked [6, 9].

Unlike type analysis, points-to analysis combined with RTA is not required to generate efficient machine code or bytecode. A compiler and a static checker serve different purposes. The sole purpose of a compiler is to generate the most efficient machine code or bytecode, while the purpose of a static checker is to find potential issues in the source code. For example, the Java compiler (javac) performs the necessary static analyses to generate bytecode, but it does not include the more advanced analyses like points-to analysis combined with rapid type analysis. Consequently, programmers only using javac miss out on this analysis and the help in finding possible null pointer dereferences.

In this paper we aim to implement a combination of a points-to analysis and a RTA for the Java programming language. The goal is to superimpose the points-to analysis on the abstract syntax tree (AST), while interleaving the RTA with the points-to analysis. The analysis will be built on top of the ExtendJ Java compiler. ExtendJ is an extensible Java compiler implemented in JastAdd [11]. The implementation that is the basis of this paper is in a proof-of-concept state. Currently the result from the rapid type analysis are inconsistent and as a fallback the simpler call graph analysis class hierarchy analysis [13] was used.

Interleaving multiple analyses traditionally requires painful manual work, by implementing the analysis declaratively the need to manually manage worklist communication can be automated [2]. To the best of our knowledge ExtendJ is the only Java compiler implemented in a suitable declarative framework, namely JastAdd. JastAdd's reference attribute grammars will be utilized both to interleave the points-to analysis with the RTA and to superimpose the analysis on the AST. JastAdd's reference attribute grammars are defined declaratively [3, 8], which means that both the RTA and the points-to analysis can be implemented declaratively. The declarative implementation allows for the analyses to be mutually dependent on each other. JastAdd allows for reference attributes to be computed using a fixed-point iteration [8], which is used to interleave both analyses. JastAdd allows us to superimpose the analysis on the AST, since all equations, which are used to find the values for different attributes, are defined on AST nodes.

Our approach differs from Lhoták et al. [6] and Milanova et al. [9], by interleaving the points-to analysis with the RTA. Both Lhoták et al. [6] and Milanova et al. [9] run both analyses in separate phases making the call-graph analysis dependent on the points-to analysis instead of making them mutually dependent on each other. The second distinction that is made is that in this paper the analysis is superimposed on the AST.

Course paper, EDAN70, Lund University, Sweden
January 30, 2025.

2 Background

The points-to analysis used in this paper is Steensgaard's algorithm, see Section 2.1, which will be implemented using reference attribute grammars in JastAdd.

2.1 Steensgaard's algorithm

One approach to implementing a points-to analysis is by use of Steensgaard's algorithm [10]. The result from Steensgaard's algorithm is a collection of points-to sets. A points-to set contains the memory locations that each pointer may reference. This points-to set is built using a unification algorithm [10, 15]. The points-to-set is constructed by finding all assignments to the pointer, which are then unified.

In the example found in Listing 1 the points-to set of `a` contains two memory locations and null. This means that `a` may reference either `m1` or `m2` or be a null pointer. During unification, when the points-to sets are built, `a`'s points-to set will be unified each of its assignments' points-to sets. When `a` is declared it is assigned the value of memory location `m1`. In the first step of unification `a`'s points-to set is a singleton set containing only the value `m1`. Steensgaard's analysis is flow insensitive [15], meaning that the order of execution does not matter and there is no distinction between different points in the program execution [10]. The insensitive nature of the analysis means that both branches of the if-statement will be considered during unification. Therefore, `a` will be unified with both `m2`'s and `null`'s points-to sets.

Listing 1. In this simple code example the points-to-set of `a` would be $pts(a) = \{m1, m2, null\}$

```
Object a = new(); // m1
if (...) { a = new(); /* m2 */ }
else { a = null; }
```

2.2 JastAdd and reference attribute grammars

Reference attribute grammars (RAGs) is an extension of attribute grammars (AGs) [8] that aims to improve on the limitations of AGs [5]. With RAGs declarative equations are defined on AST nodes. These nodes can reference any other AST node and be defined circularly [8]. A circular attribute is an attribute that is defined recursively and then computed using fixed-point iteration.

A collection attribute is an attribute in the reference attribute grammar that aggregates values from an unspecified number of nodes in the AST [7]. In JastAdd the attributes are defined on the AST node that are collecting the different values from other nodes. The other AST nodes can contribute values to the different collection attributes. When a node contributes a value, it can specify both when and for which instance of the AST node with the collection attribute, it should contribute the value to [7].

JastAdd is a meta-compilation system that allows the user to implement a reference attribute grammar, while also allowing the user to implement imperative code on AST nodes [3]. JastAdd's declarative part is implemented using RAGs, while also allowing the user to write plain old Java code for the different AST nodes [3]. JastAdd allows the user to call the imperative parts of the code form the declarative parts of the code and the other way around as well, meaning that JastAdd is a mix between a declarative and an imperative language [3]. This mix of programming paradigms is utilized when implementing the analysis proposed in this paper. See Section 3 for more details.

An externally visible side effect is a side effect that can be seen outside of a method, like printing to standard out or updating an object (that is visible outside of the method). Creating an object within a method and updating it is considered a side effect, but not an externally visible side effect, since it cannot be seen outside of the method. JastAdd does not allow externally visible side-effects in an attribute equation [3].

3 Implementation considerations

In this paper we want to interleave a points-to analysis with an RTA and superimpose the combined analysis on the AST. An interprocedural analysis is an analysis that combine results between different Java methods [10]. We want our combination of the points-to analysis to be interprocedural.

3.1 Memory locations

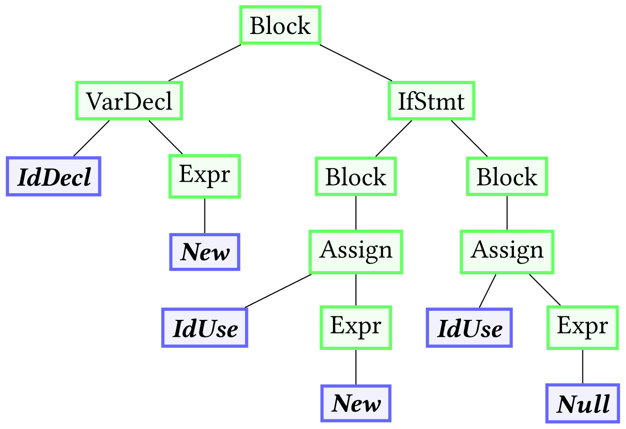

Figure 1. A simplified AST of the code in Listing 1. The green nodes are ordinary AST nodes, while ***blue*** nodes are AST nodes that are memory locations.

To superimpose the points-to analysis on the AST we need to distinguish between regular AST nodes, which are not memory locations, and the AST nodes that are memory locations. The memory location nodes are AST nodes that have been extended with special properties required to run Steensgaard's algorithm (see Section 3.2). In Figure 1, which is a simplified AST based on Listing 1, we see the distinction between regular non-memory location AST nodes and memory location AST nodes. The green nodes (in Figure 1) are regular AST nodes, while the ***blue*** nodes are AST nodes that have been extended to memory locations.

To collect points-to information, it is essential distinguish between memory location nodes and regular AST nodes. Examples of memory locations in programs include string literals, object instantiations and lambda expressions. The AST nodes corresponding to memory locations in the program need to be extended to memory location nodes. In programs pointers are used to reference different memory locations in the program. Although pointers are not memory locations in a program the corresponding AST nodes of pointers are memory location nodes. Making pointers memory location nodes allows for easy unification between memory locations, in the program, and the pointers.

3.2 Unification

Steensgaard's algorithm is a unification-based salgorithm [10, 15] and a union-find data structure have near constant time unification. When superimposing the analysis on the AST we want to mimic the behaviour of the union-find data structure to get fast unification. This is achieved using the special properties of the memory location nodes, which are `union` and `find`.

The purpose of the special property `union` is to unify two memory locations' points-to sets with each other, while the purpose of `find` is to collect the points-to set of a memory location. JastAdd does not allow for externally visible side effects [3], which is a problem for a classic implementation of the union-find data structure. To prevent this an immutable structure is required, which stores a grouping of memory locations. When `union` unifies two memory locations a new extended grouping is created and `find` returns the biggest grouping for that memory location.

3.3 Interprocedural analysis

To make the analysis interprocedural the formal arguments of the method declaration were unified with the actual arguments of the method call. In the Java programming language both method overloading and virtual methods exist. Method overloading and virtual methods means that there could be more than one method declaration for a single method. Method overloading can easily be determined at compile time, but what virtual method is being invoked is non-trivial to determine at compile time.

To figure out which method declarations may be invoked during compile time first a set of all possible method declarations for a given method call was constructed. The possible method declarations were than pruned using the points-to information from Steensgaard's algorithm. The pruned information was then used to make Steensgaard's algorithm more precise, which could then be used again to further prune the possible method declarations. The possible methods and the result from Steensgaard's algorithm continue to improve each other until a fixed point is achieved.

3.4 Null

Two approaches were considered in handling null, which were having a global null and having multiple local nulls. The first approach with a global null would mean that instead of a memory location null could be handled as a flag, which would lead to efficient computation. The drawback of the global null is that is that it would result in all pointers that may be null would be unified with each other resulting in precise results. The second alternative with multiple local nulls does not have that drawback, since then each null would be handled as a different memory location. It is more computationally heavy, but a lot more precise and therefore the second approach was used.

4 Implementation of the analysis

The implementation of the concepts explained in Section 3 consist of two main parts. These were to interleave the RTA with the points-to analysis and superimpose the combined analysis on the AST. Both parts had one common task and that was the unification required to implement Steensgaard's analysis. The analysis could not be superimposed on the AST if there were no memory location nodes with their special properties (see Section 3.2) and the points-to analysis and the RTA were interleaved during unification (see Section 3.3). The interleaving of both analyses came for free due to the way JastAdd handles the computation of circular attributes [12].

To superimpose the analysis on the AST a new interface was introduced that represented a memory location node. This interface had two methods `union` and `find`, which corresponds to the special properties of the memory location nodes (see Section 3.2). The `MemoryLocation`-interface was implemented by all AST node classes, generated by JastAdd, that represents a memory location (see Section 3.1).

4.1 Collection attributes

In contrast to a more traditional program analysis, JastAdd evaluates everything on demand. To ensure that the on-demand evaluation find a correct result each memory location needs to know what other memory locations it depends on. This was solved with two collection attributes *dataflowTargets* and *dataflowSources*. The collection attribute *dataflowTargets* contains all memory locations that a memory location assigned its value to, while the second attribute, *dataflowSources*, contained all memory locations that a memory location was assigned to.

These collection attributes, *dataflowSources* and *dataflowTargets*, were used in the implementation of the special property `find` on memory location nodes (see Section 3.1). `find` is defined as the union of a memory location and all the memory locations it could possible represent. In the implementation `find` was defined as the union between a memory location and all its *dataflowSources* and *dataflow-Targets*.

4.2 Meta memory locations

JastAdd does not allow externally visible side-effects in an attribute equation [3]. To handle unification without using externally visible side-effects, we introduce a new memory location, the meta memory location. This meta memory location is an immutable grouping that is extended by creating a new larger instance of the immutable grouping and thereby bypassing the externally visible side-effects. To unify two meta memory locations the first grouping is extended with the content of the second meta memory location. When unifying a non-meta memory location and a meta memory location the grouping of the meta memory location is extended with the non-meta memory location. The union of two non-meta memory locations resulted in a meta memory location containing both non-meta memory locations.

4.3 Circular attributes

As mentioned in Section 3.3 a fixed-point computation was used to interleave the rapid type analysis with the points-to analysis. To implement a fixed-point computation in JastAdd, circular attributes were used. The result of RTA was based on the result of find, which itself used the result from RTA, i.e., find and RTA were mutually dependent on each other. This meant that both RTA and find needed to be circular attributes.

To ensure that the fixed-point iteration terminates the memory locations must be compared with each other. For the meta-memory location, the content of the grouping was compared meaning that two groupings are equal if and only if they contain the same elements. For the non-meta memory locations an index was used. All AST nodes were given an index start with zero for the top AST node and then each of its children got an index that was one plus its sibling and so on. These indices were then used in comparing the AST nodes between different iterations.

5 Evaluation

The original plan for validating the correctness of the analysis in this paper was to compare the result of the implemented analysis with the result from SootUp [4]. Due to the unfinished state of SootUp's documentation it was decided to fallback to another evaluation approach.

The evaluation approach was to manually generate points-to information, by hand, for a Java implementation of Jonas Skeppsted's and Christian Söderberg's version of integer linear programming (intopt)[1] [14] and use that as a ground truth. Then the ground truth was compared to the printed output of the proof-of-concept version of the analysis.

[1]Link to implementation on BitBucket.

5.1 Result & Limitations

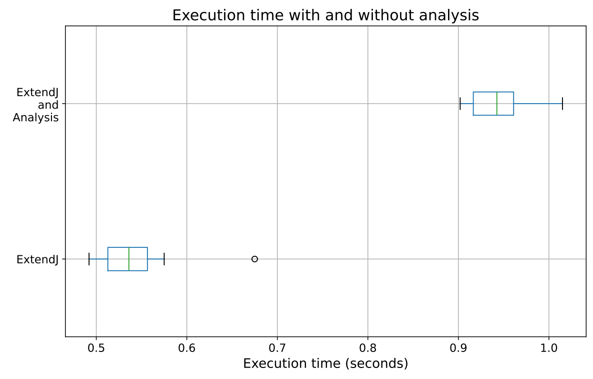

Figure 2. The execution time of ExtendJ compiling intopt with and without running the analysis. The bottom boxplot shows the measurements when running ExtendJ by itself and the top boxplot shows the measurements when running ExtendJ and the proof-of-concept analysis.

The intopt program consisted of 579 lines of Java code, which corresponded to 4697 AST nodes when compiled with the ExtendJ Java compiler. As can be seen in Figure 2 the execution time of just ExtendJ is around 0.55 seconds and the execution time of ExtendJ and the proof-of-concept analysis is around 0.95 seconds. The delta between the two figures are approximately 0.4 seconds, which approximates the execution time of the proof-of-concept analysis for 4697 AST nodes.

The points-to information produced by the proof-of-concept analysis were a sound over-approximation of the ground truth, with some limitations. The observed limitations were array dereferences and generics. The proof-of-concept analysis does not take array dereferences into account. This means that if something is stored inside of an array the analysis would completely disregard of that memory location. The points-to information is then only a sound over-approximation of the ground truth, with intopt as input, if array dereferences are disregarded. If generics are used in the input source code, then the analysis crashes and no output is produced, meaning that any program using generics cannot be analysed.

The premise of this paper is that humans are bad at generating points-to information, which means that the evaluation using human generated points-to information is a limitation. The first version of the points-to information used as ground truth did not take array dereferences into account and was nearly missed. It cannot be ruled out that the ground truth is incorrect, but we argue that the ground truth is good enough to prove that the proof-of-concept is feasible.

The sample size of the evaluation is small consisting of only one small program is a limitation. This makes it difficult to draw any conclusions on how the proof-of-concept analysis scales on larger inputs, meaning that the performance, both in execution time and precision, on larger programs is unknown.

6 Related work

Lhoták and Hendren utilized a call-graph to improve their precision of their points-to-analysis. In many ways their approach is similar to what is discussed in this paper. The big difference is that our analysis is imposed on the abstract syntax tree, which theirs is not. The other thing distinguishing the two papers is that their points-to analysis depended on their call-graph analysis, while our analyses are mutually dependent on each other.

Lhoták et al. [6] and Milanova et al. [9] implemented the same basic idea of combining a points-to analysis with a rapid type analysis. Both papers are quite old and uses older Java versions. In this paper we differentiate also from these papers by using a more modern version Java (version 11) and see if the same idea is still feasible.

Bacon and Sweeney [1] showed that RTA was better at determining which virtual method is being called than class hierarchy analysis. The distinguishing factor here is that Bacon and Sweeney only did the RTA, while we combined it with a points to analysis.

7 Further work

For now, the RTA is not working as expected (as discussed in Section 5) and a direction to further elaborate on this work is to fully implement it. This would enable a more grounded conclusion to be drawn on the feasibility of combining a points-to analysis with a rapid type analysis.

The evaluation was limited (see Section 5.1) and it is currently unknown how the proof-of-concept analysis scales on larger inputs. This could show how well the analysis scales with code size and if it is actually feasible to use in real world development.

8 Conclusion

The proof-of-concept analysis proves that it is possible to superimpose a combination of a call-graph analysis with a points-to analysis on the AST, meaning that the idea is feasible. The scalability of the purposed analysis is still unknown and requires further work to determine.

The proof-of-concept analysis consisting of class hierarchy analysis as the call-graph analysis combined with a points-to analysis produced points-to information that were a sound over-approximation of the ground truth it was evaluated against. We argue that if a rapid type analysis were used as the call-graph analysis instead of a class hierarchy analysis then the generated points-to information would be at least as precise as the one generated by the proof-of-concept analysis. We argue this, since the RTA is at least as precis as the class hierarchy analysis and often more precise if all virtual methods are not utilized in the program for that memory location.

Acknowledgments

A big thank you to Christoph Reichenbach for great supervision of the project and endless help and valuable feedback throughout the project. Thanks to Erik Präntare for help with SootUp.

References

[1] David F. Bacon and Peter F. Sweeney. 1996. 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)*. Association for Computing Machinery, New York, NY, USA, 324–341. https://doi.org/10.1145/236337.236371

[2] Alexandru Dura, Christoph Reichenbach, and Emma Söderberg. 2021. JavaDL: automatically incrementalizing Java bug pattern detection. *Proc. ACM Program. Lang.* 5, OOPSLA, Article 165 (Oct. 2021), 31 pages. https://doi.org/10.1145/3485542

[3] Görel Hedin and Eva Magnusson. 2003. JastAdd—an aspect-oriented compiler construction system. *Science of Computer Programming* 47, 1 (2003), 37–58. https://doi.org/10.1016/S0167-6423(02)00109-0 Special Issue on Language Descriptions, Tools and Applications (L DTA'01).

[4] Kadiray Karakaya, Stefan Schott, Jonas Klauke, Eric Bodden, Markus Schmidt, Linghui Luo, and Dongjie He. 2024. SootUp: A Redesign of the Soot Static Analysis Framework. In *Tools and Algorithms for the Construction and Analysis of Systems*, Bernd Finkbeiner and Laura Kovács (Eds.). Springer Nature Switzerland, Cham, 229–247.

[5] Donald E. Knuth. 1968. Semantics of Context-Free Languages. *Mathematical Systems Theory* 2, 2 (1968), 127–145. https://doi.org/10.1007/BF01692511

[6] Ondřej Lhoták and Laurie J. Hendren. 2006. Context-Sensitive Points-to Analysis: Is It Worth It?. In *International Conference on Compiler Construction*. https://api.semanticscholar.org/CorpusID:1165880

[7] Eva Magnusson, Torbjörn Ekman, and Görel Hedin. 2009. Demand-driven evaluation of collection attributes. *Automated Software Engineering* 16, 2 (01 Jun 2009), 291–322. https://doi.org/10.1007/s10515-009-0046-z

[8] Eva Magnusson and Görel Hedin. 2007. Circular reference attributed grammars — their evaluation and applications. *Sci. Comput. Program.* 68, 1 (Aug. 2007), 21–37. https://doi.org/10.1016/j.scico.2005.06.005

[9] Ana Milanova, Atanas Rountev, and Barbara G. Ryder. 2005. Parameterized object sensitivity for points-to analysis for Java. *ACM Trans. Softw. Eng. Methodol.* 14, 1 (Jan. 2005), 1–41. https://doi.org/10.1145/1044834.1044835

[10] Anders Møller and Michael I. Schwartzbach. 2024. Static Program Analysis. https://cs.au.dk/~amoeller/spa/. (June 2024). Department of Computer Science, Aarhus University, `http://cs.au.dk/~amoeller/spa/`.

[11] Jesper Öqvist. 2018. ExtendJ: extensible Java compiler. In *Companion Proceedings of the 2nd International Conference on the Art, Science, and Engineering of Programming (Programming '18)*. Association for Computing Machinery, New York, NY, USA, 234–235. https://doi.org/10.1145/3191697.3213798

[12] Idriss Riouak, Niklas Fors, Jesper Öqvist, Görel Hedin, and Christoph Reichenbach. 2024. Efficient Demand Evaluation of Fixed-Point Attributes using Static Analysis. In *Proceedings of the 17th ACM SIGPLAN International Conference on Software Language Engineering (SLE '24)*.

Association for Computing Machinery, New York, NY, USA, 56–69. https://doi.org/10.1145/3687997.3695644

[13] Jason Sawin and Atanas Rountev. 2011. Assumption Hierarchy for a CHA Call Graph Construction Algorithm. In *2011 IEEE 11th International Working Conference on Source Code Analysis and Manipulation*. 35–44. https://doi.org/10.1109/SCAM.2011.20

[14] Jonas Skeppstedt and Christian Söderberg. 2020. *Writing Efficent C Code - a thorough introduction* (third ed.). Skeppberg AB.

[15] Bjarne Steensgaard. 1996. Points-to analysis in almost linear time. In *Proceedings of the 23rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '96)*. Association for Computing Machinery, New York, NY, USA, 32–41. https://doi.org/10.1145/237721.237727