Using Dependence Graphs for Slicing Functional Programs

Extended Abstract

Vadim Zaytsev
Universiteit van Amsterdam, The Netherlands
vadim@grammarware.net

One of the popular ways to perform automated analysis of programs is by manipulating slices — reduced executable programs derived from the originals by removing some steps in such a way that they replicate parts of the original behaviour [18]. Such slices are useful for debugging and comprehension, refactoring and restructuring, reverse engineering and maintenance, model checking, as well as other tasks related to similar problems. At the core of this family of methods are the slicing strategies (backward slicing [18], forward slicing [8], as well as other variations [19], including higher-order lazy functional slicing [13]) and the base data structures. In rare cases analysis, slicing and transformation are performed directly on abstract syntax trees enriched with annotations [3]. However, it is more productive to operate on a data structure specifically designed to facilitate these tasks. Many such structures were proposed in the last three decades:

- Program Dependence Graph [6, 10] was the first and the most classic of the data structures used for slicing. It was meant to represent imperative programs in Pascal-like languages and essentially is a pseudograph with edges representing control and data dependences. Its creation is also an imperatively formulated algorithm that builds a control flow graph, a data dependence graph, a data dominator tree, etc.
- System Dependence Graph [8] is a generalisation of a PDG that is capable of expressing interprocedural relations and therefore capable of facilitating more global analyses and detection of interprocedural code clones.
- Dynamic Dependence Graph [1] is a variant more suitable for debugging and similar tasks that can benefit from representing runtime information. From this point of view, both PDGs and SDGs, as well as any of their extensions, are "static dependence graphs".
- Value Dependence Graph [17] is an efficient structure similar to data flow/dependence graphs but is demand-driven. It does not require a full control flow graph, but a control flow graph can be generated from it.
- United Dependence Graph [7] is a hybrid attempt to gain the best of static and dynamic dependence graphs. The static PDGs/SDGs are inefficient but complete; the dynamic ones are fragmented but expressive. A UDG contains both static and dynamic edges for both data and control dependences and is used in cases that need both static and dynamic information, such as testing.
- Java Dependence Graph (JSDG) [20] is the first attempt to extend PDGs/SDGs to a more advanced programming language with object-oriented concepts leading to relations like membership or inheritance dependences. The case study was limited to pre-functional Java but its treatise of interfaces allows straightforward extension to other OO languages with multiple inheritance; there were similar projects for other languages like C++ [9]. Polymorphism is modelled statically by overapproximation.
- Java Dependence Graph (JSysDG) [16] is a result of several incremental improvements of JSDG; its creation is also structured in the classic PDG style with milestone helper structures that represent dependences among methods, classes, interfaces. The most crucial addition is another kind of edges — summary edges that represent the transitive flow of dependence across a callsite caused by both control and data dependences.
- Functional Dependence Graph [12] was finally an application of the same principles to a functional language. The case study was Haskell, but the approach is identical or straightforwardly adaptable to ML, Scala and other functional languages with explicit type constructors. The focus of FDG is on high abstraction level constructs, so slicing is pretty sophisticated but does not cover expressions, just "functional statements".
- Term Dependence Graph [4] is a very relevant technology of modelling dependences in term rewriting systems which focuses on data constructions and function calls without supporting higher order constructions and cond-like branching.
- Behaviour Dependence Graph [15] successfully modelled pattern-driven dispatch and expression decomposition and provided advanced FDG-like functionality for Erlang.
- Probabilistic System Dependence Graph [2] enables calculation of the probability of correctness of an execution, based on learned execution-state changes over previously known correct executions. It entails heavy computations based on large volumes of collected data and is meant to be used on small critical software systems.
- Weighted System Dependence Graph [5] is another hybrid static/dynamic approach where the edges in a static dependence graph are weighed according to all (known) executions. It is basically a realistically performing simplified version of PPDG.
- Erlang Dependence Graph [14] was named after the language it was tested on, but essentially it is an approach not inherently

IFL '15, September 14–16, 2015, Koblenz, Germany.
Copyright © 2015 ACM 978-1-nnnn-nnnn-n/yy/mm. . . $15.00.
http://dx.doi.org/10.1145/nnnnnnn.nnnnnnn

limited to Erlang: its authors managed to combine benefits of SDG (interprocedural functionality), BDG (pattern matching), FDG (high level), TDG (low level) and JSysDG (summary edges). Instead of functional statements, it uses a closely related concept of program positions.

- Execution Dependence Graph [11] is a recent addition to the family that works on the level of bytecode and helps to find code clones there for the sake of library identification.

At IFL 2015, I would like to discuss various opportunities, tradeoffs and limitations in using dependence graphs of functional programs for productive and efficient slicing in the context of different activities and tasks such as functional program comprehension, optimisation, parallelisation, bidirectionalisation, etc. This will be an endeavour to contribute to the lines of research summarised above, and commodify program slicing for different kinds of functional languages.

References

[1] H. Agrawal and J. R. Horgan. Dynamic Program Slicing. In B. N. Fischer, editor, *Proceedings of the 11th Conference on Programming Language Design and Implementation*, pages 246–256. ACM, 1990. ISBN 0-89791-364-7. .

[2] G. K. Baah, A. Podgurski, and M. J. Harrold. The Probabilistic Program Dependence Graph and Its Application to Fault Diagnosis. *IEEE Transactions on Software Engineering (TSE)*, 36:528–545, 2010.

[3] C. Brown. *Tool Support for Refactoring Haskell Programs*. Phd thesis, School of Computing, University of Kent, Canterbury, Kent, UK, 2008.

[4] D. Cheda, J. Silva, and G. Vidal. Static Slicing of Rewrite Systems. In *Proceedings of the 15th Workshop on Functional and (Constraint) Logic Programming (WFLP 2006)*, volume 177 of *ENTCS*, pages 123–136, 2007. . URL http://www.sciencedirect.com/science/article/pii/S1571066107002174.

[5] F. Deng and J. Jones. Weighted System Dependence Graph. In *Proceedings of the Fifth International Conference on Software Testing, Verification and Validation (ICST)*, pages 380–389, April 2012. .

[6] J. Ferrante, K. J. Ottenstein, and J. D. Warren. The Program Dependence Graph and Its Use in Optimization. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 9(3):319–349, July 1987. ISSN 0164-0925. .

[7] I. Forgács, Á. Hajnal, and É. Takács. Regression Slicing and Its Use in Regression Testing. In *Proceedings of the 22nd International Computer Software and Applications Conference (COMPSAC)*, pages 464–469. IEEE Computer Society, 1998. . URL http://doi.ieeecomputersociety.org/10.1109/CMPSAC.1998.716697.

[8] S. Horwitz, T. W. Reps, and D. Binkley. Interprocedural Slicing Using Dependence Graphs. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 12(1):26–60, 1990. . URL http://doi.acm.org/10.1145/77606.77608.

[9] D. Liang and M. J. Harrold. Slicing Objects Using System Dependence Graphs. In *ICSM*, pages 358–367. IEEE Computer Society, 1998.

[10] K. J. Ottenstein and L. M. Ottenstein. The Program Dependence Graph in a Software Development Environment. In *Proceedings of the Software Engineering Symposium on Practical Software Development Environments*, pages 177–184. ACM, 1984. ISBN 0-89791-131-8. .

[11] J. Qiu, X. Su, and P. Ma. Library functions identification in binary code by using graph isomorphism testings. In Y.-G. Guéhéneuc, B. Adams, and A. Serebrenik, editors, *Proceedings of the 22nd International Conference on Software Analysis, Evolution and Reengineering*, pages 261–270. IEEE, 2015. ISBN 978-1-4799-8469-5. .

[12] N. F. Rodrigues and L. S. Barbosa. Component identification through program slicing. In *Proceedings of Formal Aspects of Component Software (FACS'05)*, volume 160 of *ENTCS*, pages 291–304. Elsevier, 2006. . URL http://dx.doi.org/10.1016/j.entcs.2006.05.029.

[13] N. F. Rodrigues and L. S. Barbosa. Higher-Order Lazy Functional Slicing. *Journal of Universal Computer Science*, 13(6):854–873, 2007. . URL http://dx.doi.org/10.3217/jucs-013-06-0854.

[14] J. Silva, S. Tamarit, and C. Tomás. System Dependence Graphs in Sequential Erlang. In *Proceedings of the 15th International Conference on Fundamental Approaches to Software Engineering*, volume 7212 of *LNCS*, pages 486–500. Springer, 2012.

[15] M. Tóth, I. Bozó, Z. Horváth, L. Lövei, M. Tejfel, and T. Kozsik. Impact Analysis of Erlang Programs Using Behaviour Dependency Graphs. In *CEFP*, volume 6299 of *LNCS*, pages 372–390. Springer, 2009. .

[16] N. Walkinshaw, M. Roper, and M. Wood. The Java System Dependence Graph. In *Proceedings of the Third International Workshop on Source Code Analysis and Manipulation*, pages 55–64. IEEE Computer Society, 2003. ISBN 0-7695-2005-7. .

[17] D. Weise, R. F. Crew, M. D. Ernst, and B. Steensgaard. Value Dependence Graphs: Representation without Taxation. In H.-J. Boehm, B. Lang, and D. M. Yellin, editors, *Conference Record of the 21st Symposium on Principles of Programming Languages*, pages 297–310. ACM Press, 1994. ISBN 0-89791-636-0. .

[18] M. Weiser. Program Slicing. *IEEE Transactions on Software Engineering (TSE)*, 10(4):352–357, 1984.

[19] B. Xu, J. Qian, X. Zhang, Z. Wu, and L. Chen. A Brief Survey of Program Slicing. *SIGSOFT Software Engineering Notes*, 30(2):1–36, Mar. 2005. ISSN 0163-5948. . URL http://doi.acm.org/10.1145/1050849.1050865.

[20] J. Zhao. Applying Program Dependence Analysis To Java Software. In *Proceedings of the Workshop on Software Engineering and Database Systems*, pages 162–169, 1998.