Technical Communications of the International Conference on Logic Programming, 2010 (Edinburgh), pp. 241–247
http://www.floc-conference.org/ICLP-home.html

PROGRAM ANALYSIS FOR CODE DUPLICATION IN LOGIC PROGRAMS

CÉLINE DANDOIS

University of Namur
Faculty of Computer Science
rue Grandgagnage 21
B-5000 Namur (Belgium)
E-mail address: cda@info.fundp.ac.be
URL: http://www.fundp.ac.be/info

Abstract. In this PhD project, we deal with the issue of code duplication in logic programs. In particular semantical duplication or redundancy is generally viewed as a possible seed of inconvenience in all phases of the program lifecycle, from development to maintenance. The core of this research is the elaboration of a theory of semantical duplication, and of an automated program analysis capable of detecting such duplication and which could steer, to some extent, automatic refactoring of program code.

1. Introduction and problem description

Program understanding or *program comprehension* refers to the process of acquiring knowledge about the structure and the functioning of a computer program [Rug96]. Such knowledge proves useful in support of a variety of software-engineering related activities including documentation, corrective and adaptive maintenance, migration and evolution of existing software systems [Sto06]. As a research area, program comprehension spans several subfields ranging from cognitive science and software psychology [Cou83, Shn93], over the development of abstract comprehension models [Bro83, Sol84, Sne98] and software visualization techniques [Bal96, JTS98] to using *program analysis* to (partially) automate program comprehension.

In this project, we will investigate program analysis techniques that allow to detect *duplication* within the source code of a given program. In its most general form, the notion of duplication refers to code fragments that are related in the sense that they subsume the same functionality. Note that this definition covers not only code fragments that are *textually* similar ("copy-paste programming") but also code fragments that are *functionally* similar but possibly implemented in a different way. Since duplication constitutes an existential and non-trivial program property, the discovery of duplication is obviously an undecidable problem that can nevertheless be approximated by using program analysis.

1998 ACM Subject Classification: D.1.6, D.2.7, F.3.2.
Key words and phrases: logic programming, program comprehension, static program analysis, code duplication, code clone, software engineering.

Technical Communications of the 26th International Conference on Logic Programming, Edinburgh, July, 2010
Editors: Manuel Hermenegildo, Torsten Schaub
LIPIcs - Leibniz International Proceedings in Informatics. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, Germany
Digital Object Identifier: 10.4230/LIPIcs.ICLP.2010.241

Detailed knowledge about duplication in a program is valuable for various reasons :

- Several studies show that software in which code duplication is present is more *error-prone* and difficult to understand and maintain than software without duplication [Kos06, Wal07a]. Hence the presence of certain forms of duplication is generally considered a bad smell [Fow99] and believed to have a negative impact on software evolution [Gei06]. Although there is some disagreement as to know whether duplication removal is always beneficial [Kap06], there seems to be a consensus that duplication should at the very least be detected [SD02, Joh94, Man06, JM96, Fow99].

- From a purely technical viewpoint, duplication reflects *redundancy* in the source code since it contains distinct code fragments that are semantically equivalent. Although removing such redundancy may offer a practical interest (e.g. *code compaction* that aims at decreasing the executable program size [Bes03]), it also raises an interesting theoretical question [Kos06] whether and to what extend program code can be *normalized* – i.e. being brought into a form such that it contains as few duplication as possible – in a sense similar to the normalization of relational databases [Kho02, Lee95].

- Identifying duplication in a program allows to steer advanced analyses and transformations on the program code such as refactoring [Fow99], cliche recognition [Rug96], aspect mining [JZ08, AK07], virus detection [Wal07b] and plagiarism detection [Lan04]. In addition, an analysis for detecting duplication could be integrated in the code development process in such a way that the creation of duplicates of a certain size is avoided from the beginning [Lag97].

2. Background and overview of the existing literature

No consensus exists in the literature about an exact definition of code duplication. One also finds the notion of *code clones*, although this notion sometimes describes textually similar fragments, sometimes refers to a more general case of duplication [Kos06, Wal07a]. The scope of the proposed definition often depends on the detection technique.

This lack of standardization about code duplication is clearly addressed in three important recent papers which constitute complete overviews of this domain: [Kos06], [Roy07] and [Roy09]. In addition to code duplication terminology, they treat the reasons for code duplication and its consequences, they give the general code duplication detection process, they describe, then evaluate and compare, the existing detection techniques and tools, they talk about visualization, removal, avoidance and management of duplication, they explain evolution and quality analyses based on duplication, they synthesize the applications and related research for code duplication detection, and finally, they expose the open problems in this research field.

Among the vast amount of research done about code duplication during the last decade, to the best of our knowledge, the problem has not received much attention in a logic programming setting. The majority of the results have indeed been reported upon in the context of *imperative* and *object-oriented languages*, as showed in the above references. In the *functional paradigm*, even if it has not really more success than the logic paradigm, some works are emerging. We can point [Li09] which presents the tool Wrangler, able of

detecting duplication, among other things, in Erlang programs. [Bro10] is another work, complementing the latter, which proposes a code duplication detection technique for Haskell, built into the framework of the Haskell Refactorer (HaRe). Some language-independent detection tools exist but, as experimented in [Roy09], such tools lose in precision what they gain with their versatility since they cannot be tuned for the target language.

Furthermore, despite the fact that useful techniques and tools have been developed, most of these code duplication detection techniques are based on the *text*, *syntax* and *structure* of a particular programming language. As such, they are capable of detecting duplicated syntactical programming constructs (real duplication in the terminology of [Roy07]) rather than duplicated *program logic* within the source code of a program [Kos06, Roy07]. Nevertheless, concentrating on the program logic or the computations performed by the program rather than its syntactical appearance is deemed essential [Kos06, Roy07] if duplication detection techniques are to become more powerful, more generally applicable and independent of particular programming language constructs.

3. Goal of the research

The cornerstone of this project is to study duplication in programs written in a logic programming language. Compared with other programming paradigms, logic programming languages have an arguably simple syntax and a small, clear and well-defined semantics. These characteristics make the development of a duplicated-code analysis both more manageable (less dependent on cumbersome syntax) and at the same time more powerful. Indeed, a logic program basically specifies a number of relations that hold between data objects rather than the algorithms to compute certain results as is the case in an imperative language. Consequently, rather than comparing syntactical algorithmic constructs, one can directly compare control- and data-flow relations, in particular when the program is augmented with mode information [Sma00].

4. Current status of the research

This project is still in its infancy and no result have been produced yet. For the moment, we focus on the dissection of the state of the art and we familiarize with some previous recent work realized inside our research group. Indeed, in [Van05], an initial study was made on the concept of code duplication in logic programs and the outline for a basic code duplication analysis of logic programs has been reported upon in [Van08]. The current project presents the natural continuation of both papers. The first ideas extracted from these works were presented at the GRASCOMP 2010 Contact Day, as a hotbed for future development.

5. Open issues and expected achievements

Although promising, devising a duplicated-code analysis for logic programs remains a daunting and non-trivial task. Finding duplication between the data-flow relations exhibited by a program is equivalent to finding isomorphic subgraphs in the data-flow graph of a program which is known to be NP-hard [Kri01, Kom01]. Consequently, sophisticated algorithms guided by heuristics will be necessary in order to make the analysis scalable to medium- and large-size programs. Topics of interest that will be studied in this project

include the following:

- Elaborate a *theory* of duplication in logic programs, including a classification of different kinds of duplication. On the one hand, this will allow to formally define what duplication is about and to state and prove certain results on analyses that try to detect duplicated code. On the other hand, these results could pave the way to develop a theory of normalization of logic programs by removing redundancies. A possible starting point for the latter topic is [Deg07] in which a first attempt was made to define a normal form in the restricted setting of Mercury [Som96] programs. However, [Deg07] does not deal with a number of important issues such as the normalization of data terms and predicate arguments.

- Based on the theory developed above, we will aim at developing an *analysis* that is able to detect duplication into a logic program up to a certain degree. Issues that need to be taken into account include *precision* (maximize the number of real duplicates while possibly minimizing the number of false positives), *granularity* (what is considered a useful duplicate), and *scalability* of the analysis. With respect to granularity, it seems that the notion of a *useful* duplicate may depend on the context of the analysis or the transformation aimed for. Hence, it seems desirable to design an analysis that can be parameterized with respect to the characteristics of the duplication it should search for.

- We will study the relation with advanced programming analysis and transformation techniques. A first topic of interest is automatically detecting opportunities for *refactoring* source code. Preliminary work on refactoring of logic programs [Ser08, Van05] has showed that a number of interesting refactorings can effectively be based on knowledge about duplication in a program. Nevertheless, the exact coupling between duplication in a program and the possibilities for automatic refactoring remains an open problem.

  Another topic of interest is the automatic detection of *cross-cutting concerns*, sometimes called *aspect-mining* [Kic96, AK07]. Basically, a cross-cutting concern is a functionality of the program that is implemented by a set of semantically similar code fragments that are scattered through the source code of a program (a typical example being the "logging" functionality within an application). Recent research has showed that duplication detection techniques can be beneficial for aspect-mining but stronger techniques capable of finding *semantically* related program code are necessary [Bru05, AK07].

Finally, since a logic program can be seen as a set of relations capturing data-flow information, we expect our research on automatically finding duplication within logic programs to be beneficial for analyses trying to find semantically related code in other programming languages and paradigms.

Acknowledgement

This PhD research is done under the supervision of Professor Wim Vanhoof.

References

[AK07] Kim Mens Andy Kellens and Paolo Tonella. A survey of automated code-level aspect mining techniques. *Transactions on Aspect-Oriented Software Development*, 4(4640):143–162, 2007.

[Bal96] Thomas Ball and Stephen G. Eick. Software visualization in the large. *IEEE Computer*, 29(4):33–43, 1996.

[Bes03] Árpád Beszédes, Rudolf Ferenc, Tibor Gyimóthy, André Dolenc, and Konsta Karsisto. Survey of code-size reduction methods. *ACM Computing Surveys*, 35(3):223–267, 2003. doi:http://doi.acm.org/10.1145/937503.937504.

[Bro83] Ruven Brooks. Towards a theory of the comprehension of computer programs. *International Journal of Man-Machine Studies*, 18(6):543–554, 1983.

[Bro10] Christopher Brown and Simon Thompson. Clone detection and elimination for haskell. In *PEPM '10: Proceedings of the 2010 ACM SIGPLAN workshop on Partial evaluation and program manipulation*, pp. 111–120. ACM Press, 2010.

[Bru05] Magiel Bruntink, Arie van Deursen, Remco van Engelen, and Tom Tourwé. On the use of clone detection for identifying crosscutting concern code. *IEEE Trans. Software Eng*, 31(10):804–818, 2005. doi:http://doi.ieeecomputersociety.org/10.1109/TSE.2005.114.

[Cou83] Neal S. Coulter. Software science and cognitive psychology. *IEEE Transactions on Software Engineering*, 9(2):166–171, 1983.

[Deg07] François Degrave and Wim Vanhoof. Towards a normal form for mercury programs. In Andy King (ed.), *LOPSTR*, *Lecture Notes in Computer Science*, vol. 4915, pp. 43–58. Springer, 2007. doi:http://dx.doi.org/10.1007/978-3-540-78769-3_4.

[Fow99] Martin Fowler, Kent Beck, John Brant, William Opdyke, and Don Roberts. *Refactoring: improving the design of existing code*. Object Technology Series. Addison-Wesley, 1999.

[Gei06] Reto Geiger, Beat Fluri, Harald Gall, and Martin Pinzger. Relation of code clones and change couplings. In Luciano Baresi and Reiko Heckel (eds.), *Fundamental Approaches to Software Engineering, 9th International Conference, FASE 2006 March 27-28, 2006, Proceedings*, *Lecture Notes in Computer Science*, vol. 3922, pp. 411–425. Springer, 2006. doi:http://dx.doi.org/10.1007/11693017_31.

[JM96] Claude Leblanc Jean Mayrand and Ettore M. Merlo. Experiment on the automatic detection of function clones in a software system using metrics. In *Proceedings of the 12th International Conference on Software Maintenance (ICSM '96)*, pp. 244–253. 1996.

[Joh94] John Howard Johnson. Substring matching for clone detection and change tracking. In *Proceedings of the International Conference on Software Maintenance (ICSM '94)*, pp. 120–126. 1994. doi:10.1109/ICSM.1994.336783.

[JTS98] Marc H. Brown John T. Stasko, John B. Domingue and Blaine A. Price. *Software Visualization: Programming As a Multimedia Experience*. Cambridge, Mass: MIT Press, 1998.

[JZ08] Yuehua Lin Jing Zhang, Jeff Gray and Robert Tairas. Aspect mining from a modelling perspective. *Int. J. of Computer Applications in Technology*, 31:74–82, 2008.
URL `http://www.inderscience.com/link.php?id=17720`

[Kap06] Cory Kapser and Michael W. Godfrey. Cloning considered harmful. In *Proceedings of the 13th Working Conference on Reverse Engineering (WCRE2006)*, pp. 19–28. IEEE Computer Society, 2006. doi:http://doi.ieeecomputersociety.org/10.1109/WCRE.2006.1.

[Kho02] V. V. Khodorovskii. On normalization of relations in relational databases. *Program. Comput. Softw.*, 28(1):41–52, 2002. doi:http://dx.doi.org/10.1023/A:1013759617481.

[Kic96] G. Kiczales. Aspect-oriented programming. *ACM Computing Surveys*, 28(4es), 1996.

[Kom01] Raghavan Komondoor and Susan Horwitz. Using slicing to identify duplication in source code. In *Proceedings of the 8th International Symposium on Static Analysis (SAS)*. Springer-Verlag, Paris, France, 2001.
URL `http://www.cs.wisc.edu/~raghavan/sas01.pdf`

[Kos06] Rainer Koschke. Survey of research on software clones. In Rainer Koschke, Ettore Merlo, and Andrew Walenstein (eds.), *Duplication, Redundancy, and Similarity in Software*, *Dagstuhl Seminar Proceedings*, vol. 06301. Internationales Begegnungs- und Forschungszentrum fuer Informatik (IBFI), Schloss Dagstuhl, Germany, 2006.
URL `http://drops.dagstuhl.de/opus/volltexte/2007/962`

[Kri01] Jens Krinke. Identifying similar code with program dependence graphs. In *Proceedings Eigth Working Conference on Reverse Engineering (WCRE'01)*, pp. 301–309. IEEE Computer Society, 2001. doi:10.1109/WCRE.2001.957835.

[Lag97] Bruno Laguë, Daniel Proulx, Jean Mayrand, Ettore Merlo, and John P. Hudepohl. Assessing the benefits of incorporating function clone detection in a development process. In *ICSM*, pp. 314–321. 1997.

[Lan04] Thomas Lancaster and Culwin Finta. A comparison of source code plagiarism detection engines. *Computer Science Education*, 14(2):101–112, 2004.

[Lee95] Heeseok Lee. Justifying database normalization: a cost/benefit model. *Inf. Process. Manage.*, 31(1):59–67, 1995. doi:http://dx.doi.org/10.1016/0306-4573(94)E0011-P.

[Li09] Huiqing Li and Simon Thompson. Clone detection and removal for erlang/otp within a refactoring environment. In *PEPM '09: Proceedings of the 2009 ACM SIGPLAN workshop on Partial evaluation and program manipulation*, pp. 169–178. ACM Press, 2009.

[Man06] Zoltán Ádám Mann. Three public enemies: Cut, copy, and paste. *IEEE Computer*, 39(7):31–35, 2006. doi:http://doi.ieeecomputersociety.org/10.1109/MC.2006.246.

[Roy07] C. K. Roy and J. R. Cordy. A survey on software clone detection research. Tech. rep., 2007. TR 2007-541 School of Computing Queen's University at Kingston Ontario, Canada.

[Roy09] Chanchal K. Roy, James R. Cordy, and Rainer Koschke. Comparison and evaluation of code clone detection techniques and tools: A qualitative approach. *Science of Computer Programming*, 74(7):470–495, 2009.

[Rug96] Spencer Rugaber. Program understanding. *Encyclopedia of Computer Science and Technology*, 1996.

[SD02] Stéphane Ducasse Serge Demeyer and Oscar Nierstrasz. *Object-Oriented Reengineering Patterns*. Morgan Kaufmann, 2002.
URL `http://www.iam.unibe.ch/~scg/OORP`

[Ser08] Alexander Serebrenik, Tom Schrijvers, and Bart Demoen. Improving prolog programs: Refactoring for prolog. *TPLP*, 8(2):201–215, 2008. doi:http://dx.doi.org/10.1017/S1471068407003134.

[Shn93] Ben Shneiderman. Software psychology: Sparks of innovation in human-computer interaction, 1993.

[Sma00] J.-G. Smaus, P. Hill, and A. King. Mode analysis domains for typed logic programs. In A. Bossi (ed.), *LOPSTR*. Springer-Verlag, 2000.
URL `http://www.cs.kent.ac.uk/pubs/2000/1011`

[Sne98] Gregor Snelting. Concept Analysis — A New Framework for Program Understanding. In *SIGPLAN/SIGSOFT Workshop on Program Analysis for Software Tools and Engineering (PASTE)*, pp. 1–10. ACM Press, Montreal, Canada, 1998.

[Sol84] Elliot Soloway and Kate Ehrlich. Empirical studies of programming knowledge. *IEEE Transactions on Software Engineering*, 10(5):595–609, 1984. Special Issue on Software Reusability.

[Som96] Z. Somogyi, F. Henderson, and T. Conway. The execution algorithm of Mercury: an efficient purely declarative logic programming language. *Journal of Logic Programming*, 29(1–3), 1996.

[Sto06] Margaret-Anne D. Storey. Theories, tools and research methods in program comprehension: past, present and future. *Software Quality Journal*, 14(3):187–208, 2006. doi:http://dx.doi.org/10.1007/s11219-006-9216-4.

[Van05] W. Vanhoof. Searching semantically equivalent code fragments in logic programs. In S. Etalle and Springer-Verlag (eds.), *Proceedings of LOPSTR 2004*, *LLNCS*, vol. 3573. 2005.

[Van08] W. Vanhoof and F. Degrave. An algorithm for sophisticated code matching in logic programs. In M. Garcia de la Banda, E. Pontelli, and Springer-Verlag (eds.), *Proceedings of ICLP 2008*, *LLNCS*, vol. 5366. 2008.

[Wal07a] Andrew Walenstein, Mohammad El-Ramly, James R. Cordy, William S. Evans, Kiarash Mahdavi, Markus Pizka, Ganesan Ramalingam, and Jürgen Wolff von Gudenberg. Similarity in programs. In Rainer Koschke, Ettore Merlo, and Andrew Walenstein (eds.), *Duplication, Redundancy, and Similarity in Software*, no. 06301 in Dagstuhl Seminar Proceedings. Internationales Begegnungs- und Forschungszentrum für Informatik (IBFI), Schloss Dagstuhl, Germany, Dagstuhl, Germany, 2007.
URL `http://drops.dagstuhl.de/opus/volltexte/2007/968`

[Wal07b] Andrew Walenstein and Arun Lakhotia. The software similarity problem in malware analysis. In Rainer Koschke, Ettore Merlo, and Andrew Walenstein (eds.), *Duplication, Redundancy, and Similarity in Software*, no. 06301 in Dagstuhl Seminar Proceedings. Internationales Begegnungs- und Forschungszentrum für Informatik (IBFI), Schloss Dagstuhl, Germany, Dagstuhl, Germany, 2007.
URL `http://drops.dagstuhl.de/opus/volltexte/2007/964`