Baltic J. Modern Computing, Vol. 7 (2019), No. 3, 430-435
https://doi.org/10.22364/bjmc.2019.7.3.09

Recursive AOP for Reversible Software

Vaidas GIEDRIMAS

Siauliai University, Visinskio 38, Siauliai, Lithuania

vaidas.giedrimas@gmail.com

Abstract. Innovative paradigm of reversible computing (RC) is on the focus of current computer science. The main idea of it is to enable the execution of programs not forward only (as it is now) but backwards as well. The set of possible applications of RC includes software debugging, fault-tolerance increasing and quantum computing. In order to do reversible computing operational there is a need to code (manually) or to generate opposite "reverse" element for each "forward" element. However, this emerging paradigm still needs the methods for expressing opposite language elements in advance (during design-time). On the other hand, classic Aspect oriented paradigm (AOP) helps to deal with crosscutting concerns such as security, transactions etc. at design time, allows to separate development tasks by proficiency instead of by modules and to run weaved program at run-time. In this paper author present the idea to express opposite language elements by using recursive aspects.

Keywords: Reversibility, Recursive algorithms, Programming languages, Aspect oriented programming

Introduction

This paper is focused on the intersection of 3 different concepts: reversible computing, aspect-oriented programming and recursion.

Reversible computing is on the focus of current computer science. The main idea of it is to enable the execution of the programs not forward only (as it is now) but backwards as well (Phillips and Thomsen, 2014; Mezzina and Schlatte, 2014). The need for reversible computing is coming from various communities including quantum computing community. Reversible computing could be used in such areas as control systems and robots with a high level of autonomy, self-healing robust and secure software systems, automated software testing and debugging etc. There are made already different approaches for the formalization (Paolini et al., 2016; Kaarsgaard et al, 2017; Gluck and Kaarsgaard, 2018) process automation and support of reversible software (Salmi and Parsa, 2009). However current achievements in the tools for reversible computing are insufficient in comparison to forward only computing. Most of the languages and tools are working either in the theoretical level or on the structural programming paradigm. There are also some attempts to adopt the concept of reversible computing in object-oriented and component-based development paradigms, however this still do not empower developers enough. We need more flexible and more automated tools.

The proposed approach in this paper is based on aspect-oriented programming (AOP) paradigm, which is relatively new (Kiczales and Hilsdale, 2001; Raheman et al., 2018; Li et al, 2011; Fabry et al., 2015). AOP presents the form of modularity with crosscutting concerns. It is quite wide used for development of complex distributed systems (Li et al, 2011; Singh et al., 2017). AOP consist three main concepts: *aspect*, *advice* and *pointcut*. When describing the aspect, we must to specify exactly what code should be injected (the advice) to what particular places (the *pointcut*). The *pointcut* usually is expressed as a query over the signature of the functions, defining what pattern of function's name, number and type of parameters etc. should be affected. Usually AOP is used when it is the need to write (and execute in the run-time) the same code in many places of the program.

The concept of the recursion in this paper is used as it is understood in software development domain. It (almost) has no relation with the definition of "recursive partial functions" in theoretical computer science.

The main goal of this paper is to present the idea to express opposite language elements by using recursive aspects. The paper is organized as follows. Section 1 presents the key points of the model of reversible software execution, made by authors of this paper. Section 2 exposes the details of aspect oriented approach for reversible software development and execution. Section 3 outlines related work, and finally the conclusions are made and the future work trends are presented.

1. The model of reversibility

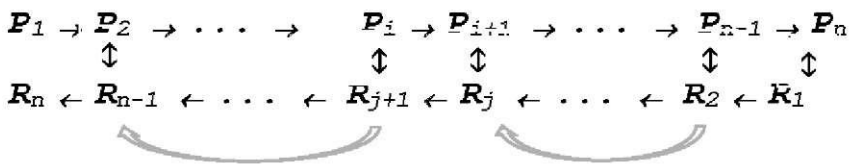

Figure 1. A ladder model of reversible computing

The program could be treated as the set of the states or the sequence of the actions. Because our context is reversible computing, we are focusing on actions. So, any forward only program could be expressed as the sequence P_1, P_2, ... P_n, where each P_i have one-to-one relation with i-th code line. Completely reverse (or backwards only) program will looks like the sequence R_1, R_2, … R_n (Figure 1). The number of actions in the P program and in the R in simplified case is the same. So the actions can be paired: $<P_1, R_n>$, $< P_2, R_{n-i} >$, ... Each action in P have the opposite action in R. Having this, it is possible to execute the program forward and backwards and the number of steps is not important. In general case (if we are using not Janus-style structural reversibility but checkpoint-based reversibility) it is possible to make "big hop" transitions, when few P_i actions could be "undone" by one additional R_u action.

2. Aspect oriented approach

Despite of the type of the reversibility, in order to execute some action(s) backwards we need to have the action(s) which would return the program to initial state, which was before the action(s). "Opposite" program consisting such opposite actions could be developed manually during the developing of main program. However, this could be very inefficient and could demand more human and time resources. We believe that "opposite" programs consisting "opposite" actions must be created in automated way using the chosen model of reversibility.

One of possible algorithms to generate the code of opposite program is to take each line of forward-only starting from the last one, and generate line-by-line of "opposite" program with the opposite actions. For trivial actions such as increment/decrement, other arithmetic operations, conditional sentence and loop this could be done easy. However, if we take the premise that our initial program could consist of functions calls then the generation of the code becomes more complicated. It is impossible to cover all the functions in the model, so the existing functions must be analyzed and decomposed to more trivial actions. If we allow to use sub-functions, then even after the first iteration of decomposition we do not have trivial actions only and will need to perform second, third etc. iterations until all the code will be decomposed to trivial parts. The software generator, which should decompose initial program and to generate opposite program should be based on recursive algorithms.

The recursive decomposition and processing steps lets to generate opposite programs for the software of nearly any complexity. In Janus-style reversibility approach this could be done (Lutz, 1986; Yokoyama, 2010; Gluck and Kaarsgaard, 2018). But if we have checkpoint based reversibility style (as, e.g., in (Lanese et al., 2018)), then the return to the checkpoint could be (and often is) very different from the way forward. In that case human help is necessary. Only human could make imperative code, which could perform the backwards transition. However, the transitions could be not very unique because the actions for forward transition between states are not unique as well. As some forward actions could be used few times in different places of the program, the backward actions could be used as well.

Aspect oriented programming is very often used, when it is necessary to separate the concerns (e.g. security, logging, transaction support aspects). The *advice* of the aspect once written is viewed in the code in many places. So the developer could not care in what exact lines in the code it is weaved and do not check is the required code written in exactly the same order or not. Similarly, we offer to extend the scope of aspect oriented programming by introducing additional advices such as instead or reverse. As classic AOP advices such as *before* or *after* helps to specify what additional code should be weaved in the final program, the advice instead could help to specify what exactly code lines cover the action R_j, which must be executed in order to eliminate the results of action P_{j+i} (Figure 1). If the code lines and actions from P sequence are unique, then AOP approach would not be efficient. However, in any program some lines (or its groups) have the same pattern. It is obvious that in order to prepare R program besides of P, we need to operate not only with functions (as classical *pointcuts* do) but with the anonymous fragments of the code as well. For this reason, there is the need for additional type of *pointcut* describing not function, but a pattern of code lines.

AOP is still evolving paradigm, however it is already formalized in some extent. Typical algorithms of AOP are pointed described in (Raheman et al., 2018; Bennett et al., 2010; Li et al, 2011). This algorithm is used in such AOP implementations as AspectJ (Gang et al., 2013; Pearce et al., 2007) and Spring-AOP (Laddad, 2009). We offer to use extended AOP algorithm for weaving (Zhang and Khedri, 2016) in order to support reversibility in aspect oriented programs.

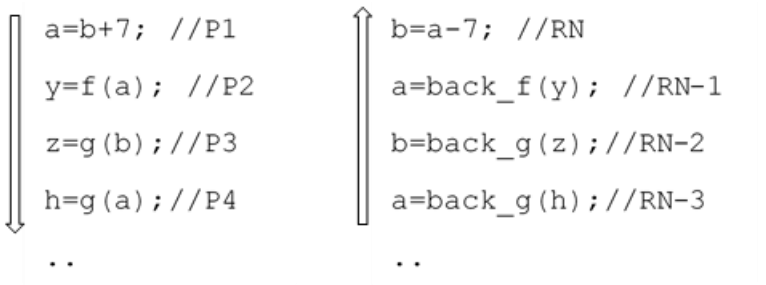

Figure 2. Example of classical and reverse code lines.

3. Related work

The approach of Singh et al. (2017) is focused on the program slicing. Program slicing is a type of program analysis technique that studies the impact of a statement in a program on other statements of the same or different programs. They present the analysis tool - D-Aspect slicer, which is able to decompose the program to "slices". The slicing algorithm (Singh et al., 2017) generates smaller slices in less time as compared to other two existing algorithms during the time, when D-Aspect slicer was presented. This approach is similar to ours because it exposes the possibility to analyze and even change the AOP elements dynamically. However, in Singh et al. (2017) main focus is on AOP for forward only software and the presented algorithm is targeted software analysis, testing and debugging purposes, not reversible execution.

The paper (Hoffman and Eugster, 2007) exposes the details how the power of AspectJ is increased by allowing direct (explicit) communication between the aspects and base code instead of using explicit join points (EJPs). The cooperative aspect-oriented programming (Co-AOP) paradigm, presented in (Hoffman and Eugster, 2007) is one of the successful cases exposing the possibilities to extent existing AOP, however it is not related with reversible computing.

The concepts of recursion and AOP are mentioned in various papers, e.g. in (Bodden et al., 2006). The focus of Bodden et al. (2006) is on the avoidance of self-recursion, i.e. the phenomena when the AOP advices can access and change the code of themselves when the code of advice meets the criteria defined by pointcut query. This would lead to infinitive recursion and waste of resources.

In short, the authors of this paper do not find any evidences that there are some applications of Aspect oriented programming for reversible software development and/or support during run-time. However most of related works expose encouraging premises for this.

4. Conclusions and Future Work

The following conclusions are made:

- The algorithm of generative development of reversible program depends on chosen style of reversibility. In Janus-style reversibility case the code generator should use recursion. In checkpoint-based reversibility case the elements of aspect oriented programing are necessary as well.

- The general AOP approach could be extended by adding two elements:

    - the *advice*, which must be used for describing the actions required for backwards transition; and

    - additional type of *pointcut* describing not standalone function, but a pattern of code lines instead;

The following research actions are planned as the future work:

- Work with more test cases in order to get statistically representative set of data to judge about the performance, safety and other aspects of proposed approach.

- It's necessary to make comparative study of proposed approach in the context of different notions of reversibility. This could help to evaluate the different reversibility styles it selves.

- The tune-up for this approach working with other AOP *advices* such as *Around* is necessary.

- The results of this paper can be used by both scientists working on reversible programming and by scientist focused on the aspect oriented paradigm.

References

Bennett et al. (2010). Aspect-oriented model-driven skeleton code generation: A graph-based transformation approach. In *Science of Computer Programming* 75(8), 689 – 725.

Bodden, E. et al. (2006). Avoiding infinite recursion with stratified aspects. In: NODe/GSEM. LNI, vol. 88, pp. 49-64. GI

Fabry, J. et al. (2015). A taxonomy of domain-specific aspect languages. *ACM Comput. Surv.* 47(3), 40:1-40:44 (Feb 2015). https://doi.org/10.1145/2685028

Gang, X. et al. (2013). A semantics of poincuts in AspectJ. In *IERI Procedia 4*, 323 - 330, 2013 International Conference on Electronic Engineering and Computer Science (EECS 2013)

Gluck, R., Kaarsgaard, R. (2018). A categorical foundation for structured reversible flowchart languages. In *Electronic Notes in Theoretical Computer Science 336*, 155 - 171, the Thirty-third Conference on the Mathematical Foundations of Programming Semantics (MFPS XXXIII)

Hoffman, K., Eugster, P. (2007). Cooperative aspect-oriented programming. Science of Computer Programming 74(5), 333 – 354, In *Special Issue on Principles and Practices of Programming in Java* (PPPJ 2007)

Kaarsgaard, R. et al. (2017). Join inverse categories and reversible recursion. *Journal of Logical and Algebraic Methods in Programming* 87, 33 – 50.

Kiczales, G., Hilsdale, E. (2001). Aspect-oriented programming. *SIGSOFT Softw. Eng. Notes* 26(5), 313-.

Laddad, R. (2009). *AspectJ in Action: Enterprise AOP with Spring Applications*. Manning Publications Co., Greenwich, CT, USA, 2nd ed.

Lanese, I. et al. (2018). A theory of reversibility for Erlang. *Journal of Logical and Algebraic Methods in Programming* 100, 71 - 97

Li, L. et al. (2011). Semantic based aspect-oriented programming for context-aware web service composition. *Information Systems* 36(3), 551 - 564, special Issue on WISE 2009 - Web Information Systems Engineering

Lutz, C. (1986). *Janus: a time-reversible language*. Available at http://tetsuo.jp/ref/janus.html, Letter to R. Landauer.

Mezzina, C.A., Schlatte, R. (2014). *Reversible computation:extending horizons of computing, state of the art report*, Working group 2, software and systems, Available at http://topps.diku.dk/ic1405/wg2_soar.pdf

Paolini, L. et al. (2016). A class of reversible primitive recursive functions. In *Electronic Notes in Theoretical Computer Science 322*, 227 – 242, Proceedings of ICTCS 2015, the 16th Italian Conference on Theoretical Computer Science

Pearce, D.J. et al. (2007). Profiling with AspectJ. *Softw. Pract. Exper*. 37(7), 747-777

Phillips, I., Thomsen, M.K. (2014). *Reversible computation: extending horizons of computing, state of the art report*, Working group 1, Foundations, Available at http://topps.diku.dk/ic1405/wg1_soar.pdf

Raheman, S.R. et al. (2018). Aspect oriented programs: Issues and perspective. *Journal of Electrical Systems and Information Technology*.

Salmi, M.F., Parsa, S. (2009). Automatic detection of infinite recursion in AspectJ programs. In: *Future Generation Information Technology, First International Conference* FGIT 2009, Jeju Island, Korea, December 10-12, 2009. Proceedings. pp. 190-197

Singh, J. et al. (2017). Dynamic slicing of distributed aspect-oriented programs: A context-sensitive approach. *Computer Standards Interfaces* 52, 71 - 84

Yokoyama, T. (2010). Reversible computation and reversible programming languages. Electronic Notes in Theoretical Computer Science 253(6), 71 – 81, In *Proceedings of the Workshop on Reversible Computation* (RC 2009)

Zhang, Q., Khedri, R. (2016). On the weaving process of aspect-oriented product family algebra. *Journal of Logical and Algebraic Methods in Programming* 85(1, Part 2), 146 – 172, Sp. Issue on Formal Methods for Software Product Line Engineering.

Received September 9, 2019, accepted September 21, 2019