Refactoring in Automatically Generated Programs

Fernando E. B. Otero, Colin G. Johnson, Alex A. Freitas, and Simon J. Thompson
School of Computing
University of Kent
United Kingdom
{*F.E.B.Otero,C.G.Johnson,A.A.Freitas,S.J.Thompson*}*@kent.ac.uk*

***Abstract*—Refactoring aims at improving the design of existing code by introducing structural modifications without changing its behaviour. It is used to adjust a system's design in order to facilitate its maintenance and extendability. Since deciding which refactoring to apply and where it should be applied is not a straightforward decision, search-based approaches to automating the task of software refactoring have been proposed recently. So far, these approaches have been applied only to human-written code. Despite many years of computer programming experience, certain problems are very difficult for programmers to solve. To address this, researches have developed methods where computers automatically create program code from a description of the problem to be solved. One of the most popular forms of automated program creation is called Genetic Programming (GP). The aim of this work is to make GP more effective by introducing an automated refactoring step, based on the refactoring work in the software engineering community. We believe that the refactoring step will enhance the ability of GP to produce code that solves more complex problems, as well as result in evolved code that is both simpler and more idiomatically structured than that produced by traditional GP methods.**

***Keywords*-refactoring; genetic programming; automated design improvement;**

I. Introduction

Software systems undergo incremental changes over time in order to deal with new requirements. Since the original design is not prepared for every new requirement in general, the addition of functionality brings the risk of degrading the quality of the design (structure) of the system. A common approach to mitigate this risk involves the use of refactoring. Refactoring aims at improving the design of existing code by introducing structural modifications without changing its behaviour. The motivation for refactoring the code of a system is that a well-designed system is generally easier to maintain and extend. Refactoring is now a core part of software engineering practice, and is supported by the inclusion of refactoring tools in well-used integrated development environments. While refactoring can help to improve a software design, deciding which refactoring to apply and where it should be applied is not a straight forward decision.

Recently, search-based approaches to automate the application of refactoring have been proposed [1], [2]. These approaches cast the refactoring as an optimisation problem, where the goal is to improve the design quality of a system based on a set of software metrics. After formulating the refactoring as an optimisation problem by defining the solution representation, search operators and fitness function, several different methods can be applied to the problem of automated refactoring—e.g. hill climbing, simulated annealing and genetic algorithms. So far, the idea of automatic refactoring has been applied only to human-written code.

Genetic Programming (GP) [3], [4] is an evolutionary technique, based on Darwin's principle of natural selection, which aims at automatically evolving computer programs. In the GP context, a computer program is a solution to the problem at hand, which can be represented as a mathematical equation, a sequence of instructions or an arbitrary combination of input values. GP uses the principle of natural selection to find solutions to complex problems by evolving initially poor solutions into near-optimal ones using a set of genetic operators and a fitness measure. In recent years GP has been applied to a number of problems of practical significance, and has produced a number of solutions to problems that are *human-competitive* [5]—for example with a GP algorithm producing a solution comparable with one which has been patented.

In this work, we aim at testing the hypotheses that adding in an automated refactoring step will enhance the ability of the evolutionary process in GP to produce code that solves more complex problems than traditional GP methods; and that by adding in the refactoring step the code evolved is simpler and more idiomatically structured—and therefore more readily understood and analysed by human programmers—than that produced by traditional GP methods. The research will draw upon work in the software engineering community in automatically applying refactoring steps that have proven effective for human software engineers. Furthermore, it will automatically identify which refactoring steps prove to be most effective during the GP process in order to apply these explicitly in future GP runs.

II. Methodology

Essentially, a GP algorithm consists of a population of candidate solutions to the target problem and an iterative selection process that mimics an evolutionary process. A traditional GP involves four main steps, as illustrated in

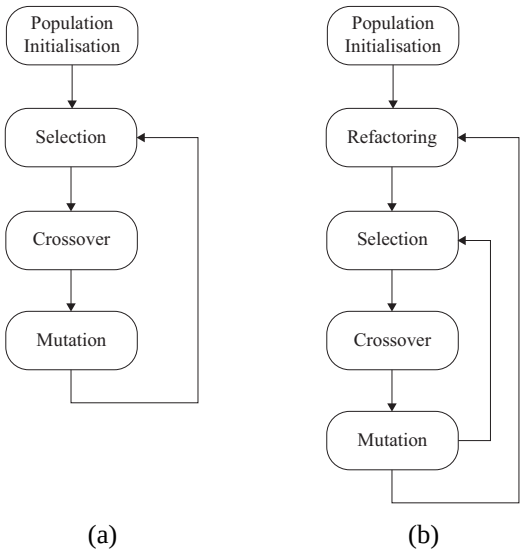

Figure 1. Introducing a refactoring step into genetic programming. In (a) the traditional GP steps; (b) the GP steps with the addition of a refactoring step.

Figure 1(a). Firstly, a population of random programs is generated. The remaining stages form an iterative loop. The programs are evaluated, e.g. by running them on a set of test data and measuring the resultant performance on the problem at hand—this measurement is called the fitness score. Based on this fitness score a number of the fitter programs are chosen to form a basis for the next generation—these are called the parent programs for the next generation. This next generation is then created by performing mutation and crossover steps on programs chosen from the set of parents. This loop continues until a particular fitness value is found, or until a particular number of iterations have been carried out.

The core idea in this work is to explore the addition of a *refactoring step* into the genetic programming iteration. That is, there will be an additional loop in which refactoring steps drawn from a catalogue of such steps will be applied to individuals of the population. The contrast between this and traditional GP is illustrated in Figure 1(b). These refactoring steps will be based on traditional software engineering, and developed to take into account factors that are important for GP (e.g., reducing the amount of unused code).

Given that programs are generated by an evolutionary process in GP, the introduced refactoring step adopts a *search-based* approach to automatically carrying out refactorings. Individuals are selected from the population and undergo a refactoring step, checking whether the preconditions for that refactoring obtain, and if so, applying the refactoring.

III. Ongoing Work

As a first case study, we are currently working on implementing a refactoring step to replace duplicated code within an individual, dubbed *code duplication refactoring*. This refactoring consists on improving the structure of an individual by identifying duplicated code, encapsulating it as a single instruction and replacing each occurrence by the single instruction. The encapsulation of the duplicated code could help the evolutionary process by preserving useful blocks of code from the effects of genetic operators, as well as providing a more compact structure for the individuals. For example, in a symbolic regression problem where individuals represent arithmetic expressions, an individual corresponding to the expression '$y+(x*x)-(2*(x*x))$'—with x and y representing numeric variables—undergoing the code duplication refactoring would be transformed to '$y + E - (2 * E))$'. In this case, the variable E would encapsulate the expression '$(x * x)$'.

A variation of the code duplication refactoring is the example of *function extraction*, where a program is refactored so that a code block is removed from its context and replaced by a function call. The block of code removed forms the body of that function. As with the code duplication refactoring, the function extraction could provide a way of forming blocks of code—or modules—for the evolutionary process to reuse, something which has already been shown to be useful in GP [6]. A complementary *unfold refactoring* step will also be investigated, which has the opposite effect of the code duplication and function extraction refactorings. The unfold refactoring replaces a variable or a function call by the corresponding encapsulated expression, allowing expression to evolve independently from that point on.

References

[1] O. Seng, J. Stammel, and D. Burkhart, "Search-based determination of refactorings for improving the class structure of object-oriented systems," in *Proceedings of the Genetic and Evolutionary Computation Conference (GECCO 2006)*. ACM, 2006, pp. 1909–1916.

[2] M. O'Keeffe and M. O. Cinnéide, "Search-based refactoring: an empirical study," *Journal of Software Maintenance and Evolution: Research and Practice*, vol. 20, no. 5, pp. 345–364, 2008.

[3] J. Koza, *Genetic Programming: On the Programming of Computers by Means of Natural Selection*. MIT Press, 1992.

[4] W. Banzhaf, P. Nordin, R. Keller, and F. Francone, *Genetic Programming—an introduction: on the automatic evolution of computer programs and its applications*. Morgan Kaufmann, 1998.

[5] J. Koza, M. Keane, M. Streeter, W. Mydlowec, J. Yu, and G. Lanza, *Genetic Programming IV: Routine Human-Competitive Machine Intelligence*. Kluwer Academic Publishers, 2003.

[6] J. Koza, *Genetic Programming II: Automatic discovery of reusable programs*. MIT Press, 1994.