Scalable metaheuristics for automated program synthesis

1 Project objectives

1.1 Background

This proposal concerns *automated program synthesis* (APS), a research area within computational intelligence where the objective of an intelligent system is to synthesize a program that meets a specification provided as a set of examples or constraints.

The problem statement of program synthesis can be phrased more formally in the following way. We assume that a program is a discrete structure (typically a list or nested hierarchy of lists) of instructions from a given programming language. When applied to an input $in \in I$, a program p produces an output $p(in) \in O$. A *programming task* (program synthesis task) is defined as a set of *tests* (*examples*, *cases*) $T \subseteq I \times O$, each of them being a pair of program input $in \in I$ and the corresponding *desired output* $out \in O$; put in machine learning terms, T forms the *training set.*

A synthesis algorithm *solves* a programming task T if it produces a program p that behaves as required by the tests, i.e., for every test $(in, out) \in T$, p yields the desired output: $p(in) = out$. When posed in this way, a programming task forms a *search problem.* Optionally, it can be augmented with an *objective function* f which, when applied to a program, expresses the degree to which it attains the search goal; such a programming task forms an *optimization problems.* In either case, the search space is discrete, and hosts all admissible programs in the programming language. Importantly, the synthesized program is expected to generalize beyond the training set.

An example of a simple yet nontrivial programming task is the synthesis of a sorting program. In that case, every test in T is a pair of program input, i.e., a list to be sorted (e.g., (9,3,7,1,6)), and the corresponding desired output, i.e., the appropriate sorted list (here: (1,3,6,7,9)). Given a set of such tests, a program synthesis method is supposed to come up with a sorting program. This search problem could be advanced to an optimization problem by augmenting its specification with a objective function f. In the simplest case, f could count the number of tests passed by a program. A more sophisticated fitness function could, for instance, measure Kendall's correlation coefficient between the actual program output and the desired sorted list.

1.2 Genetic programming (GP)

There are a few approaches to APS, pursued quite independently in various branches of computer science [47, 23, 43]. Of them, in this project we employ *genetic programming* (GP, [24, 55]), the domain of heuristic program synthesis inspired by the neo-Darwinian principles of natural evolution. Technically, GP engages evolutionary computation [12, 17] to iteratively synthesize and improve a population of candidate solutions (programs). By combining the 'mechanics' borrowed from the biological evolution with the expressive power of computer programs, GP forms a powerful paradigm of computational intelligence.

GP reaches beyond the automatic programming as it is practiced in inductive logic programming [43], deductive program synthesis [47], or search-based software engineering [15]. For instance, GP is not constrained to evolving programs in conventional programming languages, or even programs that are legible for humans. GP programs can represent arbitrary data types, including not only categorical and continuous, but also compound data structures like lists, sets, or even images (see, e.g., [26, 27]). GP is thus substantially different from, e.g., inductive logic programming [43] which focuses on the categorical datatypes.

The contemporary GP features a few major program representations (expression trees similar to abstract syntax trees [23], graphs [50], sequences [7], nested lists [59]), dozens of search operators, and a large number of evolutionary search algorithms. GP is now a rich, diversified research field, with

well identified virtues and vices. It has been successfully applied to hundreds of tasks of scientific and practical nature, and attained human-competitive performance on many of them [55], like designing radio antennas [46], solving challenging problems in pure mathematics [58, 9], design of quantum computing algorithms [2, 48], detecting bugs in existing software and repairing them [64], and automatic software improvement [40]. The contributions of GP have been featured in the top scientific journals, like *Science* [57] and *Nature* [44, 1]. The number of entries in the genetic programming bibliography is close to passing the 10,000 mark [39, 38].

1.3 Problem Statement and Main Objective

Apart from trivial cases, programming tasks are challenging. There are a few of reasons for this state of affairs. Firstly, the search space of programs is usually large (or often infinite), and grows exponentially with the length of considered programs. Secondly, the relationship between program code (syntax) and its effects (semantics) is very complex: a minute modification in program code can drastically change its behavior, i.e., the output it produces (an example will be given in Section 3.3). On the other hand, the same behavior can be implemented by many programs – as a matter of fact, there are usually infinitely many programs that behave in exactly the same way. As a result, the so-called *fitness landscape* [21], i.e., the fitness function plotted against the search space, is very rugged, features many local optima as well as plateaus. Finally, task specification is usually incomplete, i.e., the set of tests T does not contain all correct input-output pairs, so a synthesis method is expected to generalize beyond the training sample (and in this sense perform induction) while avoiding overfitting.

As a result, APS methods, including GP algorithms, find it hard to scale well with task difficulty and instance size. For example, one of the many common benchmarks in GP is the parity task [49], where each test is an input list of bits of a fixed length accompanied with the desired output - the corresponding parity bit. A program solving this task for lists of length 5 can be relatively easily synthesized with a baseline, unsophisticated variant of GP. However, synthesizing a program that realizes this functionality just for 7 is already much harder [54], and for 10 or more bits becomes extremely difficult.

The proposed project is intended to advance the current state of knowledge in automated program synthesis by developing a new generation of scalable GP-based program synthesis algorithms. The main **hypothesis** is that this can be attained by making search process *better informed* about the semantics of candidate programs, their internal behavior, and by guiding the search process in a more considerate way. The particular aspects of this hypothesis are motivated and addressed by the particular tasks of this proposal, detailed in Section 'Work plan'.

2 Significance

Computer programs are believably the most general formalism available and, thanks to capacity of conducting any Turing-complete computation, can in principle express solutions to any type of problem. In other words, programs, whether written by a human or synthesized automatically, can be applied to and solve problems of any nature, like learning problems, optimization problems, search problems, problem solving (in AI sense), etc. Programs are capable of expressing algorithms that are *active* and capable of processing data of arbitrary types (in contrast to conventional search and optimization problems, where such candidate solutions are'passive' in the above sense). By advancing APS in this project, we look forward to improving the availability of APS in several contexts discussed in the following.

In the context of AI and computational intelligence, computer programs offer powerful representation of intelligent agents. In a natural way, the percepts received by an agent can be fed into a program and the output generated by that program can determine agent's actions. This makes it possible to express behavior of intelligent agents in all sorts of environments and enables GP to to be applied to a broad spectrum of problems in science and technology, from discovery of natural laws [57], to synthesis of completely new hardware designs [18], quantum computing [2], machine learning and pattern recognition [27].

In software engineering (SE), there is more evidence now that computer programming, once an exclusively human domain, becomes more open to certain forms of automation. Given the growing complexity of software, this becomes a necessity. The advent of search-based software engineering

([15, 14]; see also `http://ssbse.org/2014/`), a novel branch of SE, is a clear sign of this trend. In this context, GP and similar methods prove capable of efficient bug detection and repair [13], improvement of functional and non-functional properties of code (e.g., like execution time, memory occupancy, or length) [41], generating new tests [42], etc. With moderate effort, the techniques developed in this project should be applicable not only to program synthesis meant as a process that starts from scratch, but to aid humans cope with the growing complexity of code and other software artifacts.

The prospective outcomes of the proposed project can substantially contribute to the above disciplines. This in turn can have immense leverage on society as a whole, because software is pervasive, essential to the functioning of many aspects of society, and forms a multi-billion euro industry. There are opportunities to use automated methods in the design and maintenance of software, resulting in great decreases in cost and effort, and increases in reliability and efficiency.

Last but not least, some of the approaches elaborated here can be extrapolated beyond program synthesis, in particular to design better metaheuristic algorithms. For instance, providing a search process with more detailed information on the outcomes of evaluation of a candidate solution (cf. behavioral evaluation, Task 2 below) is a viable option in application domains where this process involves some form of simulation. Evolutionary design (like, e.g., [18]) is an obvious example here. Another growing application area involves climbing one level higher on the 'abstraction ladder' and using advanced APS in *hyperheuristics*, i.e., metaheuristics that generate (or operate on) metaheuristics (see, e.g., [53]).

The proposed project obviously cannot address all the above actual and prospective contexts and applications. Nevertheless, by focusing on the fundamental research question concerning APS (i.e., how to control/drive the synthesis process to efficiently find a feasible solution?) we find it realistic to advance the contemporary APS methods and in particular make them better scalable. We base this claim on the encouraging preliminary results obtained in the applicant's team [51, 34, 45, more details in the next section] on one hand, and on the growing interest in the APS and search-based software engineering community.

The project tasks detailed in the next section pave new ways for tackling the contemporary challenges for APS outlined in Section 1.3. The concepts to be tested within Task 1 build on a new and promising concept of semantic GP; the research proposed for Task 2 and Task 3 forms original and otherwise unexplored directions, a preliminary drafts of which already met with positive reception [35, 33]. In Task 4, the synthesis algorithm developed within the project will be verified on a diversified suite of problems of scientific and practical character. However, universality of the computer programming paradigm sets no limits to implications in other application areas.

3 Work plan

The assumed work plan is intended to provide a better understanding of the shortcomings of existing approaches, and to design, implement, and experimentally verify a new family of automated program synthesis algorithms that scale better with task complexity. In particular, each of the research tasks below targets a specific limitation of contemporary program synthesis methods (and in particular GP), which we will motivate individually for every task.

3.1 Task 1: Semantic extensions of genetic programming

Motivation. Most of contemporary approaches to program synthesis strive to be generic, and in doing so are purely syntax-based[1]. A typical GP search operates at the level of source code, i.e., as abstract symbols, regardless of whether the actual computation takes place in the continuous domain (so-called symbolic regression), Boolean domain, or concerns Turing-complete computation involving conditional statements and loops.

As an illustration, let us shortly characterize the conventional search operators used in the most common, *tree-based* variant of GP, where candidate programs are represented as expression trees [24, 55]. Within that framework, program mutation consists in replacing a randomly selected subexpression of a program with another, randomly generated, subexpression (Fig. 1). The way in which such

[1]This remark applies not only to GP, but also to other paradigms of automated programming; see, for instance, the Syntax-Guided Synthesis Competition, http://sygus.org/

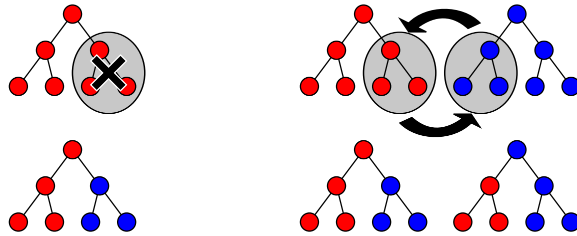

Figure 1: Conventional mutation (left) and crossover (right) for tree-based GP. Top row: parents. Bottom row: offspring.

modifications propagate to the *effects* (behavior) of program execution, i.e., program output, is very complex and depends on the code of the program as well as on the choice of location at which the operator is applied. As a result, it might happen that the mutated program behaves in a way that is often very different from that of its parent, i.e., returns largely different outputs for the considered tests. This contradicts the conventional role of mutation, which is normally intended to introduce *minor* changes in candidate solutions.

Analogously, crossover operator in evolutionary computation is intended to produce a candidate a solution that 'blends' the features of its parents. The most commonly used crossover operator in tree-based GP swaps two randomly selected subexpressions (subprograms) between the parent individuals. Unfortunately, an offspring solution resulting from such an act is often not much of a 'mixture' of its parents in terms of the output it produces.

It should not then come as a surprise that such a blind syntactic search cannot work well for different problems, across various domains, nor scale well with problem size. In the end, it is the *meaning* of a program that determines its fitness and, *en masse*, determines how successful the search is at solving the problem. And that meaning, i.e., *program semantics*, is largely neglected in conventional approaches to program synthesis.

This issue has been identified and characterized from different perspectives by referring to concepts like *locality* [56, 62], meant as the degree of distortion introduced by the genotype-phenotype mapping. Put in these terms, in GP genotypes (program code) map into phenotypes (program behavior) at low locality, so that even a small change of code can translate into huge difference in behavior.

However, that past work largely concentrated on quantitative analysis of genotype-phenotype mapping, and in most cases was not constructive in leading to new search operators. Only recently more systematic approaches to this problem have emerged, which explicitly take into account program semantics, leading to the emerging area of *semantic genetic programming* (SGP). Contrary to conventional GP where fitness is the only computation effect of interest, SGP brings the effects of program execution to the foreground and inspects program behavior separately for particular examples, using the detailed information acquired in this way to improve search efficiency. The particular extensions based on SGP concern semantic-aware search operators [3, 5, 61, 52, 63, 28, 51, 32, 31, 65, 54], population initialization [4, 19] and selection [11]. SGP attracted great deal of attention in the GP community, evidenced by the recent successful workshop at the *Parallel Problem Solving from Nature* (PPSN) conference (`http://www.cs.put.poznan.pl/smgp2014/`) and special issue of the *Genetic Programming and Evolvable Machines* journal (`http://ncra.ucd.ie/GENP_SemanticMethodsinGP.pdf`).

In particular, our recent study [51] formally defines semantics of candidate programs and casts them on a multidimensional *semantic space*, which spans the dimensions defined by particular tests. This enables design of semantics-aware search operators and evaluation metrics that enjoy, among others, certain geometric properties of the resulting semantic spaces. Preliminary results with this so-called *geometric SGP* demonstrate dramatic improvement of success rate on a wide spectrum of benchmarks and attractive theoretical potential (guarantees of search progress).

Task description. Within this task, we plan to develop computationally efficient semantic-aware extensions of GP within the SGP framework, more specifically:

- Geometric semantic search operators, which exploit the geometric properties of semantic space while keeping program size within acceptable limit (which is the major challenge for the geometric semantic operators proposed to date). Our preliminary studies [54] suggests that this can be achieved by partial reversal of program execution.

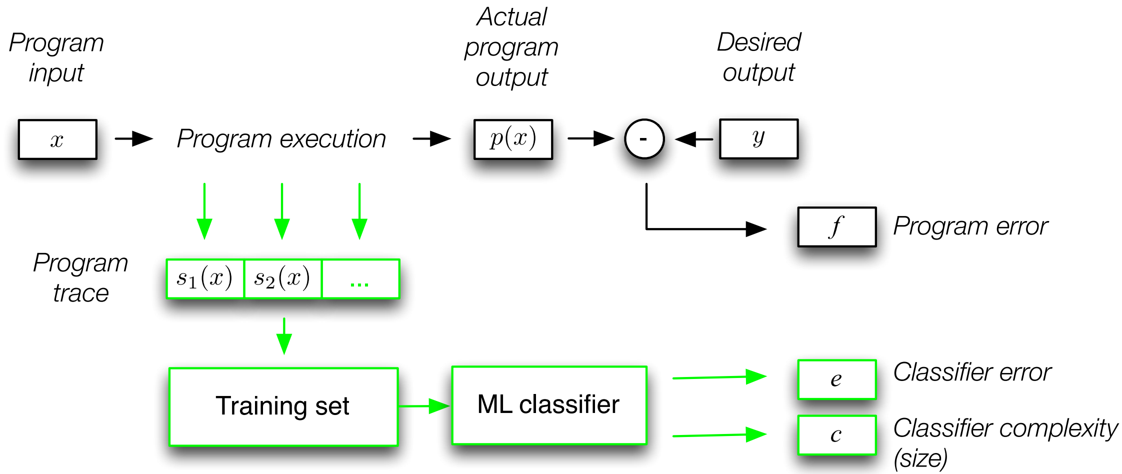

Figure 2: Behavioral evaluation of programs in genetic programming. A trace of an executing program is fed into a machine learning classifier. The classifier learns from multiple such traces of the same program and attempts to predict the desired output (as specified by the given programming task). The properties of the trained classifier (classification error and complexity) reflect the usefulness of the information revealed in the execution traces. Indirectly, they also indicate the quality of subprograms contained in the evaluated program (details in [35, 29]).

- Geometric initialization operators, which initialize the sample of programs (population) in a way that improves the convergence of search process to the optimal solution (or, put in other words, increases the likelihood of solving the programming task within the assumed computational budget). The key idea we propose to follow here is the concept of convex hull of semantics of programs in the population. Once the desired semantics is included in that hull, there is guarantee that the optimal solution can be constructed from those programs via crossover.
- Geometric, semantics-based mate selection, which pairs parents programs in a semantic-aware way, i.e., so as to increase the likelihood of the offspring program improving over the parents.

3.2 Task 2: Methods for behavioral evaluation of programs

Motivation. Conventionally in GP, program quality (fitness) depends only on the final effect of program execution, which it achieves after termination. The intermediate effects of program execution have no direct impact on its fitness. For instance, in the tree-based GP, only the value returned by the root node of program tree directly determines its fitness. The subtrees influence the program outcome only to the extent determined by the instructions located above them in the tree.

At first sight, this seems commonsensical: in the end, it is only the final outcome of program's computation that determines whether a programming task has been solved or not (or how well a program approximates the desired output). This remains however in stark contrast to the process of programming as exercised by humans. When faced with a nontrivial task, programmers split it into subtasks and attempt to solve them independently or semi-independently. This strategy proves effective, because human programmers often know in advance which *intermediate execution states* are desired. Consider for instance a programming task of synthesizing an algorithm that calculates the median of an list of numbers. In such a task, a desired intermediate state is the input list sorted in ascending or descending order[2].

The ability to decompose tasks allows human programmers to excel on tasks that automatic programming methods still struggle to solve. It is then highly desirable to equip the GP algorithms with an analogous capability, which we postulated in our previous studies [37, 25, 36]. Recently, we proposed *behavioral evaluation* [35, 29], an extension of conventional fitness function, which uses machine learning techniques to detect potentially valuable subprograms in program code by analyzing its execution traces when applied to the training tests (where by execution trace we mean the sequence of

[2]More efficient algorithms for median calculation exist, but for the sake of argument we stick here to the conventional approach that sorts the entire list.

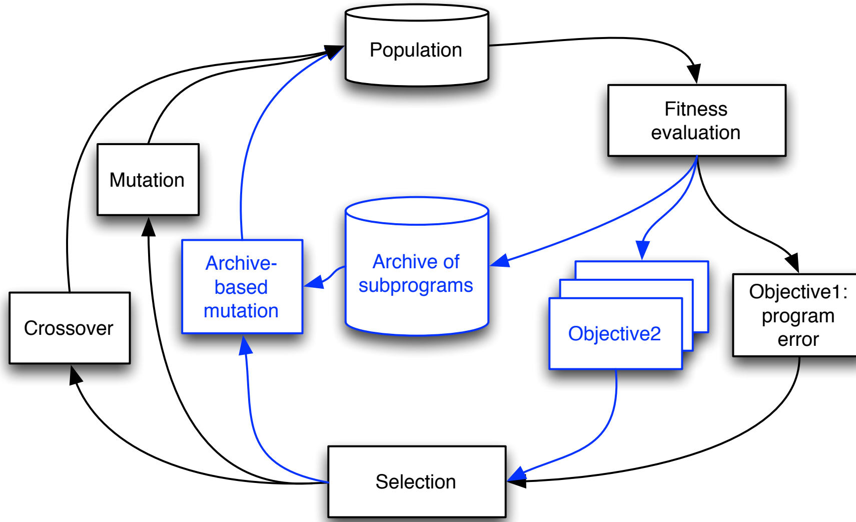

Figure 3: The workflow of a GP algorithm extended with behavioral evaluation and an archive of subprograms. The subprograms identified as promising by behavioral evaluation (cf. Fig. 2) are gathered in an archive, a repository of code fragments maintained throughout the entire process of evolutionary run. The code fragments are subsequently reused in the new candidate programs by search operators (archive-based mutation).

semantic states traversed by an executing program when applied to a specific input data). This concept is outlined in Fig. 2. By this virtue, behavioral evaluation can promote a program that features prospectively useful code fragments, even if such a program does not perform very well as a whole (note that semantic GP considered in Task 1 considers only final program outcomes). Early experimental results show that behavioral evaluation measures provide better correlation with the actual distance from the optimal program that conventional fitness [30]. In our recent publication [29], awarded with the Best Paper Award at the Genetic and Evolutionary Computation (GECCO) conference in July 2014, we extended this idea with an archive of subprograms (cf. Fig. 3). This lead to further dramatic improvements and solving several problem instances that conventional GP algorithms hardly ever solve.

Task description. In this task, we plan to build upon and employ the idea of behavioral evaluation to perform implicit *decomposition* of a programming task and use it as a means to achieve better scalability of program synthesis. More specifically we intend to:

- Investigate other (than machine learning-based) methods for detecting regularities in program traces that reveal valuable intermediate program behaviors. This may include, among others, information theory-based measures.
- Design methods that explicitly detect analogies between program behavior on particular tests, and use them to modify programs in a directed way. The tentative idea is to employ commutative diagrams for this purpose.
- Generalize the concept of behavioral evaluation to Turing-complete programs, and verify the feasibility of this concept on the more difficult tasks of program synthesis. The challenge here is the fact that detecting potentially valuable subprograms becomes hard when programs are allowed to use conditional statements and loops, and as such requires a substantially different approach.

3.3 Task 3: Alternative search drivers

Motivation. The conventional objective functions designed to evaluate candidate programs in automated program synthesis (fitness functions in GP) feature low fitness-distance correlation [60] (where distance is the number of search steps required to reach the optimal solution-program). This implies in practice that they are not necessarily good at guiding the search process towards the useful parts of search space.

As an illustration of this issue, let us reconsider the automated synthesis of a sorting program mentioned at the end of Section 1.1. For a given input list, an ideal program is expected to return the corresponding list sorted in ascending order. An exemplary test for this problem could be thus composed of program input (9,3,7,1,6) and desired output (1,3,6,7,9). Assume now that evolutionary search happens to produce a program that sorts the array in a reverse order, i.e., returns (9,7,6,3,1) for this particular fitness case. In terms of error, measured for instance by Kendall rank correlation coefficient, this program performs particularly bad and will be penalized. Nevertheless, it is likely that a very small modification, possibly a single replacement of the arithmetic comparison operator '<' with operator '>' somewhere in the program code, will render it optimal. In other words, a very small modification (technically, a single GP mutation) may turn an utterly bad candidate solution into an ideal one. Two programs distanced by just one search step may exhibit a completely different behavior. In this sense, many program synthesis problems are *deceptive*, by analogy to famous deceptive problems considered in generic evolutionary computation, like trap functions, NK-landscapes, or Hierarchical If-and-only-if (HIFF) problems.

This example clearly illustrates that the objective function, despite reflecting the objective quality of candidate solutions, is not necessarily the best 'driver' to guide the search. Blind reliance of program error as the only objective to be followed during search process can be detrimental. Other means of guiding search should be sought for. In the above example of sorting program, an alternative search driver could for instance reward the candidate programs for *any* ordering of list elements, whether ascending or descending; in this way, the abovementioned 'close miss' program would be potentially promoted and likely mutated into an optimal program. Ideally, an alternative search driver would not be designed by hand, but discovered by an APS algorithm online, while solving the problem.

Task description. To address this problem, we propose the concept of *search driver*, meant as an imperfect substitute for objective function that correlates reasonably well with the distance yet reflects only selected aspects of the quality of candidate solution. In the context of GP, where candidate solutions are evaluated on *multiple* tests, a natural example of a search driver is an evaluation measure that captures program's performance only on a *subset* of tests. A search driver can be also 'subjective' in the sense of reflecting program's performance only *relatively* to the other programs in the population, in a way characteristic for coevolutionary algorithms [8].

By analogy with classifier ensembles in machine learning, we posit that in the difficult program synthesis tasks, a search guided by multiple such 'weak' search drivers can be more efficient than the search guided by conventional objective function. In a recent work [45], we demonstrated that this can be indeed true when evolving a game strategy for the famous Iterated Prisoner's Dilemma problem [10]. A submission currently in review verified this positively also in the area of GP (work submitted to EvoStar'15 conference).

The concept of search driver may sound similar to *surrogate fitness function* [6, 16]. However, surrogate fitness function is usually designed manually by a human expert to mitigate the high computational cost of the original fitness function, and intended to mimic the original fitness as close as possible. Search driver, on the other hand, is usually automatically derived from the problem (or from the outcomes of search algorithm's interactions with the problem), and is not assumed to approximate the original fitness function. Other related concepts are *multiobjectivization* of problems [22] and automatic extraction of objectives in test-based problems [20].

In this project task, we would like to:

- Formalize the concept of search driver. In general, we allow search drivers to be partial, i.e., applicable only in parts of search space. In this way, we enable division of competence across the search space.
- Define features that a search driver has to exhibit to be effective at guiding a search process.
- Delineate the class of problems that can be efficiently solved using search drivers.
- Devise a method for automatic *learning* of such search drivers from the interactions between candidate solutions and tests, which proceeds on-line with the iterative program synthesis.

3.4 Task 4: Experimental verification

Motivation. By focusing on the fundamental issues of program synthesis and formulating problems in generic manner, the proposed project will pursue basic research. Nevertheless, the algorithms and methods proposed here cannot be gauged otherwise than by applying them to some test problems of scientific and/or practical importance.

Task description. In this task, we will demonstrate the concepts and algorithms developed in Tasks 1–3 on the following types of problems:

- symbolic regression tasks, where the objective is to synthesize a formula that realizes a mapping from one or more continuous input variables to a continuous dependent variable,
- classification tasks, where an evolved program is expected to classify the inputs to one of the predefined decision classes (as in conventional machine learning); this includes the special case of synthesis of Boolean functions,
- synthesis of programs implementing operations on lists, like concatenation, reversal, sorting,
- selected problems in search-based software engineering, , in particular improvement of non-functional program properties (like execution time or memory occupancy).

Another material outcome of this tasks, in combination with the remaining ones, will be a library of evolutionary algorithms for program synthesis, licensed according to Open Source standards. The prototype of that library, written in the powerful object-oriented and functional programming language Scala[3], is already under preparation.

4 Methodology

The topics formulated in Tasks 1, 2, and 3 will be executed by following the methodology typical for research in metaheuristics and machine learning. New algorithms will be devised, targeted at the weaknesses of the contemporary methods (as identified in Tasks 1–3). The algorithms will be subject to computer implementation and testing according to good practices of software engineering. Only then will they be verified on the existing suites of problems, mentioned in the description of Task 4. The primary source of benchmarks will be the GP-benchmark initiative of the community of GP researchers (`http://gpbenchmarks.org/`). Where possible, formal background will be developed and hybrids of the above concepts will be considered.

The algorithms developed here will be objectively assessed using well-defined performance indicators, such as success rate, time-to-success, and program error. Stochastic algorithms will obviously require multiple runs to provide for statistical significance. The obtained values will undergo rigorous analysis using appropriate statistical tests and procedures. In every experiment, an appropriate control setup (baseline) will be provided to ease the assessment of the effect size of the extension in question.

The results will be thoroughly scrutinized and disseminated via presentations at respected conferences (ACM GECCO, EvoStar, PPSN) and publications in indexed scientific journals (*IEEE Trans. Evol. Comp., Genetic Programming and Evolvable Machines, Evolutionary Computation Journal*). A website describing the project and reporting its major results will be set up and maintained during project lifetime and possibly for a couple of years after its completion.

The Institute of Computing Science and the Faculty of Computing host appropriate computer infrastructure required to conduct the research planned in this project. In particular, several computer laboratories are integrated into a unified framework of distributed computing via SLURM (Simple Linux Utility for Resource Management [66]). Our previous experience suggests that this infrastructure is sufficient for conducting the research plan envisioned in this proposal. For this reason, we do not plan purchasing hardware equipment in this project.

[3] `http://www.scala-lang.org/`

References

[1] Wolfgang Banzhaf, Guillaume Beslon, Steffen Christensen, James Foster, Francois Kepes, Virginie Lefort, Julian Miller, Miroslav Radman, and Jeremy J. Ramsden. From artificial evolution to computational evolution: A research agenda. *Nature Reviews Genetics*, 7(9):729–735, September 2006.

[2] Howard Barnum, Herbert J Bernstein, and Lee Spector. Quantum circuits for OR and AND of ORs. *Journal of Physics A: Mathematical and General*, 33(45):8047–8057, 17 November 2000.

[3] Lawrence Beadle and Colin Johnson. Semantically driven crossover in genetic programming. In *Proceedings of the IEEE World Congress on Computational Intelligence*, pages 111–116, Hong Kong, 1-6 June 2008. IEEE Computational Intelligence Society, IEEE Press.

[4] Lawrence Beadle and Colin G. Johnson. Semantic analysis of program initialisation in genetic programming. *Genetic Programming and Evolvable Machines*, 10(3):307–337, September 2009.

[5] Lawrence Beadle and Colin G Johnson. Semantically driven mutation in genetic programming. In *2009 IEEE Congress on Evolutionary Computation*, pages 1336–1342, Trondheim, Norway, 18-21 May 2009. IEEE Computational Intelligence Society, IEEE Press.

[6] A.J. Booker, Jr. Dennis, J.E., P.D. Frank, D.B. Serafini, V. Torczon, and M.W. Trosset. A rigorous framework for optimization of expensive functions by surrogates. *Structural optimization*, 17(1):1–13, 1999.

[7] Markus Brameier and Wolfgang Banzhaf. *Linear Genetic Programming*. Number XVI in Genetic and Evolutionary Computation. Springer, 2007.

[8] Anthony Bucci and Jordan B. Pollack. A mathematical framework for the study of coevolution. In Kenneth A. De Jong, Riccardo Poli, and Jonathan E. Rowe, editors, *Foundations of Genetic Algorithms 7*, pages 221–236. Morgan Kaufmann, San Francisco, 2003.

[9] David M. Clark. Evolution of algebraic terms 1: Term to term operation continuity. *International Journal of Algebra and Computation*, 23(05):1175–1205, August 2013.

[10] Paul J Darwen and Xin Yao. Why more choices cause less cooperation in iterated prisoner's dilemma. In *Evolutionary Computation, 2001. Proceedings of the 2001 Congress on*, volume 2, pages 987–994. IEEE, 2001.

[11] Edgar Galvan-Lopez, Brendan Cody-Kenny, Leonardo Trujillo, and Ahmed Kattan. Using semantics in the selection mechanism in genetic programming: a simple method for promoting semantic diversity. In *2013 IEEE Conference on Evolutionary Computation*, volume 1, pages 2972–2979, Cancun, Mexico, June 20-23 2013.

[12] David E. Goldberg. *Genetic Algorithms in Search, Optimization and Machine Learning*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1st edition, 1989.

[13] Mark Harman. Automated patching techniques: The fix is in. *Communications of the ACM*, 53(5):108, June 2010.

[14] Mark Harman. Software engineering meets evolutionary computation. *Computer*, 44(10):31–39, October 2011. Cover feature.

[15] Mark Harman, Ub Ph, and Bryan F. Jones. Search-based software engineering. *Information and Software Technology*, 43:833–839, 2001.

[16] Torsten Hildebrandt and Juergen Branke. On using surrogates with genetic programming. *Evolutionary Computation*. Forthcoming.

[17] John Henry Holland. *Adaptation in natural and artificial systems: an introductory analysis with applications to biology, control, and artificial intelligence*. Bradford Books. MIT Press, 1992.

[18] Gregory. S. Hornby, Jason D. Lohn, and Derek S. Linden. Computer-automated evolution of an X-band antenna for NASA's space technology 5 mission. *Evolutionary Computation*, 19(1):1–23, Spring 2011.

[19] David Jackson. Phenotypic diversity in initial genetic programming populations. In *Proceedings of the 13th European Conference on Genetic Programming, EuroGP 2010*, volume 6021 of *LNCS*, pages 98–109, Istanbul, 7-9 April 2010. Springer.

[20] Wojciech Jaskowski. *Algorithms for Test-Based Problems*. PhD thesis, Institute of Computing Science, Poznan University of Technology, Poznan, Poland, May 2011.

[21] Kenneth E. Kinnear, Jr. Fitness landscapes and difficulty in genetic programming. In *Proceedings of the 1994 IEEE World Conference on Computational Intelligence*, volume 1, pages 142–147, Orlando, Florida, USA, 27-29 June 1994. IEEE Press.

[22] Joshua D. Knowles, Richard A. Watson, and David Corne. Reducing local optima in single-objective problems by multi-objectivization. In *EMO '01: Proceedings of the First International Conference on Evolutionary Multi-Criterion Optimization*, pages 269–283, London, UK, 2001. Springer-Verlag.

[23] John R. Koza. *Genetic Programming: On the Programming of Computers by Means of Natural Selection*. MIT Press, Cambridge, MA, USA, 1992.

[24] John R. Koza. *Genetic Programming: On the Programming of Computers by Means of Natural Selection*. MIT Press, Cambridge, MA, USA, 1992.

[25] Krzysztof Krawiec. On relationships between semantic diversity, complexity and modularity of programming tasks. In Terry Soule, Anne Auger, Jason Moore, David Pelta, Christine Solnon, Mike Preuss, Alan Dorin, Yew-Soon Ong, Christian Blum, Dario Landa Silva, Frank Neumann, Tina Yu, Aniko Ekart, Will Browne, Tim Kovacs, Man-Leung Wong, Clara Pizzuti, Jon Rowe, Tobias Friedrich, Giovanni Squillero, Nicolas Bredeche, Stephen Smith, Alison Motsinger-Reif, Jose Lozano, Martin Pelikan, Silja Meyer-Nienberg, Christian Igel, Greg Hornby, Rene Doursat, Steve Gustafson, Gustavo Olague, Shin Yoo, John Clark, Gabriela Ochoa, Gisele Pappa, Fernando Lobo, Daniel Tauritz, Jurgen Branke, and Kalyanmoy Deb, editors, *GECCO '12: Proceedings of the fourteenth international conference on Genetic and evolutionary computation conference*, pages 783–790, Philadelphia, Pennsylvania, USA, 7-11 July 2012. ACM.

[26] Krzysztof Krawiec and Bir Bhanu. Visual learning by coevolutionary feature synthesis. *IEEE Transactions on System, Man, and Cybernetics – Part B*, 35(3):409–425, June 2005.

[27] Krzysztof Krawiec and Bir Bhanu. Visual learning by evolutionary and coevolutionary feature synthesis. *IEEE Transactions on Evolutionary Computation*, 11(5):635–650, October 2007.

[28] Krzysztof Krawiec and Pawel Lichocki. Approximating geometric crossover in semantic space. In *GECCO '09: Proceedings of the 11th Annual conference on Genetic and evolutionary computation*, pages 987–994, Montreal, 8-12 July 2009. ACM.

[29] Krzysztof Krawiec and Una-May O'Reilly. Behavioral programming: a broader and more detailed take on semantic GP. In Christian Igel, Dirk V. Arnold, Christian Gagne, Elena Popovici, Anne Auger, Jaume Bacardit, Dimo Brockhoff, Stefano Cagnoni, Kalyanmoy Deb, Benjamin Doerr, James Foster, Tobias Glasmachers, Emma Hart, Malcolm I. Heywood, Hitoshi Iba, Christian Jacob, Thomas Jansen, Yaochu Jin, Marouane Kessentini, Joshua D. Knowles, William B. Langdon, Pedro Larranaga, Sean Luke, Gabriel Luque, John A. W. McCall, Marco A. Montes de Oca, Alison Motsinger-Reif, Yew Soon Ong, Michael Palmer, Konstantinos E. Parsopoulos, Guenther Raidl, Sebastian Risi, Guenther Ruhe, Tom Schaul, Thomas Schmickl, Bernhard Sendhoff, Kenneth O. Stanley, Thomas Stuetzle, Dirk Thierens, Julian Togelius, Carsten Witt, and Christine Zarges, editors, *GECCO '14: Proceedings of the 2014 conference on Genetic and evolutionary computation*, pages 935–942, Vancouver, BC, Canada, 12-16 July 2014. ACM. Best paper.

[30] Krzysztof Krawiec and Una-May O'Reilly. Behavioral search drivers for genetic programing. In Miguel Nicolau, Krzysztof Krawiec, Malcolm I. Heywood, Mauro Castelli, Pablo Garcia-Sanchez, Juan J. Merelo, Victor M. Rivas Santos, and Kevin Sim, editors, *17th European Conference on Genetic Programming*, volume 8599 of *LNCS*, pages 210–221, Granada, Spain, 23-25 April 2014. Springer.

[31] Krzysztof Krawiec and Tomasz Pawlak. Approximating geometric crossover by semantic backpropagation. In *GECCO '13: Proceeding of the fifteenth annual conference on Genetic and evolutionary computation conference*, pages 941–948, Amsterdam, The Netherlands, 6-10 July 2013. ACM.

[32] Krzysztof Krawiec and Tomasz Pawlak. Locally geometric semantic crossover: a study on the roles of semantics and homology in recombination operators. *Genetic Programming and Evolvable Machines*, 14(1):31–63, March 2013.

[33] Krzysztof Krawiec and Jerry Swan. Guiding evolutionary learning by searching for regularities in behavioral trajectories: A case for representation agnosticism. In Sebastian Risi, Joel Lehman, and Jeff Clune, editors, *How Should Intelligence Be Abstracted in AI Research: MDPs, Symbolic Representations, Artificial Neural Networks, or ...*, number FS-13-02 in 2013 AAAI Fall Symposium Series, pages 41–46, Arlington, Virginia, USA, 15-17 November 2013. AAAI Press.

[34] Krzysztof Krawiec and Jerry Swan. Pattern-guided genetic programming. In Christian Blum et al., editor, *GECCO '13: Proceeding of the fifteenth annual conference on Genetic and evolutionary computation conference*, pages 949–956, Amsterdam, The Netherlands, 6-10 July 2013. ACM.

[35] Krzysztof Krawiec and Jerry Swan. Pattern-guided genetic programming. In Christian Blum, En-

rique Alba, Anne Auger, Jaume Bacardit, Josh Bongard, Juergen Branke, Nicolas Bredeche, Dimo Brockhoff, Francisco Chicano, Alan Dorin, Rene Doursat, Aniko Ekart, Tobias Friedrich, Mario Giacobini, Mark Harman, Hitoshi Iba, Christian Igel, Thomas Jansen, Tim Kovacs, Taras Kowaliw, Manuel Lopez-Ibanez, Jose A. Lozano, Gabriel Luque, John McCall, Alberto Moraglio, Alison Motsinger-Reif, Frank Neumann, Gabriela Ochoa, Gustavo Olague, Yew-Soon Ong, Michael E. Palmer, Gisele Lobo Pappa, Konstantinos E. Parsopoulos, Thomas Schmickl, Stephen L. Smith, Christine Solnon, Thomas Stuetzle, El-Ghazali Talbi, Daniel Tauritz, and Leonardo Vanneschi, editors, *GECCO '13: Proceeding of the fifteenth annual conference on Genetic and evolutionary computation conference*, pages 949–956, Amsterdam, The Netherlands, 6-10 July 2013. ACM.

[36] Krzysztof Krawiec and Jerry Swan. Pattern-guided genetic programming. In *Proceedings of the 15th international conference on Genetic and evolutionary computation conference*, GECCO '13, Amsterdam, The Netherlands, 2013. ACM.

[37] Krzysztof Krawiec and Bartosz Wieloch. Analysis of semantic modularity for genetic programming. *Foundations of Computing and Decision Sciences*, 34(4):265–285, 2009.

[38] W. B. Langdon and S. M. Gustafson. Genetic programming and evolvable machines: ten years of reviews. *Genetic Programming and Evolvable Machines*, 11(3/4):321–338, September 2010. Tenth Anniversary Issue: Progress in Genetic Programming and Evolvable Machines.

[39] William B. Langdon. A bibliography for genetic programming. In Peter J. Angeline and K. E. Kinnear, Jr., editors, *Advances in Genetic Programming 2*, appendix B, pages 507–531. MIT Press, Cambridge, MA, USA, 1996.

[40] William B. Langdon and Mark Harman. Optimising existing software with genetic programming. *IEEE Transactions on Evolutionary Computation*. Accepted.

[41] William B. Langdon and Mark Harman. Genetically improved CUDA C++ software. In Miguel Nicolau, Krzysztof Krawiec, Malcolm I. Heywood, Mauro Castelli, Pablo Garcia-Sanchez, Juan J. Merelo, Victor M. Rivas Santos, and Kevin Sim, editors, *17th European Conference on Genetic Programming*, volume 8599 of *LNCS*, pages 87–99, Granada, Spain, 23-25 April 2014. Springer.

[42] William B. Langdon, Mark Harman, and Yue Jia. Efficient multi-objective higher order mutation testing with genetic programming. *Journal of Systems and Software*, 83(12):2416–2430, December 2010.

[43] Nada Lavrac and Saso Dzeroski. *Inductive logic programming - techniques and applications*. Ellis Horwood series in artificial intelligence. Ellis Horwood, 1994.

[44] Hod Lipson and Jordan B. Pollack. Automatic design and manufacture of robotic lifeforms. *Nature*, (406):974–978, 31 August 2000.

[45] Paweł Liskowski and Krzysztof Krawiec. Discovery of implicit objectives by compression of interaction matrix in test-based problems. In Thomas Bartz-Beielstein, Jürgen Branke, Bogdan Filipič, and Jim Smith, editors, *Parallel Problem Solving from Nature – PPSN XIII*, volume 8672 of *Lecture Notes in Computer Science*, pages 611–620. Springer, 2014.

[46] Jason D. Lohn and Gregory S. Hornby. Evolvable hardware using evolutionary computation to design and optimize hardware systems. *IEEE Computational Intelligence Magazine*, 1(1):19–27, February 2006.

[47] Zohar Manna and Richard Waldinger. Fundamentals of deductive program synthesis. *IEEE Transactions on Software Engineering*, 18:674–704, 1992.

[48] Paul Massey, John A. Clark, and Susan Stepney. Human-competitive evolution of quantum computing artefacts by genetic programming. *Evolutionary Computation*, 14(1):21–40, Spring 2006. Best of GECCO 2004 special issue.

[49] James McDermott, David R. White, Sean Luke, Luca Manzoni, Mauro Castelli, Leonardo Vanneschi, Wojciech Jaskowski, Krzysztof Krawiec, Robin Harper, Kenneth De Jong, and Una-May O'Reilly. Genetic programming needs better benchmarks. In *GECCO '12: Proceedings of the fourteenth international conference on Genetic and evolutionary computation conference*, pages 791–798, Philadelphia, Pennsylvania, USA, 7-11 July 2012. ACM.

[50] Julian F. Miller, editor. *Cartesian Genetic Programming*. Natural Computing Series. Springer, 2011.

[51] Alberto Moraglio, Krzysztof Krawiec, and Colin G. Johnson. Geometric semantic genetic programming. In *Parallel Problem Solving from Nature, PPSN XII (part 1)*, volume 7491 of *Lecture Notes in Computer Science*, pages 21–31, Taormina, Italy, September 1-5 2012. Springer.

[52] Quang Uy Nguyen, Xuan Hoai Nguyen, and Michael O'Neill. Semantics based mutation in genetic

programming: The case for real-valued symbolic regression. In R. Matousek and L. Nolle, editors, *15th International Conference on Soft Computing, Mendel'09*, pages 73–91, Brno, Czech Republic, June 24-26 2009.

[53] Su Nguyen, Mengjie Zhang, Mark Johnston, and Kay Chen Tan. Dynamic multi-objective job shop scheduling: A genetic programming approach. In A. Sima Uyar, Ender Ozcan, and Neil Urquhart, editors, *Automated Scheduling and Planning*, volume 505 of *Studies in Computational Intelligence*, pages 251–282. Springer, 2013.

[54] Tomasz P. Pawlak, Bartosz Wieloch, and Krzysztof Krawiec. Semantic backpropagation for designing search operators in genetic programming. *IEEE Trans. on Evol. Computation*, 2014.

[55] Riccardo Poli, William B. Langdon, and Nicholas Freitag McPhee. *A field guide to genetic programming.* Published via `http://lulu.com` and freely available at `http://www.gp-field-guide.org.uk`, 2008. (With contributions by J. R. Koza).

[56] Franz Rothlauf. *Representations for genetic and evolutionary algorithms.* Springer, pub-SV:adr, second edition, 2006. First published 2002, 2nd edition available electronically.

[57] Michael Schmidt and Hod Lipson. Distilling free-form natural laws from experimental data. *Science*, 324(5923):81–85, 3 April 2009.

[58] Lee Spector, David M. Clark, Ian Lindsay, Bradford Barr, and Jon Klein. Genetic programming for finite algebras. In Maarten Keijzer, Giuliano Antoniol, Clare Bates Congdon, Kalyanmoy Deb, Benjamin Doerr, Nikolaus Hansen, John H. Holmes, Gregory S. Hornby, Daniel Howard, James Kennedy, Sanjeev Kumar, Fernando G. Lobo, Julian Francis Miller, Jason Moore, Frank Neumann, Martin Pelikan, Jordan Pollack, Kumara Sastry, Kenneth Stanley, Adrian Stoica, El-Ghazali Talbi, and Ingo Wegener, editors, *GECCO '08: Proceedings of the 10th annual conference on Genetic and evolutionary computation*, pages 1291–1298, Atlanta, GA, USA, 12-16 July 2008. ACM.

[59] Lee Spector and Alan Robinson. Genetic programming and autoconstructive evolution with the push programming language. *Genetic Programming and Evolvable Machines*, 3(1):7–40, March 2002.

[60] Marco Tomassini, Leonardo Vanneschi, Philippe Collard, and Manuel Clergue. A study of fitness distance correlation as a difficulty measure in genetic programming. *Evolutionary Computation*, 13(2):213–239, Summer 2005.

[61] Nguyen Quang Uy, Nguyen Xuan Hoai, Michael O'Neill, R. I. McKay, and Edgar Galvan-Lopez. Semantically-based crossover in genetic programming: application to real-valued symbolic regression. *Genetic Programming and Evolvable Machines*, 12(2):91–119, June 2011.

[62] Nguyen Quang Uy, Nguyen Xuan Hoai, Michael O'Neill, R. I. McKay, and Dao Ngoc Phong. On the roles of semantic locality of crossover in genetic programming. *Information Sciences*, 235:195–213, 20 June 2013.

[63] Nguyen Quang Uy, Nguyen Xuan Hoai, Michael O'Neill, R. I. McKay, and Dao Ngoc Phong. On the roles of semantic locality of crossover in genetic programming. *Information Sciences*, 235:195–213, 20 June 2013.

[64] Westley Weimer, Stephanie Forrest, Claire Le Goues, and ThanhVu Nguyen. Automatic program repair with evolutionary computation. *Communications of the ACM*, 53(5):109–116, June 2010.

[65] Bartosz Wieloch and Krzysztof Krawiec. Running programs backwards: instruction inversion for effective search in semantic spaces. In *GECCO '13: Proceeding of the fifteenth annual conference on Genetic and evolutionary computation conference*, pages 1013–1020, Amsterdam, The Netherlands, 6-10 July 2013. ACM.

[66] Andy B Yoo, Morris A Jette, and Mark Grondona. Slurm: Simple linux utility for resource management. In *Job Scheduling Strategies for Parallel Processing*, pages 44–60. Springer, 2003.