Evolutionary Program Sketching

Iwo Błądek and Krzysztof Krawiec

Institute of Computing Science, Poznan University of Technology
Piotrowo 2, 60-965 Poznań, Poland
ibladek@cs.put.poznan.pl, krawiec@cs.put.poznan.pl

Abstract. Program synthesis can be posed as a satisfiability problem and approached with generic SAT solvers. Only short programs can be however synthesized in this way. Program sketching by Solar-Lezama assumes that a human provides a partial program (*sketch*), and that synthesis takes place only within the uncompleted parts of that program. This allows synthesizing programs that are overall longer, while maintaining manageable computational effort. In this paper, we propose Evolutionary Program Sketching (EPS), in which the role of sketch provider is handed over to genetic programming (GP). A GP algorithm evolves a population of partial programs, which are being completed by a solver while evaluated. We consider several variants of EPS, which vary in program terminals used for completion (constants, variables, or both) and in the way the completion outcomes are propagated to future generations. When applied to a range of benchmarks, EPS outperforms the conventional GP, also when the latter is given similar time budget.
Keywords: program synthesis, satisfiability modulo theory, program sketching, genetic programming.

1 Introduction

Program synthesis (PS), as many other search tasks, can be posed as a *satisfiability problem*: given a *contract*, i.e., a logical predicate that describes the desired input-output behavior, and an encoding of program's structure parametrized with Boolean variables, determine whether a valuation of those variables exists that makes the program satisfy the contract. To obtain this valuation, called in propositional logic a *model*, the synthesis formula is passed to a SAT *solver*, which produces a feasible variable assignment, and thus a program that is guaranteed to meet the contract, or otherwise states that the sought program does not exist. In practice, the solver is equipped with an additional abstraction layer, a *theory* that enables reasoning in terms of, for instance, integer arithmetic. This leads to the concept of *satisfiability modulo theories* (SMT) used in a range of past works on program synthesis [5,6,18].

SMT solvers implement exact algorithms supported by heuristics which are guaranteed to find the sought program in the prescribed search space (unless they time out). Nevertheless, the above approach to PS suffers from poor scalability: the assumed maximum length of a synthesized program determines the number of involved variables, and search space grows exponentially with that

number. Additionally, solving an SMT problem involves solving a SAT problem as a subtask, which is known to be NP-complete. Even with contemporary sophisticated SMT solvers, only short programs can be synthesized with this approach in a reasonable time.

To address this problem, Solar-Lezama proposed *program sketching* [16]. Therein, one assumes that a partial program (*sketch*) is provided, with one or more *holes* marking the locations of missing code pieces. The synthesis takes place only in the holes, while the sketch remains intact. In this way, the total length of the program (sketch length plus the length of the synthesized code pieces) is increased, while the task remains manageable for the solver.

In the original sketching, it is assumed that a human provides the sketch. Indeed, there are plausible scenarios in which a programmer may come up with an overall program structure, yet fails to implement all the details. This endows sketching with certain interactive flavor, which is often desirable in software development. On the other hand, human-provided sketch can be suboptimal for completion, or in an extreme case incorrect, i.e., such that cannot be completed to satisfy the contract.

Evolutionary Program Sketching (EPS) we propose in this paper substitutes the human sketcher with an evolutionary process. A GP algorithm evolves partial programs with holes. When evaluating a program, the solver attempts to complete the holes with code pieces; in this preliminary study, we fill the holes with constants and input variables. Fitness measures the extent to which the holes can be completed to meet the contract. We consider several variants of EPS, which vary in the way they handle code completions, in particular in whether the holes persist after evaluation. Experimental verification proves EPS feasible and points to interesting potential extensions of this approach.

2 Program Sketching

Presentation of program sketching requires brief introduction to SMT-based program synthesis. The synthesis task is given by a *contract*, typically a pair of logical formulas: a *precondition* Pre – the constraint imposed on program input, and a *postcondition* $Post$ – a logical clause that should hold upon program completion. The content (code) of a candidate program is controlled by a vector of variables $\mathbf{b}$. For instance, when synthesizing sequential programs n instructions long, with each instruction taken from an instruction set of cardinality k, one would assume $\mathbf{b} \in [1, k]^n$, i.e., that each program corresponds one-to-one to a vector of n variables controlling the choices of instructions on particular positions.

Let $p_{\mathbf{b}}$ denote a program determined by a specific vector $\mathbf{b}$, and let $p_{\mathbf{b}}(in)$ denote the output produced by $p_{\mathbf{b}}$ when applied to input in. Solving a synthesis task $(Pre, Post)$ is equivalent to proving that

$$\exists_{\mathbf{b}} \forall_{in} Pre(in) \implies Post(in, p_{\mathbf{b}}(in)), \tag{1}$$

where $Pre(in)$ is the precondition valuated for the input in, and $Post(in, p_{\mathbf{b}}(in))$ is the postcondition valuated for the input in and the output produced by $p_{\mathbf{b}}$ for in.

For illustration, consider synthesizing a program that calculates the maximum of two integers (x, y). For this synthesis task, the contract is defined as follows:

$$\begin{aligned} Pre((x,y)) &\iff (x,y) \in \mathbb{Z}^2 \\ Post((x,y),o) &\iff o \in \mathbb{Z} \wedge o \geq x \wedge o \geq y \wedge (o = x \vee o = y) \end{aligned} \tag{2}$$

This is an example of *a complete specification*, which defines the desired behavior of the sought program for *all possible inputs*, the number of which happens to be infinite here. If programs are expressed in terms of the theory known to the solver, the solver can prove (1), and so determine $\mathbf{b}$ and the sought program $p_{\mathbf{b}}$. The solver achieves this without actually running any program, because the properties of the output can be logically, *modulo the theory*, inferred from the properties of the input and the properties of program code. For the above problem to be solved, it would be sufficient to provide the solver with the Linear Integer Arithmetic (LIA) theory [3]. The resulting synthesized program $p_{\mathbf{b}}$ is guaranteed to adhere to the contract or, in other words, it is correct by construction.

This approach to synthesis, as elegant as it seems, is nevertheless feasible only when the sought program is short. As the above example of sequential programs shows, the cardinality of the search space grows exponentially with program length n, and even the modern SMT solvers, equipped with sophisticated heuristics for prioritizing search, become quickly computationally inefficient.

Program sketching [16,17] extends the effective program length while keeping the computational expense of synthesis at bay. This is achieved by assuming that the sought program is to some extent fixed, and the fixed part forms a *sketch*, a template that should be completed by a solver. For the $\max(x, y)$ synthesis problem mentioned above, a template could have the following form:

$$\text{if } (h_1) \text{ then } h_2 \text{ else } h_3, \tag{3}$$

where h_1, h_2 and h_3 are *holes*, i.e. program parts not specified by the sketch. Crucially, only the instructions in the holes can be varied by manipulating the control variables in $\mathbf{b}$. The number and domains of these control variables depend on assumed structure for the missing code. This structure is usually defined by a grammar, and variables in $\mathbf{b}$ determine the traversal of production rules of that grammar.

Example 1. Consider the $\max(x, y)$ problem presented above. The user starts by constructing a sketch of the solution as in (3). She assumes that h_1 should be filled with a Boolean expression of the form $var\ op\ var$, where op is an arithmetic operator ($\{>, =, <\}$) and vars are either x or y, and that both h_2 and h_3 should be also filled with var. All such hole completions can be enumerated (encoded) with five variables: four binary variables that control whether x or y should be

be filled in for h_2, h_3, and for both sides of *op*, and one ternary variable that controls the choice of *op*. These five variables together determine the search space for sketching (of size $2^4 \cdot 3 = 48$) and form the vector $\mathbf{b}$ that is controlled by the solver. □

We present here only the aspects that are relevant for this paper; the original program sketching involves more mechanisms, among them maintaining a set of test cases and augmenting them with counterexamples produced by the solver. Overall, sketching has multiple merits: it not only increases the effective program length, but delegates some control on the synthesis process to a human. Allowing a user to express her *intent* in this way is often desirable in the practice of software development. Nevertheless, there are limitations too. A human might find it difficult to come up with a sketch featuring a number of holes small enough for a solver to find a solution in an acceptable time. The provided sketch can be suboptimal in not forming the most elegant (or the shortest) solution to a given problem, or not enabling the solver to find the solution fast enough. In the worst scenario, a human may propose a wrong sketch, which cannot be completed so as to satisfy the contract. Evolutionary Program Sketching detailed in the next section addresses some of these issues.

3 Evolutionary Program Sketching

EPS evolves partial programs (sketches) and evaluates them based on the substitution for the missing parts determined by an SMT solver. The workflow of the method is presented in Fig. 1, and in the following we detail its key components.

3.1 Problem specification

As the conventional GP, EPS assumes that a synthesis problem is given by a set of instructions I of which the programs are to be built, and a set of examples T (tests) on which the programs are evaluated. Each example is a pair (in, out) of an input *in* and the corresponding desired output *out*. Such specification is *partial*: the universal quantifier in (2) is bound to T ($\forall_{(in,out)\in T}$), the precondition is always *true*, and the postcondition simply checks whether $p_b(in) = out$. This stands in contrast to the complete formal specification of the $\max(x, y)$ example in the previous section, and places GP and EPS in the realm of inductive approaches to PS, where the behavior of the synthesized program beyond the set of examples cannot be in general predicted.

3.2 Instruction set

As in sketching (Section 2), we allow for incomplete (partial) programs. To this aim, we extend the instruction set I with a set of terminal instructions H containing a symbol for each kind of hole allowed in a program. The kind of a

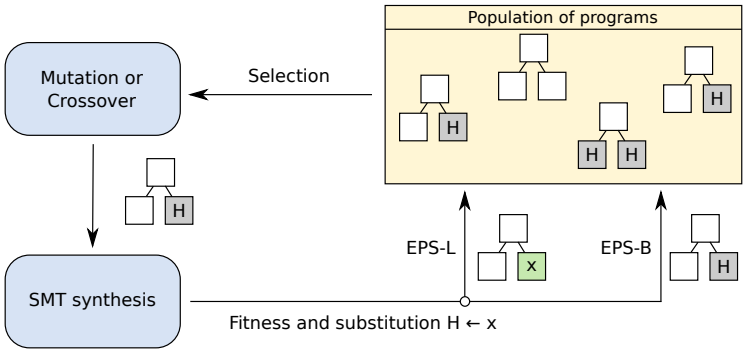

Fig. 1. The flow of candidate programs in EPS. H represents a hole, and x the content assigned to the hole. The EPS-L and EPS-B variants are described in Section 3.4.

hole determines the content it may be filled with, e.g. an integer or Boolean expression, or linear or nonlinear arithmetic expression. In conclusion, the GP process works with the set of instructions $I \cup H$, where each $h \in H$ is treated like other terminal symbols (i.e., it is subject to search operators, and in the case of strongly-typed GP it has an assigned type).

3.3 Fitness function

Partial programs are incomplete and thus cannot be evaluated in the common GP fashion, i.e. by executing them on examples in T. This is, however, not a problem if evaluation is to be based on a query to an SMT solver which can substitute the missing code pieces, as in the original sketching. But the outcome of the completion process described in Section 2 is only twofold: either a perfect completion (and thus a correct program) is found and the search terminates, or there is no feasible completion[1]. There is no obvious way of eliciting a fine-grained fitness from this binary outcome.

To address this issue, we reformulate the synthesis problem, originally posed as a search problem in (1), as an *optimization problem*, asking the solver to determine the hole completion that maximizes the number of tests passed by the evaluated program. The number of passed tests becomes the (maximized) fitness of the program:

$$f(p) = \max_{\mathbf{b}} |(in, out) \in T : \; Pre(in) \implies Post(in, p_{\mathbf{b}}(in))|. \tag{4}$$

Optimization is beyond the original formulation of SMT satisfiability problem, and thus SMT solvers cannot be expected to handle it. However, in the case of $f(p)$, a bisection algorithm may be used, because we have a discrete and

[1] Technically, the solver may also time-out, which we interpret as lack of feasible completion too.

bounded set of possible fitness values. By halving intervals and adding appropriate constraints, it is possible to determine the largest $f(p)$ for which the synthesis formula is still satisfied, using only $\log_2 |T|$ solver queries. Alternatively, there are solvers with a built-in capability for optimization, like Z3 [13] we use in Section 5. Our implementation is based on the latter, because it proved to be more efficient.

3.4 Exploiting the feedback from hole completion

The optimization process that calculates fitness in (4), apart from the number of passed tests, produces also the optimal completion of holes (the model). Let $\mathbf{b}^*$ be the associated optimal assignment of variables found in (4). In the default scenario, we discard it. However, one may argue that the assignment defined by $\mathbf{b}^*$ contains useful knowledge that can be leveraged. Thus, we consider an alternative variant in which the completion defined by $\mathbf{b}^*$ is incorporated in the evaluated program, and the modified program replaces the original candidate solution in the population (in other words, f has a *side effect* consisting in p being modified).

In both variants, the process of hole completion can be seen as a local search, or in evolutionary terms as an adaptation that takes place during individual's lifetime. It seems thus justified to liken the former variant to *Baldwinian evolution*, in which such adaptations impact individual's fitness, but do not get explicitly inherited, and the latter to *Lamarckian evolution*, in which the acquired adaptations do get inherited directly. We will refer to these variants in this way and use the respective acronyms EPS-B and EPS-L.

4 Related work

The work most directly related to EPS is obviously program sketching [16,17], presented in Section 2. There are, however, other studies that involve the two distinguishing features of EPS:

- its formal (and unusual in GP) approach to program evaluation,
- evolution of partial programs.

We group them according to these characteristics.

Concerning the **use of formal techniques** in GP, Johnson [7] was probably the first to use *model checking* for calculating fitness in GP. Model checking is a specific approach to *formal verification* of programs and systems, which essentially consists in determining whether a given program p meets the contract $(Pre, Post)$:

$$\forall_{in} Pre(in) \implies Post(in, p(in)). \tag{5}$$

Verification applies to an existing program and is thus computationally less demanding than synthesis (1). In [7], Johnson used temporal logic to express formal specifications that describe the desired time-wise behavior of finite state

machines. A fairly conventional GP algorithm was employed to evolve candidate state machines, with fitness defined as the number of fulfilled constituent clauses in the contract. The work demonstrated successful application of this approach to synthesis of control programs for a vending machine.

Temporal logic and GP are also the underlying mechanisms in other related work by Katz and Peled [8]. Similarly to [7] – and in contrast to this study – the authors evolve complete programs. When evaluating them, they distinguish four levels of program correctness: first, in which no scenario of program execution can satisfy the contract; second, in which some program executions satisfy the contract; third, in which all terminating executions meet the contract, and the highest level, in which all program executions meet the contract. Given this distinction, the fitness measure counts the satisfied postconditions. The authors apply the methods to examples from [19] and to synthesis of mutual exclusion algorithms and correction of erroneous programs. To verify programs, they consider both model checking and an SMT solver. Their use of SMT solver is, however, different than in this paper. In EPS, SMT solver is used to synthesize the content of a hole, while in [8] it is used solely for verification and producing counterexamples.

Concerning **evolving and completing partial programs**, the latter EPS feature identified at the beginning of this section, the related work is very limited. Though partial programs are occasionally considered in GP (cf., e.g., program contexts in semantic GP [12]), no GP approach known to us explicitly maintains them in population. However, program completion in EPS is limited to single-node terminals, and as such can be likened to optimizing constants in programs, which attracted significant attention in GP research. Among past contributions, Sarafopoulos [15] hybridized GP with evolutionary strategies (ES), where the ES component was responsible solely for fine-tuning the constants in candidate programs. Azad and Ryan [1] extended GP with a simple local search that tunes the instructions of individuals (including the internal nodes of program trees), and implemented a caching mechanism to reduce the computational overhead of tuning. When evaluated on a range of benchmarks, their approach synthesizes fitter and smaller programs than standard GP. The cited work features comprehensive review of analogous techniques, which we redirect an interested reader to.

In a broader perspective, EPS capability to improve candidate programs pertains also to *memetic* approaches, researched in numerous studies in the past. For instance, semantic backpropagation [14] and memetic semantic genetic programming [4] offer search operators that improve programs locally, i.e. at the level of particular instructions/subprograms.

5 Experimental evaluation

Objectives. We compare the Baldwinian (EPS-B) and Lamarckian (EPS-L) variants of EPS on a range of problems, in a few configurations detailed in the following, within the conventional tree-based GP. Our goal is to find out which of

Fig. 2. The NIA grammar defining the set of considered programs. c stands for an integer constant, v_i for the ith input variable, and h_j for the jth hole allowed in the program. `ite` stands for *if-then-else*, the conventional conditional statement.

```
I ::= I + I | I - I | I * I | I / I | ite(B,I,I) | c
      | v_1 | v_2 | ... | v_k
      | h_1 | h_2 | ... | h_l
B ::= I < I | I <= I | I = I | B = B | I >= I | I > I
```

EPS variants and configurations fair the best and how its performance compares to that of standard GP.

Domain. As follows from Section 2, applicability of EPS is conditioned on a theory that supports the reasoning conducted by the SMT solver. Past research led to elaboration of several popular theories and associated logics, now systematized by the SMT-LIB standard [2,3]. The theories vary in the data types they support (e.g., Booleans, bit vectors, integers, floating point, reals) and in logics that constrain the form of expressions/formulas (e.g., linear, nonlinear)[2]. Wider logics offer more expressibility but typically require higher computational effort from the solver. In this preliminary study, we settle on a mid-way compromise in that trade-off, the Nonlinear Integer Arithmetic (NIA) logic. This choice determines:

1. The types that can be used in expressions: integer (`I`) and Boolean (`B`),
2. The set of expressions that can be passed to the solver, which, by the design of EPS, becomes also the instruction set to be used by the evolutionary process.

A solver equipped with NIA can prove theorems that obey the grammar shown in Fig. 2. For the sake of synthesizing programs that are k-ary integer functions ($\texttt{I}^k \rightarrow \texttt{I}$), we assume that the starting symbol of the grammar is `I`, even though the top-level type of the predicates passed to the solver is naturally `B`, as follows from the synthesis formula (1). The grammar diverges from the conventional NIA in two ways:

- It features additional terminal symbols h_i, which implement the holes to be substituted by the solver in EPS.
- It does not contain some of the less common nonterminals, e.g. `mod` and `abs`.

Configurations. In sketching as introduced by Solar-Lezama [16], holes can be filled by arbitrary code pieces (of the compatible type). However, the larger the code pieces one considers to substitute for holes, the larger the search space and the more expensive synthesis becomes. In this study, we consider the simplest approach, i.e., we allow the holes to be substituted only with single-instruction code pieces, more precisely the integer-valued terminals available in the NIA

[2] `http://smtlib.cs.uiowa.edu/logics.shtml`

Table 1. Compared configurations.

Configuration	*Terminals that can be substituted for holes*	
	Constants c	Input variables v_i
GP		
EPS_c	✓	
EPS_v		✓
EPS_{cv}	✓	✓

Table 2. Program synthesis benchmarks.

Benchmark	*#vars.*	*Formula*	*Tests*	*#tests*
`Keijzer12`	2	$x_1^4 - x_1^3 + x_2^2/2 - x_2$	$x_1, x_2 \in \{-3, \ldots, 0, \ldots, 3\}$	49
`Koza1`	1	$x^4 + x^3 + x^2 + x$	$x \in \{-5, -4, \ldots, 0, \ldots, 4, 5\}$	11
`Koza1-p`		$3x^4 - 2x^3 + 6x^2 + 3x - 4$		
`Koza1-2D`	2	$x_1^4 + x_2^3 + x_1^2 + x_2$	$x_1, x_2 \in \{-3, \ldots, 0, \ldots, 3\}$	49
`Koza1-p-2D`		$3x_1^4 - 2x_2^3 + 6x_1^2 + 3x_2 - 4$		

grammar. Nevertheless, even this simple design choice leads to several configurations summarized in Table 1, which vary in the terminals that are substituted for holes: constants only (EPS_c), variables only (EPS_v), or both (EPS_{cv}). These three configurations together with two EPS variants (EPS-B, EPS-L) lead to six setups. Naturally, standard GP cannot handle holes, so the hole terminals are removed from the grammar for this method.

Benchmarks. NIA allows us to use benchmarks that are similar in spirit to symbolic regression, albeit dwell in the integer domain. We employ the benchmarks presented in Table 2; these are based on their real-valued counterparts from the GP benchmarks suite [11], but by necessity use integer inputs, typically from a wider interval than in the original benchmark.

Search operators. The presence of two types (`I` and `B`) implies that the GP part of EPS implementation has to be typed. We impose the correct typing by means of a grammar in Fig. 2 and constrain the actions of initialization, subtree mutation and tree-swapping crossover operators, so that they guarantee producing programs that follow the grammar. When generating a random program for the initial population, we traverse the grammar rules starting from the `I` symbol, reducing the probability of nonterminals when closing to the allowable tree height (Table 3); if the resulting program exceeds that limit, we scrap it and initialize a new program. For subprograms to be inserted by mutation we proceed similarly, however starting either from the `I` or from the `B` symbol, depending on the type of the instruction being replaced. The crossover operator picks a random location from the first parent, draws a random location of the same type in the second parent, and swaps the subtrees rooted at those locations. Should it fail to find a type-compatible location in the second parent, it discards both parents and starts anew with another selected pair of parents (this may happen

Table 3. Parameters of the evolutionary algorithm.

Parameter	*Value*
Population size	250
Maximum height of initial programs	4
Maximum height of subprograms inserted by mutation	4
Constant terminals drawn from interval	$[0, 5]$
Maximum number of generations	100
Probability of mutation	0.5
Probability of crossover	0.5
Tournament size	7

only for the `B` type, as at least one instruction of type `I` is guaranteed to exist in every program).

Solver budget. In EPS, the solver is given the computational budget of 1.5 seconds for a single query. If it fails to find an optimal assignment in this time, the evaluated program receives the worst possible fitness of zero. Handling timeouts is essential, because it is very hard in general to estimate the upper bound on solver's computation time.

Implementation. EPS has been implemented in authors' PySV (Python Synthesis and Verification) framework (responsible for constructing queries to Z3 solver) and SMTGP Scala framework (responsible for running evolution with holes). Sources of both of these frameworks are accessible on Github[3]. The latter framework is based on two Scala libraries: *Functional Evolutionary Algorithms* (FUEL) and *Synthesis with Metaheuristics* (SWIM), both originating in [9] and also available on GitHub[4]. The SMTGP framework implements the conventional GP workflow, with the exception of fitness function that passes the individuals with holes to the PySV framework, which in turn handles the call to the SMT solver. The communication with the solver is realized using the SMT-LIB standard [2]. We employ the Z3 SMT solver by Microsoft [13], one of the most efficient and powerful non-commercial solvers.

Results. Table 4 presents the success rate of particular configurations on individual benchmarks, and Table 5 and Fig. 3 the average fitness of the best-of-run programs. Applying the configurations that substitute holes with variables only (EPS-L_v and EPS-B_v) to univariate benchmarks is pointless, so such cases are excluded from presentation. The figure reveals clear, repetitive pattern of relative performances of individual configurations. EPS-B_c fares the best: it tops the other configurations in terms of success rate, and reliably produces an optimal program in each run for Koza1 and Koza1-p. The figure suggests that EPS-L_{cv} and EPS-L_c are the two competing runners-up; however, Table 4 leaves no doubts that they are much less likely to synthesize a correct program.

[3] `https://github.com/iwob`
[4] `https://github.com/kkrawiec`

Table 4. The number of optimal solutions found (maximum: 100).

	GP			*EPS*					
	GP	GP_T	GP_{5000}	L_c	L_v	L_{cv}	B_c	B_v	B_{cv}
`Keijzer12`	0	0	5	0	0	1	39	1	0
`Koza1`	19	68	96	33	-	32	100	-	100
`Koza1-p`	0	0	0	5	-	3	100	-	100
`Koza1-2D`	1	12	20	2	0	11	80	21	23
`Koza1-p-2D`	0	0	0	0	0	1	75	0	0

Table 5. Average end-of-run fitness.

	GP			*EPS*					
	GP	GP_T	GP_{5000}	L_c	L_v	L_{cv}	B_c	B_v	B_{cv}
`Keijzer12`	15.85	23.02	25.06	23.92	18.05	27.77	39.05	20.45	17.47
`Koza1`	5.89	9.74	10.87	9.93	-	9.83	11.00	-	11.00
`Koza1-p`	2.59	4.45	3.98	9.05	-	8.78	11.00	-	11.00
`Koza1-2D`	16.54	29.73	33.18	23.39	19.47	31.29	45.42	27.36	23.70
`Koza1-p-2D`	9.29	17.18	14.60	22.60	10.66	29.47	46.23	12.56	15.41

Overall, the configurations that complete the holes with variables only (EPS-L_v and EPS-B_v) fare the worst. This suggests that substituting with constants, available in the other configurations of EPS, is essential. This capability is particularly important for the benchmarks considered here, which feature at most two variables, and manipulating them does not leave much space for improvement.

Table 6 presents the average runtimes of configurations on particular benchmarks, which reveals that engaging the SMT solver comes at a price: EPS runs take up to four orders of magnitude longer than standard GP. One may question thus whether comparing EPS with short-timed GP is entirely fair. To address this issue, we devise another configuration, GP_T, in which genetic programming uses the same parameters as previously (Table 3), except for the maximum number of generations, which is replaced by the time limit, equal to the average runtime of the EPS configurations on a given benchmark. For instance for the Keijzer12 benchmark, GP_T is allowed to run for 11,300 seconds.

By definition, GP_T should not be worse than GP, which is confirmed in Fig. 3, where the non-overlapping inter-quartile boxes suggest superiority of the former to the latter. GP_T also manages to produce more fit best-of-run individuals than EPS-L_v and EPS-B_v. However, it seems incapable to catch up with the other EPS configurations, in particular with the leading EPS-B_c.

We also include configuration GP_{5000}, which is identical to GP except for population holding 5000 programs. It proves to be much better than GP_T on all the tested benchmarks and often outperforms all EPS-L configurations. However, in terms of success rate it still fares worse than EPS-B_C.

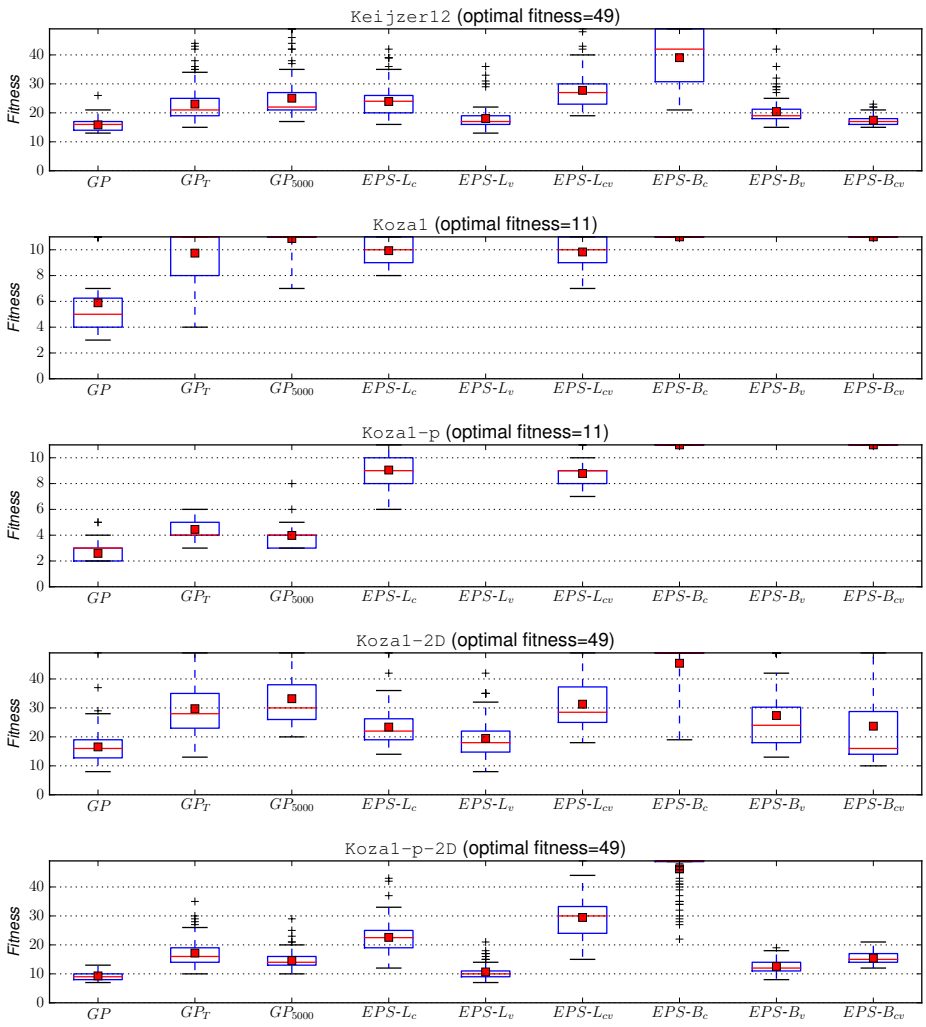

Fig. 3. Box-and-whiskers plots of the (maximized) fitness of the final solutions across all configurations and benchmarks. Boxes mark lower and upper quartiles, red line – median, red square – mean, whiskers – 1.5 of inter-quartile range below/above the corresponding quartile, and crosses – the outliers. The missing plots for EPS-L_v and EPS-B_v applied to Koza1 and Koza1-p are due to those benchmarks being univariate, which makes variable substitution pointless.

Table 6. Average runtime in seconds.

	GP			*EPS*					
	GP	GP_T	GP_{5000}	L_c	L_v	L_{cv}	B_c	B_v	B_{cv}
`Keijzer12`	14.8	11330.7	493.0	772.3	488.0	1578.6	15439.8	21172.6	28354.0
`Koza1`	4.8	291.0	46.3	699.8	-	801.4	652.0	-	695.8
`Koza1-p`	4.5	962.9	344.0	892.3	-	971.6	978.2	-	982.0
`Koza1-2D`	16.0	7635.8	431.9	793.1	478.7	1790.6	9076.9	16280.5	23033.8
`Koza1-p-2D`	15.4	9206.1	515.9	750.4	511.3	1725.7	11986.4	12390.8	27875.4

6 Discussion

Overall, the EPS-B configurations perform better than or at least as good as the corresponding EPS-L configurations in terms of average end-of-run fitness (Table 5). This holds for 10 out of 13 pairs of corresponding EPS-B and EPS-L configurations. It seems thus that EPS favors the Baldwinian approach, in which local, within-lifetime modifications (hole completions) affect individual's fitness but do not propagate to its offspring. Our working explanation is that, by propagating to offspring, the unfilled holes in EPS-B prospectively make it possible to find even better completions. In the Lamarckian variant, to the contrary, fitness evaluation leaves no holes in the evaluated individuals. The only supply of 'fresh' holes is the mutation operator, which affects on average only half of the offspring (Table 3), but even in them it is not guaranteed to introduce any new holes. Apparently, those holes are not sufficient in numbers to permit completions that would make EPS-L outperform EPS-B.

The above pattern is however reversed for the configurations that permit completion with both constants and variables, when applied to bivariate benchmarks, i.e., EPS-B_{cv} vs. EPS-L_{cv} on Keijzer12, Koza1-2d and Koza-1-p-2d. This may result from the overall worse performance of configurations that complete holes with variables: when juxtaposing such configurations with their counterparts that do not involve variables (i.e., EPS-B_v vs. EPS-B_c, EPS-L_v vs. EPS-L_c, EPS-B_{cv} vs. EPS-B_c, and EPS-L_{cv} vs. EPS-L_c), the former almost always fare worse. The only exception is the last pair, EPS-L_{cv} vs. EPS-L_c, where the former may occasionally perform better (for Koza-1-2d and Koza-p-2d), but the differences do not seem to be statistically significant. As signaled in the previous section, this could be to some extent explained with the very low number of variables in considered benchmarks: with only two variables at its disposal, the solver has limited chance to complete the holes in a way that leads to high fitness. However, this argument is unconvincing for the mixed configurations (cv), where both variables and constants can be substituted.

Explanation for this phenomenon turns out to be of a different nature: in EPS-B_{cv}, with the possibility of completing with both constants and variables, and with relatively many holes to complete (due to following the Baldwinian process), the search space of possible completions is on average the largest compared to the other configurations. As a consequence, the solver faced with such large problems is more likely to fail to return a definitive answer within the

prescribed computational budget of 1.5 seconds. This results with assigning the worst possible fitness to an evaluated program and likely loss of potentially useful code it may contain. This is confirmed by the statistics on solver behavior we gathered: in EPS-B$_{cv}$, the solver fails to provide optimal completion on time in roughly 35 percent of evaluations, compared to only 13 percent for EPS-L$_{cv}$. Interestingly however, this does not turn out to be problematic for EPS-B$_c$, which also suffers from quite high incidence of such cases (around 25 percent), yet performs the best. Overall, these relatively high numbers suggest that solver timeouts may have significant impact on search dynamics. However, this phenomenon does not need to be pathological. To the contrary, it can serve as a natural parsimony pressure: in EPS-L as well as in EPS-B, large programs tend to have higher number of holes than small programs, and large number of holes makes solver timeout more likely.

Interestingly, program evaluation in EPS-B can be said in resulting in *prospective fitness*: the fitness that the partial program being evaluated *could* achieve in the future, given the optimal completion of its holes. This concept bears some resemblance to potential fitness considered in past work [10].

Concerning the runtime, it is not surprising that the Baldwinian configurations are, by a huge margin, more time consuming (except for the simpler Koza1 and Koza1-p benchmarks) than the corresponding Lamarckian variants – after all they contain more holes. It is also interesting that the Lamarckian configurations generally achieve end-of-run fitness (and, to a lesser extent, success rate) that is comparable to GP_T, which was granted much larger time budgets. This however may be a result of bloat, which could have lessened the effectiveness of GP.

7 Conclusion

This paper presented Evolutionary Program Sketching, a novel approach to program synthesis that combines selected elements of genetic programming and formal synthesis methods. EPS evolves partial programs and uses an SMT solver to complete them so as to maximize the number of passed test cases. The experiments have shown that EPS has the potential to be more efficient than standard GP in some scenarios. Nevertheless, empirical evaluation conducted here was rather constrained, which makes approaching a larger and more diverse benchmark suite our priority in further work on this topic.

As in all SMT-based approaches to program synthesis, the fact that candidate programs are never executed opens interesting possibilities. In principle, EPS can be used synthesize programs written in programming languages for which no interpreter exists or program execution is particularly costly (albeit the theory that backs up the language has to be known).

EPS as presented in this paper is an inductive synthesis method. However, engaging a solver for evaluation opens up an interesting possibility of using GP to synthesize programs from a complete formal specification (like the $\max(x, y)$ specification in Formula (2)). Apart from this, future work may include filling

holes with more complex content than just constants and variables, and optimizing the mechanism of querying the solver.

Acknowledgments. This work was supported by grant 2014/15/B/ST6/05205 funded by the National Science Centre, Poland.

References

1. Azad, R.M.A., Ryan, C.: A simple approach to lifetime learning in genetic programming based symbolic regression. Evolutionary Computation 22(2), 287–317 (Summer 2014), http://www.mitpressjournals.org/doi/abs/10.1162/EVCO_a_00111
2. Barrett, C., Fontaine, P., Tinelli, C.: The SMT-LIB Standard: Version 2.5. Tech. rep., Department of Computer Science, The University of Iowa (2015), available at www.SMT-LIB.org
3. Barrett, C., Fontaine, P., Tinelli, C.: The Satisfiability Modulo Theories Library (SMT-LIB). www.SMT-LIB.org (2016)
4. Ffrancon, R., Schoenauer, M.: Memetic semantic genetic programming. In: Silva, S., Esparcia-Alcazar, A.I., Lopez-Ibanez, M., Mostaghim, S., Timmis, J., Zarges, C., Correia, L., Soule, T., Giacobini, M., Urbanowicz, R., Akimoto, Y., Glasmachers, T., Fernandez de Vega, F., Hoover, A., Larranaga, P., Soto, M., Cotta, C., Pereira, F.B., Handl, J., Koutnik, J., Gaspar-Cunha, A., Trautmann, H., Mouret, J.B., Risi, S., Costa, E., Schuetze, O., Krawiec, K., Moraglio, A., Miller, J.F., Widera, P., Cagnoni, S., Merelo, J., Hart, E., Trujillo, L., Kessentini, M., Ochoa, G., Chicano, F., Doerr, C. (eds.) GECCO '15: Proceedings of the 2015 on Genetic and Evolutionary Computation Conference. pp. 1023–1030. ACM, Madrid, Spain (11-15 Jul 2015), https://hal.inria.fr/hal-01169074/document, gP Track best paper
5. Gulwani, S., Jha, S., Tiwari, A., Venkatesan, R.: Component based synthesis applied to bitvector programs. Tech. Rep. MSR-TR-2010-12 (February 2010), http://research.microsoft.com/apps/pubs/default.aspx?id=119146
6. Jha, S., Gulwani, S., Seshia, S.A., Tiwari, A.: Oracle-guided component-based program synthesis. In: 29th International Conference on Software Engineering (ICSE '10). pp. 215–224 (May 2010)
7. Johnson, C.: Genetic programming with fitness based on model checking. In: Ebner, M., O'Neill, M., Ekárt, A., Vanneschi, L., Esparcia-Alcázar, A.I. (eds.) Proceedings of the 10th European Conference on Genetic Programming. Lecture Notes in Computer Science, vol. 4445, pp. 114–124. Springer, Valencia, Spain (11-13 Apr 2007), https://kar.kent.ac.uk/14594/1/Genetic.pdf
8. Katz, G., Peled, D.: Synthesis of parametric programs using genetic programming and model checking. In: Clemente, L., Holik, L. (eds.) Proceedings 15th International Workshop on Verification of Infinite-State Systems. EPTCS, vol. 140, pp. 70–84. Hanoi, Vietnam (14 Oct 2013), http://www.fit.vutbr.cz/~holik/INFINITY13/, invited talk
9. Krawiec, K.: Behavioral Program Synthesis with Genetic Programming, Studies in Computational Intelligence, vol. 618. Springer International Publishing (2015), http://www.cs.put.poznan.pl/kkrawiec/wiki/?n=Site.BPS
10. Krawiec, K., Polewski, P.: Potential fitness for genetic programming. In: Ebner, M., Cattolico, M., van Hemert, J., Gustafson, S., Merkle, L.D., Moore, F.W., Congdon, C.B., Clack, C.D., Moore, F.W., Rand, W., Ficici, S.G., Riolo, R., Bacardit, J., Bernado-Mansilla, E., Butz, M.V., Smith, S.L., Cagnoni, S., Hauschild,

M., Pelikan, M., Sastry, K. (eds.) GECCO-2008 Late-Breaking Papers. pp. 2175–2180. ACM, Atlanta, GA, USA (12-16 Jul 2008), http://www.cs.bham.ac.uk/~wbl/biblio/gecco2008/docs/p2175.pdf
11. McDermott, J., White, D.R., Luke, S., Manzoni, L., Castelli, M., Vanneschi, L., Jaskowski, W., Krawiec, K., Harper, R., De Jong, K., O'Reilly, U.M.: Genetic programming needs better benchmarks. In: Soule, T., Auger, A., Moore, J., Pelta, D., Solnon, C., Preuss, M., Dorin, A., Ong, Y.S., Blum, C., Silva, D.L., Neumann, F., Yu, T., Ekart, A., Browne, W., Kovacs, T., Wong, M.L., Pizzuti, C., Rowe, J., Friedrich, T., Squillero, G., Bredeche, N., Smith, S.L., Motsinger-Reif, A., Lozano, J., Pelikan, M., Meyer-Nienberg, S., Igel, C., Hornby, G., Doursat, R., Gustafson, S., Olague, G., Yoo, S., Clark, J., Ochoa, G., Pappa, G., Lobo, F., Tauritz, D., Branke, J., Deb, K. (eds.) GECCO '12: Proceedings of the fourteenth international conference on Genetic and evolutionary computation conference. pp. 791–798. ACM, Philadelphia, Pennsylvania, USA (7-11 Jul 2012)
12. McPhee, N.F., Ohs, B., Hutchison, T.: Semantic building blocks in genetic programming. In: O'Neill, M., Vanneschi, L., Gustafson, S., Esparcia Alcazar, A.I., De Falco, I., Della Cioppa, A., Tarantino, E. (eds.) Proceedings of the 11th European Conference on Genetic Programming, EuroGP 2008. Lecture Notes in Computer Science, vol. 4971, pp. 134–145. Springer, Naples (26-28 Mar 2008)
13. de Moura, L., Bjørner, N.: Z3: An efficient SMT solver. In: Ramakrishnan, C., Rehof, J. (eds.) Tools and Algorithms for the Construction and Analysis of Systems, Lecture Notes in Computer Science, vol. 4963, chap. 24, pp. 337–340. Springer Berlin / Heidelberg, Berlin, Heidelberg (2008), http://dx.doi.org/10.1007/978-3-540-78800-3_24
14. Pawlak, T.P., Wieloch, B., Krawiec, K.: Semantic backpropagation for designing search operators in genetic programming. IEEE Transactions on Evolutionary Computation 19(3), 326–340 (Jun 2015), http://dx.doi.org/10.1109/TEVC.2014.2321259
15. Sarafopoulos, A.: Evolution of affine transformations and iterated function systems using hierarchical evolution strategy. In: Miller, J.F., Tomassini, M., Lanzi, P.L., Ryan, C., Tettamanzi, A.G.B., Langdon, W.B. (eds.) Genetic Programming, Proceedings of EuroGP'2001. LNCS, vol. 2038, pp. 176–191. Springer-Verlag, Lake Como, Italy (18-20 Apr 2001), http://www.springerlink.com/openurl.asp?genre=article&issn=0302-9743&volume=2038&spage=176
16. Solar-Lezama, A.: Program Synthesis by Sketching. Ph.D. thesis, Electrical Engineering and Computer Science, University of California, Berkeley, USA (fall 2008), http://people.csail.mit.edu/asolar/papers/thesis.pdf
17. Solar-Lezama, A.: Program sketching. International Journal on Software Tools for Technology Transfer 15(5), 475–495 (2013), http://dx.doi.org/10.1007/s10009-012-0249-7
18. Srivastava, S., Gulwani, S., Foster, J.S.: From program verification to program synthesis. In: Proceedings of the 37th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages. pp. 313–326. POPL '10, ACM, New York, NY, USA (2010), http://doi.acm.org/10.1145/1706299.1706337
19. Warren, H.S.: Hacker's Delight. Addison Wesley (2002)