Genetic Programming on Program Traces as an Inference Engine for Probabilistic Languages

Vita Batishcheva[2] and Alexey Potapov[1,2]

[1]ITMO University, St. Petersburg, Russia
[2]St. Petersburg State University, St. Petersburg, Russia
elokkuu@gmail.com, potapov@aideus.com

Abstract. Methods of simulated annealing and genetic programming over probabilistic program traces are developed firstly. These methods combine expressiveness of Turing-complete probabilistic languages, in which arbitrary generative models can be defined, and search effectiveness of meta-heuristic methods. To use these methods, one should only specify a generative model of objects of interest and a fitness function over them without necessity to implement domain-specific genetic operators or mappings from objects to and from bit strings. On the other hand, implemented methods showed better quality than the traditional mh-query on several optimization tasks. Thus, these results can contribute to both fields of genetic programming and probabilistic programming.

Keywords: probabilistic programming, genetic programming, program traces

1 Introduction

Two crucial approaches in AGI are cognitive architectures and universal algorithmic intelligence. These approaches start from very different points and sometimes are even treated as incompatible. However, we believe [1] that they should be united in order to build AGI that is both efficient and general. However, a framework is required that can help to intimately combine them on the conceptual level and the level of implementation. Probabilistic programming could become a suitable basis for developing such a framework. Indeed, on the one hand, query procedures in the Turing-complete probabilistic programming languages (PPLs) can be used as direct approximations of universal induction and prediction, which are the central components of universal intelligence models. On the other hand, probabilistic programming has already been successfully used in cognitive modeling [2].

Many solutions in probabilistic programming utilize efficient inference techniques for particular types of generative models (e.g. Bayesian networks) [3, 4]. However, Turing-complete languages are much more promising in the context of AGI. These PPLs allow for specifying generative models in the form of arbitrary programs including programs which generate other programs. Inference over such generative models automatically results in inducing programs in user-defined languages. Thus, the same inference engine can be used to solve a very wide spectrum of problems.

On the one hand, the performance of generic inference methods in PPLs can be rather low even for models with a small number of random choices [5]. These methods

are most commonly based on random sampling (e.g. Monte-Carlo Markov Chains) [2, 6]. There are some works on developing stronger methods of inference in Turing-complete probabilistic languages (e.g. [5, 7]), but they are not efficient for all cases, e.g. for inducing programs, although some progress in this direction is achieved [8]. Thus, more appropriate inference methods are needed, and genetic programming (GP) can be considered as a suitable candidate since it has already been applied to universal induction [9] and cognitive architectures [10].

On the other hand, wide and easy applicability of inference in PPLs is also desirable by evolutionary computations. Indeed, one would desire to be able to apply some existing implementation of genetic algorithms simply by defining the problem at hand without developing binary representations of solutions or implementing problem-specific recombination and mutation operators (and some attempts to overcome this also exist in the field of genetic programming, e.g. [11]).

Consequently, it is interesting to combine generality of inference over declarative models in Turing-complete PPLs and strength of genetic programming. This combination will give a generic tool for fast prototyping of genetic programming methods for arbitrary domain specific languages simply by specifying a function generating programs in a target language. It can also extend the toolkit of PPLs, since conventional inference in probabilistic programming is performed for conditioning, while genetic programming is intended for optimization of fitness functions.

In this paper, we present a novel approach to inference in PPLs based on genetic programming and simulated annealing, which are applied to probabilistic program (computation) traces. Each program trace is the instantiation of the generative model specified by the program. Recombinations and mutations of program traces guarantee that their results can be generated by the initial probabilistic program. Thus, program traces are used as a "universal genetic code" for arbitrary generative models, and it is enough to only specify such a model in the form of a probabilistic program to perform evolutionary computations in the space of its instantiations. To the best of our knowledge, there are no works devoted to implementing an inference engine for PPLs on the base of genetic programming, so this is the main contribution of our paper.

In [12] authors indicated that "current approaches to Probabilistic Programming are heavily influenced by the Bayesian approach to machine learning" and the optimization approach is promising since "optimization techniques scale better than search techniques". That is, our paper can also be viewed as the work in this direction, which is much lesser explored in the field of probabilistic programming.

2 Background

Probabilistic Programs

Since general concepts of genetic programming are well known, we will concentrate on probabilistic programming. Some PPLs extend existing languages preserving their semantics as a particular case. Programs in these languages typically include calls to (pseudo-)random functions. PPLs use an extended set of random functions corresponding to different common distributions including Gaussian, Beta, Gamma, multi-

nomial, etc. Evaluation of such a program with random choices is performed in the same way as evaluation of this program in the base (non-probabilistic) language.

However, programs in PPLs are treated as generative models defining distributions over possible return values [5], and their direct evaluation can be interpreted as taking one sample from corresponding distributions. Multiple evaluation of a program can be used to estimate an underlying distribution.

PPLs go further and support programs defining conditional distributions. Such a program contains a final condition indicating whether the result of program evaluation should be accepted or not (in some languages an "observe" statement can be placed anywhere to impose conditions on intermediate results). The simplest way to sample from conditional distributions is the rejection sampling, in which a program is simply evaluated many times while its final condition is not satisfied. However, the rejection sampling can be extremely inefficient even for rather simple models.

One can utilize a method for efficient inference of conditional probabilities without sampling for a restricted set of models, but generic inference methods should be used for Turing-complete languages. One of such widely used methods is based on Monte-Carlo Markov Chains (MCMC), namely, the Metropolis-Hastings (MH) algorithm. This algorithm uses stochastic local search to sample such instances, for which the given condition will remain true. To implement it for models specified by probabilistic programs, one needs to introduce small changes to the return values of elementary random procedures called in these programs, so these values should be memoized [2].

MCMC can be much more efficient than rejection sampling for evaluating posterior distributions. However, without utilizing some additional techniques it can be as bad as the rejection sampling (or even worse due to overheads) in retrieving the first appropriate sample. One can easily check this on the example of the following simple Church program (mh-query 1 1 (define xs (repeat 20 flip)) xs (all xs)).

In this program, the list of 20 random Boolean values is defined, and this list is returned, when all its values are true. If one replaces "mh-query 1 1" with "rejection-query", calculation time will slightly decrease. However, retrieving many samples by mh-query will be much faster than executing rejection-query many times. Thus, the unsolved problem here is the problem of finding the first admissible instantiation of a model. This is done blindly in both MCMC and the rejection sampling.

In many practical problems, a user can convert a strict condition into a soft one or even can initially have a task with the goal to optimize some function. Thus, query procedures which accept a fitness-function for optimization instead of a strict condition for satisfying can be used as a part of MCMC sampling as well as they can be used independently for solving optimization problems.

Implemented Language

Since exploration of solution spaces in probabilistic programming require manipulations with random choices made during program evaluation, development of new query procedures is connected with interfering in the evaluation process. Since no language supports flexible enough external control of this process, it was easier for us to implement a new interpreter. However, we decided not to develop a new language, but to reproduce (using Scheme as the host language) some basic functionality of Church [2] including a number of simple functions (+, -, *, /, and, or, not, list, car, cdr, cons, etc.), several random functions (flip, random_integer, gaussian, multinomial), declaration of variables and functions (define, let), function calls with recursion.

Also, “quote” and “eval” were implemented. For example, the following program is acceptable (which is passed to our interpreter as the quoted list)

```
'((define (tree) (if (flip 0.7) (random-integer 10)
                     (list (tree) (tree))))
  (tree))
```

Traditional Lisp interpreters will return different results on each run of such programs. Interpreters of PPLs provide for query functions, which are used for calculating posterior probabilities or to perform sampling in accordance with the specified condition. We wanted to extend this language with GP-based query procedures, which accept fitness-functions instead of strict conditions. Let’s consider how genetic operators can be implemented in these settings.

3 Genetic Operators For Computation Traces

Mutations

To combine genetic programming with probabilistic languages we treat each run of a program as a candidate solution. The source of variability of these candidate solutions comes from different outcomes of random choices during evaluation. Mutations consist in slight modifications of the random choices performed during the previous evaluation that resembles some part of the MH-algorithm. All these choices should be cached and bound to the execution context, in which they were made. To do this, we implemented the following representation of program traces, which idea (but not its implementation) is similar to that used in the mh-query implementation in Church [2].

In this representation, each expression in the original program during recursive evaluation is converted to the structure (struct IR (rnd? val expr) #:transparent), where IR is the name of the structure, rnd? is #t if random choices were made during evaluation of the expression expr; val is the result of evaluation (one sample from the distribution specified by expr). interpret-IR-prog function was implemented for evaluating programs (lists of expressions) given in symbolic form. Consider some examples.

- (interpret-IR-prog '(10)) → (list (IR #f 10 10)) meaning that the result of evaluation of the program containing only one expression 10 is 10 and it is not random.
- (interpret-IR-prog '((gaussian 0 1))) → (list (IR #t -0.27 (list 'gaussian (IR #f 0 0) (IR #f 1 1)))) meaning that the result of evaluation of (gaussian 0 1) was -0.27.
- (interpret-IR-prog '((if #t 0 1))) → (list (IR #f 0 (list 'if (IR #f #t #t) (IR #f 0 0) 1))) meaning that only one branch was evaluated.
- In the more complex case, random branch can be expanded depending on the result of evaluation of the stochastic condition: (interpret-IR-prog '((if (flip) 0 1))) → (list (IR #t 1 (list 'if (IR #t #f '(flip)) 0 (IR #f 1 1)))).
- In definitions of variables only their values are transformed to IR: (interpret-IR-prog '((define x (flip)))) → (list (list 'define 'x (IR #t #f '(flip)))). Evaluation of definitions results in changes of the environment as usual. Let-expressions have similar behavior, but they have a body to be evaluated. Function definitions are kept unchanged. Non-library function application is replaced by its body and let-binding of its arguments.

- Symbols, which can be found in the environment, are replaced by their values: (interpret-IR-prog '((define x (random-integer 10)) x)) → (list (list 'define 'x (IR #t 5 (list 'random-integer (IR #f 10 10)))) (IR #t 5 'x)).

The evaluated program can be evaluated again, and previously made random choices can be taken into account during this re-evaluation. We extended interpret-IR-prog in such a way that it can accept both initial programs and their IR expansions (since re-evaluation process can run into branches not expanded yet, such unification is necessary to deal with mixed cases also).

During each following re-evaluation of the expanded program, deterministic expressions are not evaluated again, but their previous values are used. All stochastic expressions are evaluated in the same way as during the first run except calls to the basic random functions, which behavior is changed. These functions are modified in order to take previously returned values into account. The mutation speed parameter p added to interpret-IR-prog indicates, how close new values should be to previous values. For example, the previous result of (flip) is changed with probability equals to p. Re-evaluation of (IR #t v (gaussian x0 s)), where v is the previously returned value, x0 is mean and s is sigma, will correspond to (gaussian v (* p s)), but in other implementations it could be biased towards x0.

For example, the result of re-evaluation of the IR expression (list (IR #t -0.27 (list 'gaussian (IR #f 0 0) (IR #f 1 1)))) using p=0.01 can be (list (IR #t -0.26 (list 'gaussian (IR #f 0 0) (IR #f 1 1)))).

Simulated Annealing

The described interpreter is already enough to implement an optimization query based on simulated annealing. Let the program be given, which last expression returns the value of energy (fitness) function to be minimized. Then, we can re-evaluate this program many times preferring evaluation results with lower value of the last expression.

Simulated annealing maintains only one program trace (in the form of IR expansion). It executes interpret-IR-prog to generate new candidate solutions (with transition probabilities derived by the interpreter for the given program and parameterized by the temperature), and accepts them with probability (/ 1 (+ 1 (exp (/ dE t)))), where dE is the difference of energies of the candidate and current solutions, and t is the current temperature (other acceptance probabilities can be used).

On each iteration, candidate solutions are generated until acceptance (although the number of tries is limited), and the temperature is decreased from iteration to iteration. We implemented annealing-query on the base of this approach.

Crossover

Crossover also utilizes program traces. However, it requires dual re-evaluation of two expansions of a program. These expansions are interpreted together as the same program, while their structures match (and they should match except variations caused by random choices). The main difference is in application of the basic random functions since the previously returned values from both parents should be taken into account.

For example, in our implementation, the dual flip randomly returns one of the previous values, and the dual Gaussian returns (+ (* v1 e) (* v2 (- 1 e))), where v1 and v2 are the previous values, and e is the random value in [0, 1] (one can bias the result

of this basic element of crossover towards initial Gaussian distribution). Mutations are introduced simultaneously with crossover for the sake of efficiency.

However, such a branch can be encountered during re-evaluation that has not been expanded yet in one or both parents. In the latter case, this branch is evaluated simply as it was the first execution. Otherwise it is re-evaluated for the parent, for which it has already been expanded (without crossover, but with mutations). It is not expanded for another parent, since this expansion will be random and not evaluated by fitness function, so it will simply clutter information from the more relevant parent.

Children can contain earlier expanded, but now unused branches, which can be activated again in later generations due to a single mutation or even crossover. These parts of the expanded program resemble junk DNA and fast genetic adaptations.

Let us consider one simple, but interesting case for crossover, namely a recursive stochastic procedure '((define (tree) (if (flip 0.7) (random-integer 10) (list (tree)))). Expansion of (tree) can produce large computation traces, so let us consider results of crossover on the level of values of the last expression.

'(6 9) + '(8 0) → '(7 6)
'(7 9) + '((0 7) (7 4)) → '(7 (7 4))
'((3 (7 (1 7))) 5) + '((5 2) 2) → '((4 (7 (1 7))) 2)

It can be seen that while the structure of trees matches, two program traces are re-evaluated together and results of random-integer are merged in leaves, but when the structure diverges, a subtree is randomly taken from one of the parents (depending on the result of re-evaluating (flip 0.7)). This type of crossover for generated trees automatically follows from the implemented crossover for program traces. Of course, someone might want to use a different crossover operator based on the domain-specific knowledge, e.g. to exchange arbitrary subtrees in parents. The latter is difficult to do in our program trace representation (and additional research is needed to develop a more flexible representation). On the other hand, recombination of program traces during dual re-evaluation guarantees that its result can be produced by the initial program, and also provides for some flexibility.

Using the described genetic operators, evolution-query was implemented.

4 Empirical Evaluation

We considered three tasks, each of which can be set both for conditional sampling and fitness-function optimization. Three query functions were compared – mh-query (web-church), annealing-query and evolution-query (Scheme implementation). mh-query was used to retrieve only one sample (since we were interested in its efficiency on this step; moreover, the tasks didn't require posterior distributions).

Curve Fitting

Consider the generative polynomial model $y=poly(x|\mathbf{ws})$, which parameters defined as normally distributed random variables should be optimized to fit observations $\{(x_i, y_i)\}$. Implementation of *poly* is straightforward. The full generative model should also include noise, but such a model will be useless since it is almost impossible to blindly guess noise values. Instead, MSE is used in annealing-query and evolution-query, and the following condition is used in mh-query.

```
(define (noisy-equals? x y) (flip (exp (* -30 (expt (- x y) 2)))))
(all (map noisy-equals? ys-gen ys))
```

noisy-equals? can randomly be true, even if its arguments differ, but with decreasing probability. Speed of this decrease is specified by the value, which equals 30 in the example code. The smaller this value, the looser the equality holds. We chose such the value that mh-query execution time approximately equals to that of annealing-query and evolution-query (which execution time is controlled by the specified number of iterations), so we can compare precision of solutions found by different methods. Of course, such comparison is quite loose, but it is qualitatively adequate since linear increase of computation time will yield only logarithmic increase of precision. The results for several functions and data points are shown in Table 1.

TABLE I. AVERAGE RMSE

Task		**RMSE**		
		mh-query	***annealing-query***	***evolution-query***
$4x^2+3x$	xs=(0 1 2 3)	1.71	0.217	0.035
$4x^2+3x$	xs=(0 0.1 0.2 0.3 0.4 0.5)	0.94	0.425	0.010
$0.5x^3-x$	xs=(0 0.1 0.2 0.3 0.4 0.5)	0.467	0.169	0.007

It can be seen that mh-query is inefficient here – it requires very loose noise-equals? yielding imprecise results. Stricter condition results in huge increase of computation time. Evolution-query is the most precise, while annealing-query works correctly, but converges slower. The worst precision is achieved, when wn is selected incorrectly. It is important to see, how crossover on program traces results in children's "phenotypes". Consider the example of how crossover affects ws values

```
'(1.903864    -11.119573  4.562440) +
'(-20.396958 -12.492696 -0.735389  3.308482) →
'(-5.232313   -11.462677  2.3152821 3.308482)
```

The values in same positions are averaged with random weights (although these weights are independent for different positions as in geometric semantic crossover). If lengths (i.e. wn) of parent's vector parameters ws differ, the child's wn value will correspond to that of one of the parents or will be between them.

Subset Sum Problem

In the subset sum problem, a set of integer numbers is given, and a nonempty subset should be found such that its sum equals a given integer value (we will assume that the sum equals 1 and skip the check of non-triviality of the solution for the sake of simplicity). Integers in each set were generated as random numbers from a certain range, e.g. -10000 to 10000. A random subset was selected, and the last number was calculated as 1 minus sum of elements in this subset. The following program specifies the generative model for this task.

```
(define xs '(9568 5716 8382 7900 -5461 5087 1138 -1111 -9695 -5468 6345
             -1473 -7521 -4323 9893 -9032 -4715 3699 5104 1551))
(define (make-ws n) (if (= n 0) '() (cons (flip) (make-ws (- n 1)))))
(define ws (make-ws (length xs)))
(define (summ xs ws) (if (null? xs) 0
```

```
                (+ (if (car ws) (car xs) 0) (summ (cdr xs) (cdr ws)))))
(define subset-sum (summ xs ws))
```

mh-query was executed using the condition (equal? subset-sum 1), while annealing-query and evolution-query were executed to minimize (abs (- subset-sum 1)). Direct comparison of different queries appeared to be difficult on this task. However, the results are qualitatively similar. All methods either stably find correct solutions (this is the case, when the task dimensionality is about 15 or less or when the range of numbers is small and many subsets can sum up to the desirable value) or all the methods fail (to achieve this, one should take numbers from a larger range and take the set size 20 or more).

However, for certain task complexities intermediate results can be obtained. In particular, the following results were obtained for the range [-10000, 10000] and the set sizes 20÷25. With as much as twice time limit of annealing-query and evolution-query, mh-query was able to find correct solutions in 83% cases. annealing-query and evolution-query yielded approximately 75% correct solutions (their performance vary depending on settings, and annealing-query showed more stable results, while our simple form of genetic programming had some tendency to get stuck in local extrema). It should be pointed out that in the rest 25% cases the solution found is almost optimal (error equals to 1).

Let's make sure that these slightly superficial results of genetic programming are not due to its invalid functioning. Consider the following typical effect of crossover over program traces on the "phenotype" level.

'(#t #f #t #t #t #t #f #t #t) + '(#f #f #t #t #f #t #f #t #t #f) → '(#t #f #t #t #t #t #f #f #t #f)

One can see that this is the uniform crossover. It is possibly not the most interesting one, but it is correct. From this example, it can also be seen that optimization queries on probabilistic programs fit well for solving deterministic problems.

Integer Number Sequence Prediction

One more test task we considered was the task of integer number sequence prediction. We restricted the set of possible sequences to polynomials, but it can easily be extended to the wider class of sequences defined by recurrence relations. Consider the following fragment of the generative model.

```
; recursively generating expressions
(define xs '(1 2 3 4 5 6))
(define ys '(3 7 13 21 31 43))
(define (gen-expr) (if (flip 0.6)
                       (if (flip) 'x (random-integer 10))
                       (list (multinomial '(+ - *) '(1 1 1)) (gen-expr) (gen-expr))))
(define (f x) (eval (list 'let (list (list 'x x)) expr)))
```

After these definitions, the function f(x) is used to map all xs and check if the result matches ys or to calculate the total deviation depending on the query type.

We ran tests for different sequences and compared the results. mh-query wasn't able to find a solution in each run. Depending on the web browser, it either finished with "Maximum call stack size exceeded" error or worked extremely long in some runs. annealing-query and evolution-query also were not able to find precise solutions in each case and terminated with imprecise solutions. Percentages of runs, in which correct solutions were found, are shown in Table 2. The value of xs was '(0 1 2 3 4 5).

TABLE II. PERCENTAGE OF CORRECT SOLUTIONS

ys	Correct answers, %		
	mh-query	***annealing-query***	***evolution-query***
'(0 1 2 3 4 5)	90%	100%	100%
'(0 1 4 9 16 25)	20%	100%	100%
'(1 2 5 10 17 26)	10%	70%	80%
'(1 4 9 16 25 36)	0%	90%	80%
'(1 3 11 31 69 131)	0%	90%	60%

mh-query yielded surprisingly bad results here, although its inference over other recursive models can be successful. evolution-query also yielded slightly worse results than annealing-query. The reason probably consists in that this task could not be well decomposed into subtasks, so crossover doesn't yield benefits, but annealing can approach to the best solution step by step.

Nevertheless, it seems that crossover operator over program traces produces quite reasonable results in the space of phenotypes. If the structure of the parents matches, each leaf is randomly taken from one of the parents, e.g. '(+ (+ 3 x) x) + '(- (- x x) x) → '(- (+ x x) x). In nodes, in which the structure diverges, a subtree is randomly taken from one of the parents, e.g.

'(- (- (* (* 3 (* x x)) 3) (- x 8)) (* (- x 0) x)) +'(- 3 (- 5 x)) → '(- 3 (* (- x 0) x))

'(* (+ 4 x) x) + '(* (* 2 (- 1 x)) 7) → '(* (* 4 x) 7)

"Phenotypic" crossover effect is somewhat loose, but not meaningless, and it produces valid candidate solutions, which inherit information from their parents.

Conclusion

We developed the methods of simulated annealing and genetic programming over probabilistic program traces. For the best of our knowledge, this is the first implementation of such methods. The same functions for genetic operators over program traces were used to solve optimization problems for very different types of objects including parametrically defined functions, sets, and symbolic expressions without producing invalid candidate solutions. Our implementation corresponds to the uniform crossover. Other types of genetic operators are to be implemented, since our implementation showed advantage over annealing only in the task of learning real-valued models. It is interesting to combine probabilistic programming with advanced genetic programming systems such as MOSES [10].

In spite of simplicity of the used meta-heuristic search methods, they outperformed the standard mh-query. Although this comparison doesn't mean that annealing-query or evolution-query can replace mh-query since they solve different tasks, it shows that they can be combined and also optimization queries can be useful to extend semantics of PPLs. Still, efficiency of general inference methods is insufficient, and this could be one of the principle obstacles in the path to AGI. Possibly, one general inference method cannot be efficient in all problem domains, so it should be automatically spe-

cialized w.r.t. each domain encountered by an AGI-agent implying that such methods should be deeply combined with cognitive architectures.

Acknowledgements

This work was supported by Ministry of Education and Science of the Russian Federation, and by Government of Russian Federation, Grant 074-U01.

References

1. Potapov, A., Rodionov, S., Myasnikov, A., Begimov, G.: Cognitive Bias for Universal Algorithmic Intelligence. arXiv:1209.4290v1 [cs.AI] (2012)
2. Goodman, N.D., Mansinghka, V.K., Roy, D.M., Bonawitz, K., Tenenbaum, J.B.: Church: a language for generative models. arXiv:1206.3255 [cs.PL] (2008)
3. Minka, T., Winn, J.M., Guiver, J.P., Knowles, D.: Infer.NET 2.4. Microsoft Research Camb., http://research.microsoft.com/infernet (2010)
4. Koller, D., McAllester, D.A., Pfeffer, A.: Effective Bayesian inference for stochastic programs. Proc. National Conference on Artificial Intelligence (AAAI), pp. 740–747 (1997)
5. Stuhlmüller, A., Goodman, N. D.: A dynamic programming algorithm for inference in recursive probabilistic programs. arXiv:1206.3555 [cs.AI] (2012)
6. Milch, B., Russell, S.: General-purpose MCMC inference over relational structures. Proc. 22nd Conference on Uncertainty in Artificial Intelligence, pp. 349–358 (2006)
7. Chaganty, A., Nori A.V., Rajamani, S.K.: Efficiently sampling probabilistic programs via program analysis. Proc. Artificial Intelligence and Statistics, pp. 153–160 (2013)
8. Perov, Y., Wood, F.: Learning Probabilistic Programs. arXiv:1407.2646 [cs.AI] (2014)
9. Solomonoff, R.: Algorithmic Probability, Heuristic Programming and AGI. In: Baum, E., Hutter, M., Kitzelmann, E. (Eds). Advances in Intelligent Systems Research, vol. 10 (proc. 3rd Conf. on Artificial General Intelligence), pp. 151–157 (2010)
10. Goertzel, B., Geisweiller, N., Pennachin, C., Ng, K.: Integrating Feature Selection into Program Learning. In: Kühnberger, K.-W., Rudolph, S., Wang, P. (Eds.): AGI'13, LNAI 7999, pp. 31–39 (2013)
11. McDermott, J., Paula, C.: Program Optimisation with Dependency Injection. In: Proc. 16th European Conference on Genetic Programming, EuroGP (2013)
12. Gordon, A.D., Henzinger, Th.A., Nori, A.V., Rajamani, S.K.: Probabilistic programming. In: Proc. International Conference on Software Engineering (2014)