Combinatorially Efficient Exploration of Program Transformations for Automatic Programming

Henrik Berg and Roland Olsson
Østfold College
1750 HALDEN
Norway
{Henrik.Berg,Roland.Olsson}@hiof.no http://www-ia.hiof.no/∼rolando

Abstract: Program induction, where one or more parts of a potentially huge software system are automatically synthesized, is an emerging technology that will become more and more industrially useful as more computing power becomes available, for example in the form of computing grids. The scalability of program induction primarily depends on the combinatorial properties of the program transformations that are employed. In this paper, we improve the scalability of the ADATE automatic programming system by using a standard tool from combinatorial design theory, namely covering arrays, to more efficiently explore combinations of program transformations. The paper presents a series of 18 experiments which show that the covering array transformation algorithm is a highly useful supplement to the old ADATE transformations.

Keywords: Program synthesis, automatic programming, scalability, combinatorial design theory, functional programming, program transformation

1 Introduction

Automatic Design of Algorithms through Evolution (ADATE) [7] is the leading system for induction of functional programs and is likely to be able to generate one or more correct programs for practically any algorithmic problem given enough CPU time.

Under certain assumptions, we have shown [8] that the time required for program induction with ADATE grows as the fourth power of the size of the code to be synthesized, which means that ADATE is very computationally demanding. Therefore, the most immediate practical application is to improve one or more small parts of a potentially huge software system instead of automatically generating the whole system from scratch. ADATE requires only that there exists one or more software performance measures that are to be be optimized, which holds for almost all software.

For example, consider the software controlling the Electrolux Trilobite robot vacuum cleaner, which was the world's first fully automatic domestic vacuum cleaner when commercially introduced in 2001 [3]. Given a simulation of the Trilobite and its environment, ADATE may automatically optimize part after part of its software with respect to cleaning ability for various room geometries and furniture configurations.

In this paper, we show how to reduce the time required for program synthesis by more efficiently exploring combinations of program transformations and achieve better scalability with respect to the size of synthesized code than the one mentioned above.

2 A brief overview of the ADATE system

ADATE maintains a population of programs that is organized so that each program must be better than all smaller ones found so far. Program transformations in varying combinations are employed to produce new programs that become candidates for insertion into the population.

The most fundamental program transformation in ADATE is *replacement*, abbreviated R, which replaces all or parts of a subexpression in the program with a newly synthesized expression. The algorithm for expression synthesis has sophisticated heuristics to guarantee that a large fraction of recursive calls are terminating, to avoid introducing dead code and quota systems guiding the use of local and global variables and functions.

There are three kind of R transformations that differ in how they reuse old code. The first kind of R transformation simply replaces an entire subexpression with a synthesized one. For example, consider the following factorial function, which is much too primitive to be used for testing ADATE but is useful in this overview since it is so simple.

```
f(n) = if n=0 then 1 else n*f(n-1)
```

Replacing the subexpression `n=0` with the synthesized expression `n<=1` is an example of the first kind of R transformation. The second and third kinds of R transformations reuse either the entire subexpression or some part of it respectively. For example, assume that the entire subexpression `n*f(n-1)`, below represented by `E`, is to be reused and that the synthesized expression is `n+E+n`.

The program resulting from this kind of R transformation, also called insertion, is

```
f(n) = if n=0 then 1 else n+n*f(n-1)+n
```

Note that this last R transformation does not preserve semantics whereas the previous one did and therefore also is a so called REQ transformation, where REQ stands for "replacement preserving equality". More exactly, REQ transformations are found by trying R transformations and selecting the ones that do not make the program worse. Note that REQs are expensive to find since lots of programs resulting from various R transformations are compiled, executed and checked for REQ-hood.

Recall that the first transformation given above was neutral. Another neutral transformation is an insertion reusing `f(n-1)` and that yields the new subexpression

```
if n-1=0 then 1 else f(n-1)
```

Combining these two neutral transformations gives the following program that has the same semantics as the original one.

```
f(n) = if n<=1 then 1 else
  n*(if n-1=0 then 1 else f(n-1))
```

The old version of ADATE systematically combines 2, 3 or 4 REQ transformations whereas the current version that uses the techniques from this paper may combine dozens or hundreds.

A given combination of REQ transformations may be viewed as a neutral walk in the search landscape. Such walks are quite important in ADATE as well as in many other optimization methods [9] and natural as well as artificial evolution [6, 4]. Neutral walks explore a plateau in the search landscape in order to find a point where it is easy to jump to the next higher plateau.

In addition to the R and REQ transformations, ADATE contains abstraction (ABSTR) that introduces new functions and case/lambda-distribution/lifting (CASE-DIST) that changes the scope of variables and functions. Both of the latter two transformations are neutral and also less combinatorially challenging than the R and REQ transformations.

To produce children programs from a given parent program, ADATE uses so-called compound transformations that are suitable combinations of R, REQ, ABSTR and CASE-DIST transformations. An example of a compound *transformation form* is ABSTR REQ REQ R, where one or both of the two REQ transformations typically use the function introduced by the ABSTR or its parameters.

The total number of children produced from a given parent is determined by a *cost limit* that is deepened iteratively, meaning that more and more children are generated from the parent as an ADATE run progresses.

3 A combinatorial argument

In any program-searching system like ADATE, the number of programs tested during the search tends to grow fast. For example, if ADATE needs to transform a program of size N using a combination of i REQ transformations, assuming that k different REQs can be generated at each position, the number of different ways to combine the i REQs is approximately $\binom{N}{i}k^i \in O(N^i k^i)$.

To simplify the discussion, we will introduce some notation for denoting these transformations, combinations and programs. At each position j, we denote the number of different REQs by k_j or sometimes just k, assuming they are all equal. Each of the different REQs will be denoted by a number in the interval $0..(k_j - 1)$.

Denoting REQs this way, we can represent a generated program by a line, where each position in the line corresponds to a position in the program, and each position is either empty, which is denoted by a "-" and means that no transformation was applied at that position, or contains a number in the range $0..(k_j - 1)$ that tell us which REQ was applied. The length of the line will of course be N, which is the size of the program.

As an illustration, Figure 1 contains such representations of the programs ADATE would generate if $N = 3$ and $k = 2$. The number of programs generated would be $\binom{3}{2}2^2 = 12$.

However, when the program-sizes increase, this seems like a big waste of time. If the system applies a couple of neutral transformations at a few different locations in the program, one might expect it to be possible to apply other neutral transformations at other locations at the same time, thus increasing the number of combinations tested, and decreasing the number of programs generated.

```
0 0 -
0 1 -
1 0 -
1 1 -
0 - 0
0 - 1
1 - 0
1 - 1
- 0 0
- 0 1
- 1 0
- 1 1
```

Figure 1:

For example, consider the array in Figure 2. For any pair of columns in the array, all the four possible combinations $\{00, 01, 10, 11\}$ occur at least once. And yet it only consists of four lines, one third of the array above.

This last array was an example of a *covering array* [11]. Covering and orthogonal arrays [5] are standard tools from combinatorial design theory to guarantee coverage of all pair-wise parameter combinations, for example, and were one of the major inspirations when we started this work. Generally, a *covering array with N columns, k levels and strength t* is an N-column array with the property that whichever set of t columns you choose, all the k^t different combinations of "levels", usually numbers in the range $0..(k - 1)$, occur at least once in the selected columns. A main objective is to make such an array as short as possible. This is a difficult task for the cases $t \geq 4$ but the literature contains numerous results concerning the cases $t \leq 3$ [2]. It is straightforward to construct a covering array with N columns, k levels and strength two with $O(k^2 log N)$ lines [10], which would be a big improvement over the current search in ADATE that tries $\binom{N}{2}k^2 \in O(k^2N^2)$ lines.

```
0 0 1
0 1 0
1 0 0
1 1 1
```

Figure 2:

4 Construction of covering arrays for program transfomations

When combining REQ transformations, ADATE does not know in advance how many REQs that will be needed. Therefore, it is not possible to know with certainty which strength that a covering array should have. We have chosen to guarantee total coverage only for strength one and then use a probabilistic array construction to cover pairs, triples, quadruples and other higher strength combinations of REQs.

When combining transformations that are individually neutral, it may happen that the combination is not neutral because two or more transformations interfere with each other. We handle such interference for a line in a covering array by partitioning the line into two or more lines that contain empty positions sufficient to eliminate interference. First, we greedily take as many REQs as possible from the original line without destroying neutrality into one partition. Then, we take as many of the remaining alleles as possible into a second partition, and so on until all the alleles have been partitioned into a hopefully quite small number of partitions, each of which is neutral with respect to fitness. Note that this partitioning strategy only guarantees coverage for strength one.

Our probabilistic algorithm for covering arrays of strength greater than one repeatedly runs the above strength one algorithm with a new random ordering of the REQs at each position for each new run.

With k different REQs at each position, the probability that a random line covers a given combination of two REQs is $\frac{1}{k^2}$. In a random array with s lines, the probability of a given combination of two REQs to be covered is $1 - (1 - \frac{1}{k^2})^s$. Setting the number of lines $s = Ak^2$ for some small constant A, this can be approximated by

$1 - (1 - \frac{1}{k^2})^{(Ak^2)} \approx 1 - e^{-A}$.

So, in an array with Ak^2 lines, the probability that a given pair is covered, is approximately $1 - e^{-A}$. For example, if the number of lines in our array is $3k^2$, the probability that a given pair is covered is about 95%, and thus 95% of all the pairs in the array should on average be covered.

In general, a randomly generated array with Ak^t lines will cover on average $1 - e^{-A}$ of all the t-tuples for any $1 \leq t \leq N$. So even though we only guarantee total coverage for the case $t = 1$, we are still able to cover a decent number of t-tuples for higher values of t by repeatedly running our algorithm, randomly reordering the alleles at each position each time.

In order to maximize the number of covered tuples for strength two and greater, we should strive for dense arrays. However, the above partitioning may give sparse arrays.

To remedy this, we loop through each of the lines in the array again, and try to insert a randomly chosen REQ at each empty position. Currently we only try once per position. If inserting the random REQ fails, meaning that it leads to a non-neutral line, we just skip it and leave it empty. We could of course try several times on failure. This would probably result in slightly denser arrays, but would also require slightly longer execution time.

5 Experiments

Since our work concerns the combination of neutral transformations in order to more efficiently search the local neutral fitness space, we decided to let the experiments concentrate on the local search for transformations from one specific program rather than a full scale population based search. We ran the main part of the ADATE system, namely the local search algorithms, from a number of different test programs.

Each experiment started from one given test program, and iteratively increased the cost limit used to transform that program. Each time a new and different program better than the test program was found, it was noted in the logfile, together with the transformation form used to generate the program (like REQ CASE-DIST R, or CA if the new covering array algorithm was used).

In order to be able to compare the new CA transformations with the old REQ-transformations, the time the system spent doing REQs was split into two: 50% of the CPU time was given to the old REQ algorithm, while the remaining 50% of the CPU time was given to our new CA algorithm.

The experiment was repeated 18 times, using several different specifications, and one, two or three different test programs for each specification. We used the following specifications in our experiments:

- **ASMB** - a specification for a function to parse and compute simple arithmetical expressions coded as a list of three bit binary words representing arbitrary precision binary numbers, parentheses, subtraction and multiplication, following the standard rules of precedence.
- **BOXPACK** - a specification for a function to pack as many boxes of different sizes as possible into another box, given the geometries of the boxes as parameters.
- **CHESS1** - a specification for a function to discover as many legal moves as possible given a representation of a chess board as its parameter. The specification contained only training input with one king and/or one rook in the same color, but none where they were able to reach each other.
- **CHESS1b** - the same specification as **CHESS1** but with a higher number of training inputs. The test program we used for this specification was the result of running ADATE for approximately 10 times as long as for the **CHESS1** experiment, and should therefore be much harder to improve.
- **CHESS2** - the same specification as **CHESS1**, but this time including training inputs where the king and the rook can reach each other, in other words the function has to check for collisions and jumping.
- **GS** - learning the syntax and semantics of a simple language as described in [8]. GS stands for Grounded Semantics, as used in linguistics, since the actions carried out by a synthesized program in response to input sentences are "grounded" in a simulated world.
- **PERMS** - a specification for generating all the possible permutations of a given input-list.

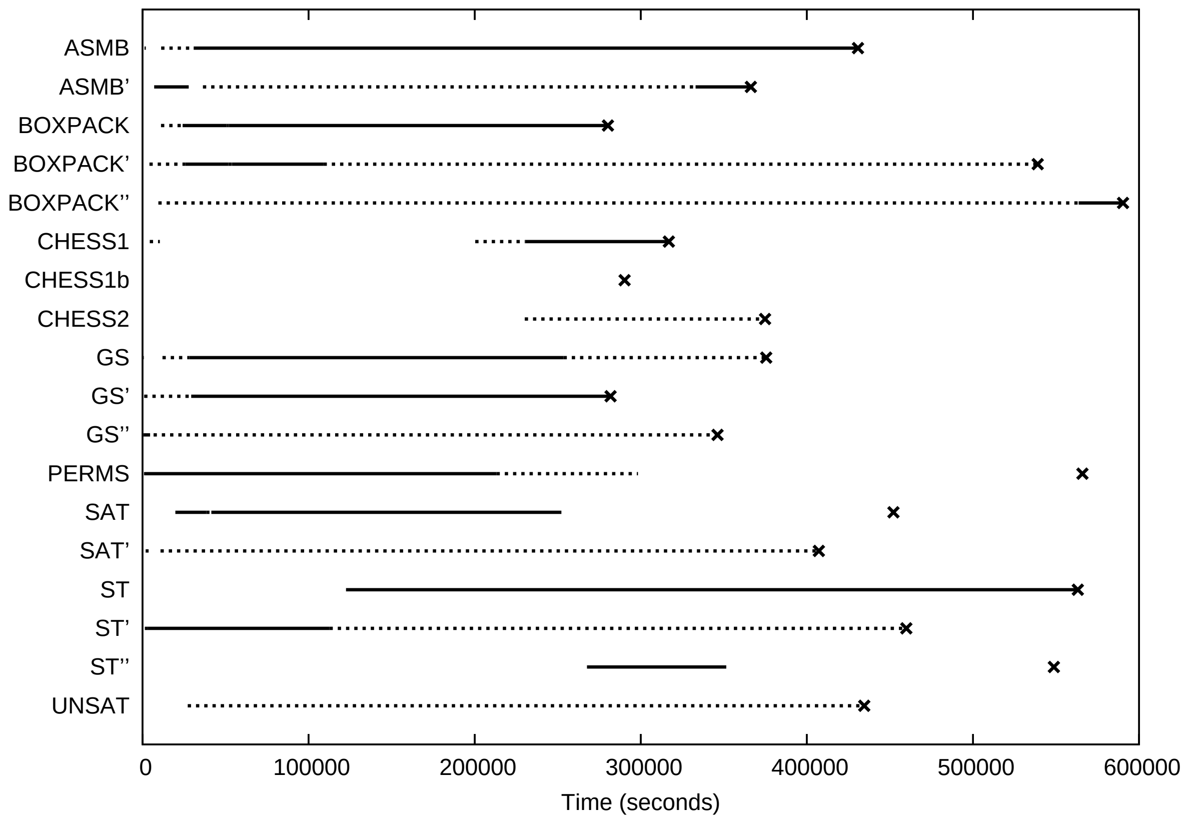

Figure 3: Timelines of the experiments we have done, representing which transformations that were used to produce the currently best programs at any time.

- **SAT** - a specification for the Satisfiability problem, which is the most fundamental NP-complete problem.
- **ST** - a specification for a function to act as a Stock Trader that tries to maximize profit when buying and selling stock every day. The simulated trading used to compute the evaluation function is based on 10 years of stock prices from the Swedish Stock Exchange for four different stocks.
- **UNSAT** - a specification for a function that uses extended resolution to try to solve the Unsatisfiability problem which is another fundamental problem in algorithm design and likely to be even more difficult than the SAT problem.

For each of the specifications, the best program found by a run of ADATE, that typically was given one week on a single CPU, was used as a test program. Being the best programs found by ADATE, these programs should be difficult to improve further. Additionally, some random test programs, marked with ' or " in the table, were chosen from old logfiles for some of the more interesting specifications. So we had a mix of "difficult" and "average" programs for our experiments.

Each of the experiments was run for about a week on a single-CPU (2.8GHz) computer. Afterwards, we extracted from the logfiles the transformation used each time a new program was constructed that was better than any of the previously constructed programs.

Each timeline in Figure 3 shows the points in time our new CA transformation was used to create the currently best program, and at which points in time the currently best program was created by using the REQ transformation. A solid line in a timeline represent segments of time in which CA was used in the construction of the currently best program, while a dashed line represents segments of time in which the currently best program was constructed using one or more REQs. At the empty areas of the timelines, no program better than the test program has been found, or the currently best program was created using neither CA nor REQ transformations.

6 Conclusions

In section 3, we theoretically showed that covering arrays may dramatically reduce the time complexity of ADATE. For example, if it is necessary to simultaneously perform two REQ transformations anywhere in a program of size N, the time complexity may be reduced by a factor $N^2/\log N$. If only one REQ is needed, the reduction can be a factor N.

In practice, the benefit of covering arrays depends on how often REQ transformations are needed, how many that need to be combined and how much various transformations interfere with each other as discussed in section 4.

In our experiments, the best final program was produced by covering arrays in 7 out of 18 runs in competition with all the old ADATE transformations. As synthesized program increase in size beyond the relatively small programs used in our experiments, the covering array transformation is likely to become more and more superior to the old REQ transformation algorithm.

The conclusion is that covering arrays or related methods, for example variants of genetic algorithms, are indispensable for the scalability of automatic programming as in ADATE.

References

[1] Brassard, G. and P. Bratley (1996). *Fundamentals of Algorithms.* Prentice-Hall, New Jersey, pp. 133 – 134.

[2] Chateauneuf, M. and D.L. Kreher (2000). On the state of strength-three covering arrays. *Journal of Combinatorial Designs*, *Vol. 10, Nr. 4*, pp. 217–238.

[3] Christensen, H.I. (2001). Intelligent Home Appliances. Centre for Autonomous Systems, Department of Numerical Analysis and Computer Science, Royal Institute of Technology, Stockholm, Sweden.

[4] Geard, N., J. Wiles, J. Hallinan, B. Tonkes and B. Skellett (2002). A comparison of neutral landscapes - NK, NKp and NKq. *Proceedings of the 2002 Congress on Evolutionary Computation*, pp. 205 – 210.

[5] Hedayat, A.S., N.J.A. Sloane and J. Stufken (1999). *Orthogonal Arrays.* Springer, New York.

[6] Kimura, M (1983). *The Neutral Theory of Molecular Evolution.* Cambridge University Press, Cambridge.

[7] Olsson, R (1995). Inductive functional programming using incremental program transformation. *Artificial intelligence*, *Vol. 74, Nr. 1*, pp. 55 – 83.

[8] Olsson, R. and D.M.W. Powers (2003). Machine learning of human language through automatic programming. *International Conference on Cognitive Science*, University of New South Wales, pp. 507 – 512.

[9] Selman, B., H. Kautz and B. Cohen (1996). Local search strategies for satisfiability testing. *DIMACS Series in Discrete Mathematics and Theoretical Computer Science*, *Vol. 26.* AMS.

[10] Sherwood, G (2004). On the Construction of Orthogonal Arrays and Covering Arrays Using Permutation Groups. *Webpage*, http://home.att.net/~gsherwood/cover.htm

[11] Sloane, N.J.A (1994). Covering Arrays and Intersecting Codes. *Journal of Combinatorial Designs*, *Vol. 1, Nr. 1*, pp. 51–63.