The Art of Writing Specifications for the ADATE Automatic Programming System

J. Roland Olsson
Department of Computer Science
Østfold College
Os Allé 11
1757 Halden
Norway
Roland.Olsson@hiof.no

ABSTRACT

The main difficulty for the user of an automatic programming system is to write a specification that the system can employ to synthesize desirable programs. We discuss how to write specifications for the ADATE system and present examples of specifications and the corresponding synthesized programs.

We show how ADATE can be used to synthesize parts of specifications as well as desirable programs and argue that multi-stage inferences are useful when specifying quite complex problems such as natural language understanding.

1 Introduction

Automatic Design of Algorithms through Evolution (ADATE) is a system for inductive functional programming developed independently of Genetic Programming (GP) (Koza 1992) and other program induction techniques such as Evolutionary Programming (EP) (Fogel 1996) and Inductive Logic Programming (ILP) (Muggleton 1992). Please see references (Olsson 1994, Olsson 1995) for general information on the ADATE system. Executable versions of the system and more documentation are available on the web (Olsson n.d.). Since ADATE has more in common with GP than with EP and ILP, we will only compare ADATE and GP. Below is a list of differences between ADATE and an ordinary GP system. Please note that it is difficult to understand how profound these differences are when reading the brief and superficial descriptions below.

1.1 Application Areas

Many, perhaps most, applications of GP concern the inference of real-valued mathematical models. The main goal of ADATE is to synthesize symbolic and purely functional programs for typical algorithm design problems (Aho *et al.* 1983). The final goal is the synthesis of "intelligent" symbolic programs highly skilled in logic, mathematics, computer science and human language.

1.2 Programming Language

Typical GP programs are written in LISP, sometimes including assignments, while-loops and other ingredients that are not used in pure functional programming. The programs synthesized by ADATE are written in a small, carefully selected and purely functional subset of Standard ML that uses rigorous typing and syntactic constraints.

1.3 Transformations

The ADATE transformations (genetic operators) have been designed by studying genealogical chains, where a chain is a sequence $P_1 - P_2 - \ldots - P_n$ of programs such that P_i usually is just a little bigger and a little better than P_{i-1}. When constructing such a chain by hand, we seek to find a combinatorially cheap transformation yielding P_i from P_{i-1}. Note that the objective of such theoretical chain making is to find "optimal" transformations and not to discover chains that actually will be generated experimentally, which is rather futile since the chains produced by ADATE almost always are quite different from the ones produced by hand.

We studied the first chains of gradually bigger and better functional programs in 1986 and had a working implementation before even knowing that GP existed.

The main program transformations used by GP are crossover, mutation and ADF. The first two of these seem to have been designed by rather direct modelling of natural Darwinian evolution, whereas the ADATE transformations have been specifically designed for artificial evolution of purely functional programs.

ADATE does not use crossover and always produces offspring from one single parent. The main transformations in ADATE are replacement (R), equivalence preserving replacement (REQ), abstraction (ABSTR), embedding (EMB) and case-distribution (CASE-DIST).

The R transformation replaces existing code with code syn-

thesized by a complex expression generator that for example considers the termination properties of recursive calls. An R can also insert code by cutting a branch in the syntax tree and reuse a subtree below the cut during expression synthesis. ADATE systematically tries all cuts that "make sense" considering the relationship to preceding transformations. The R transformation algorithm also tries substitutions i.e., replacing several equal subtrees with the same newly synthesized expression.

The REQ transformation moves along a plateau or ridge in the fitness landscape, trying to find suitable "jump points".

ABSTR has the same role as ADF in GP but with several key differences. The most important difference is that an ABSTR basically is the inverse of an inlining or β-expansion as done by an optimizing compiler, which means that the created function definition is based on already synthesized and hopefully "proven" code rather than being constructed from scratch as in ADF. Of course, the function definition usually continues to evolve even after being introduced by the ABSTR. ADATE tries all possible abstractions subject to constraints imposed by preceding transformations before possibly finding an ABSTR that is worth keeping.

EMB adds a parameter to an existing function and uses the expression generator or inverse inlining to update each call to the function with an extra argument.

CASE-DIST moves code one or several positions inside or outside case-expressions or let-expressions so that semantics is preserved.

ADATE combines these five basic transformations according to so-called coupling rules, yielding compound transformations such as CASE-DIST ABSTR REQ R, where only the R normally changes the semantics of a program.

1.4 Population-Based Search

GP produces one or two children from one or two parents and uses randomized selection. ADATE uses systematic, but carefully weighted, compound transformations to produce thousands or millions of children from one single parent. Zero, one or a small number of these children are selected on-the-fly for insertion into the population based on a size-fitness ordering. In contrast to GP, the size of the population varies dynamically. The number of children to produce from a given parent is deepened iteratively. Please see (Olsson 1998) for details on population management in ADATE.

Below, we discuss how to write specifications enabling incremental and evolutionary growth of desirable programs. In particular, we show how the ADATE system can be used not only to synthesize programs, but also to write parts of specifications, for example output evaluation functions (fitness functions). The paper assumes that the reader is fluent in functional programming (Reade 1989, Wikström 1987).

2 The Basics of Writing ADATE Specifications

A specification contains

1. Data type definitions describing the input and the output. These type definitions have the usual functional programming form and are analogous to a context-free grammar (Wikström 1987).

2. Primitives and other help functions to be used in synthesized programs.

3. Sample inputs.

4. The output evaluation function.

For example, ADATE uses the following specification to synthesize programs that add arbitrary, unsigned binary numbers. The best such program is shown in Figure 1.

1. **Type definitions.**

```
datatype bit = O | I
datatype unsigned =
    digit of bit
  | mku of unsigned * bit
```

2. **Help functions.** `false, true, O, I, digit, mku`

3. **Sample inputs.** The following set of pairs in binary form:

$$\{0,1,2,3,4,5,6,7\}^2 \cup \{100,1000,10000,100000\}^2$$

The last 16 inputs, which are "big" are used to improve the time complexity of the synthesized programs.

4. **Output evaluation function.** Uses input-output pairs to check if the output is the sum of the inputs.

The four specification ingredients above should be chosen according to the following guidelines.

1. **Type definitions.** The type definitions should be highly specific, which is generally accepted as good functional programming practice. In other words, the grammar equivalent to the type definitions should generate a small number of sentences of a given maximum complexity.

   When inferring complex programs, it may be necessary to let the type definitions evolve along with the programs, but more research is needed before this coevolution can be optimally implemented.

2. **Help functions.** The run time of an inference is naturally shortened if the set of help functions include exactly the functions needed to solve the problem at hand. However, the specifier frequently cannot anticipate which help functions that are needed and must not be required to do so. In contrast to other inductive inference systems, ADATE can invent its own recursive help functions as they are needed.

```
fun uadd( (V2_5, V2_6) : unsigned * unsigned ) : unsigned =
  case V2_5 of
    digit( V2_1bc9 ) => (
      case V2_1bc9 of
        O => V2_6
      | l =>
      case V2_6 of
        digit( V2_fe9956 ) => (
          case V2_fe9956 of O => V2_5 | l => mku( V2_6, O ) )
      | mku( V2_fe9957, V2_fe9958 ) =>
          mku( uadd( digit( V2_fe9958 ), V2_fe9957 ),
               case V2_fe9958 of O => l | l => O ) )
  | mku( V2_1bca, V2_1bcb ) =>
  case V2_6 of
    digit( V2_9ac20f4 ) => uadd( V2_6, V2_5 )
  | mku( V2_9ac20f5, V2_9ac20f6 ) =>
  case V2_1bcb of
    O => mku( uadd( V2_1bca, V2_9ac20f5 ), V2_9ac20f6 )
  | l =>
      mku( uadd( uadd( V2_1bca, digit( V2_9ac20f6 ) ), V2_9ac20f5 ),
           case V2_9ac20f6 of O => V2_1bcb | l => O )
```

Figure 1 The best synthesized program for binary addition.

3. **Sample inputs**. In general, the sample inputs should be "pedagogical" and support gradual and evolutionary learning. Elementary school textbooks in mathematics and English provide inspiration for the art of pedagogical input design. However, the ADATE system often manages well even if the specifier does not possess the impressive pedagogical skills of elementary textbook authors.

   One general principle is to "cover all cases" and to include even the very simplest inputs. A simple example of a design strategy is to choose all inputs not bigger than some small maximum size. It is usually wise to include some big inputs, which helps to eliminate inferred programs with bad time complexity and also contributes to avoiding rote learning.

   In general, specifications of boolean functions require very many sample inputs. Intuitively, the reason is that the probability of producing a correct output for a given input by just "random guessing" is 50%, which is much greater than for most non-boolean outputs. Thus, a specification of a predicate requires particularly many sample inputs to avoid synthesized programs that are "lucky" on all inputs in the specification but unable to solve unseen inputs i.e., programs without generalizing ability.

4. **The output evaluation function.** The simplest output evaluation function uses input-output pairs to decide if an output is correct or wrong. Input-output pair specifications are easy to write for non-programmers, but also quite limited, representing only a drop in the sea of all ADATE specifications. It is difficult to give general rules for writing output evaluation functions since the possibilities are almost endless and primarily restricted by the specifier's creativity.

   ADATE uses the output evaluation function to grade synthesized programs so that even microscopic program improvements can be identified, which is essential for effective evolutionary program synthesis. However, a grading scale that is too fine-grained will introduce misleading improvements and increase run times because more programs need to be evaluated. A coarse-grained scale, on the other hand, gives a small population, but may increase run times due to missing links in the genealogical chain of programs leading from the initial program to a desirable program.

3 Semi-Automatic Specification

This section examines and exemplifies how the ADATE system can write parts of the specification. We will use the system to construct two of the four specification ingredients given in the previous section namely the help functions and the output evaluation function.

3.1 Using the System to Produce Help Functions

Sometimes, the specifier knows that a certain help function will be needed but cannot or do not want to define it. There are then two obvious options:

1. Expecting the system to invent the help function on-the-fly during the synthesis of the main function.

2. Writing an ADATE specification of the help function and synthesizing it before starting the synthesis of the main function.

Since a help function may need other help functions, these options may be applied recursively. We will now present an example where it is natural to use option two.

Example. The mathematics education in elementary school follows a hierarchy according to a carefully designed plan. For example, addition is always taught before multiplication. Consider synthesizing programs for adding and multiplying binary numbers of arbitrary length. We first specified addition as described in Section 2, ran the ADATE system and then used the synthesized function, shown in Figure 1, as a help function in the specification of multiplication. The specification for multiplication and the synthesized multiplication program are available on the web (Olsson n.d.).

3.2 Using the System to Produce Output Evaluation Functions

For many interesting and complex problems, the output evaluation function seems to be about as difficult to write as the main function. In such cases, it is natural to first use the system to produce the difficult part of the output evaluation function and then the main function. Several of these problems are too difficult for the current version of the ADATE system using the available computational resources, but they should still be studied with great care in order to guide the future development of the system. Here is such a problem.

Example. Consider the problem of reading, comprehending and writing English using a formal semantics as in (Muskens 1995). Assume that the sample inputs consist of a number of formalized texts with associated questions that a synthesized program should be able to answer. Given a text, a question and an answer produced by a synthesized program, the output evaluation function needs to grade the answer. Unless the formal language is a trivial subset of a natural language, it seems extremely difficult to write such an output evaluation function. Therefore, the following two-stage inference may be necessary.

1. Write a specification where each input consists of a text, a question and an answer. The same text and question typically occur several times but with different answers. It is assumed that the specifier knows how to rank his/her/its own different answers to the same question. The job of a synthesized program is to grade an answer. The output evaluation function checks how well the program's grading agrees with the ranking given by the specifier.

2. Use the best synthesized program from stage one in the output evaluation function in the specification of programs that produce answers to questions.

Here is a much easier problem that the ADATE system has solved and that also benefits from a multi-stage inference.

Example. Given a list `Xs`, the problem is to compute a list of lists `Yss` that contains all permutations of `Xs`. For example, if the input `Xs` is [1,2,3], one possible output `Yss` is [[1,2,3], [1,3,2], [2,1,3], [2,3,1], [3,1,2], [3,2,1]].

The list of lists `Yss` is required to only contain sublists that are permutations of `Xs` and each permutation may occur only once. These two requirements may be checked using two functions `is_perm` and `is_set` that work as follows. The call `is_perm(Xs,Ys)` returns true if and only if `Ys` is a permutation of `Xs`. The call `is_set Yss` returns true if and only if `Yss` does not contain duplicates.

The following three stage inference was employed to produce a permutation generation program:

- We specified `is_perm` using input-output pairs and used ADATE to produce an implementation of this function. Note that `is_perm` is a predicate and that predicates require especially many sample inputs as discussed in Section 2. The specification of `is_perm`, available through (Olsson n.d.), contains 59 input-output pairs.

- Also using input-output pairs, we employed ADATE to produce an implementation of `is_set`. Please see (Olsson n.d.) for details.

- We wrote the following specification of permutation generation:

  1. **Type definitions.**

     ```
     datatype int_list =
         nil
       | cons of int * int_list

     datatype int_list_list =
         nil'
       | cons' of int_list * int_list_list
     ```

  2. **Help functions.** `false, true, nil, nil', cons, cons', append, append'.`

     Note that the functions `append` and `append'`, which concatenate lists and lists of lists respectively, are predefined as a single polymorphic function in practically all functional languages

  3. **Sample inputs.** The following five lists written in simplified Standard ML notation:

     `[], [1], [1,2], [1,2,3], [1,2,3,4]`

     Only five inputs are needed since several desirable outputs are reasonably complex and difficult to produce just through "lucky guesses".

  4. **Output evaluation function.** The output `Yss` is checked using `is_set Yss` and each sublist `Ys` in `Yss` is checked using `is_perm(Xs,Ys)`. If `Yss` passes this check, it is given a grade $-n$ where n is the number of sublists in `Yss`. Since ADATE strives to minimize grades, we use $-n$ instead of n.

With this specification, ADATE produced the program in Fig. 2 that contains two recursive help functions invented "on-the-fly" by ADATE. The topmost help function apparently produces all rotations of a list. The first parameter to this function seems to be the list to be rotated whereas the second parameter is used to count the number of rotations performed so far. The other help function applies the rotation function to a list of lists.

4 Summary, Conclusions and Future Work

We have discussed the form of ADATE specifications and given some general advice on how to make them pedagogical i.e., how to design them for effective machine learning. However, the primary contribution of the paper is the automatic synthesis of parts of specifications, particularly output evaluation functions. This automatization will be necessary when synthesizing complex programs for which the output evaluation function is too difficult to write manually. We have also presented several non-trivial examples of ADATE specifications and the resulting synthesized programs.

The multi-stage inference process proposed in the paper is useful now, but will become indispensable when specifying and synthesizing programs that are many orders of magnitude more complex than the ones produced today. The ADATE system may already have most of the techniques needed for such complex inferences. The primary requirement for running them may be efficient parallelization of the ADATE system and a parallel computer with a few thousand CPUs.

A particularly exciting area for future research is automatic self-improvement using the following technique. Select a part of the ADATE code, for example the expression synthesis function (Olsson 1994), specify it and run the ADATE system to try to produce a code part that is better than the one used in the current version of the ADATE system. Since both the ADATE code and the synthesized programs are written in Standard ML, it is feasible to replace a part of the system code with synthesized code. If a better code part is synthesized and used as ADATE code, the overall performance of the ADATE system is improved which may enable it to synthesize code that is better than some other part of itself. Continuing in this manner, we may obtain cyclic self-improvement.

References

Aho, Alfred V., John E. Hopcroft and Jeffrey D. Ullman (1983). *The Design and Analysis of Computer Algorithms*. Addison-Wesley.

Fogel, David B. (1996). *Evolutionary Computation: Toward a New Philosophy of Machine Intelligence*. IEEE Press.

Koza, John R. (1992). *Genetic Programming: on the Programming of Computers by Means of Natural Selection*. MIT Press.

Muggleton, Stephen, Ed.) (1992). *Inductive Logic Programming*. Academic Press.

Muskens, Reinhard (1995). *Meaning and Partiality*. CSLI Publications and FoLLI.

Olsson, Jan Roland (1994). Inductive Functional Programming using Incremental Program Transformation and Execution of Logic Programs by Iterative-Deepening A* SLD-Tree Search. PhD thesis. University of Oslo. ISBN 82-7368-099-1.

Olsson, Jan Roland (1995). Inductive functional programming using incremental program transformation. *Artificial Intelligence* **74**(1), 55–83.

Olsson, Jan Roland (1998). Population management for automatic design of algorithms through evolution. In: *International Conference on Evolutionary Computation*. IEEE Press.

Olsson, Jan Roland (n.d.). Web page for automatic design of algorithms through evolution. http://www-ia.hiof.no/∼rolando/adate_intro.html (current Jun. 26, 1997).

Reade, Chris (1989). *Elements of Functional Programming*. Addison-Wesley.

Wikström, Åke (1987). *Functional Programming using Standard ML*. Prentice Hall International.

```
fun f (V2_18) =
  case V2_18 of
    nil => cons’( V2_18, nil’ )
  | cons( V2_8a7fd, V2_8a7fe ) =>
  let
    fun g2_4b8d36f (( V2_4b8d370, V2_4b8d371 )) =
      case V2_4b8d370 of
        nil => ?_NA_2_804fcaa
      | cons( V2_4b8d36b, V2_4b8d36c ) =>
      cons’( V2_4b8d370,
        case V2_4b8d371 of
          nil => nil’
        | cons( V2_4b7c434, V2_4b7c435 ) =>
        g2_4b8d36f( append( V2_4b8d36c, cons( V2_4b8d36b, nil ) ), V2_4b7c435 )
        )
  in
    let
      fun g2_804ea1c (V2_804ea1d) =
        case V2_804ea1d of
          nil’ => V2_804ea1d
        | cons’( V2_7ff6207, V2_7ff6208 ) =>
        append’( g2_4b8d36f( cons( V2_8a7fd, V2_7ff6207 ), V2_7ff6207 ),
          g2_804ea1c( V2_7ff6208 )
          )
    in
      g2_804ea1c( f( V2_8a7fe ) )
    end
  end
```

Figure 2 The best synthesized program for permutation generation.