Automatic Design of Algorithms Through Evolution

(version 0.3)

—

User Manual

Roland Olsson

May 16, 1997

1 Introduction

This document describes how to use and install the latest release of the system **Automatic Design of Algorithms Through Evolution** (ADATE). ADATE itself is described in the document found in the file `01-report.ps.Z`.

The only input to the ADATE system is a specification file, which contains requirements for the programs to be synthesized. The specification tells the system *what* synthesized programs should do, but not necessarily *how* these programs do their job. Let us consider a program that is almost impossible to write for human programmers, for example one that can analyze and synthesize English text as well as (or better than) humans. Assume that this program is called $P_{intelligent}$. Using a variant of so-called Montague semantics, it is feasible to write a pedagogical ADATE specification of $P_{intelligent}$ in a few months. Given vast amounts of CPU time, the ADATE system may very well succeed in synthesizing $P_{intelligent}$. We believe that the first computer program ever to exhibit general intelligence is likely to be $P_{intelligent}$.

2 The Basic Form of an ADATE Specification

In the very simplest kind of specification, the requirements are input-output pairs. The user of the ADATE system needs to provide the following information when writing an input-output pair specification.

1. Data type definitions describing the input and the output.
2. A set of input-output pairs.
3. The primitives and other help functions to be used in synthesized programs.

Example. In order to illustrate an input-output pair specification, consider reversing a list of integers, which is a simple problem indeed. Assume that the specification file is called `rev.spec` and that the user has added the following.

1. The data type for lists of integers. This type may be defined as follows in Standard ML:

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

   This definition says that a list either is empty, denoted by `nil` or has the form `cons(` X`,` Xs `)`, where X has type `int` and Xs has type `list`. Thus, the type expression `int * list` is the domain of the constructor `cons` and means the Cartesian product of `int` and `list`.

For example, `cons( 95, cons( 49, cons( 37, nil ) ) )` denotes the list containing 95, 49 and 37.

The function to be inferred is always called `f` and has the following input and output types in this example:

```
type input_type = list
type output_type = list
```

2. The input-output pairs:

```
val Input_output_pairs = [
  ( nil, nil ),
  ( cons(37,nil), cons(37,nil) ),
  ( cons(49,cons(37,nil)), cons(37,cons(49,nil)) ),
  ( cons(95,cons(49,cons(37,nil))), cons(37,cons(49,cons(95,nil))) )
  ]
```

Four input-output pairs are given above. The first input is an empty list. The second is a singleton list. The third input is the list 49, 37. The third output is 37, 49. The fourth input is 95, 49, 37. The corresponding output is 37, 49, 95.

3. The predefined functions that may be used during synthesis, namely:

```
val Funs_to_use = [ "nil", "cons" ]
```

Thus, the only auxiliary functions to be used are the constructors `nil` and `cons`, which were defined in the data type definition for `list`

Using this specification, the ADATE system produces the following list reversal program in about 10 minutes on a PC with a 200 MHz PentiumPro processor:

```
fun f (V2_4) =
  let
    fun g2_86a47 (( V2_87117, V2_86a48 )) =
      case V2_86a48 of
        nil => V2_87117
      | cons( V2_872ff, V2_87300 ) =>
      g2_86a47( cons( V2_872ff, V2_87117 ), V2_87300 )
  in
    g2_86a47( nil, V2_4 )
  end
```

The specification file `rev.spec` contains default declarations in addition to the ones above. These default declarations are the same for all input-output pair specifications and may be copied when writing a new specification file. Figure 1 shows the `rev.spec` file including the default declarations.

```
datatype list = nil | cons of int * list
type input_type = list
type output_type = list

%%

val Input_output_pairs = [
  ( nil, nil ),
  ( cons(37,nil), cons(37,nil) ),
  ( cons(49,cons(37,nil)), cons(37,cons(49,nil)) ),
  ( cons(95,cons(49,cons(37,nil))), cons(37,cons(49,cons(95,nil))) )
  ]

val Inputs = map( #1, Input_output_pairs )

val Outputs = Array.fromList( map( #2, Input_output_pairs ) )

val Funs_to_use = [ "nil", "cons" ]

val Reject_funs = []
fun restore_transform D = D

structure Grade : GRADE =
struct
type grade = unit
val zero = ()
val op+ = fn(_,_) => ()
val comparisons = [ fn _ => EQUAL ]
val toString = fn _ => ""
val fromString = fn _ => SOME()
end

fun output_eval_fun( I : int, _ : input_type, Y : output_type, _ )
  : output_eval_type * Grade.grade =
  if Array.sub( Outputs, I ) <> Y then
    (wrong, ())
  else
    (correct, ())

val N = 0
val Theta = 0.25
val Max_delta_class_card = N
val Delta_class_synt_compl_ratio = if N = 0 then 1.0 else real_pow( 2.0, 1.0/real(N) )
val Max_output_class_card = N
val Max_scm_class_card = N
val Scm_class_synt_compl_ratio = if N = 0 then 1.0 else real_pow( 1.35, 1.0/real(N) )
```

Figure 1: The specification file for list reversal.

3 Choosing Data Types and Sample Inputs

Version 0.3 of the ADATE system only allows monomorphic data type definitions. The reason for this restriction was to simplify the ADATE ML compiler and to speed up compilation and execution.

A set of data type definitions is analogous to a set of rules in a context-free grammar. Each grammar rule corresponds to a `datatype` definition and each alternative in a rule has a corresponding alternative in the `datatype` definition. Each non-terminal in the grammar corresponds to a data type. Therefore, the writing of data type definitions is equivalent to defining a context-free grammar.

Example. Consider example 2.7 in the book "Compilers: Principles, Techniques and Tools" by Alfred Aho, Ravi Sethi and Jeffrey Ullman. The following context-free grammar generates sequences of instructions for a robot:

seq → *seq instr* | **begin**
instr → **east** | **north** | **west** |**south**

The corresponding `datatype` definitions are:

```
datatype seq = mk_seq of seq * instr | begin
datatype instr = east | north | west | south
```

For example, the command sequence

begin north west south

corresponds to the following term in Standard ML:

```
mk_seq( mk_seq( mk_seq( begin, north ), west ), south )
```

Due to the analogy between a set of `datatype` definitions and a grammar, we will refer to such a set as a grammar. In general, the grammar in the specification should be as specific as possible i.e., generate a minimum number of sentences of a given size. Good specificity is essential to reduce the amount of combinatorial search during program synthesis.

When the grammar has been chosen, the next step is to choose sample inputs. As a rule of thumb, one may choose all sentences of size n for $n = 0, 1, 2, \ldots$ as sample inputs. It is also wise to include a few rather long sentences, for example with $n > 30$, which helps the system to eliminate inferred programs with bad time complexity and also contributes to avoiding rote learning. The choice of sample inputs should be "pedagogical" and support gradual, incremental and evolutionary learning. One general principle is to try to cover all special cases. The authors of textbooks in mathematics and English for elementary school are masters in the art of pedagogical specification. However, the ADATE system often manages well even if the specification writer does not posess the impressive pedagogical skills of such authors.

4 How to Install and Run the ADATE System

In order to install the ADATE system, obtain at least one of the following files using anonymous ftp from the directory `/pub/adate` on `ftp.hiof.no` (158.36.33.3).

`03-adate.x86-linux.tar.Z` The ADATE system for x86 processors running Linux.

`03-adate.x86-win32.zip` The ADATE system for x86 processors running Windows 95/NT.

After having unpacked one of these files in a suitable directory, you may start the ADATE system using the command `adate`. Assuming that you want to run the system on the file called `rev.spec`, shown in Figure 1, issue the following commands

```
make_spec "rev.spec";
use "rev.spec.sml";
structure G = GFn( spec );
G.start( ".", "rev.trace", "rev.log" );
```

The `make_spec` command creates a file called `rev.spec.sml` that is loaded into the system with the `use` command. The next command creates a Standard ML structure `G`, which contains the following two commands:

`start`*(Savedir, Tracefile, Logfile)* Starts a synthesis and writes each expanded individual to *Logfile*. *Savedir* is the name of a directory where backup copies of the population will be saved in files called `saved.`n`.pop` for $n = 0, 1, 2, \ldots$. Statistics from the run will be printed to *Tracefile*.

`restart`*(Savedir, Populationfile, Tracefile, Logfile)* Restarts a run using the individuals in *Populationfile* as the initial population. Appends the output described above to *Tracefile* and *Logfile*. Immediately after restart, the system prints the best individual in *Populationfile* to *Logfile*, which means that this command also can be used for determining which individual that is the best in for example `saved.0.pop`.

Exercise 1. Write a specification of a list concatenation program. Run the ADATE system using your specification as described above. The system should produce a desirable program in about 5 minutes on a 200 MHz PentiumPro machine.

5 Advanced Specifications

Input-output pair specifications are reasonably easy to write without knowing Standard ML but also quite limited, representing only a drop in the sea of all ADATE specifications. If the specification writer knows Standard ML well, the possibilities are almost endless and primarily restricted by his, her or its creativity, where the word "its" is used to also include machines in the set of potential specification writers. This section assumes that the reader is an experienced Standard ML programmer.

As can be seen in Figure 1, the specification file consists of two parts separated by %% on a single line. In general, the part before %% contains definitions of data types and auxiliary functions, but note that no auxiliary functions are defined in Figure 1. This first part contains all information needed by the ADATE-ML compiler that compiles synthesized programs. The second part i.e., the part after the %% delimiter, contains Standard ML code that is compiled by the Standard ML of New Jersey compiler.

The following values need to be defined in the specification file:

```
type input_type
type output_type

val Inputs : input_type list
val Funs_to_use : string list
val Reject_funs : ( ('a,'b)Ast.e -> bool ) list
val restore_transform : ('a,'b)Ast.d -> ('a,'b)Ast.d
structure Grade : GRADE
val output_eval_fun : int * input_type * output_type * int ->
  Ast_lib.output_eval_type * Grade.grade

val Theta : real
val Max_delta_class_card : int
val Delta_class_synt_compl_ratio : real
val Max_output_class_card : int
val Max_scm_class_card : int
val Scm_class_synt_compl_ratio : real
```

The types `input_type` and `output_type` are defined before %%. The other values are defined after %%. The values have the following meaning:

`input_type` The domain type of the function `f` that is to be inferred.

`output_type` The range type of the function `f` that is to be inferred.

`Inputs` The sample inputs on which synthesized programs are tested.

`Funs_to_use` The predefined functions to be used when synthesizing programs. Should include all such functions i.e., constructors, functions defined before %% and functions that are built into the ADATE ML compiler.

`Reject_funs` A list of predicates used to prune synthesized expressions. Such a predicate should return `true` if and only if it is called with an expression that should be pruned. Please see the file `01-report.ps` for a definition of the type of expressions.

`restore_transform` May be used to transform programs that are saved in a file, for example "saved.0.pop", when that file is read back by the ADATE system. This makes it possible to let the specification evolve using successive such restores.

`Grade` This structure contains an ADT for grading inferred programs and has the following signature:

```
signature GRADE =
sig

type grade
val zero : grade
val op+ : grade * grade -> grade
val comparisons : ( grade * grade -> order ) list
val toString : grade -> string
val fromString : string -> grade option

end
```

The list `comparisons` contains a list of functions such that each function is one way of comparing two grades. A smaller grade is considered to be better than a greater one.

`output_eval_fun`*(I, X, Y, T)* This function is called by the ADATE system when it wants to rate an output produced by a given synthesized program. Argument I is the order number of the output, which is the same as the order number of the corresponding input in the list `Inputs`. X is the input. Y is the output. T is the number of function calls needed to produce Y from X using the given program. The type `Ast_lib.output_eval_type` contains the constants `correct` and `wrong`.

The remaining six values i.e., `Theta, Max_delta_class_card`, ..., are used to control the population-based search. The techniques controlled by these parameters are quite sophisticated and not suitable to explain here. If extreme optimization of the parameters is not required, it suffices to only vary the value `N` shown in Figure 1, which there provides values for the five last parameters. Increasing `N` improves the ability to avoid local optima and makes the population bigger. For simple problems, `N` may be chosen to 0 which gives a population of minimum cardinality.

If you have access to many CPUs, you can try `N=0` on the first CPU, `N=1` on the second, `N=2` on the third, `N=4` on the fourth, `N=8` on the fifth and so on.

Example. Appendix A contains a specification file for the problem of producing a list of lists containing all permutations of a given list. The following individual was synthesized after about 400000 seconds of run time on a 200 MHz PentiumPro CPU:

```
Individual id = 0_a31989

Ancestor ids = [ 0_a31989 0_5d34f0 0_128e7 0_fde0 0_fbab 0_f1bf 0_962b
  0_91f1 0_8677 0_83ff 0_7c6e 0_786d 0_6d0a 0_6a3d 0_62b6 0_21b2 0_4 0_1 ]

Ancestor evals = [  :  5 0 ~34 5  :  5 0 ~11 5  :  5 0 ~8 5  :  5 0 ~8 5
  :  5 0 ~7 5  :  5 0 ~7 5  :  5 0 ~7 5  :  5 0 ~7 5  :  5 0 ~7 5
  :  5 0 ~7 5  :  5 0 ~6 5  :  5 0 ~6 5  :  5 0 ~6 5  :  5 0 ~6 5
  :  5 0 ~7 5  :  5 0 ~5 5  :  5 0 ~5 5  :  0 0 0 0 ]

Max cost limit used = 10000.0

Eval fun value =
 5 0 ~34 5 253 [ 162.962004661 167.301700294 ] 23.93796455


Program =
fun f (V2_18) =
  case V2_18 of
    nil => cons'( V2_18, nil' )
  | cons( V2_1f78a8, V2_1f78a9 ) =>
  let
    fun g2_c1e297f (( V2_c1e2980, V2_c1e2981 )) =
      case V2_c1e2980 of
        nil => ?_NA_2_14cb4296
      | cons( V2_c1e297b, V2_c1e297c ) =>
      cons'(
        V2_c1e2980,
        case V2_c1e2981 of
          nil => nil'
        | cons( V2_c1ae4a9, V2_c1ae4aa ) =>
        g2_c1e297f( append( V2_c1e297c, cons( V2_c1e297b, nil ) ), V2_c1ae4aa )
        )
  in
    let
      fun g2_14cb1b96 (V2_14cb1b97) =
        case V2_14cb1b97 of
```

```
          nil' => V2_14cb1b97
        | cons'( V2_14be80de, V2_14be80df ) =>
        append'(
          g2_c1e297f( cons( V2_1f78a8, V2_14be80de ), V2_14be80de ),
          g2_14cb1b96( V2_14be80df )
          )
    in
      g2_14cb1b96( f( V2_1f78a9 ) )
    end
  end

Local trf history =
REQ [ 2 1 1 ]
CASE-DIST  [ 2 1 2 ]  [ 2 1 ]
ABSTR [ 2 1 ]
REQ [ 2 1 0 2 0 ]
R [ 2 1 0 2 ]  [ 2 1 0 2 ][ ]
R [ 2 1 0 1 ][ ]
Creation time = 403808.06  Adjusted cost limit = 36000.0
Global time = 411649.59
```

The evaluation function value is printed using the following fields:

5 The number of correct answers.

0 The number of wrong answers.

~34 The grade.

5 The number of genuinely correct answers.

253 The total number of function calls.

The next two values are two different syntactic complexity measures. An explanation of the last value is beyond the scope of this text.

Exercise 2. Write a specification for the problem of adding unsigned binary numbers using the following data type:

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

Note that the constructors for the type `bit` are the letters big O and small l. If your specification is pedagogical, the system will produce a desirable program after less than 500000 seconds of CPU time.

Exercise 3. Specify the problem of multiplying binary numbers. Use the program inferred in Exercise 2 as an auxiliary function before the %% delimiter. Watch out for synthesized multiplication algorithms with bad time complexity.

Exercise 4. Determine the optimum values of `N` for Exercise 2 and Exercise 3. Note how the choice of `N` may influence:

1. The syntactic complexity of the smallest completely correct algorithm.
2. The time required to find the first completely correct algorithm.

Exercise 5. Recall the data type for sequences of robot instructions in Section 3. In this exercise, the problem is to make the robot return to the starting point assuming that the sequence of move instructions it has followed is known. Given such a sequence, a desirable program should produce another sequence that makes the robot return to the starting point. Use the length of the output sequence as a grade. Note that `f` has the type

```
seq -> seq
```

Exercise 6. Note that this exercise is quite difficult and still unsolved. Specify the n queens problem in two stages as follows. First write a specification of an "okay" function that checks if a given list of queen positions is such that no queen can hit another and run the ADATE system to infer this function. Use a unary number data type to represent column numbers, e.g.,

```
datatype num = zero | succ of num
```

Then use the inferred "okay" function to define `output_eval_fun` in a new specification, which describes the problem of placing as many queens as possible on a board with a given initial configuration of queens i.e., list of column numbers. Run the ADATE system on this specification and send mail to `Roland.Olsson@hiof.no` if a desirable program is produced.

A The Specification File for the List Permutation Problem

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

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

type input_type = int_list
type output_type = int_list_list

fun append( (Xs,Ys) : int_list * int_list ) : int_list =
  case Xs of
    nil => Ys
  | cons(X1,Xs1) => cons(X1,append(Xs1,Ys))
```

```
fun append'( (Xss,Yss) : int_list_list * int_list_list )
    : int_list_list =
  case Xss of
    nil' => Yss
  | cons'(Xs1,Xss1) => cons'(Xs1,append'(Xss1,Yss))



%%
fun from_list nil = []
  | from_list( cons(X1,Xs1) ) = X1 :: from_list Xs1

fun from_list' nil' = []
  | from_list'( cons'(Xs1,Xss1) ) =  from_list Xs1 :: from_list' Xss1

fun to_list [] = nil
  | to_list (X::Xs) = cons( X, to_list Xs )

val Inputs = map( fn N => to_list( fromto(1,N) ), fromto(0,4) ))

val Funs_to_use = [ "false", "true", "nil", "nil'",
  "cons", "cons'", "append", "append'" ]

val NIL = string_to_symbol( func_sym, "nil" )
val NIL' = string_to_symbol( func_sym, "nil'" )
val APPEND = string_to_symbol( func_sym, "append" )
val APPEND' = string_to_symbol( func_sym, "append'" )
val CONS = string_to_symbol( func_sym, "cons" )
val CONS' = string_to_symbol( func_sym, "cons'" )

fun append_id_left( app_exp{ func, args=app_exp{func=F,...}::_, ...} ) =
      func = APPEND andalso F = NIL
  | append_id_left _ = false

fun append_id_right( app_exp{ func,
        args= _::app_exp{func=F,...}::_, ...} ) =
      func = APPEND andalso F = NIL
  | append_id_right _ = false

fun append_assoc( app_exp{ func,
        args= app_exp{ func=F, args=_::_::[], ...}::_, ...} ) =
      func = APPEND andalso F = APPEND
  | append_assoc _ = false
```

```
fun append_cons( app_exp{ func, args=app_exp{func=F,...}::_, ...} ) =
      func = APPEND andalso F = CONS
  | append_cons _ = false


fun append_id_left'( app_exp{ func, args=app_exp{func=F,...}::_, ...} ) =
      func = APPEND' andalso F = NIL'
  | append_id_left' _ = false

fun append_id_right'( app_exp{ func,
        args= _::app_exp{func=F,...}::_, ...} ) =
      func = APPEND' andalso F = NIL'
  | append_id_right' _ = false

fun append_assoc'( app_exp{ func,
        args= app_exp{ func=F, args=_::_::[], ...}::_, ...} ) =
      func = APPEND' andalso F = APPEND'
  | append_assoc' _ = false

fun append_cons'( app_exp{ func, args=app_exp{func=F,...}::_, ...} ) =
      func = APPEND' andalso F = CONS'
  | append_cons' _ = false

val Reject_funs = [
      append_id_left, append_id_right, append_assoc, append_cons,
      append_id_left', append_id_right', append_assoc', append_cons' ]

fun restore_transform D = D

structure Grade : GRADE =
struct

type grade = int
val zero = 0
val op+ = Int.+
val comparisons = [ Int.compare ]
val toString = Int.toString
val fromString = Int.fromString

end


fun make_set( Yss : int list list ) =
  fast_make_set( fn( Xs,Ys ) => list_less( op<, Xs, Ys ), Yss )
```

```
fun is_perm(Xs:int list,Ys) = Xs = sort (op<) Ys

fun output_eval_fun( _, Xs : input_type, Yss : output_type, _ )
  : output_eval_type * Grade.grade =
  let
    val Xs = from_list Xs
    val Yss = from_list' Yss
  in
    if exists( fn Ys => not(is_perm(Xs,Ys)), Yss ) then
      (wrong, 0)
    else
      case length(make_set Yss) of N =>
        if N = length Yss then
          (correct, ~N)
        else
          (wrong, 0)
  end


(* Uses 1.35s as upper limit for an scm class and 2s for a delta class.  *)

val N = 2
val Theta = 0.25
val Max_delta_class_card = N
val Delta_class_synt_compl_ratio =
  if N = 0 then 1.0 else real_pow( 2.0, 1.0/real(N) )
val Max_output_class_card = N
val Max_scm_class_card = N
val Scm_class_synt_compl_ratio =
  if N = 0 then 1.0 else real_pow( 1.35, 1.0/real(N) )
```