Combining Analytical and Evolutionary Inductive Programming*

Neil Crossley and **Emanuel Kitzelmann** and **Martin Hofmann** and **Ute Schmid**
Faculty Information Systems and Applied Computer Science, University of Bamberg, Germany
neil.crossley@stud.uni-bamberg.de, {emanuel.kitzelmann, martin.hofmann, ute.schmid}@uni-bamberg.de

Abstract

Analytical inductive programming and evolutionary inductive programming are two opposing strategies for learning recursive programs from incomplete specifications such as input/output examples. Analytical inductive programming is data-driven, namely, the minimal recursive generalization over the positive input/output examples is generated by recurrence detection. Evolutionary inductive programming, on the other hand, is based on searching through hypothesis space for a (recursive) program which performs sufficiently well on the given input/output examples with respect to some measure of fitness. While analytical approaches are fast and guarantee some characteristics of the induced program by construction (such as minimality and termination) the class of inducable programs is restricted to problems which can be specified by few positive examples. The scope of programs which can be generated by evolutionary approaches is, in principle, unrestricted, but generation times are typically high and there is no guarantee that such a program is found for which the fitness is optimal. We present a first study exploring possible benefits from combining analytical and evolutionary inductive programming. We use the analytical system IGOR2 to generate skeleton programs which are used as initial hypotheses for the evolutionary system ADATE. We can show that providing such constraints can reduce the induction time of ADATE.

Introduction

Automated programming research addresses the old dream of AI having computer systems which can automatically generate computer programs (Green *et al.* 1983; Biermann, Guiho, & Kodratoff 1984). Such systems would mimic the cognitive ability and expertise of human programmers. Deductive approaches to automated programming might reflect the use of general and specific knowledge about a programming language and the domain of the given problem which is available to experienced programmers. But neither proof-based approaches (Manna & Waldinger 1975; 1992) nor transformational approaches (Burstall & Darlington 1977) seem to be plausible cognitive strategies. Furthermore, as programming assistants such systems can only be used by highly trained experts, since programs must be completely and correctly specified in some formal language. Inductive approaches, on the other hand, might reflect strategies used by human programmers with limited experience in recursive programming. Common to all approaches to inductive programming is that recursive programs are constructed from *incomplete* specifications, typically samples of the desired input/output behavior and possibly additional constraints such as length or time efficiency. Such kind of information can be much more easily provided by programmers without special training and therefore, inductive programming approaches are good candidates for the development of programming assistants. There are two distinct approaches to inductive programming: analytical and evolutionary inductive programming.

Analytical inductive programming is data-driven and often relies on specifications which consist only of a small set of positive input/output examples. A recursive program is learned by detecting recurrence relations in the input/output examples and generalization over these regularities (Summers 1977; Kitzelmann & Schmid 2006). Typically, analytical approaches are fast and they can guarantee certain characteristics for the constructed program such as minimality of the generalization with respect to the given examples and termination. However, the class of learnable programs is necessarily restricted to such problems which can be specified by small sets of input/output examples. The scope of learnable programs can be somewhat widened by allowing the use of background knowledge (Kitzelmann 2008). Analytical inductive programming mimics a strategy often used by human programmers with limited experience in coding recursive programs: Explicitely write down the behavior of the desired program for the first possible inputs, observe the regularities between succeeding examples which reflect how the problem of size n can be solved using the solution of the problem with size $n-1$ and use this information to construct the recursive solution (Kahney 1989; Kruse 1982; Pirolli & Anderson 1985).

Evolutionary inductive programming is based on search through the hypothesis space of possible pro-

*Research was supported by the German Research Community (DFG), grant SCHM 1239/6-1.

grams given some (syntactically restricted) programming language. A hypothesis is returned as a solution if it performs sufficiently well on the input/output examples with respect to some measure of fitness, typically involving code length and time efficiency. The scope of programs learnable with an evolutionary approach is, in principle, unrestricted. But, generation times are typically high and there is no guarantee that the returned program is the optimal solution with respect to the fitness function. Evolutionary inductive programming follows a generate-and-test strategy which – to some extend – might be used by inexperienced programmers when they do have a clear idea about the desired program behavior but no clear idea about the algorithm. A cognitively more plausible search strategy is hill climbing, that is, searching for a solution by stepwise transforming the current solution such that it becomes more similar to the desired goal by covering more of the positive input/output examples and having a more desirable fitness. This idea is also incorporated in the means-end strategy (Newell & Simon 1961) and was shown as a strategy often exhibited in human problem solving (Greeno 1974). That is, to make evolutionary programming a more plausible strategy and at the same time to make it more efficient, it would be helpful to provide a program skeleton as initial seed which is afterwards stepwise refined with respect to coverage of examples and fitness.

Therefore, we propose to use analytical inductive programming to generate initial seeds for evolutionary programming. The combination of both approaches should be such that if a solution can be generated by analytical means alone, this fast and reliable approach should be used exclusively. If the problem is out of scope for analytical programming, at least a partial solution could be provided which then can be used as input for program evolution. In the following, we first describe the evolutionary programming system ADATE and the analytical programming system IGOR2. Afterwards we will introduce different strategies for the analytical generation of program seeds with IGOR2 and their incorporation in ADATE. We will report results of our first experiments and give a short conclusion.

Evolutionary Programming with ADATE

ADATE (Olsson 1995; Vattekar 2006) was initially proposed in the nineties and has been continually extended ever since. To our knowledge, it is the most powerful approach to inductive programming which is currently available. ADATE constructs programs in a subset of the functional language ML, called ADATE-ML. The problem specification presented to ADATE consists of: a set of data types and a set of primitive functions; a set of sample inputs; an evaluation function; an initial declaration of the goal function f. Sample inputs typically are input/output pairs. It is enough to give only positive examples, but it is additionally possible to provide negative examples. There are a number of predefined evaluation functions, each using different measures for syntactic complexity and time efficiency of the goal program. These are completed by a callback evaluation function given in the problem specification which evaluates the return value of a inferred function for a given input example. In general, the search heuristic is to prefer smaller and faster functions. As typical for evolutionary approaches, there are sets of individuals which are developed over generations such that fitter individuals have more chances to reproduce. If no additional knowledge is provided, in contrast to usual approaches, ADATE starts with a single individual – the empty function f.

The function declarations of all constructed program candidates use the declaration of f, differing only in the program body. To construct program bodies, only the programming constructs available in ADATE-ML can be used together with additionally data types and primitive functions provided in the problem specification.

The search operators are transformations used in reproduction to generate new individuals. These transformations include: replacements of expressions in the program body, abstraction of expressions by introducing a call to a newly introduced function, distributing a function call currently outside a case expression over all cases, and altering the number and type of function arguments by various embedding techniques. From these ADATE constructs compound transformations, consisting of multiple atomic transformations, depending of the current stage of the search. Through management of an upper bound for the number of compound transformations used to generate individuals, ADATE can employ iterative deepening in its exploration of the problem space. In the current version of ADATE crossover is realized by applying a compound transformation from one individual to another (Vattekar 2006).

In our experiments we used ADATE with the same set of only positive input/output examples which can be presented to the analytical system IGOR2. No data types and function primitives beyond ADATE-ML were used. Performance was evaluated with the predefined measures. Correctness was evaluated by a user-defined all-or-nothing function. The to be learned program f was either presented with an empty body or with one of four variants of initial seeds constructed by analytical induction. A simple example is given in figure 1.

With only the minimum necessary background knowledge, such as necessary data types and atomic functions, ADATE is able to find reasonable solutions given enough time. Additional background knowledge can reduce the required transformations to infer correct solutions, which can also reduce search time. However, additional background knowledge exacts deeper insights into the problem on behalf of the problem specifier. From a programming assistance system perspective, it would be not convenient to expect that a human programmer could provide exactly such knowledge which would increase the speed of generating a solution. This would presuppose (a) that the user understands the in-

Examples (represented in MAUDE notation):

```
sorts elem list.
ops a b c d: -> elem list [ctor].
op nil: -> list [ctor].
op _,_ : elem list -> list [ctor].

eq swap(nil) = nil.
eq swap((a,nil)) = (a,nil).
eq swap((a, b, nil)) = (b, a, nil).
swap((a, b, c, nil)) = (c, b, a, nil).
swap((a, b, c, d, nil)) = (d, b, c, a, nil).
```

Program induced by IGOR2:

```
1. swap( nil )      = nil.
2. swap( (X, XS) )  = (last((X,XS)), sub((X,XS))).
3. last( (X, nil) ) = X.
4. last( (X, XS) )  = last(XS).
5. sub ( (X, nil) ) = nil.
6. sub ( (X,Y,XS) ) = swap((X , sub((Y,XS)))).
```

Best program induced by ADATE with empty seed:

```
fun f Xs =
 case Xs of
   nill => Xs
 | cons( V144C, V144D) =>
 case V144D of
   nill => Xs
 | cons( V63EC5, V63EC6 ) =>
 case f( V63EC6 ) of
   nill => cons( V63EC5, cons( V144C, V63EC6 ) )
 | cons( V66B8B, V66B8C ) =>
     cons( V66B8B, cons( V63EC5, f( cons( V144C, V66B8C ) ) ) )
```

Figure 1: `Swap` specified for ADATE and IGOR2 and resulting programs

ner workings of ADATE and (b) has a deep insight in the programming problem at hand. From a cognitive perspective, such additional knowledge to guide ADATE's search might be gained by a closer inspection of the structure of the input/output examples, thereby providing ADATE with a helpful initial hypothesis.

Analytical Inductive Programming with IGOR2

IGOR2 (Kitzelmann 2008) – to our knowledge – is currently the most powerful system for analytical inductive programming. Its scope of inducable programs and the time efficiency of the induction algorithm compares very well with classical approaches to inductive logic programming and other approaches to inductive programming (Hofmann, Kitzelmann, & Schmid 2008). IGOR2 continues the tradition of previous work in learning LISP functions from examples (Summers 1977) as the successor to IGOR1 (Kitzelmann & Schmid 2006).

The system is realized in the constructor term rewriting system MAUDE. Therefore, all constructors specified for the data types used in the given examples are available for program construction. IGOR2 specifications consist of: a small set of positive input/output examples, presented as equations, which have to be the first examples with respect to the underlying data type and a specification of the input data type. Furthermore, background knowledge for additional functions can (but must not) be provided.

IGOR2 can induce several dependent target functions (i.e., mutual recursion) in one run. Auxiliary functions are invented if needed. In general, a set of rules is constructed by generalization of the input data by introducing patterns and predicates to partition the given examples and synthesis of expressions computing the specified outputs. Partitioning and searching for expressions is done systematically and completely which is tractable even for relatively complex examples because construction of hypotheses is data-driven. An example of a problem specification and a solution produced by IGOR2 is given in figure 1.

Considering hypotheses as equations and applying equational logic, the analytical method assures that only hypotheses entailing the provided example equations are generated. However, the intermediate hypotheses may be unfinished in that the rules contain unbound variables in the rhs, i.e., do not represent functions. The search stops, if one of the currently best hypotheses is finished, i.e., all variables in the rhss are bound.

IGOR2's built-in inductive bias is to prefer fewer case distinctions, most specific patterns and fewer recursive calls. Thus, the initial hypothesis is a single rule per target function which is the least general generalization of the example equations. If a rule contains unbound variables, successor hypotheses are computed using the following operations: (i) Partitioning of the inputs by replacing one pattern by a set of disjoint more specific patterns or by introducing a predicate to the righthand side of the rule; (ii) replacing the righthand side of a rule by a (recursive) call to a defined function (including the target function) where finding the argument of the function call is treated as a new induction problem, that is, an auxiliary function is invented; (iii) replacing subterms in the righthand side of a rule which contain unbound variables by a call to new subprograms.

Refining a Pattern. Computing a set of more specific patterns, case (i), in order to introduce a case distinction, is done as follows: A position in the pattern p with a variable resulting from generalising the corresponding subterms in the subsumed example inputs is identified. This implies that at least two of the subsumed inputs have different constructor symbols at this position. Now all subsumed inputs are partitioned such that all of them with the same constructor at this position belong to the same subset. Together with the corresponding example outputs this yields a partition of the example equations whose inputs are subsumed by p. Now for each subset a new initial hypothesis is computed, leading to one set of successor rules. Since more than one position may be selected, different partitions may be induced, leading to a set of successor rule-sets.

For example, let

$$\begin{array}{lcl} reverse([]) & = & [] \\ reverse([X]) & = & [X] \\ reverse([X,Y]) & = & [Y,X] \end{array}$$

be some examples for the *reverse*-function. The pattern of the initial rule is simply a variable Q, since the example input terms have no common root symbol. Hence, the unique position at which the pattern contains a variable and the example inputs different constructors is the root position. The first example input consists of only the constant [] at the root position. All remaining example inputs have the list constructor *cons* as root. Put differently, two subsets are induced by the root position, one containing the first example, the other containing the two remaining examples. The least general generalizations of the example inputs of these two subsets are [] and $[Q|Qs]$ resp. which are the (more specific) patterns of the two successor rules.

Introducing (Recursive) Function Calls and Auxiliary Functions. In cases (ii) and (iii) help functions are invented. This includes the generation of I/O-examples from which they are induced. For case (ii) this is done as follows: Function calls are introduced by matching the currently considered outputs, i.e., those outputs whose inputs match the pattern of the currently considered rule, with the outputs of any defined function. If all current outputs match, then the rhs of the current unfinished rule can be set to a call of the matched defined function. The argument of the call must map the currently considered inputs to the inputs of the matched defined function. For case (iii), the example inputs of the new defined function also equal the currently considered inputs. The outputs are the corresponding subterms of the currently considered outputs.

For an example of case (iii) consider the last two *reverse* examples as they have been put into one subset in the previous section. The initial rule for these two examples is:

$$reverse([Q|Qs]) = [Q2|Qs2] \qquad (1)$$

This rule is unfinished due two the two unbound variables in the rhs. Now the two unfinished subterms (consisting of exactly the two variables) are taken as new subproblems. This leads to two new examples sets for two new help functions $sub1$ and $sub2$:

$$\begin{array}{lcllcl} sub1([X]) & = & X & sub2([X]) & = & [] \\ sub1([X,Y]) & = & Y & sub2([X,Y]) & = & [X] \end{array}$$

The successor rule-set for the unfinished rule contains three rules determined as follows: The original unfinished rule (1) is replaced by the finished rule:

$$reverse([Q|Qs]) = [sub1([Q|Qs] \mid sub2[Q|Qs]]$$

And from both new example sets an initial rule is derived.

Finally, as an example for case (ii), consider the example equations for the help function $sub2$ and the generated unfinished initial rule:

$$sub2([Q|Qs] = Qs2 \qquad (2)$$

The example outputs, $[], [X]$ of $sub2$ match the first two example outputs of the *reverse*-function. That is, the unfinished rhs $Qs2$ can be replaced by a (recursive) call to the *reverse*-function. The argument of the call must map the inputs $[X], [X,Y]$ of $sub2$ to the corresponding inputs $[], [X]$ of *reverse*, i.e., a new help function, $sub3$ is needed. This leads to the new example set:

$$\begin{array}{lcl} sub3([X]) & = & [] \\ sub3([X,Y] & = & [X] \end{array}$$

The successor rule-set for the unfinished rule contains two rules determined as follows: The original unfinished rule (2) is replaced by the finished rule:

$$sub2([Q|Qs] = reverse(sub3([Q|Qs]))$$

Additionally it contains the initial rule for $sub3$.

Analytically Generated Seeds for Program Evolution

As proposed above, we want to investigate whether using IGOR2 as a preprocessor for ADATE can speed-up ADATE's search for a useful program. Furthermore, it should be the case that the induced program should be as least as efficient as a solution found unassisted by ADATE with respect to ADATE's evaluation function. Obviously, coupling of IGOR2 with ADATE becomes only necessary in such cases where IGOR2 fails to generate a completed program. This occurs if IGOR2 was presented with a too small set of examples or if analytically processing the given set of examples is not feasible within the given resources of memory and time. In these cases IGOR2 terminates with an incomplete program which still contains unbound variables in the body of rules, namely, with missing recursive calls or auxiliary functions.

To have full control over our initial experiments, we only considered problems which IGOR2 can solve fully automatically. We artificially created partial solutions by replacing function calls by unbound variables. We investigated the following strategies for providing ADATE with an initial seed:

For a given ADATE-ML program of the form

```
fun f ( ... ) : myType = raise D1
fun main ( ... ) : myType = f ( ... )
```

- the function f is **redefined** using the partial solution of IGOR2,
- or the problem space becomes **restricted** from the top-level by introducing the partial solution in the function *main*.
- Any IGOR2 induced auxiliary **functions** can also be included: as an atomic, predefined function to be called by f or as an inner function of f also subject to transformations.

Experiments

We presented examples of the following problems to IGOR2:

switch(X) = Y iff the list Y can be obtained from the list X by swapping every element on an odd index in X with the element with the next incremental even index.

sort(X) = Y iff the list Y is a permutation of X with all elements sorted in increasing order.

swap (X) = Y iff the list Y is identical to the list X, except that the first and last element are swapped in around in Y.

lasts(X) = Y iff X is a list of lists and Y is a list containing the last element of each list in X in the order those lists appear in X.

shiftR(X) = Y iff the list Y is identical to the list X, except that the last element in X is on the first position in Y and all other elements are shifted one position to the right.

shiftL(X) = Y iff the list Y is identical to the list X, except that the first element in X is on the last position in Y and all other elements are shifted one position to the left.

insert(X, Y) = Z iff X is a list of elements sorted in an ascending order and Z is a list of elements X + Y sorted in an ascending order.

To generate an initial seed for ADATE, typically the righthand side of a recursive rule was replaced by an unbound variable. For example, the solution for *switch* provided by IGOR2 was

```
switch ( [] )  = []
switch ( [X] ) = [X]
switch ( [X,Y|XS] ) = [Y, X, switch(XS)]
```

and the third rule was replaced by

```
switch ( [X,Y|XS] ) = Z.
```

If IGOR2 induced solutions with auxiliary functions, either the function calls on the righthand side of the rules were made known to ADATE (see section *Analytically Generated Seeds for Program Evolution*) or this information was obscured by again replacing the complete righthand side by a variable.

For example, for *swap*, IGOR2 inferred one atomic function *last* and inferred that the solution consists of two functions that recursively call each other as shown in figure 1. ADATE was presented with the rule 1, 2, 5 and 6 from figure 1 where the righthand side of rule 6 was replaced with an unbound variable.

The results were ascertained by analysing the log files produced to document an ADATE run. To effectively compare the specifications we evaluated each according to the time taken to generate the most correct functions. Because ADATE infers many incorrect programs in the search process, we restricted our focus to those programs that:

- were tested by ADATE against the complete set of given training examples,
- terminated for each training example, and
- generated a correct output for each training example.

This allowed us to achieve a meaningful overview of the performance of the specifications. While an analysis of the inferred programs with poorer performance provides insights into the learning process, it is outside of our scope. Nonetheless, ADATE generates a very complete overview of inferred programs in the log files. For the analysis of the ADATE runs we needed only the following information:

- the elapsed **time** since the start of the search until the creation of the program,
- the breakdown of the results the function produced for the examples, which in our case is the number of results evaluated as correct, incorrect or timed-out. Due to our accepted evaluation restrictions, we filtered out all inferred functions which did not attain 100% correct results with the test examples.
- an ADATE time evaluation of the inferred function. This is the total **execution** time taken by the function for all the test examples as defined by ADATEs built in time complexity measure. This is compareable to a count of all execution operations in runtime, including method and constructor calls and returns.

Because all the specifications designed to solve the same problem included exactly the same examples, we could now compare the respective runs with each other. Table 1 is a summary comparing the most efficient solutions of the unassisted specifications with those of the best specification for the same problem. Included is the problem name, the specification type (either unassisted or the type of assistance), the creation time of the solution, the execution time necessary for the same set of examples.[1]

Table 1: Results for the best strategy (Time in seconds, Execution see text)

Problem	Type	Time	Execution
switch	Unassisted	4.34	302
	Restricted	0.47	344
	Redefined	3.96	302
sort	Unassisted	457.99	2487
	Restricted	225.13	2849
swap	Unassisted	292.05	1076
	Restricted + functions	41.43	685
lasts	Unassisted	260.34	987
	Restricted	6.25	1116
shiftR	Unassisted	8.85	239
	Redefined + functions	1.79	239
shiftL	Unassisted	4.17	221
	Restricted	0.61	281
insert	Unassisted	7.81	176
	Restricted	18.37	240

[1] To run ADATE only the 1995 ML compiler can be used. The technical details are given in a report

IGOR2 produced solutions with auxiliary functions for the problems *sort*, *swap* and*shiftR*. In the case of *sort* the best result for ADATE was gained by giving no information about the auxiliary functions.

Attention should be drawn to the uniformly quicker inference times achieved by the assisted ADATE specifications with the noteable exception of *insert*. Two assisted specifications – *swap* and *shiftR* resulted in better results that were also inferred sooner, whereas the remaining assisted specifications produced results between 14% and 27% less efficient than their unassisted counterparts. All in all, one could summarise, that this relatively small comparative inefficiency is more than compensated by the drastically reduced search time, just over 41 times quicker in the case of *lasts*. That is, our initial experiments are promising and support the idea that search in a generate-and-test approach can be guided by additional knowledge which can be analytically obtained from the examples.

Conclusions

In inductive programming, generate-and-test approaches and analytical, data-driven methods are diametrically opposed. The first class of approaches can in principal generate each possible program given enough time. The second class of approaches has a limited scope of inducable programs but achieves fast induction times and guarantees certain characteristics of the induced programs such as minimality and termination. We proposed to marry these two strategies hoping to combine their respective strengths and get rid of their specific weaknesses. First results are promising since we could show that providing analytically generated program skeletons mostly guides search in such a way that performance times significantly improve.

Since different strategies showed to be most promising for different problems and since for one problem (*insert*) providing an initial solution did result in longer search time, in a next step we hope to identify problem characteristics which allow to determine which strategy of knowledge incorporation into ADATE will be the most successful. Furthermore, we hope either to find a further strategy of knowledge incorporation which will result in speed-up for *insert*, or – in case of failure – come up with additional criteria to determine when to refrain from providing constraining knowledge to ADATE. Our research will hopefully result in a programming assistant which, given a set of examples, can determine whether to use IGOR2 or ADATE stand alone or in combination.

References

Biermann, A. W.; Guiho, G.; and Kodratoff, Y., eds. 1984. *Automatic Program Construction Techniques*. New York: Macmillan.

Burstall, R., and Darlington, J. 1977. A transformation system for developing recursive programs. *JACM* 24(1):44–67.

Green, C.; Luckham, D.; Balzer, R.; Cheatham, T.; and Rich, C. 1983. Report on a knowledge-based software assistant. Technical Report KES.U.83.2, Kestrel Institute, Palo Alto, CA.

Greeno, J. 1974. Hobbits and orcs: Acquisition of a sequential concept. *Cognitive Psychology* 6:270–292.

Hofmann, M.; Kitzelmann, E.; and Schmid, U. 2008. Analysis and evaluation of inductive programming systems in a higher-order framework. In Dengel, A.; Berns, K.; Breuel, T. M.; Bomarius, F.; and Roth-Berghofer, T. R., eds., *KI 2008: Advances in Artificial Intelligence (31th Annual German Conference on AI (KI 2008) Kaiserslauten September 2008)*, number 5243 in LNAI, 78–86. Berlin: Springer.

Kahney, H. 1989. What do novice programmers know about recursion? In Soloway, E., and Spohrer, J. C., eds., *Studying the Novice Programmer*. Lawrence Erlbaum. 209–228.

Kitzelmann, E., and Schmid, U. 2006. Inductive synthesis of functional programs: An explanation based generalization approach. *Journal of Machine Learning Research* 7(Feb):429–454.

Kitzelmann, E. 2008. Analytical inductive functional programming. In Hanus, M., ed., *Pre-Proceedings of the 18th International Symposium on Logic-Based Program Synthesis and Transformation (LOPSTR 2008, Valencia, Spain)*, 166–180.

Kruse, R. 1982. On teaching recursion. *ACM SIGCCE-Bulletin* 14:92–96.

Manna, Z., and Waldinger, R. 1975. Knowledge and reasoning in program synthesis. *Artificial Intelligence* 6:175–208.

Manna, Z., and Waldinger, R. 1992. Fundamentals of deductive program synthesis. *IEEE Transactions on Software Engineering* 18(8):674–704.

Newell, A., and Simon, H. 1961. GPS, A program that simulates human thought. In Billing, H., ed., *Lernende Automaten*. München: Oldenbourg. 109–124.

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

Pirolli, P., and Anderson, J. 1985. The role of learning from examples in the acquisition of recursive programming skills. *Canadian Journal of Psychology* 39:240–272.

Summers, P. D. 1977. A methodology for LISP program construction from examples. *Journal ACM* 24(1):162–175.

Vattekar, G. 2006. Adate User Manual. Technical report, Ostfold University College.

by Neil Crossley available at http://www.cogsys.wiai.uni-bamberg.de/teaching/ss07/p_cogsys/adate-report.pdf.