Inductive Synthesis of Recursive Functional Programs
A Comparison of Three Systems

Martin Hofmann, Andreas Hirschberger, Emanuel Kitzelmann, and Ute Schmid
University of Bamberg, Faculty of Information Systems and Applied Computer Science
{martin.hofmann, andreas.hirschberger}@stud.uni-bamberg.de,
{emanuel.kitzelmann, ute.schmid}@wiai.uni-bamberg.de

Abstract. We compare three systems for the task of synthesising functional recursive programs, namely ADATE, an approach through evolutionary computation, the classification learner ATRE, capable of simultaneously learning mutually dependent, recursive multi-class target predicates, and the inductive/abductive logic program synthesiser DIALOGS-II. An overview over the functionality of all three systems is given and their capabilities are systematically evaluated under equal premises with a variety of recursive problems. We propose to combine ADATE’s expressive power with DIALOG-II’s search heuristic. The accessory adoption of ATRE’s k-beam search strategy to learn mutually dependent recursive target functions should only be adopted, if ADATE’s search time could be reduced significantly.

1 Introduction

One of the most challenging subfields, and a still little researched niche of machine learning, is the inductive synthesis of recursive programs from incomplete specifications, such as examples for the desired input/output behavior [1–4]. The special appeal of an inductive approach to automated program construction is that the user only provides some examples of the desired program behaviour, such as \([A, B, C] \rightarrow [C, B, A]\), as input to the synthesis system and a general program (here for reversing a list) is created. Potential applications for automatic program induction are to enable end-users to create their own simple programs, to assist professional programmers or even to automatically invent new and efficient algorithms.

Inductive program synthesis shares the principal characteristics of all approaches to inductive learning, that is, the learning system can be characterised by an inductive bias due to some restriction of the hypothesis language or to the strategy which determines the sequence in which hypotheses are constructed [5]. In contrast to most learning approaches, especially in contrast to classification learning, specialised approaches to inductive program synthesis typically expect
a small collection of only positive examples as input for the learner and the con-
structed program is accepted only if all examples given are correctly transformed
into the specified output [2].

Since no broadly accepted fundamentals and approaches prevail in the field
of inductive program synthesis, this paper systematically evaluates three sys-
tems to inductively synthesise functional recursive programs,\(^1\) which are based
on three fundamentally different induction methods. We omit prominent and well
researched ILP systems such as FOIL/FFOIL [6, 7], PROGOL [8, 9] and GOLEM
[10] for several reasons: These systems are already extensively researched and
evaluated [2], they were not designed specifically for learning recursion, and fi-
nally, they focus on the learning of classifications and not on general programs
or algorithms, which are in our focus of interest.

In the next Section 2, we introduce the information required to generate a
solution for a problem, the specification, the capabilities of each compared system
and each system’s underlying strategy for program generation is explained. In
Section 3 the systems are evaluated for their performance on a set of selected
problems. Expected and observed results are discussed. Section 4 contains the
conclusions drawn from the comparison and gives recommendations for further
work.

2 Three Prototypical Synthesis Systems

Existing inductive program synthesis systems use either a search-based or an
analytical approach. While the most promising are allocated in the subfield of
program synthesis, also concept learners with extended codomain and especially
ILP-based systems exhibit success worthy to mention.

Although our focus is on the synthesis of recursive functions, our nomen-
clature is, for the sake of convenience, based on the ILP glossary. Therefore,
we understand background knowledge as any additional, user provided, problem-
specific information, as e.g. predefined predicates or helper-functions and predi-
cate invention as the automated generation of new predicates or subfunctions to
solve the problem, neither mentioned in the background knowledge, nor in the
training examples.

The introduced systems are the synthesis program ADATE, following a search
strategy based on evolutionary computing, the search-based concept learner
ATRE, capable of simultaneously learning recursive concepts, and program syn-
thesiser DIALOGS-II, using an analytical approach based on inductive logic pro-
gramming. For each system first the problem specification, the learning and
synthesis strategy and finally the limits and capabilities are discussed.

\(^1\) By “functional” we refer to the mapping of each input to a unique output by a
synthesised program and not to a specific programming paradigm.
2.1 ADATE

This section introduces ADATE [11, 12] (Automatic Design of Algorithms Through Evolution), a system for automatic programming utilising evolutionary computing for program generation.

Problem Specification ADATE’s specification is written in a sub-language of ML, which is then embedded in ADATE’s own ML source code. First of all, those functions and datatypes with their constructors which should be used within the inferred function, have to be provided as background knowledge. These can either be user defined or taken from the standard library, such as comparators or arithmetical operations. Additionally, a prototype \( f \) of the target function, which defines number and types of the input parameters and the type of the result, has to be provided. It is used as the first individual, which is then expanded using evolutionary operators. Usually \( f \) is provided as the most simple ADATE-ML function, which just raises an exception. Instead of I/O examples only inputs are given, but with an output evaluation function to compare the desired output with the one computed by the evaluated individual to determine its correctness. ADATE also offers many other customisation parameters and the possibility to use whole code fragments during the synthesisation, but for these we refer to the manual [13].

Learning/Synthesis Strategy ADATE as an evolutionary computation system employs search techniques that are inspired by basic biological principles of evolution, like for instance mutation and crossover. The search for programs conducted is global and mostly systematic, that is, without randomisation. A series of heuristics is utilised to increase search efficiency. Therefore ADATE builds a collection of programs during the search called the kingdom in analogy with a kingdom in biological taxonomy. Programs are viewed as individuals. The kingdom consists initially of only one individual, which is provided as function \( f \) by the user in the problem specification. New individuals are generated by applying transformations to programs in the kingdom. If a generated individual performs better than programs already in the kingdom it is inserted into the kingdom possibly replacing programs that perform worse. To evaluate an individual ADATE uses three internal functions, which are based on the number of correct and wrong training examples, the number of memory and time limit overflows, and an user defined evaluation function. By continuing to produce and insert individuals in this manner, ADATE will find better and better programs as the search progresses.

Limits and Capabilities Being a system based on evolutionary computation, ADATE is able to generate every syntactically correct program using the provided functions and datatypes. Thereby ADATE is theoretically able to solve every problem a valid ADATE-ML program can be found for. However, whether the problem can be solved in reasonable time depends strongly on the user
provided functions and datatypes. It should be clear, that the more additional functions are provided to the system, the more complex problems are solvable. Since ADATE requires little knowledge about the way those problems can be solved, the user does not have to think about the solution of a problem in detail, he simply provides every function that might be needed to solve it.

Nevertheless the capability of ADATE is capped by the required search time, due to the fact that the complexity of the problem together with the amount and quality of the provided background knowledge are highly correlated with the search time. To speed up the search for a given problem the search space has to be restricted by extending the problem by e.g. a more compact representation or more specific helper functions.

Nonetheless, ADATE runs infinitely until the user terminates it, as there is no way to estimate the time needed to find a good solution. This is necessary because ADATE can always find other, maybe “better” solutions during continuing search, because new individuals are created by adding complexity to previously generated ones. So “simple” solutions are always found first, but a more efficient even though more complex solution might be hard to generate by repeatedly adding small pieces of code. While ADATE tries to improve the correct but simple solution other individuals are neglected.

2.2 ATRE

Although we know that the concept learner ATRE [14–16] was not specifically designed for program synthesis, we consider it here, because (a) it is capable of handling learning tasks with multiple target predicates involved which might be mutually dependent on each other and (b) we also want to know if it is sufficient to enrich concept learners with recursive abilities to succeed in program synthesis.

**Problem Specification** Since ATRE is a multi-class concept learner, it adopted a form of functional style defining the target concept. Each output literal of the target predicate is either a value or a closed interval over a continuous range of values, defining the concept class this instance belongs to. The classes are non-overlapping and mutual exclusive, so a distinct training example belongs exactly to one concept class, such that positive examples of one class are negative examples of the others. The background knowledge represents relevant information about the domain in terms of definite clauses, but also additional information about the context in which the training examples occur as classified literals. In a saturation phase the information that is implicit in the background knowledge is made explicit via logic entailment.

**Learning Strategy** ATRE belongs to the family of sequential covering algorithms [5]. The core of such algorithms is the LEARN-ONE-RULE strategy, which means, that they learn one clause at one time, covering some of the positive training examples. The covered examples are then removed from the training set and
the process is repeated until all positive examples are covered by some rule. In the same way Atre incrementally builds up a theory, starting with the empty theory $T_0 = \emptyset$ and adding at each step a new clause $C$ learnt in one LEARN-ONE-RULE run, thus obtaining a sequence of theories $T_0, T_1, \ldots, T_i, T_{i+1}, \ldots, T_n = T$, where for each $i T_{i+1} = T_i \cup \{C\}$ holds and all theories are complete and consistent with respect to the training set.

In fact, ATRE performs a general-to-specific parallel beam search in the space of definite clauses ordered by generalised implication [14]. The search space can be described as a forest of several search trees. A search tree is rooted with the most general instance of the seed, i.e. the a target predicate with only variables as arguments, and a directed arc from node $C$ to node $C'$ exists, if $C'$ can be obtained by a single refinement step from clause $C$. In one refinement step a descendant is created by adding one literal to the body of the parent clause, where only predicates are used specified in training examples and background knowledge. The level$_0$ of every search tree represents a clause with an empty body and the seed-predicate in the head, whereas level$_i+1$ contains all clauses that can be obtained by one refinement step from a clause in level$_i$. Note that also the target predicates can be added to a clause during a refinement step, and thus, recursive target concepts are possible to learn.

In each LEARN-ONE-RULE run all search trees are traversed concurrently by as many tasks as seeds were selected, top-down from general-to-specific. Each task is lead by some heuristic and can adopt its own search strategy and decide which clauses are worth testing, but synchronises with the other tasks at the same level. The clauses are tested against the training examples and the evidence derived in the saturation phase. A supervisor task compares the found clauses of each level and decides whether to continue search or select the “best” clauses according to some user’s preference criterion.

However, in a multiple predicate learning setting the problem may arise, that when adding new consistent clauses to a consistent theory their conjunction may become inconsistent [14]. A recovery strategy fixes this problem by syntactic changes in the theory by predicate renaming and adding an auxiliary clauses. It is noteworthy, that although now “new” predicates may occur in the learnt theory, ATRE is not capable of predicate invention in the common sense, or as it was defined in [2], since this is only due to syntactic reformulation.

Limits and Capabilities Atre was developed as a concept learning system capable to learn recursive theories in the domain of document recognition. Nevertheless we want to tackle some the issues concerning the field of program synthesis and describing ATRE’s capabilities in this domain, even though we know that this is not what the system was meant to do.

Although ATRE can concurrently learn several recursive theories, which by themselves might be mutually dependent on each other, while exploring the search space in the LEARN-ONE-RULE phase only user defined predicates are added and therefore no predicate invention [2] is done. Hence, no recursive theories can be learnt that require recursive auxiliary predicates as a recursive com-
pose operator, e.g. reverse/2, addlast/2, assuming only the usual head/tail decomposition is given as background knowledge. ATRE cannot handle such problems as it does not invent predicates during the LEARN-ONE-RULE phase.

As in usual ILP settings, evidence is presented to ATRE as positive and negative training examples, but the system supports the classical negation rather than negation by failure as under a closed world assumption. Therefore, there exists no possibility to describe negative examples in another way than including each in the set of evidence as a negative example. Omitting negative evidence may lead to rules, describing accidental irregularities in the positive evidence, which may be preferred to the “correct” rules by the systems search bias as no negative evidence disproved them.

2.3 DIALOGS-II

DIALOGS-II (Dialogue-based Inductive and Abductive LOGic program Synthesiser)\cite{17, 18} is a schema-guided, interactive, inductive and abductive recursion synthesiser that takes the initiative and queries a (possibly computational naive) specifier for evidence in her/his conceptual language.

**Problem Specification** As mentioned before, DIALOGS-II belongs to the class of interactive synthesis programs as it collects all required evidence during a dialogue with the specifier.

First of all, DIALOGS-II prompts the user for a predicate consisting of at least one induction and one result parameter and one (if any) passive parameter, i.e. a parameter that remains unchanged in a recursive call, and their accordant types.

As a schema-guided synthesiser, DIALOGS-II provides the user with two schemas and corresponding strategies (though only one per schema), namely a descending-generalisation-schema \cite{19} which introduces an accumulator parameter and a divide-and-conquer-schema, to synthesise programs. Generally, in DIALOGS-II a schema represents a whole program family with a specific data flow and certain constraints concerning the computations performed by these programs. It is represented as an open program, also called template program, that is closed during the synthesisation process using the evidence collected from the specifier and satisfying its constraints \cite{17}. An open program is a set of clauses were some predicates are not defined by the set itself.

Finally the user has to define the decomposition operator, that divides the non-minimal case into smaller subcases to process them in recursive calls. DIALOGS-II supports a partition into $n$ heads and one tail, the partition of a list into two sublists according to a pivot element and the partition into two halves.

\footnote{It is noteworthy, that we were not able to synthesise a correct program using halves/3 or partition/4 during evaluation, what brings up the question whether DIALOGS-II can handle other than linear recursion at all.}

In the case of the type nat represented as a Peano number, the decomposition operator specifies more an in-/decrement with the invariable $n$ than a proper decomposition into head and tail.
**Synthesis Strategy** In the specification dialogue, all structural decisions about parameter roles and their types, the template program and the decomposition operator are made so far. This again results in an open program, where the sole open predicates are the open relation for the *solve* part $p$ of the non-recursive clause and the open relation for the *combine* part $q$ of the recursive clause (i.e. the recursive call).

From now on, DIALOGS-II starts querying the specifier in order to abduct evidence for the open relations $p$ and $q$. DIALOGS-II does not use I/O examples, but rather asks under what conditions the predicate $r(I, R, P)$ with probably some additional assumptions holds. It starts with the smallest, most general instance of the induction parameter and increases its complexity from query to query, e.g. it first queries for the empty list, then for a list with one element (i.e. a list containing one variable), and so on. $I$ denotes some instance of the induction parameter, $R$ the result parameter and $P$ the passive parameter, if present.

Since the induction parameter in the queried predicate is always in its most general form of a given size, e.g. a list of three variables for a list with three elements, the user can use system primitives as equality, ordering relations, simple arithmetic operators and self-introduced predicates to exactly specify the conditions under which the target predicate holds in clause form, i.e. as disjunctions over conjunctions. This Nevertheless, this has to be done exhaustively with respect to the induction parameter $I$ to cover all its possible parameters, because it is in its most general form of a given size.

Before each new query, DIALOGS-II applies the decomposition operator to the new instance of the induction parameter and tries to find evidence from the previous queries how to process the tail (the complex part of the decomposition) in the recursive call. If the evidence is ambiguous, i.e. disjunctions were used during past specifications, DIALOGS-II adds additional assumptions to a query. An example run for *delOdds/2*, which deletes all odd elements in an integer list, with normal head/tail decomposition would look like as follows:

What conditions on <R0> must hold such that delOdds([],R0) holds?
  | R0=[]
What conditions on <A1,R1> must hold such that delOdds([A1],R1) holds?
  | odd(A1), R1=[]; not(odd(A1)), R1=[A1]
What conditions of <B1,B2,R2> must hold such that delOdds([B1,B2],R2) holds, assuming odd(B2)? ... 

Note here, that the predicate *odd/1* was only used by the system, after the user mentioned it in the specification.

The result of the abduction for each query are evidence pairs for the non-recursive relation $p$ and its accordant recursive relation $q$, stating under which conditions they hold for a certain instance of the induction parameter.

To close the open relations, a strategy based on the least general generalization under $\theta$-subsumption is used [20] to generalise over the evidence pairs and generate the recursive clause(s).
Nevertheless, this method assumes, that the open relation \( q \) of the recursive clause of the open program is finite and non-recursive, but this is not always the case [17]. Therefore there might be a recursive one and the system needs to do necessary predicate invention. This is done by calling the system recursively. However, it does not detect this need by itself and so the specifier has to decide whether it is appropriate to call DIALOGS-II again to close the open relation \( q \) when the predicates could not been closed the first time, or not.

**Limits and Capabilities** DIALOGS-II strength and weaknesses could be described appropriately under the notion of an inductive bias. The language bias is determined by its used schemas, which implicitly define the search space, combined with the decomposition operators which fix the access to the inductive parameter. Obviously, problems requiring different schemas or decomposition, e.g. multiple inductive parameter, are not solvable. Although the user supports the system during the specification process with a lot of background knowledge, information like system primitives etc. are only used when explicitly used by the specifier. The provided background knowledge occurs only in the non-recursive clauses, where all recursive clauses themselves only consist of the predicates already defined in the schemas and the decomposition parameters. Predicate invention is therefore only limited to find probably recursive solution for the recursive clause \( q \) of the open program and does not include possible predicate invention in the evidence clause.

DIALOGS-II’s search bias explains the system’s need for self-contained and noise-free evidence, since it traverses the search space, from the most general (the schema) program to the most specific, guided by the clauses for its open predicates induced from the evidence provided by the specifier. Self-contained evidence means, it must describe the behaviour of the target predicate completely, given a most general instance of the induction parameter of a specific size. Consider for example the program for `flatten/2`, where `flatten(I, O)` is true, iff \( I \) is a list probably containing other lists and \( O \) is a list containing the elements of \( I \) in the same order, but with no nested lists. DIALOGS-II fails with such a problem, because the description of conditions under which `flatten([A], O)` holds is already infinite as \( A \) could again be a list of any length. Additional, since the evidence is not presented to the system as specific positive or negative training examples, but moreover as a logical relation which is directly used in the following abduction process in unfolding and resolution operations. However, if the evidence wrong, the traversal is immediately misled, but if the DIALOGS-II “does not know were to go” it queries the specifier.

3 **Empirical Setup and Results**

The following section covers first some general issues to justify our setting and gives a description of the example problems and the provided background for each system and then evaluates them in each problem class.
3.1 Description of the Test Setting

We tried to set up the tests as fair as possible, nevertheless, due to the inhomogeneity of the system there still might be discrepancies. First, we describe the applied problem classes, followed by the programs we intended to synthesise and finally the background knowledge used for each system.

Problem Classes Although all three synthesis systems provide to some extend the same means to learn recursive programs, they still remain quite inhomogeneous concerning their underlying concepts. To properly evaluate them, we were forced to some common denominator for their problem space. The Venn diagram (Fig. 1) shall illustrate the extend of the capabilities of the three systems and explains the identified classes used in the evaluation process:

- **Single recursive call without predicate invention** (I.): solvable with a single recursive call in the body of the predicate definition; no predicate or variable invention is required.
- **Single recursive call with predicate invention** (II.): at least the invention of an auxiliary predicate is required.
- **Multiple recursive call** (III. + IV.): at least a second recursive call is necessary (either of another recursive predicate or of the target predicate itself)
- **Miscellaneous** (V. + III.): emphasises the individual strengths of a certain system.

![Venn Diagram of System Capabilities](image)

**Fig. 1.** Venn Diagram of System Capabilities

**Description of Problems** We especially concentrated on problems over lists, because with their simple structure they allowed us to tailor problems with a minimum of necessary background knowledge and also remained more or less unchanged for all systems, to make the whole evaluation more transparent and facilitate the comparison of results between the different systems. In the following we provide a short description of our examined problems. They contain typical, already researched problems [10], as well as problems that appeared interesting to us:

**Single Recursive Call without Predicate Invention**

- **evenpos(X,Y)** holds iff list Y contains all elements of list X at an even position in unchanged order.
- **insert(X,Y,Z)** holds iff X is a list with its elements in a not decreasing order, and Z is X with Y inserted on the right place.
- **inslast(X,Y,Z)** holds iff Z is the list X with Y inserted at the end.
- **last(X,Y)** holds iff Y is the last element of the list X.
- **length(X,Y)** holds iff Y is the length of the list X.
- **member(X,Y)** holds iff X is a list containing the element Y.

---

3 Classes III. and VI. were combined, since DIALOGS-II is not capable of multiple recursive calls and an ATRE specification for such a problem would result in an extensive enumeration in input/output pairs.
**switch**($X, Y$) holds iff list $Y$ can be obtained from list $X$ were all elements on an odd position changed place with their right neighbour.

**unpack**($X, Y$) holds iff $Y$ is a list of lists, each containing one element of $X$ in unchanged order.

**Single Recursive Call with Predicate Invention**

**i-sort**($X, Y$) holds iff the list $Y$ is a permutation of list $X$ with elements in a non decreasing order.

**multlast**($X, Y$) holds iff the list $Y$ contains nothing but the last element of list $X$ as many times as the number of elements in $X$.

**reverse**($X, Y$) holds iff list $Y$ could be derived from list $X$ by shifting the first element to the end.

**swap**($X, Y$) holds iff list $Y$ could be derived from list $X$ by swapping the first and the last element.

**Multiple Recursive Call with(out) Predicate Invention**

**lasts**($X, Y$) holds iff $X$ is a list of lists, and $Y$ contains the last elements of each list in $X$ in the correct order.

**flatten**($X, Y$) holds iff $Y$ is the flattened version of the list of lists $X$.

**Miscellaneous Problems**

**mergelists**($X, Y, Z$) holds iff the list $Z$ could be derived from the lists $X$ and $Y$ such that $Z = [x_1, y_1, x_2, y_2, \ldots]$ where each $x_n$ and $y_n$ is the $n^{th}$ of the list $X$ and $Y$, respectively.

**odd**($X$) / **even**($X$) holds iff $X$ is an odd, respectively even number, and each predicate is defined in terms of **zero**($X$) and the other.

**Background Knowledge** Although the systems are based on quite different concepts, background knowledge provided to the systems was kept as similar as possible for all of them.

**ADATE** As mentioned before, ADATE uses a simplified version of ML, in which lists are defined as recursive datatypes. Since most of the problems do not require the elements to be of some defined type the built-in datatype int is used, because built-in integer operations simplify the problem specification. Several problems require additional datatypes and functions. For the problems **lasts**/2 and **flatten**/2 the datatype list of list of int is needed. For **member**/2, **insert**/3 and **sort**/2 the comparator "$< "$ and the constructors "true" and "false" of the datatype bool are additionally needed. **Length**/2 requires "$0 "$, "$+ " and "$1 " to allow counting.

**ATRE** Since ATRE operates in the usual ILP setting, data structures such as lists are in general unknown to the system. Therefore, we represented all lists up to four elements and no repetition as atoms and introduced a new predicate **components**($L, H, T$), which is true iff $L$ is the atom representation of a list and $H$ is the atom representation of the head of $L$ and $T$ the atom representation of the tail of $L$. Every possible list up to four elements was either represented in terms of a grounded instance of the predicate **components**/3 or as a constant, **nil**($[ ]$) for the empty list. To learn the predicate **length**/2 we additionally introduced a grounded predicate for 0, **zero**(0), and every integer was represented
as a grounded instance of the predicate \( \text{succ}(X,Y) \), where \( \text{succ}(X,Y) \) holds iff \( X \) is the atom representation of a natural number which successor is the atom representation of the natural number \( Y \). For every input list up to length four, the correct input output pair was given to the system as correct evidence and the input with all permutations of the correct output as negative evidence.

**Dialogs-II** Unless not otherwise stated, for all problem synthesisations the divide-and-conquer schema was used, as well as the simple head-tail-decomposition operator. A descending generalisation schema was inappropriate for most problems. Although it would have been possible to synthesise a program with this schema, it allows only the extension of the accumulator variable via \text{cons} or \text{append}. The former requires a program where the output is somehow the reverse from its input (perfect for \text{reverse}/2) or the order does not matter, the latter would have made additional information as background knowledge available. Elsewise all specifications were done as required by the system.

### 3.2 Results of the Test Setting

Table 1 shows the results of the test runs with different systems on problems from classes described above. 4

**ADATE**

*Expected Outcome* Neither the need for predicate invention nor multiple recursive calls should be a major obstruction for ADATE. However, the search time is expected to increase with the complexity of the processed problems.

*Results* All of the provided problems where solved and the search time rose as the complexity of problems increased as expected. For all problems with single recursive call the first solution found was optimal, except for \text{swap}/2 and \text{sort}/2 the code was correct but inefficient. The results of the more complex problems still were acceptable.

For \text{swap}/2 a first solution was found after 232s which was an inefficient variation of \text{i-sort}/2, using more recursive calls than necessary. The second solution found after 315s performed better. Both solutions use multiple recursion calls. When the computation was stopped after 10 minutes the individuals tended to classically overfit the training examples.

The first solution for \text{sort}/2 was found after 70s. It is an inefficient variation of insertion sort using more recursive calls then necessary. When the process was terminated after 15 minutes several other solutions were found, which performed

---

4 All synthesis systems were executed on an Intel Celeron processor with 2.80 GHz. ADATE was compiled with MLton under Debian 2.6.8. Dialogs-II also ran under Debian 2.6.8 using SWI Prolog 5.6.9. Atre was used in version 2.3.0 and since no source code was available to us, but only a compiled version we had to run it under Windows XP SP2.
Table 1. Problem runtimes on different systems(— not tested × failed ⊥ wrong)

<table>
<thead>
<tr>
<th></th>
<th>ADATE</th>
<th>ATRE</th>
<th>DIALOGS-II</th>
</tr>
</thead>
<tbody>
<tr>
<td>Single recursive call without predicate invention</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>member/2</td>
<td>2.0s</td>
<td>91.61s</td>
<td>0.03s ⊥</td>
</tr>
<tr>
<td>unpack/2</td>
<td>1.5s</td>
<td>×</td>
<td>0.05s</td>
</tr>
<tr>
<td>length/2</td>
<td>1.2s</td>
<td>17.90s</td>
<td>0.04s</td>
</tr>
<tr>
<td>last/2</td>
<td>0.2s</td>
<td>6.35s</td>
<td>0.03s ⊥</td>
</tr>
<tr>
<td>inslast/3</td>
<td>2.7s</td>
<td>×</td>
<td>0.03s</td>
</tr>
<tr>
<td>switch/2</td>
<td>2.8s</td>
<td>1983.26s ⊥</td>
<td>0.19s ⊥</td>
</tr>
<tr>
<td>evenpos/2</td>
<td>1.6s</td>
<td>156.09s ⊥</td>
<td>×</td>
</tr>
<tr>
<td>insert/3</td>
<td>16.0s</td>
<td>—</td>
<td>0.06s</td>
</tr>
<tr>
<td>Single recursive call with predicate invention</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>reverse/2</td>
<td>78.0s</td>
<td>—</td>
<td>0.07s</td>
</tr>
<tr>
<td>i-sort/2</td>
<td>&gt;70.0s</td>
<td>—</td>
<td>0.09s ⊥</td>
</tr>
<tr>
<td>swap/2</td>
<td>232.0s</td>
<td>—</td>
<td>0.15s</td>
</tr>
<tr>
<td>shift/2</td>
<td>15.0s</td>
<td>—</td>
<td>0.11s</td>
</tr>
<tr>
<td>multlast/2</td>
<td>4.3s</td>
<td>—</td>
<td>0.13s ⊥</td>
</tr>
<tr>
<td>Multiple recursive calls with(out) predicate invention</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>flatten/2</td>
<td>110.0s</td>
<td>—</td>
<td>×</td>
</tr>
<tr>
<td>lasts/2</td>
<td>822.0s</td>
<td>—</td>
<td>×</td>
</tr>
<tr>
<td>Miscellaneous problems</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>odd/even/1</td>
<td>—</td>
<td>0.05s</td>
<td>—</td>
</tr>
<tr>
<td>mergelists/3</td>
<td>.80 00s</td>
<td>—</td>
<td>—</td>
</tr>
</tbody>
</table>

better or were syntactically simpler. The solution found after 87s was almost 24 times faster than the first one. The fastest solution found stored the result of the first recursive call to speed up the computation. The insert part of the algorithm is a recursive call of \( f \), causing an unnecessary sort of the already sorted elements. Since ADATE is able to generate sub functions a solution might be found that takes advantage of the presorted part of the list. The search time for a good solution could be reduced by splitting up the problem in two parts, first generating a solution for insert and then using that solution to solve sort.

Also for the more complex problems as lasts/2 and flatten/2 the search time increased dramatically compared to the other problems. Mergelists/3 is an example of problems using more than one induction parameter. The problem itself is solvable by a single recursive call without additional sub-functions. Although ADATE found the optimal solution after 80s, it took more time than expected, since the solution is syntactically very simple.

Generally speaking, ADATE, being able to generate almost every valid ADATE-ML program, should find a solution if even more time is provided. Nevertheless, as mentioned earlier, the same problem could be solved much faster by dividing it into the necessary sub-problems—all of them were solvable problems tested before—and providing the generated solutions as helper functions to ADATE.

**ATRE**

*Expected Outcome* Since ATRE is not capable of neither predicate nor variable invention, it only was expected to solve problems from the first class, provided
that it is supported with sufficient training evidence. Therefore we did not test ATRE with other problems, although some of them are solvable without predicate invention, however with an extremely extensive specification.

Results Indeed ATRE was able to synthesise correct, although not always minimal programs for member/2, length/2 and last/2. For switch/2 and evenpos/2 ATRE has found only a partially correct program, since the recursive clause and one base clause were correct but instead of the second base clause, ATRE came up with a non-recursive clause representing accidental similarities in the positive evidence, which have nothing in common with the intended program. This was due to the lack of sufficient negative evidence presented to the system, that would have disproven such a rule and forced the system to a different one. To overcome this, one should have either crafted the evidence in a very sophisticated way to lead the system to the correct clause or just provided all possible input output pairs over lists up to a certain length to the system.

The processing of the specifications for inslast/2 and unpack/2 was aborted manually with no result, since the system did not terminate after a sufficient time. We also suspect, that this is due to a lack of helpful training evidence as mentioned above. The remaining problems were not tested since writing the specification would have been to laborious and the expected result would not provide additional findings. The enormous deviations in the run time could be explained, that our chosen positive and negative evidence was for some problems more appropriate than for others. ATRE could therefore sometimes find a recursive theory more quickly than other times.

The synthesis of odd/1 and even/1 in parallel is one of ATRE’s prime examples. It concurrently learnt the recursive definition of odd/1 and even/1 in terms of the respective other target predicate. As background knowledge constant for 0, zero/1 and the successor function succ/2 over natural numbers together with positive and negative evidence for each predicate over natural numbers up to twenty was provided.

DIALOGS-II

Expected Outcome Since the divide-and-conquer-schema provides DIALOGS-II with the means for the necessary predicate invention to solve problems with the need for a recursive compose operators, the system should be successful with tasks from the first two classes. With respect to its schemas, we expected DIALOGS-II to be able to solve cascading recursive problems, too.

Results Surprisingly, DIALOGS-II still failed with evenpos/2, although decomposition into two heads and one tail was used. The system was unable to abduct enough evidence to close the open relation for the compose clause without a recursive call, but run into a loop when trying to close the compose predicate recursively.

Member/2, last/2 and switch/2 could not be solved without calling the system recursively. In the case of switch/2 the recursive compose operator was
needed to perform the switch of the two elements. The more simple possible solution with the two-headed decompose operator was not successful. This again questions, whether the system can really deal with other than the head tail decomposition. Nevertheless, still then the programs remained open as the last recursive clause of the compose predicate could not been closed by a second recursive call of the system, because it always ran into a non-terminating loop when called again. This is even more surprising as the closing clauses were not very complex, but simple head tail composition and DIALOGS-II mastered this in other problems without difficulties.

In the case of length/2, DIALOGS-II needed a recursive self-call when synthesising with a divide-and-conquer schema. This was due to the constructive definition of the integer numbers as Peano numbers as the successor predicate was invoked in the recursive call of the compose operator and built there constructively. Nevertheless, it still failed closing the recursive clause. Only with a descending-generalisation schema it was successful.

Similar with problems with single recursive call and predicate invention (sort/2 and multlast/2). DIALOGS-II again encountered difficulties while closing the open relation of the last recursive clause. The fact, that DIALOGS-II is fixed to its schemas and does not recognise when there is no need for another recursive relation might explain its failure here.

Additionally with more complex problem, it was either not possible to specify the problems in DIALOGS-II’s setting at all (mergelists/3) or it was not possible to specify the result with respect to one instance of the induction parameter exhaustively, as it is required by the system. Consequently DIALOGS-II failed completely (flatten/2) or ran into a non-terminating loop (lasts/2).

4 Conclusion

Since in the field of inductive program synthesis no broadly accepted fundamentals and approaches prevail, this paper systematically evaluated on the basis of four problem classes three systems, using fundamentally different induction methods. These are (a) ADATE, using a search based and (b) DIALOGS-II an analytical program synthesis approach and (c) the multi-class concept learner ATRE. We found out, that program synthesis is up till now characterised by a very strong trade-off between the restriction of the search space and the time needed for synthesis. ADATE operates in a quite unrestricted search space, capable of finding powerful solution for complex problems, whereas DIALOGS-II successively confines the search space, but with disadvantageous loss of expressional power. ATRE is an indicative example, that extending a concept learner with recursive abilities is not sufficient for satisfactory program synthesis. The goal of future research should be to combine the DIALOGS-II’s search bias with the unrestricted search space of ADATE and the expressional power of functional languages. This could for example be done by using the input/output examples during ADATE’s search not only for validation but also as a heuristic. Nevertheless, ATRE can still serve as a salutary example, since the accessory adoption of
ATRE’s $k$-beam search strategy could make it possible to learn mutually dependent recursive target functions, provided ADATE’s search time could be reduced significantly.

References

20. Erdem, E., Flener, P.: A redefinition of least generalizations and its application to inductive logic program synthesis