Inductive Synthesis of Recursive Programs – A Comparison of Three Systems

Andreas Hirschberger
andreas.hirschberger@stud.uni-bamberg.de

Martin Hofmann
martin.hofmann@stud.uni-bamberg.de

Abstract

The paper we present compares the three systems for program synthesis, namely ADATE, an approach through evolutionary computation, the inductive/abductive logic program synthesizer DIALOGS-II and the classification learner ATRE, capable of simultaneously learning mutually dependent, recursive target predicates. It gives an overview over the functionality of all three systems, and evaluates their capabilities under equal premises. As a consequence, we propose to combine ADATE's expressional power with DIALOG-II's search heuristic.

Contents

1 Introduction **4**

2 Three Prototypical Synthesis Systems **4**
2.1 Choice of Synthesis Systems . . . 4
2.2 ADATE . . . 5
2.2.1 Problem Specification . . . 5
2.2.2 Learning/Synthesis Strategy . . . 6
2.2.3 Limits and Capabilities . . . 7
2.3 ATRE . . . 8
2.3.1 Problem Specification . . . 8
2.3.2 Learning Strategy . . . 10
2.3.3 Limits and Capabilities . . . 12
2.4 DIALOGS-II . . . 13
2.4.1 Problem Specification . . . 13
2.4.2 Synthesis Strategy . . . 16
2.4.3 Limits and Capabilities . . . 17

3 Empirical Setup and Results **19**
3.1 General . . . 19
3.1.1 Hardware and System Setting . . . 19
3.1.2 Problem Classes . . . 19
3.1.3 Description of Problems . . . 21
3.1.4 Background Knowledge . . . 22
3.2 Single recursive call without predicate invention . . . 23
3.2.1 ADATE . . . 24
3.2.2 ATRE . . . 24
3.2.3 DIALOGS-II . . . 25
3.3 Single recursive call with predicate invention . . . 25
3.3.1 ADATE . . . 25
3.3.2 ATRE . . . 26
3.3.3 DIALOGS-II . . . 26
3.4 Multiple recursive call . . . 28
3.4.1 ADATE . . . 28
3.4.2 DIALOGS-II . . . 28
3.5 Miscellaneous Problems . . . 28
3.5.1 ADATE . . . 29
3.5.2 ATRE . . . 29

4 Conclusion **29**

References **32**

A Appendix **34**
A.1 Example run DIALOGS-II, member/2 . . . 34
A.2 ATRE specification for length/2 . . . 35
A.3 ATRE result of length/2 . . . 38
A.4 ADATE first result of swap/2 . . . 38
A.5 ADATE second result of swap/2 . . . 38

A.6 ADATE first result of sort/2 39
A.7 ADATE fastest result of sort/2 39
A.8 ADATE result of merge/2 39

1 Introduction

One of the most interesting sub-fields of machine learning is automatic programming. In automatic programming solutions for problems are found by providing information about the desired program behavior and not about the way to solve it. The creation of a program that performs the desired way often requires detailed knowledge of the problem domain. While a human developer might have the ability to create a program that solves the problem, he has to be provided with detailed information about the domain. The lack of those informations may cause the program to behave in an unexpected, possibly wrong way.

Being able to automatically generate runnable code from a problem specification would allow the development of reliable solutions for domain specific problems. The generated program would only depend on the specification, which could be provided by an expert of that domain.

Today there are several approaches toward automated program generation. The deductive program generation finds a solution for a specified problem by searching through the space created by the syntactically correct programs. To be able to find solutions within an acceptable time span the program language is often restricted and strong heuristics are used, creating domain specific systems. Inductive program synthesis provides another approach. Instead of searching for a program that behaves the desired way, the solution is constructed using the provided samples. This allows a mostly systematic construction of the solution instead of searching in a wide space.

The present paper provides an evaluation of three systems for automated program generation. The three systems were chosen from different approaches, which are based on inductive logical programming and evolutionary computing.

In section 2 the three classes and the representing systems are assessed. Especially the information required to generate a solution for a problem, the specification, and the capabilities of each system are determined. To understand the results provided in section 3 the underlying strategy for program generation is explained for each system. 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 this paper and gives recommendations for further work.

2 Three Prototypical Synthesis Systems

For our evaluation setting we chose three different systems covering three different approaches. The next sections justify the choice and introduce each systems and describing their functionalitites.

2.1 Choice of Synthesis Systems

Program synthesis is a not fully researched subfield of machine learning. Today several scientific approaches exist providing individual advantages and disadvantages. In this section three of those approaches, which are inductive logical programming, recursive concept learning, and evolutionary programming, will be assessed for their capabilities, their individual strengths and weaknesses. For

each of the three chosen approaches several implementations exist, which themselves have different abilities. At the end three systems where chosen each one being the most powerful and most advanced represent of its approach. These systems are ATRE which is a concept learner with the ability of learning recursions, DIALOGS-II, a system based on inductive logical programming, and ADATE which search strategy is based on evolutionary computing. Although the latest available version of DIALOGS-II is form the year 1999 it is of special interest, as it is the only system for program synthesis that is dialog based.

2.2 ADATE

Adate (Automatic Design of Algorithms Through Evolution) is a system for automatic programming developed by Roland Olsson at Ostfold University College. Adate utilizes evolutionary computing for program generation.

The following sections cover the problem specification, the procedure to generate a solution, and an overview of its limitations and capabilities.

2.2.1 Problem Specification

To be able to solve a problem Adate needs to know what the synthesized program should do. For Adate this specification consists of a text file containing Adate-ML/ML code. The provided code is then embedded in Adate's source code which is compiled with the MLton compiler. The specification is divided in two parts.

The ADATE-ML Part The first part consists of Adate-ML code. Adate-ML is a simplified subset of ML which is purely functional. In this part those datatypes and functions are defined, which will be used within the inferred function. Providing well chosen functions reduces the search space and helps reducing the search time. The user should always try to minimize the set of functions the solution may contain. It is on the other hand important that the information is adequate to describe the problem or it might be impossible, or at least more difficult, to find valid and reliable solutions. Therefore, specifications are often written with some unnecessary information to ensure possible and correct inferences. The user can utilize Adates built-in datatypes *bool*, *int* and *real* to specify the problem. For those datatypes a set of basic functions, such as comparators or arithmetical operations, are also available. Additionally, a prototype of the function to learn must be provided. The function prototype f defines number and datatypes of the input parameters and the datatype of the result. It is used as the first individual which is then expanded using evolutionary operators. This function may contain any Adate-ML code to provide problem specific knowledge to assist the search process. Usually f is provided as the most simple Adate-ML function, which just raises an exception. Finally, the main function has to be defined. It must call f at least once, but can contain additional operations that might be needed for the given problem. This allows f to be an unknown part of a bigger main program.

The ML Part The second part of the specification file contains the user callbacks and any ML code that might be necessary. A user callback is a value

that is used in the ADATE main code, but must be implemented by the user. The following callbacks are described in the ADATE Manual:

- Inputs: An ML list containing the training inputs.
- Validation_inputs: An ML list containing the validation inputs (if used).
- Abstract_types: A list of types whose values can be inspected but not constructed.
- Funs_to_use: The function set.
- Reject_funs: A list of functions that check for redundant code.
- restore_transform: A function that can alter individuals to accommodate a new f function.
- Grade: An ML structure used to grade the individuals for evaluation value calculations.
- Output_eval_fun: Function to evaluate individuals.
- Max_output_class_card: Number of individuals that can exist in an output genus.
- Max_time_limit and Max_time_limit_base: Time limit on the running of individuals.

For simple specifications the most important callbacks are *Inputs*, *Funs_to_use* and *Output_eval_fun*. *Inputs* is a list containing the training inputs. To determine the correctness of a generated individual the input and the calculated output is provided to the function *Output_eval_fun*. Here the computed output can be compared to the desired one to evaluate the individual. The desired output can be constructed using any necessary ML code. The output can, for example, be provided as a list allowing input/output pairs for training samples. For that purpose the index of the input value is also provided to the function. *funs_to_use* is a set of functions that can be utilized to generate a solution. Especially the constructors of datatypes that are used need to be provided. This function set is defined as a list of strings. The samples represent the specified problem. For ADATE to find a solution for a problem it is not necessary to provide specifically chosen samples. Like in all machine learning ADATE might find a solution covering the samples but not solving the problem. While a larger set of samples increase the possibility of finding a valid solution the search time also increases. It can be advantageous to remember that ADATE performs a learning process. This is relevant to ADATE as its learning can be made more efficient by following pedagogical principles. One should keep in mind how ADATE learns and write the specification to make gradual and evolutionary learning possible.

2.2.2 Learning/Synthesis Strategy

ADATE belongs to the field of evolutionary computation (EC) with other techniques such as genetic algorithms, evolution strategies and genetic programming. Common for EC systems is that they employ search techniques that

are inspired by basic biological principles of evolution, like for instance mutation and crossover. The search for programs conducted by ADATE is global and mostly systematic, that is, without randomization. A series of heuristics is utilized to increase the efficiency of the search. 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 within the problem specification. ADATE then generates new individuals by applying transformations to programs in the kingdom. If an individual is generated, which 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 a user defined grade. By continuing to produce and insert individuals in this manner, ADATE will find better and better programs as the search progresses.

2.2.3 Limits and Capabilities

Capability 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 able to solve every problem a valid ADATE-ML program can be found for. Whether the problem can be solved depends strongly on the user provided functions and datatypes. Providing a large set of functions ADATE can use during the construction of individuals problems of high complexity may be solvable. ADATE requires little knowledge about the way those problems can be solved making it more usable for real world problems. 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. Utilizing cluster computers allows the generation of complex solutions to some degree, which is important for scientific research.

Search time reduction While ADATE has high potential of finding solutions for complicated problems the time necessary to do so increases dramatically. To speed up the search for a given problem the search space has to be restricted. That can be attained by extending the problem providing additional background knowledge. That additional information might be provided by e.g. a more compact representation or more specific helper functions. By thinking about the additional information that can help decreasing search times the user has to think about the way the problem can be solved. The search process is forced toward the solution the user had in mind, while better ones might exist.

Search termination ADATE runs infinitely until the user terminates it. This is necessary because ADATE can always find other solutions during continuing search. Those solutions might be "better" in some way than those already found. ADATE constructs new individuals by adding complexity to previously generated. So "simple" solutions are always found first. A more efficient solution might be complex and might be hard to generate by repeatedly adding small pieces of code. While ADATE tries to improve the correct but simple solution

other approaches are neglected. For the user there is no way to estimate the time needed to find a good solution.

In addition to several other callback values the user provides how much time a generated program might use to compute the output for each given input. The time is calculated by ADATE based on Operation calls. If the time limit is exceeded the program is stopped. This is used to avoid infinite loops in the code. By increasing those time limits slow but correct solutions might be easier to find. Since this also increases the search time a reasonable limit must be defined. If no solution to a specified problem is found after some search time there is no way the user can determine the reason. ADATE might either provide a solution when the search continues or no solution might be found because the specified time limits are set to low.

2.3 ATRE

ATRE, developed by the group around Donato Malerba at the University of Bari, is a concept learning system capable of handling learning tasks with multiple target predicates involved which might be mutually dependent on each other. Its learning task is executed in the following problem setting:

Given:

- a set of concepts $C_1, \ldots, C_n$ (target predicates)
- a set of objects O described in $\mathcal{L}_O$ defining the observations ("training examples")
- background knowledge BK described in $\mathcal{L}_{BK}$
- a hypothesis language $\mathcal{L}_H$ defining the hypothesis space S_H
- a user's preference criterion PC

Find:
a (possibly recursive) theory $T \in S_H$, defining the concepts $C_1, \ldots, C_n$, such that T is complete and consistent with respect to the set of observations and BK and satisfying the preference criterion PC [Mal00].

The following sections give an overview over ATRE's problem specification and learning strategy. Since we used ATRE not in a typical concept learning environment, but for program synthesis we allow ourselves for the sake of simplicity to explain some aspects less detailed than others, but still maintain consistency. For a more elaborate description we refer to [Mal03], [Mal00], [DM04].

2.3.1 Problem Specification

In ATRE's representation languages $\mathcal{L}_O, \mathcal{L}_{BK}, \mathcal{L}_H$ a domain is described rather with a functional notation than in a logical. The basic component is the *literal*, which appears in two distinct forms:

$f(t_1, \ldots, t_n) = Value$ as a *simple literal*

$g(s_1, \ldots, s_n) = \in [a..b]$ as a *set literal*

where f and g are function symbols called *descriptors*, the t_i's and s_i's are terms, $Value$ is a constant and $[a..b]$ is a closed interval [Mal00]. A *simple literal* describes a concept with n discrete classes and *set literals* a concept which classes are defined as closed intervals over a continuous range of values. Descriptors of simple literals are called *nominal* as they are only defined over a nominal domain. Similarly, descriptors of set literals are called *linear*. Hence, no ordering relation is defined over a nominal domain, but a total ordering relation over a linear domain.

Since ATRE lacks predicate symbols in its representation languages, the first-order predicates $p(X, Y)$ and $\neg p(X, Y)$ are represented as $f_p(X, Y) = true$ and $f_p(X, Y) = false$. Note that they are a special case case of a simple literal, as $f_p(X, Y)$ describes a concept with two classes in a nominal domain. Consequently, ATRE can deal with classical negation $\neg$, but not with negation by failure not/1. Such a representation has the same expressiveness as FOL, since every possible clause of literals can be transformed in an equivalent clause in FOL, but such a representation has technical advantages for learning tasks in the domain the system was developed for [Mal03].

Concepts Concepts or target predicates are in ATRE represented as simple literals. Literals sharing the same descriptor with the same arity but different values define a multi-class problem, where the membership to a class is mutually exclusive. A trainings example is a member of exactly one class, such that positive examples of one class are negative examples of the others. Additionally, concepts can have different descriptors and can, by themselves have concept dependencies expressed by recursive theories.

Objects of Observations The so called objects of observation, described in the language of observation $\mathcal{L}_O$, represent the positive and negative evidence for the target concept and supplementary, a kind of context information in which these evidences occur. They are represented as ground multiple-head clauses, with the evidence as conjunction of simple literals of the target concept in the head and the context information in the body. For example in the family domain, an object of observation which describes the mother relation with some background knowledge may be formulated as follows:

$$\begin{aligned}
&mother(s, m) = true \wedge mother(s, f) = false \wedge mother(d, m) = true \leftarrow \\
&\quad parent(s, f) = true,\ parent(s, m) = true,\ male(f) = true, \\
&\quad male(s) = true,\ female(d),\ female(m) = true,\ brother(s, d)
\end{aligned}$$

This is just a shorthand, since every n-headed clause can be transformed into n definite clauses, each with exactly one literal from the multiple-headed clause in the head and the same body.

Background Knowledge The background knowledge represents relevant information about the domain in the language $\mathcal{L}_{BK}$ in terms of definite clauses containing both kinds of literals in the body, but only simple literals n the head. The information that is implicit in the BK is made explicit with use of trainings examples and the context information represented in the body of an object of observation via logic entailment. Following an example from the family domain:

$sister(X,Y) \leftarrow brother(Y,X), female(X)$

$sister(X,Y) \leftarrow sister(Y,X)$

2.3.2 Learning Strategy

ATRE belongs to the family of sequential covering algorithms [Mit97]. Since it is based on the strategy LEARN-ONE-RULE it learn one clause at a time and then remove the covered examples and iterating the process until all examples are covered. In this way ATRE incrementally builds up a theory T, starting with the empty theory $T_0 = \emptyset$ and adding at each step a new clause C learned in one LEARN-ONE-RULE run, thus obtaining a sequence of theories:

$$T_0, T_1, \ldots, T_i, T_{i+1}, \ldots, T_n = T$$

In the above sequence, for each i, $T_{i+1} = T_i \cup \{C\}$ holds (where C is the added clause), and all theories are complete and consistent with respect to the training set [DM04].

Learn-One-Rule In fact, ATRE performs an example driven general-to-specific parallel beam search in the space of definite clauses, whose ordering is called generalized implication [Mal00]. 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 the seed, i.e. the seed 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 $\mathcal{L}_O$ and $\mathcal{L}_{BK}$. 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. To avoided the discovery the recursive clause before the base clause of a recursive defined predicate a target predicate is only added during a refinement step if at least one clause for this predicate was already included in the theory in some earlier step.

Thus, 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 $level_i$.

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 synchronizes with the other tasks at the same level. The clauses are tested against the trainings examples and the evidence derived from the background knowledge and the objects of observation via logic entailment. 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.

A partial exploration of the search space is shown in figure 1. The selected seeds are $even(0)$ and $odd(1)$ and consistent clauses are reported in italics. The levels refer to the refinement step. Traversing the search space level by level, ATRE finds several consistent clauses to add to the empty theory T_0 but the simplest clause $even(x) \leftarrow zero(X)$ is chosen and added to T_0. Then, a second

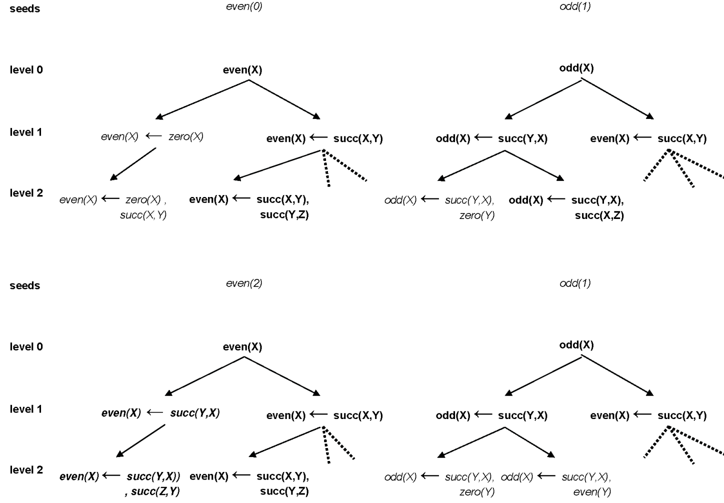

Figure 1: Parallel search for the predicates odd and even (from [Mal00], figure 2)

clause in generated by choosing the seeds $even(2)$ and $odd(1)$, but now two consistent clauses are found at the same level. Since both clauses entail the only positive example $odd(1)$ the selection is fully based on the search bias or the user's preference criterion PC and the second clause is chosen, because it is more general then the first one under ATRE's generalization hierarchy given the the partially learned theory.

Consistency Recovery Strategy 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 [Mal00]. Consider following example taken from [Mal03] with the training set and background knowledge:

$+ : \{p(5), q(1), q(4)\}$

$- : \{P(3), q(3)\}$

$BK : \{f(3), g(0), g(1), s(0,1), s(1,2), s(2,3), s(3,4), s(4,5)\}$

And following recursive theory T_2, learned after two LEARN-ONE-RULE runs, which is consistent but not complete with respect to the trainings set, since T_2 explains two positive examples $\{q(4), q(5)\}$, given BK:

C_1: $q(X) \leftarrow s(Y,X), f(Y)$

C_2: $p(Z) \leftarrow s(W,Z), q(W)$

Now a new consistent clause C, since it entails $\{q(1), q(2)\}$ given $BK \cup T_2$, is added to T_2.

C: $q(U) \leftarrow s(V,U), g(V)$

However, this makes C_2 inconsistent as it would now also entail $p(3)$. Malerba's recovery strategy fixes this problem by syntactic changes in the theory by predicate renaming and adding an auxiliary clause. T_2 remains consistent with respect to the trainings set by reformulating C_1 and C_2 as follows,

C_3: $q'(X) \leftarrow s(Y,X), f(Y)$

C_2': $p(Z) \leftarrow s(W,Z), q'(W)$

and adding following two clauses:

C_1': $q(A) \leftarrow q'(A)$

C_2': $q(U) \leftarrow s(V,U), q(W)$

It is noteworthy, that although now "new" predicates may occur in the learned theory, Atre is not capable of predicate invention in the common sense, or as it was defined by Flener [FY99], since this is only due to syntactic reformultaion.

2.3.3 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, although we know that this is not what the system was meant to do.

Learnable problem domain As mentioned several times before, Atre can concurrently learn several recursive theories which, by themselves might be mutually dependent on each other. Nevertheless, since while exploring the search space in the Learn-one-rule phase only user defined predicates are added and therefore no predicate invention [FY99] is done, no recursive theories can be learned that require recursive auxiliary predicates as a recursive compose operator. The following recursive program for *reverse/2* shall give an example of an recursive compose operator:

$\mathit{reverse}([\,],[\,])$

$\mathit{reverse}([X|Xs], Y) \leftarrow \mathit{reverse}(Xs, R), \mathit{addlast}(Y, R, Y)$

The predicate *addlast/3* acts as a composition operator. It is itself recursive, und the assumption that only the usual head/tail decomposition is known to the system. A program synthesizer with no additional background knowledge must therefore invent the predicate *addlast/3* to successfully synthesize *reverse/2*. Hence, Atre cannot handle such problems as it does not invent predicates during the the Learn-one-rule phase.

Representation of Evidence As in usual ILP settings, evidence is presented to ATRE as positive and negative trainings examples, but the system supports the classical negation rather than negation by failure. Therefore, there exists no possibility to describe negative examples in another way than including each in the set of evidence as e negative example. Omitting negative evidence may lead to rules, describing accidental regularities in the positive evidence, which may be preferred to the "correct" rules by the systems search bias as no negative evidence disproved them. Note that these problems especially occurred due to our, for ATRE unusual problem setting to synthesize recursive programs. Furthermore, writing specifications is extremely time consuming and contain normally between 200 and up to 500 lines of code. Consequently the cost-benefit ratio is quite low, especially since generating training examples is quite prone to errors due to specifiers inattention and lack of concentration after he7she has written several lines of code.

2.4 DIALOGS-II

DIALOGS-II (**D**ialogue-based **I**nductive and **A**bductive **LOG**ic program **S**ynthesizer) was developed by the group around Pierre Flener as an improvement of DIALOGS and was implemented for a master thesis in computer science by Serap Yilmaz in 1997 at the Bilkent University in Ankara [Yil97]. It is a schema-guided, interactive, inductive and abductive recursion synthesizer that takes the initiative and queries a (possibly computational naive) specifier for evidence in her/his conceptual language [Fle96].

The following sections cover the problem specification, i.e. how the synthesizer collects evidence from the specifier, the synthesis strategy, i.e. how the system induce/abduct a program and an overview of its limitations capabilities.

2.4.1 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. This is justified with the assumption that a specifier with a certain intended program in mind must have a minimal knowledge about this program, otherwise she/he would not have the need for such a program [Yil97].

Predicate Declaration and Choice of Schema 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 over the induction process, and their accordant types. DIALOGS-II supports the types {*atom, int, nat, list(_)),...*}, but since all natural numbers have to be described constructively as Peano numbers with 0 for zero and s/1 as the prefix functor for successor, induction is only carried out on *nat* and the type integer may only serve as a basis for comparisons.

As a schema-guided synthesizer, DIALOGS-II provides the user with two schemas and corresponding strategies (though only one per schema), namely a descending-generalization-schema and a divide-and-conquer-schema, to synthesize 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 synthesization process using the evidence collected from the specifier and satisfying its constraints [Yil97]. An open program is a set of clauses were some predicates are not defined by the set itself.

A general *divide-and-conquer* schema for a predicate r and parameter X,Y and Z can be described as follows. Consider the template program in Figure 2.

$$\begin{array}{lcl} r(X,Y,Z) & \leftarrow & solve_r(X,Y,Z) \\ r(X,Y,Z) & \leftarrow & decompose_r(X,HX,TX), r(TX_1,TY_1,Z),\ldots,r(TX_t,TY_t,Z), \\ & & compose_r(HX,TY,Y,Z) \end{array}$$

Figure 2: Divide-And-Conquer Schema

Let X be the induction parameter, Y the result parameter and Z an optional auxiliary or passive parameter that remains unchanged during the recursive calls. If X is minimal then Y could be directly computed from X and Z using $solve_r(X,Y,Z)$. Note that there may be more than one way to directly compute Y from X and Z, consequently there may be more than one clause in the resulting program which head is $solve_r(_,_,_)$.

In the non-minimal case, X is first decomposed into h heads and t tails via $decompose_r(X,HX,TX)$, where HX denotes h heads HX_i and TX t tails TX_i, respectively. Then the tails are processed, probably using Z, in t recursive calls, one for each TX_i, in order to get each TY_i using Z via $decompose_r(TX_i,TY_i,Z)$. Again, TX and TY denote the t tails TX_i before and the t tails TY_i after processing, respectively. Afterwards HX and TY are composed, probably using Z, by means of $compose_r(HX,TY,Y,Z)$.

Sometimes a divide-and-conquer schema turns out to be inappropriate since it becomes difficult, if not impossible to process (compose) Y from the HX and TY. To overcome this, Flener and Deville propose in [FD95] to generalize the initial the specification and then synthesize a recursive program for the generalized specification as well as a non-recursive program for the initial program as a special case of the generalized one. DIALOGS-II uses one type of problem generalization, namely *descending generalization* which generalizes a state of computation in terms of "what has already been done" and "what remains to be done" where no information is needed about the work done so far. In its simplest occurrence, descending-generalization therefore just introduces an accumulator parameter, which is progressively extended to the final result. Logic Programs for descending generalization problems are obviously not covered by a divide-and-conquer schema, because the accumulator is extended during recursive calls rather than reduced. The corresponding general schema is as follows:

Again, let X be the induction parameter, Z the optional passive parameter, Y the result parameter and let now A be the new accumulative parameter. For the non-recursive case, if X is minimal, Y is directly computed by processing X with probably the use of Z and composing the result with A. If X is not minimal, the recursive case holds and X is decomposed into one head HX and one tail TX, then the accumulator A is extended to *Anew* with the result of

$$
\begin{array}{lcl}
r(X,A,Z,Y) & \leftarrow & \mathit{solveAccu_r}(X,A,Z,Y) \\
\text{r(X, A, Z, Y)} & \leftarrow & \mathit{decompose_r}(X,HX,TX), \mathit{extendAccu_r}(HX,Z,A,Anew), \\
 & & r(TX,Anew,Z,Y)
\end{array}
$$

Figure 3: Descending Generalization Schema

processing HX with probably the use of Z. Finally the predicate r is called again on TX with the new accumulator $Anew$. Y remains unchanged here.

Decomposition Operator The last thing the user has to define initially is the decomposition operator, that divides the non-minimal case into smaller subcases to process them in recursive calls. Although it seems, regarding the template programs, that DIALOGS-II is able to handle linear as well as cascading recursion, reality looks a little bit different when considering the decomposition operators, which are defined for all its inductively defined types, i.e. lists and natural numbers.

The predefined decomposition operator for the type $list(_)$ is the usual head-tail-decomposition which decomposes a list into h heads and one tail, for $h \geq 1$:

$$
\begin{array}{lcll}
\mathit{decompose_r}(L,H,T) & \leftarrow & L = [H \mid T] & h\backslash 1,\ t\backslash 1 \\
\mathit{decompose_r}(L,H_1,H_2,T) & \leftarrow & L = [H_1,H_2 \mid T] & h\backslash 2,\ t\backslash 1 \\
\ldots & & &
\end{array}
$$

Additionally, there are only two more predefined primitives implemented in DIALOGS-II, namely $partition(L,H,T_1,T_2)$ and $halves(T,H_1,H_2)$:

$$
\begin{array}{lcll}
\mathit{decompose_r}(L,H,T_1,T_2) & \leftarrow & L = [H \mid T], partition(L,H,T_1,T_2) & h\backslash 1,\ t\backslash 2 \\
\mathit{decompose_r}(L,T_1,T_2) & \leftarrow & L = [_,_ \mid _], halves(L,T_1,T_2) & h\backslash 0,\ t\backslash 2
\end{array}
$$

The primitive $partition(L,H,T_1,T_2)$ divides an integer list L into two parts T_1 and T_2, where T_1 contains all elements of L that are smaller or equal than the pivot H and T_2 contains all elements that are greater than the pivot H. DIALOGS-II's last predefined primitive for the type $list(_)$ is $halves(L,T_1,T_2)$, which parts the list L into two halves T_1 and T_2.

The decomposition operator for the type *nat* is defined similarly and parts a natural number N typed as Peano number into one head HN and one tail TN:

$$
\mathit{decompose_r}(N,HN,TN) \quad \leftarrow \quad N = s^n(TN), HN = N \qquad h\backslash 1,\ t\backslash 1
$$

In the case of the type *nat* the decomposition operator specifies more a in-/decrement with the invariable n than a proper decomposition into head and tail. Nevertheless, for consistency reasons, the same terminology is used.

As one can see so far DIALOGS-II recursive ability is, although kept unrestraint in the template program, limited through the predefined decomposition operators to not more than two recursive calls. Furthermore, we were not able to synthesize a correct program using $halves\backslash 3$ or $partition\backslash 4$ during evaluation, what brings up the question whether DIALOGS-II can handle other than linear recursion at all.

2.4.2 Synthesis Strategy

Till now, all structural decisions about parameter roles and their types, template program and decomposition operator are made. This again results in an open program, where the sole open predicates are the open relation for the non-recursive clause p and and the open relation for the recursive clause q. In case of a divide-and-conquer template solve_r takes the role of p and *compose_r* the role of *q*; in case of a descending generalization schema the open predicates are *olveAccu_r* and *extendAccu_r*, respectively.

Abduction of Evidence From now on, DIALOGS-II starts querying the specifier in order to abduct evidence for the open relations p and q. Therefore it asks under what conditions the predicate $r(I, R, P)$ with probably some additional assumptions holds, starting with the smallest, most general instance of the induction parameter and increasing 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. In the following the dialogue is explained with respect to the divide-and-conquer schema. For a descending-generalization schema the only the accumulator parameter A has to be included in the specification.

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. Nevertheless, this has to be done exhaustively with respect to the induction parameter I to cover all its possible parameter, since it is in its most general form of a given size. Also note that, although the system primitives are known to DIALOGS-II by definition they will not be used unless they occur in an evidence clause. Similar, DIALOGS-II understands predicates introduced by the specifier as simple atoms assumed to be true during the the abduction process and later resolution as well as given during execution time of the synthesized program.

The answer should then be some formula $\mathcal{F}[R]$, where R again is the result parameter of the queried instance of the predicate r and the only free variable in $\mathcal{F}$ and therefore explaining how to compute R from I and P [Fle96]. The formula $\mathcal{F}$ consists of the target predicate with grounded I and P (denoted as i and p) and free R in the head, and of disjunctions over $\mathcal{E}_n$ in the body. Each $\mathcal{E}_n$ represents a conjunction of the evidence for which the instantiated target predicate holds:

$$\mathcal{F}[R] := r(i, p, R) \leftarrow \bigvee_n \mathcal{E}_n[R]$$

n clauses $\mathcal{F}_n[R]$ are defined as:

$$\mathcal{F}_n[R] := r(i, p, R) \leftarrow \mathcal{E}_n[R]$$

The n evidence pair of the open relations p and q are generated by matching the head of each clause $\mathcal{F}_n$ against the head of each clause in the open program and simplifying them via SLD resolution and unfolding with already collected evidence. Let now denote $\langle e_p, e_q \rangle_{\mathcal{E}_n}$ the evidence pair derived like this from clause

$\mathcal{E}_n$. These pairs are used together with the initial open template program and previously collected evidence to generate the next queries and possible assumptions again via unfolding and SLD resolution until the specifier stops the query phase. See Appendix Asec:dialogsexample) for an example run of $member/2$.

Induction of Clauses The result of the abduction for each query, i.e. for every queried instance of the induction parameter, are evidence pairs $\langle e_p, e_q \rangle_{\mathcal{E}_n}$ where e_p and e_q represent clauses describing under which conditions p, respectively q, hold with a certain instance of the induction parameter. To close the open relations, all evidence clauses for the predicate q (e_q) are divided into a minimal number of subsets, so called cliques, such that any two elements in this set have the same admissible least generalization under θ-subsumption (lgθ). If now the counterpart subset of each clique (consisting of the appropriate e_ps) has also an admissible lgθ, then the clique is deleted from the clauses for q, otherwise that counterpart subset of this clique is deleted from the clauses for p. Then the lgθs for q are taken as evidence clauses for q and the process is repeated for the clauses for p. The resulting sets are the closing clauses for the open predicates [EF].

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 [Yil97]. 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.

2.4.3 Limits and Capabilities

Since the last section gave an overview over the functionality of Dialogs-II, in this section we will discuss what programs Dialogs-II can and cannot synthesize.

Complete specification Without doubt, Dialogs-II's strength lies clearly in its abductive querying mechanism. It makes the system capable of interrogating the specifier to successively collect information about a certain domain exhaustively. This may be quite useful in relative complex domains were the specifier is not able to do so without help, but an extensive and complete specification is crucial for the desired result as for expert systems. The problems arise when it is not possible any more to completely describe the behaviour of a program 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.

Noise-free Evidence Data Along with the necessity for a complete, or let it call self-contained, description of conditions under which the synthesized predicate holds comes also the need for a noise-free description. The evidence is not

presented to the system as specific positive or negative trainings examples, but moreover, as mentioned before, as a description of conditions under which the desired predicate must hold and therefore describing a whole class of possible trainings examples on the basis of the most general instance of the induction parameter with a certain size. Consequently DIALOGS-II is more prone to errors due to noisy evidence data, not only because errors in the description have a higher impact on the synthesized program, but also because the evidence clause entered by the specifier is directly used during the following abduction process in unfolding and resolution operations.

Parameters and Schemas Another drawback of DIALOGS-II is rooted in the implementation of its schemas. All programs DIALOGS-II is able to synthesize are restricted to exactly one induction parameter, one result parameter and one passive parameter, which might be omitted. DIALOGS-II cannot handle any parameter setting deviating from the mentioned one, e.g. recursion over more than one parameter, what would be also similar to a setting with an arbitrary number of passive parameters. We also failed to synthesize predicates without result parameter representing for example simple boolean concepts like *odd/even*.

Similar, as mentioned before, despite the schemas are kept unrestraint for any type of recursion the decomposition operators support only decomposition of the induction parameter into one or two tails which can be processed in further recursive calls. DIALOGS-II should therefore be capable of handling at least cascading recursion with a branching factor of two. However, during our experiments the system failed with any problem where recursion other than linear was necessary.

Use of Background Knowledge During the specification process, the user has to support the system with a lot of background knowledge and the fact that the evidence clause is directly used to abduct evidence for the open relations has also important consequences for DIALOGS-II's use of this knowledge. Although many primitives like ordering relations or simple arithmetic are already known to the system, they cannot be used unless the specifier explicitly includes them in the evidence clause and thus makes them available to the system for unfolding and resolution. Predicate invention is 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. As already mentioned, all unknown predicates occurring in the evidence clause are assumed to be available to the final synthesized program. Thus, a pedagogical evidence description and the specifier's aid which background knowledge to attribute to which problem influences the synthesization process radically and has a crucial influence on the final result.

Inductive Bias The problems mentioned in the last paragraphs can easily explained under the notion of a inductive bias. DIALOGS-II language bias is determined by its used schemas, since it only traverses the space of programs that are direct instances of them. Thy Background Knowledge provided by the user 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. DIALOGS-II's need for self-contained and noise-free evidence is rooted in its search bias, 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. Is now 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

In this section we present the results of our evaluation and describe the test setting. The following section cover first some general issues as a justification of our setting and a description of the example problems and the provided background for each system. Then the evaluation for each problem class follows.

3.1 General

This section intends to provide the basic information describing or evaluation 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, the hardware and the used operation system are specified, followed by a description of the applied problem classes, then we describe the programs we intended to synthesize and finally the background knowledge used for each system.

3.1.1 Hardware and System Setting

All synthesis systems were executed on a Intel Celeron processor with 2.80 GHz. ADATE was compiled with MLton under Debian 2.6.8. We used always the newest the version from the still ongoing development. 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 also on a Intel Celeron machine with 2.80 GHz.

3.1.2 Problem Classes

Although all three synthesis systems provide to some extend means to learn recursive programs, they still remain quite inhomogeneous concerning their underlying concepts. To properly evaluate them, we were forced to some "greatest common divisor" for their problem space. The Venn diagram in figure 4 shall illustrate the extend of the capabilities of the three systems:

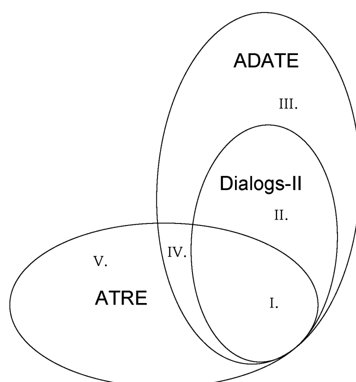

Figure 4: Expressional power of the three systems

ADATE's space of learnable programs completely subsumes that of DIALOGS-II and partly overlaps with that of ATRE. ADATE and ATRE are capable of learning programs not covered by any of the other systems.

When examining the three different systems with regard to their capabilities we could therefore identify five different classes according to whether they can invent new predicates and variables or not as well as the number of recursive calls, and multi-class recursive predicate learning. We used following classes:

Single recursive call without predicate invention (I.)
These problems are solvable with a single recursive call per iteration and do not require any predicate or variable invention.

Single recursive call with predicate invention (II.)
These problems additionally require at least the invention of an auxiliary predicate.

Multiple recursive call (III. + VI.)
To solve these problems, at least a second recursive call is necessary. This may be a call of another recursive predicate or an additional call of the target predicate itself.

Miscellaneous (V. + III.)
This class contains specific problems emphasizing the special capabilities of a certain system.

We combined the classes III. and VI., since DIALOGS-II is not capable of multiple recursive calls and although ATRE may be capable, a specification for such a problem would be too extensive. This is due to the fact the data structure of such a problem is even more complex than normal lists. Writing a specification for ATRE would be similar to enumerate every input/output tuple as grounded representations of the accordant instances of the specific data structure and labelling either as a positive or a negative training example.

This could only be efficiently done by writing a program for writing ATRE-specifiactions, but such a program itself would require the program to synthesize to label the I/O-tuples.

To demonstrate the special abilities of ADATE and ATRE we finally also introduced an additional miscellaneous class.

3.1.3 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 provided to the systems. It had also the advantage, that remained more or less unchanged for all test settings. This made the whole evaluation more transparent and facilitated the comparison of results between the different systems.

For these reasons, we abstained from numerical problems. Even very simple recursive problems already require quite extensive background knowledge, which may change from setting to setting and are also quite complex compared to list problems. For example, synthesizing a program computing the nth Fibonacci's numbers or a program to compute the greatest common divisor.

Although it is usually common sense to distinguish between structural and semantic problems, one can argue, if such a differentiation is necessary. We are certain that every semantical problem could be reduced to a structural problem by adding sufficient information to the background knowledge. A semantic problem, by definition, requires some knowledge about the domain in which it is posed or about, i.e. this knowledge is needed to solve this problem. We think, every it semantical problem could be reduced to a structural problem, be giving the knowledge that implicitly is required to solve this problem explicitely to the system as background knowledge. Therefore we treated structural and semantic problems equally. In the following we provide a short description of our examined problems. They contain typical, already researched problems, e.g. *member/2* and *reverse/2* [Sum77], [MF90], 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 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.

switch(X,Y) holds iff list Y can be obtained from list X were all elements on a 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 the list Y is the reverse of list X.

shift(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 X could be derived from list Y 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.

multi(X, Y) holds iff X is a list of lists and Y the concatenated results of applying *switch/2* to the first element. of X, *evenpos/2* to the second, *multlast/2* to the third, again *switch/2* to the next, and so on.

pack(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 nth of the list X and Y, respectively.

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

3.1.4 Background Knowledge

We tried to keep the background knowledge provided to the systems as similar as possible for all of them. Nevertheless, due to different concepts of the systems, this was only possible to some extend. In this section we describe all additional information given to the synthesizers to give the reader a basis for individual judgement.

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. A list of integers is defined as follows:

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

nill and cons are the constructors of the datatype "list" and must be added to the callback value "funs_to_use". Using elements of the datatype int allows the use of the also built-in integer operations simplifying the problem specification. Most of the callbacks required in the specification are set to the value used in the sample specification for XOR described in the ADATE manual. Several problems require additional datatypes and functions. For the problems *lasts/2*, *pack/2* and *multi/2* the datatype list of list of int is needed. It is defined by

```
datatype lists = nill' | cons' of list * lists
```

with nill' and cons' added to "funs_to_use". 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. For all problems the samples are provided by pairs of input/output values. Since ADATE uses a functional language the result parameter is replaced by the return value. The problem definition is the same, only that one parameter is implicitly provided.

ATRE Since ATRE operates in the usual ILP setting, data structure such as lists are in general unknown to the system. Therefore, we represented all lists with 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, *nill([ ])* for the empty list, in the object of observation as the body of the multi-headed clause.

For the setting 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 *succ(X,Y)*, where *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. See Appendix A.2 for a excerpt of an specification for *length/2* and Appendix A.3 for the result of the synthesization.

Dialogs-II Unless not otherwise stated, for all problem synthesizations the divide-and-conquer schema was used, as well as the simple decomposition operator, which decomposes a list into one head and one tail. A descending generalization schema was inappropriate for most problems. Although it would have been possible to synthesize a program with this schema, it allows only the extension of the accumulator variable via cons or append. The former requires a program where the output is somehow the reverse from its input (perfect for $reverse/2$) or the order does not matter, the latter would have represent an additional information of background knowledge. Elsewise all specifications were done as required by the system, but it is important to keep in mind that this by itself is already a strong bias, because as mentioned before, they are always complete and quite verbose with respect to one instance of the induction parameter.

3.2 Single recursive call without predicate invention

Table 1 shows the result of the test runs on problems with one single recursive call without predicate invention.

	ADATE	ATRE	DIALOGS-II
member/2	2.0s	91.61s	0.03s ⊥
unpack/2	1.5s	×	0.05s
length/2	1.2s	17.90s	0.04s
last/2	0.2s	6.35s	0.03s ⊥
inslast/3	2.7s	×	0.03s
switch/2	2.8s	1983.26s ⊥	0.19s ⊥
evenpos/2	1.6s	156.09s ⊥	×
insert/3	16.0s	—	0.06s

— not tested × failed ⊥ wrong

Table 1: Runtimes for problems with a single recursive call and without predicate invention

3.2.1 ADATE

Expected Outcome Since the provided problems are solvable by a single recursive call and do not require any sub-functions or additional variables ADATE was expected to find solutions for all problems of this class rather quickly.

Results All of the provided problems where solved as expected. For each problem the first program that provided correct output for all the samples was the optimal solution. An interesting observation was the increasing search times by adding complexity to a problem. While the differences between last/2 and inslast/3 and member/2 and insert/3 are rather small the increase of search time was intense.

3.2.2 ATRE

Expected Outcome Since these problems require neither predicate nor variable invention, ATRE was expected to solve problems of this class, provided that it is supported with sufficient trainings evidence.

Results Indeed ATRE was able to synthesize 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 trainings 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 cognitions.

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.

3.2.3 DIALOGS-II

Expected Outcome Since all problem are solvable with a divide-and-conquer schema and no recursive compose operator is necessary, Dialogs-II should have been able to solve all problems of this class.

Results Surprisingly, Dailogs-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, what 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, since 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 synthesizing 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-generalization schema it was successful.

3.3 Single recursive call with predicate invention

Table 2 shows the result of the test runs on problems with one single recursive call and with predicate invention.

3.3.1 ADATE

Expected Outcome The need of predicate invention should be no obstruction for Adate. It should be able to solve all problems of this class.

	ADATE	ATRE	DIALOGS-II
reverse/2	78.0s	—	0.07s
i-sort/2	>70.0s	—	0.09s ⊥
swap/2	232.0s	—	0.15s
shift/2	15.0s	—	0.11s
multlast/2	4.3s	—	0.13s ⊥

— not tested × failed ⊥ wrong

Table 2: Runtimes for problems with single recursive call and with predicate invention

Results ADATE found solutions for all specified problems. The generated program for revers/2, shift/2 and multlast/2 was optimal. The code of swap/2 and sort/2 was correct but inefficient. For swap/2 a second solution (A.5) was found after 315s which performed better then the first (A.4). Both solutions use multiple recursion calls. When the computation was stopped after 10 minutes the individuals tended to satisfy the samples directly, without the use of recursion. The first solution for sort/2 was found after 70s (A.6). It is a 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 better or were syntactically simpler. The solution found after 87s was almost 24times faster than the first one. The fastest solution found (A.7) 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.

For both problems more time might have provided better results.

3.3.2 ATRE

Since ATRE is not capable of predicate invention and with respect to very extensive specifications we considered it as superfluous to test ATRE with problem of this class.

3.3.3 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 have been successful with these tasks.

Results DIALOGS-II again encountered difficulties while closing the open relation of the last recursive clause. Isort/2 and multlast/2 remained open, although the problem was already solved after the non-recursive relation of the last recursive clause compose_xyz. The open recursive relation compose_compose_xyz was therefore unnecessary and could have been deleted or closed with a no-op

program. The fact, that Dialogs-II is fixed to its schemas and does not recognize when there is no need for another recursive relation might explain its failure here.

3.4 Multiple recursive call

Table 3 shows the result of the test runs on problems with multiple recursive calls.

	ADATE	ATRE	DIALOGS-II
pack/2	110.0s	—	×
lasts/2	822.0s	—	×
multi/2	×	—	—

— not tested × failed ⊥ wrong

Table 3: Runtimes for cascading recursive problems

3.4.1 ADATE

Expected Outcome Despite the problems of this class being more complicated ADATE should be able to solve all of the problems. The search time could become a critical problem trying to find solutions.

Results As expected the search time increased dramatically compared to the other problems. While lasts/2 and pack/2 could be solved, providing a good solution, ADATE could not find a solution for multi/2 within the provided time. Even extending the test time to 48 hours did not enable ADATE to find a solution. ADATE, being able to generate almost every valid ADATE-ML program, should find a solution if even more time is provided. 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.

3.4.2 DIALOGS-II

Expected Outcome With respect to its schemas we expected DIALOGS-II to be able to solve cascading recursive problems, too. Nevertheless, the results and our experience gained during evaluation revealed that DIALOGS-II is not. Although, the decomposition operators partition/4 and halves/3 raised hope it seems that this is not implemented completely.

Either it was not possible to specify the problems in DIALOGS-II's setting at all (multi/2 and 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 (pack/2) or ran into a non-terminating loop (lasts/2).

3.5 Miscellaneous Problems

Table 4 shows the result of the test runs on the miscellaneous problems. These problems were included to give the reader a glimpse on the capabilities of ADATE and ATRE which to look at in detail is beyond this paper.

odd_even/1 (Atre)	0.05s
mergelists/3 (Adate)	80.00s

Table 4: Runtimes miscellaneous problems

3.5.1 ADATE

Expected Outcome Mergelists is a example of problems using more then one induction parameter. The problem itself is solvable by a single recursive call without additional sub-functions.

Due to this Adate should be able to generate a solution within a small amount of time and provided the optimal solution after 80s. Since the solution is syntactically very simple (A.8) 80s is more time then was expected.

3.5.2 ATRE

This tasks is one of Atre's prime examples. It learned 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.

4 Conclusion

After all, what can we capitalise on our work done so far? Is it possible to prefer one system over another? Definitely not, since the systems still are too different for a direct comparison. Although Dialogs-II took only a fraction of time Adate needed for synthesization, it often finally failed to output a correct program. Contrary, Adate was successful on nearly all problems, provided with sufficient time. Nonetheless its programs also needed a final validation by human specifier. Although it was quite successful in dealing with the assigned tasks, a comparison with the other two systems is nearly impossible due to the application of Atre in a domain other than originally intended by its developers.

Within the setting of our work it was not possible to finally fathom all boundaries of the systems capabilities. We already showed that Atre is more powerful than expected by its developers[1], regarding the program synthesis domain, provided the system is supported with sufficient or exhaustive trainings examples. We also believe, that Atre should be able to handle problems that require multiple recursive calls, but a specification of such problems would go beyond the scope of this work.

Dialog-II's greatest upper bounds are already determined through its schemas, but within this boundary a definite statement cannot be made since unless its theoretical potentialities remain restricted trough its implementation.

Theoretically, Adate's restrictions are only provided by the expressiveness of its language of hypothesis Adate ML and the necessity of finite data types. Nevertheless, with increasing intricacy of the intended problem the search time rises explosively and makes its application to really interesting extensive real world problems if not practically impossible then inefficient.

[1]from personal communication via email with Antonio Varlaro

Regrading the examined systems, program synthesis is up till now characterized by a very strong trade-off between the restriction of the search space and the time needed for synthesization. ADATE operates in 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. 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 an heuristic.

Acknowledgements

We would like to thank Pierre Flener for his provided assistance with understanding DIALOGS-II code during the portation of from Sicstus Prolog to Swi Prolog. Also Antonio Varlaro for correcting our extensive ATRE specification and Roland Olsson for giving us access to the working prototype of ADATE. We are esspecially grateful for the support and tutoring Emanuel Kitzelmann and Ute Schmid gave us.

List of Tables

1 Runtimes for problems with a single recursive call and without predicate invention . . . 24
2 Runtimes for problems with single recursive call and with predicate invention . . . 26
3 Runtimes for cascading recursive problems . . . 28
4 Runtimes miscellaneous problems . . . 29

List of Figures

1 Parallel search for the predicates odd and even (from [Mal00], figure 2) . . . 11
2 Divide-And-Conquer Schema . . . 14
3 Descending Generalization Schema . . . 15
4 Expressional power of the three systems . . . 20

References

[DM04] M. Berardi D. Malerba, A. Varalro. Learning recursive theories with the separate-and-parallel conquer strategy. In *Proceedings of the Workshop on Advances in Inductive Rule Learning in conjunction with ECML/PKDD*, pages 179–193, 2004.

[EF] E. Erdem and P. Flener. A redefinition of least generalizations and its application to inductive logic program synthesis. citeseer.ist.psu.edu/140293.html.

[FD95] Pierre Flener and Yves Deville. Logic program transformation through generalization schemata. In *Logic Program Synthesis and Transformation*, pages 171–173, 1995.

[Fle96] P. Flener. Inductive logic program synthesis with Dialogs. In S. Muggleton, editor, *Proceedings of the 6th International Workshop on Inductive Logic Programming*, pages 28–51. Stockholm University, Royal Institute of Technology, 1996.

[FY99] Pierre Flener and Serap Yilmaz. Inductive synthesis of recursive logic programs: Achievements and prospects. *J. Log. Program.*, 41(2-3):141–195, 1999.

[Mal00] Donato Malerba. Learning recursive theories in the normal ilp setting. In J. Cussens and A. Frisch, editors, *Lecture Notes in Artificial Intelligence*, pages 93–111. Springer, Berlin, Germany, 2000.

[Mal03] Donato Malerba. Learning recursive theories in the normal ilp setting. *Fundamenta Informaticae*, 57:39–77, 2003. IOS Press.

[MF90] S. Muggleton and C. Feng. Efficient induction of logic programs. In *Proc. of the Workshop on Algorithmic Learning Theory by the Japanese Society for AI*, pages 368–381, Tokyo, 1990.

[Mit97] Tom M. Mitchell. *Machine Learning*. McGraw-Hill, New York, 1997.

[Ols94] J. R. Olsson. *Inductive functional programming using incremental program transformation and Execution of logic programs by iterative-deepening A* SLD-tree search*. Dr scient thesis, University of Oslo, Norway, 1994.

[Ols95] J. R. Olsson. Inductive functional programming using incremental program transformation. *Artificial Intelligence*, 74(1):55–83, 1995.

[Ols98a] J. Roland Olsson. The art of writing specifications for the ADATE automatic programming system. In John R. Koza, Wolfgang Banzhaf, Kumar Chellapilla, Kalyanmoy Deb, Marco Dorigo, David B. Fogel, Max H. Garzon, David E. Goldberg, Hitoshi Iba, and Rick Riolo, editors, *Genetic Programming 1998: Proceedings of the Third Annual Conference*, pages 278–283, University of Wisconsin, Madison, Wisconsin, USA, 22-25 1998. Morgan Kaufmann.

[Ols98b] Roland Olsson. Population management for automatic design of algorithms through evolution. In *Proceedings of the 1998 IEEE World Congress on Computational Intelligence*, pages 592–597, Anchorage, Alaska, USA, 5-9 1998. IEEE Press.

[Ols99] J. Olsson. How to invent functions, 1999.

[Sum77] P. D. Summers. A methodology for LISP program construction from examples. *Journal ACM*, 24(1):162–175, 1977.

[Vat] Geir Vattekar. *ADATE User Manual*.

[Yil97] Serap Yilmaz. Inductive synthesis of recursive logic programs. Master's thesis, University of Bilkent, Computer Science Department, 1997.

A Appendix

sec:appendix)

A.1 Example run DIALOGS-II, member/2

```
Predicate declaration?
|: member(L:list(term),M:term)
[dc, dg]
Schema? {dc}
|:
dc
[divide_and_conquer_strategy1]
Strategy? {divide_and_conquer_strategy1}
|:
divide_and_conquer_strategy1
Induction parameter? {L}
|:
L
Passive parameter(s)? {[M]}
|:
[M]
Decomposition Operator? {decompose(L,HL,TL)<--L=[HL|TL]}
|:
decompose(L,HL,TL)<--L=[HL|TL]
When does member([], A) hold?
|: false.
When does member([A], B) hold?
|: A=B.
When does member([A, B], C) hold?
|: A=C;B=C.
When does member([A, B, C], D) hold?
|: A=D;B=D;C=D.
When does member([A, B, C, D], E) hold?
|: stop_it.
compose evidence:

Result of the Program Closing Method:
Clauses for compose:

Clauses for solve:
   solve_member([A|B], A)<--

Please evaluate the Program Closing Method results: need for recursive synthesis? [yes/no]
|: no
A possible program is:
   member(A, B) <-- solve_member(A, B)
   member(A, C) <-- decompose_member(A, D, E),member(E, C, F),compose_member(D, C)
   decompose_member(G, H, I) <-- G=[H|I]
   solve_member([J|K], J)<--
```

```
Do you want another logic program? {yes}
no
```

A.2 ATRE specification for length/2

```
concept_list([length(_,_)=.true]).

descriptor(length(list,number)=.boolean,nominal,0).
descriptor(nil(list)=.boolean,nominal,0).
descriptor(succ(number,number)=.boolean,nominal,0).
descriptor(zero(number)=.boolean,nominal,0).
descriptor(components(list,elem,list)=.boolean,nominal,0).

(...)

starting_number_of_literals(length(_,_)=.true,0).

object(obj1,[
length([],0)=.true,
length([],1)=.false,
length([],2)=.false,
length([],3)=.false,
length([],4)=.false,

length([a],0)=.false,
length([a],1)=.true,
length([a],2)=.false,
length([a],3)=.false,
length([a],4)=.false,

(...)

length([d],0)=.false,
length([d],1)=true,
length([d],2)=.false,
length([d],3)=.false,
length([d],4)=.false,

length([ab],0)=.false,
length([ab],1)=.false,
length([ab],2)=.true,
length([ab],3)=.false,
length([ab],4)=.false,

length([ac],0)=.false,
length([ac],1)=.false,
length([ac],2)=.true,
length([ac],3)=.false,
length([ac],4)=.false,
```

```
(...)

length([dc],0)=.false,
length([dc],1)=.false,
length([dc],2)=.true,
length([dc],3)=.false,
length([dc],4)=.false,

length([abc],0)=.false,
length([abc],1)=.false,
length([abc],2)=.false,
length([abc],3)=.true,
length([abc],4)=.false,

(...)

length([dcb],0)=.false,
length([dcb],1)=.false,
length([dcb],2)=.false,
length([dcb],3)=.true,
length([dcb],4)=.false,

length([abcd],0)=.false,
length([abcd],1)=.false,
length([abcd],2)=.false,
length([abcd],3)=.false,
length([abcd],4)=.true,

length([abdc],0)=.false,
length([abdc],1)=.false,
length([abdc],2)=.false,
length([abdc],3)=.false,
length([abdc],4)=.true,

(...)

length([dcba],0)=.false,
length([dcba],1)=.false,
length([dcba],2)=.false,
length([dcba],3)=.false,
length([dcba],4)=.true
],[
zero(0)=.true,//background knowledge or context information
succ(0,1)=.true,
succ(1,2)=.true,
succ(2,3)=.true,
succ(3,4)=.true,
nil([])=.true,
components([a],a,[])=.true,
components([b],b,[])=.true,
```

```
components([c],c,[])=.true,
components([d],d,[])=.true,
components([ab],a,[b])=.true,
components([ac],a,[c])=.true,
components([ad],a,[d])=.true,
components([ba],b,[a])=.true,
components([bc],b,[c])=.true,
components([bd],b,[d])=.true,
components([ca],c,[a])=.true,
components([cb],c,[b])=.true,
components([cd],c,[d])=.true,
components([da],d,[a])=.true,
components([db],d,[b])=.true,
components([dc],d,[c])=.true,
components([abc],a,[bc])=.true,
components([abd],a,[bd])=.true,
components([acb],a,[cb])=.true,
components([acd],a,[cd])=.true,
components([adb],a,[db])=.true,
components([adc],a,[dc])=.true,
components([bac],b,[ac])=.true,
components([bad],b,[ad])=.true,
components([bca],b,[ca])=.true,
components([bcd],b,[cd])=.true,
components([bda],b,[da])=.true,
components([bdc],b,[dc])=.true,
components([cab],c,[ab])=.true,
components([cad],c,[ad])=.true,
components([cba],c,[ba])=.true,
components([cbd],c,[bd])=.true,
components([cda],c,[da])=.true,
components([cdb],c,[db])=.true,
components([dab],d,[ab])=.true,
components([dac],d,[ac])=.true,
components([dba],d,[ba])=.true,
components([dbc],d,[bc])=.true,
components([dca],d,[ca])=.true,
components([dcb],d,[cb])=.true,
components([abcd],a,[bcd])=.true,
components([abdc],a,[bdc])=.true,
components([acbd],a,[cbd])=.true,
components([acdb],a,[cdb])=.true,
components([adcb],a,[dcb])=.true,
components([adbc],a,[dbc])=.true,
components([bacd],b,[acd])=.true,
components([badc],b,[adc])=.true,
components([bcad],b,[cad])=.true,
components([bcda],b,[cda])=.true,
components([bdac],b,[dac])=.true,
components([bdca],b,[dca])=.true,
```

```
components([cabd],c,[abd])=.true,
components([cadb],c,[adb])=.true,
components([cbad],c,[bad])=.true,
components([cbda],c,[bda])=.true,
components([cdab],c,[dab])=.true,
components([cdba],c,[dba])=.true,
components([dabc],d,[abc])=.true,
components([dacb],d,[acb])=.true,
components([dbac],d,[bac])=.true,
components([dbca],d,[bca])=.true,
components([dcab],d,[cab])=.true,
components([dcba],d,[cba])=.true]).
```

A.3 ATRE result of length/2

```
rule(1,length(X2,X1)=.true,[zero(X1)=.true,nil(X2)=.true],2).

rule(2,length(X2,X1)=.true,[succ(X3,X1)=.true,
components(X2,X5,X4)=.true,length(X4,X3)=.true],2).
```

A.4 ADATE first result of swap/2

```
fun f X =
  case X of
    nill => nill
  | cons( VB3A5, VB3A6 ) =>
   case f( VB3A6 ) of
     nill => X
   | cons( V7CB8D, V7CB8E ) =>
       cons( V7CB8D, f( cons( VB3A5, V7CB8E ) ) )
```

A.5 ADATE second result of swap/2

```
fun f X =
  case X of
    nill => X
  | cons( VB3A5, VB3A6 ) =>
   case VB3A6 of
     nill => X
   | cons( VAD6EA, VAD6EB ) =>
   case f( VB3A6 ) of
     nill => (raise NA_AD6EC)
   | cons( V7CB8D, V7CB8E ) =>
       cons(
         V7CB8D,
```

```
        case V7CB8E of
          nill => cons( VB3A5, V7CB8E )
        | cons( V8F109, V8F10A ) => f( cons( VB3A5, V7CB8E ) )
        )
```

A.6 ADATE first result of sort/2

```
fun f Xs =
  case Xs of
    nill => Xs
  | cons( VE0C, VE0D ) =>
   case f( VE0D ) of
     nill => Xs
   | cons( VB750, VB751 ) =>
   case ( VB750 < VE0C ) of
     false => cons( VE0C, f( VE0D ) )
   | true => f( cons( VB750, cons( VE0C, VB751 ) ) )
```

A.7 ADATE fastest result of sort/2

```
fun f Xs =
  case Xs of
    nill => Xs
  | cons( VE0C, VE0D ) =>
   case f( VE0D ) of
     V2EBC1 =>
   case V2EBC1 of
     nill => Xs
   | cons( VB750, VB751 ) =>
   case ( VB750 < VE0C ) of
     false => cons( VE0C, V2EBC1 )
   | true => cons( VB750, f( cons( VE0C, VB751 ) ) )
```

A.8 ADATE result of merge/2

```
fun f( X, Xs ) =
  case X of
    nill => Xs
  | cons( V557, V558 ) => cons( V557, f( Xs, V558 ) )
```