Data-Driven Induction of Recursive Functions from Input/Output-Examples

Emanuel Kitzelmann

Faculty of Information Systems and Applied Computer Sciences,
University of Bamberg, 96045 Bamberg, Germany
emanuel.kitzelmann@wiai.uni-bamberg.de
http://www.cogsys.wiai.uni-bamberg.de/kitzelmann/

Abstract. We describe a technique for inducing recursive functional programs over algebraic datatypes from few non-recursive and only positive ground example-equations. Induction is data-driven and based on structural regularities between example terms. In our approach, functional programs are represented as constructor term rewriting systems containing recursive rewrite rules. In addition to the examples for the target functions, background knowledge functions that may be called by the induced functions can be given in form of ground equations. Our algorithm induces several dependent recursive target functions over arbitrary user-defined algebraic datatypes in one step and automatically introduces auxiliary subfunctions if needed. We have implemented a prototype of the described method and applied it to a number of problems.

1 Introduction

Automatic induction of recursive functional or logic programs from input/output-examples (I/O-examples) is an active area of research since the sixties (see [1] for classical methods, [2] for systems in the field of inductive logic programming, and [3] for recent research).

There exist two general approaches to tackle inductive synthesis of programs: (i) In the generate-and-test approach (e.g., the ADATE system [4]), programs of a defined class are enumerated heuristically and then tested against given examples. (ii) In the analytical approach, programs of a defined class are derived by detecting recurrences in given examples which are then generalized to recursively defined functions. That is, hypotheses[1] are (more or less) computed instead of searched. Generate-and-test methods are applicable for very general program classes since there are no principal difficulties in enumerating programs. They naturally facilitate usage of predefined functions (background knowledge) in induced programs. On the other side, generate-and-test methods are search intensive and therefore time consuming. Analytical approaches have more restricted program classes and generally do not facilitate the usage of background

[1] We adopt machine learning terminology here: The output program of an induction is called *hypothesis*, the function(s) to be induced are called *target function(s)*

knowledge. On the other side, analysis minimizes search and makes these approaches fast. The goal of the approach described in this paper was to relax the analytical approach by applying a search for hypotheses but by keeping analytical concepts within the search. The result is a functional program induction system that data-driven searches a comparatively less restricted hypothesis space and allows the use of background knowledge.

One classical and influential analytical approach is from Summers [5], who put inductive synthesis on a firm theoretical foundation. His system induces functional Lisp programs containing one function definition whose body consists of a conditional for an arbitrary number of base cases and one recursive case containing one recursive call. Parameters are restricted to be S-expressions (the general datatype in Lisp) and I/O-examples have to be linearly ordered. An interesting feature is a particular heuristic for automatically introducing an additional parameter if needed, e.g., the accumulator variable for list reversing. Analytical systems inspired by this approach are the BMWk algorithm [6,7] and the more recent system described in [8].

Another line of research is the field of inductive logic programming (ILP). Though ILP has a focus to non-recursive concept learning problems, there has also been research in inducing recursive logic programs on inductive datatypes (see [2]). One relatively recent analytical ILP method for inducing recursive logic programs is DIALOGS [9]. In order to induce more complex functions, e.g., the quicksort algorithm and to automatically invent needed subfunctions, e.g., the partitioning function for quicksort, it makes use of some general schemas, e.g., divide-and-conquer, which the user must chose and requires further information from the user. The schemas strongly restrict the hypothesis space. Moreover, DIALOGS is restricted to some predefined datatypes like lists and numbers.

The new approach described in this paper is a major extension of [10]. It induces multiple dependent target functions over arbitrary user defined algebraic datatypes in one step, facilitates the use of background knowledge and allows complex recursion patterns (nested calls of induced recursive functions, mutual recursion, tree recursion, arbitrary numbers of base- and recursive cases). Additionally needed subfunctions are introduced automatically if the calling relation fulfills some conditions. Its integrated analytical concepts lead to induction times which are very small compared to powerfull generate-and-test systems like ADATE. E.g., to induce the *Reverse*-function, our system needs milliseconds whereas ADATE needs more than a minute on the same computer. Particularly the capability of inducing multiple related target functions as for example mutual recursive definitions for *Even* and *Odd* on natural numbers is a feature not provided by most program induction systems. A recent ILP system also capable of learning multiple related recursively defined target concepts is ATRE [11].

2 Preliminaries

We represent functional programs as sets of equations (pairs of terms) over a many-sorted first order signature Σ. That is, we abstract from any concrete

functional programming language and do not consider higher order functions. Each equation specifying a function F has a left hand side (lhs) of the form $F(t_1, \ldots, t_n)$ where neither F nor the name of any other of the defined functions occur in the t_i. Thus, the symbols in the signature Σ are divided into two disjunct subsets $\mathcal{F}$ of *defined function symbols*, e.g., F, and $\mathcal{C}$ of *constructors*. Terms without defined function symbols are called constructor terms. Ground constructor terms denote values. The constructor terms t_i in the lhss of the equations for a defined function F may contain variables and are called *pattern*. This corresponds to the concept of pattern matching in functional programming languages and is the only form of case distinction. Each variable in the rhs of an equation must occur in the lhs, i.e., in the pattern. To evaluate a function defined by equations we read the equations as simplification rules from left to right. A set of simplification (or rewrite) rules is called *term rewriting system (TRS)*. TRSs whose lhss have defined function symbols as roots and constructor terms as arguments, i.e., whose lhss have the described pattern-matching form, are called *constructor term rewriting systems (CSs)*.

In order to formalize the simplification process we first introduce some standard concepts on terms: One can denote each subterm of a term t by its unique *position* within t, a sequence of positive integers. The term t itself stands at position ϵ—the empty sequence—called *root position*. If $t = f(t_1, \ldots, t_n)$ then each t_i stands at position i. If a subterm s of t_i stands at position u within t_i then it stands at position $i.u$ within t. The subterm at position u is written $t|_u$. Consider the term $t = f(a, g(x, y))$. Then, e.g., holds $t|_2 = g(x, y)$ and $t|_{2.1} = x$.

A *substitution* σ is a mapping from variables to terms and is extended to a mapping from terms to terms which is also denoted by σ and written in postfix notation; $t\sigma$ is the result of applying σ to all variables in term t. If $s = t\sigma$, then t is called *generalization* of s and we say that t *subsumes* s and that s *matches* t by σ. Given two terms t_1, t_2 and a substitution σ such that $t_1\sigma = t_2\sigma$, then we say that t_1, t_2 *unify* and call σ *unifier* of t_1 and t_2. We generalize the subsumption relation to sets of terms and say that a set of terms T subsumes another set of terms S if each term $s \in S$ is subsumed by a term $t \in T$. Given a set of terms, $S = \{s, s', s'', \ldots\}$, then there exists a term t which subsumes all terms in S and which is itself subsumed by each other term subsuming all terms in S. The term t is called *least general generalization (lgg)* of the terms in S [12].

A *context* is a term that contains a distinguished symbol $\Box$, denoting *wholes*, at at least one position. If C is a context containing n wholes then $C[t_1, \ldots, t_n]$ denotes the term resulting from replacing the n wholes in C by the t_i from left to right. The *rewrite relation* $\rightarrow_R$ established by a CS R is defined as follows: A term t rewrites to s according to R, written $t \rightarrow_R s$ iff there exists a rule $l \rightarrow r$ in R, a substitution σ, and a context C such that $t = C[l\sigma]$ and $s = C[r\sigma]$. Evaluating an n-ary function F for an input $i_1, \ldots, i_n$ consists of repeatedly rewriting the term $F(i_1, \ldots, i_n)$ w.r.t. the rewrite relation until the term is in *normal form*, i.e., cannot be further rewritten. A sequence of (in)finitely many rewrite steps $t_0 \rightarrow_R t_1 \rightarrow_R \cdots$ is called *derivation*. If a derivation starts with term t and results in a normal form s, then s is called normal form of t, written

$t \xrightarrow{!} s$. We say that t normalizes to s. In order to define a *function* on a domain (a set of ground terms) by a CS, no two derivations starting with the same ground term may lead to different normal forms, i.e., normal forms must be unique. A sufficient condition for this is that no two lhss of a CS unify; then the CS is *confluent*. A CS is *terminating* if each possible derivation terminates. A sufficient condition for termination is that the arguments/inputs of recursive calls strictly decrease within each derivation and w.r.t. a well founded order.

3 Analytical Induction of Recursive Functions

For better readability we write t^n for a sequence of terms $t_1, \ldots, t_n$.

If i^n is an input to a recursively defined function F with a corresponding output term o and $i^{n\prime}$ is the input to F resulting from a recursive call of F within computing i, then o contains the output term o' for $i^{n\prime}$ as subterm. Using this structural regularity between computations of recursively defined functions in order to infer the recursive definition from I/O-examples is the core of analytical function induction as proposed by Summers [5]. Examples for a target function F are equations of the form $F(i^n) = o$ where the i^n and o are ground constructor terms and are called example inputs and outputs respectively. A necessary condition for applying the described principle is that for each example input, all inputs resulting from recursive calls are also included in the example set. The following definition states this condition formally and extended to more than one target function.

Definition 1 (Recursively Subsumed Examples). *Let R be a CS which correctly computes a set of example equations. The example equations are called* recursively subsumed *w.r.t. R if for all example inputs i^n hold: Let $F(p^n) \to t$ be a rule in R such that i^n matches p^n by substitution σ. Then for each (recursive) call $F'(r^m)$ of a defined function F' of R in t the instantiation $r^m\sigma$ is contained as an example input in the example equations.*

4 Function Induction by Pattern Refinement, Matching, and Ubiquitous Subprogram Introduction

We require that induced CSs are terminating and that they represent functions, i.e., that they have unique normal forms. With regard to the given examples we require that a hypothesis is *correct*:

Definition 2. *A hypothesis, i.e., a CS R is* consistent/complete *w.r.t. a set of example equations iff for each example equation $F(i^n) = o$ holds*

consistent: $F(i^n) \xrightarrow{!}_R o$ *or* $F(i^n) \xrightarrow{!}_R s$ *for a* non-*constructor term s,*
complete: $F(i^n) \xrightarrow{!}_R s$ *for a constructor term s.*

A hypothesis is correct *iff it is both consistent and complete.*

The consistency condition assures that if the induced function is defined for an input then the corresponding function value is the specified output. The completeness condition assures that the induced function is total on the example inputs. We do not require the induced function to be total w.r.t. *all* Σ-constructor terms. Fig. 1 shows example equations for the *Reverse*-function and the equations induced by our system. Only the example equations and the corresponding datatype definitions were provided. Note that the example equations contain variables. Using variables where possible reduces the amount of needed example equations and makes the induction more time efficient. Two subfunctions have been introduced automatically, *Last* and *Init*, which compute the last element of a list and the list without the last element respectively. Note that automatically introduced subfunctions are simply named "Sub1", "Sub2" etc. by the system.

Example equations:

$$
\begin{aligned}
1.\quad & \mathit{Reverse}([]) &&= [] \\
2.\quad & \mathit{Reverse}([X]) &&= [X] \\
3.\quad & \mathit{Reverse}([X,Y]) &&= [Y,X] \\
4.\quad & \mathit{Reverse}([X,Y,Z]) &&= [Z,Y,X] \\
5.\quad & \mathit{Reverse}([X,Y,Z,V]) &&= [V,Z,Y,X]
\end{aligned}
$$

Induced CS:

$$
\begin{aligned}
&\mathit{Reverse}([]) && \rightarrow [] \\
&\mathit{Reverse}([X|Xs]) && \rightarrow [\mathit{Last}([X|Xs])|\mathit{Reverse}(\mathit{Init}([X|Xs]))] \\
&\mathit{Last}([X]) && \rightarrow X \\
&\mathit{Last}([X_1,X_2|Xs]) && \rightarrow \mathit{Last}([X_2|Xs]) \\
&\mathit{Init}([X]) && \rightarrow [] \\
&\mathit{Init}([X_1,X_2|Xs]) && \rightarrow [X_1|\mathit{Init}([X_2|Xs])]
\end{aligned}
$$

Fig. 1. Example equations and the induced solution for the *Reverse*-function

The induction of a terminating, confluent, correct CS is organized as a kind of best first search. During search, a hypothesis is a set of equations entailing the example equations and constituting a terminating and confluent CS *but potentially with variables in the rhss not occurring in the lhss*. That is, the equations of a hypothesis during search do not necessarily represent *functions*. We call such equations and hypotheses containing them *unfinished* equations and hypotheses. A goal state is reached, if at least one of the best—according to a criterion explained below—hypotheses is finished, i.e., does not contain unfinished equations. Such a finished hypothesis is terminating and confluent by construction and since its equations entail the example equations, it is also correct.

Our induction bias is to prefer CSs whose patterns partition the example inputs into fewer subsets. This corresponds to preferring programs with fewer case distinctions. This leads, in some sense, to a most general hypothesis. Regarding

one defined function, this bias prefers a CS with fewer rules, since the pattern of each rule determines one unique subset. But consider the solution for the *Reverse*-function with the subfunctions *Last* and *Init* as shown in Fig. 1. The solution contains six rules but the number of induced example input subsets is only three, because *Last* and *Init* induce the same subsets (pattern $[X]$ subsumes the second example, pattern $[X_1, X_2|Xs]$ examples 3 - 5) which are again partitions of the subset induced by pattern $[X|Xs]$ of *Reverse* such that pattern [] from *Reverse* remains and induces the subset containing the first example input. So if we would have chosen fewer rules as preference bias then obviously the five example equations themselves would have been favored over the solution with *Init* and *Last* such that no generalization would have taken place.

With respect to the described bias and in order to get a complete hypothesis w.r.t. the examples, the initial hypothesis is a CS with one rule per target function such that its pattern subsumes all example inputs. In most cases (e.g., for all recursive functions) one rule is not enough and the rhss will remain unfinished. Then for one of the unfinished rules successors will be computed which leads to one or more (unfinished) hypotheses. Now repeatedly unfinished rules of currently best hypotheses are replaced until a currently best hypothesis is finished. Since one and the same rule may be member of different hypotheses, the successor rules originate successors of *all* hypotheses containing this rule. Hence, in each induction step several hypotheses are processed.

4.1 Initial Rules

Given a set of example equations for one target function, the initial rule is constructed by first antiunifying [12] all example inputs.[2] This leads to the lgg of the example inputs, i.e., to the most specific pattern subsuming all example inputs. Second, the example outputs are antiunified w.r.t. the substitutions resulting from antiunification of the inputs. This gives the lgg of all outputs were variables from the pattern are used if possible. Considering only lggs of example inputs as patterns narrows the search space. It does not constrain completeness of hypotheses regarding the example equations as shown by the following lemma.

Lemma 1. *Let R be a CS with non-unifying patterns and which is correct regarding a set of recursively subsumed example equations. Then there exists a CS R' such that R' contains exactly one pattern p' for each pattern p in R, each p' is the lgg of all example inputs matching the corresponding pattern p, and R and R' compute the same normal form for each example input.*

Proof. It suffices to show (i) that if pattern p of a rule r is replaced by the lgg of the example inputs matching p then also the rhs of r can be replaced by a new rhs such that the rewrite relation remains the same for the example inputs matching p, and (ii) that if the rhs of r contains a call to a defined function

[2] Note that for functions with arity > 1 inputs are sequences i^n, $n > 1$, of terms. We may consider such a sequence as *one* term by assuming a distinguished constructor symbol as root and the i^n as direct subterms.

then each instance of this call regarding the example inputs matching p is also an example input (matched by p or another pattern and, hence, also matched by patterns constituting lggs of example inputs). Proving these two conditions suffices because (i) assures equal rewrite relations for the example inputs and (ii) assures that each resulting term of one rewrite step which is not in normal form regarding R is also not in normal form regarding R'.

The second condition is assured if the example inputs are recursively subsumed. To show the first condition let p be a pattern in R which is *not* the lgg of the input examples matching it. Then there exists a position u with $p|_u = x, x \in Var(p)$ and $p'|_u = s \neq x$ if p' is the lgg of the input examples matching p. First assume that x does not occur in the rhs of the rule r with pattern p, then replacing x by s in p does not change the rewrite relation of r for the example inputs because still all example inputs are subsumed and are rewritten to the same term as before. Now assume that x occurs in the rhs of r. Then the rewrite relation of r for the input examples remains the same if x is replaced by s in p as well as in the rhs. □

4.2 Processing Unfinished Rules

This section describes the three methods for replacing unfinished rules by successor rules. All three methods are applied to a chosen unfinished rule. The first method, *splitting rules by pattern refinement*, replaces an unfinished rule with pattern p^n by at least two new rules with more specific patterns in order to establish a case distinction on the example inputs. The second method, *introducing function calls*, implements the principle described in Sec. 3 in order to introduce recursive calls or calls to other defined functions. Other defined functions can be further target functions or background knowledge functions. The third method, *introducing subfunctions*, generates new induction problems, i.e., new example equations, for the unfinished subterms of an unfinished rhs. These new problems are treated the same way as the "original" problems, i.e., this method implements the capability to automatically find auxiliary subfunctions.

Splitting Rules by Pattern Refinement The first method for generating successors of a rule is to replace its pattern p^n by a set of more specific patterns, such that the new patterns induce a partition of the example inputs matching p^n. This results in a *set* of new rules replacing the original rule and—from a programming point of view—establishes a case distinction.

Suppose a rule with pattern p^n which is the lgg of the example inputs matching it. Then the examples whose inputs match p^n have to be partitioned into a minimum number of at least two subsets and p^n has to be replaced by the lggs of the inputs of the respective subsets. It has to be assured that no two of the new lggs unify. This is done as follows: First a position u is chosen at which a variable stands in p^n. Since p^n is the lgg of all inputs matching it it holds that at least two inputs have different constructor symbols at position u. Then respectively all example inputs with the *same* constructor at position u are taken into the

same subset. This leads to a partition of the example inputs. Finally, for each subset the lgg is computed. The new lggs do not unify, since they have different constructors at at least one position.

Possibly different positions of variables in pattern p^n lead to different partitions. Then all partitions and the corresponding sets of specialized patterns are generated. Each new pattern determines the lhs of a new rule. The corresponding initial rhss are computed as lggs of the respective outputs as described in Sect. 4.1. Since the refined patterns subsume fewer examples, the number of variables in the initial rhss which are not contained in the corresponding lhs (non-strictly) decreases with each refinement step. Eventually, if no correct hypothesis with fewer partitions exists, each example input is subsumed by itself such that the example equations are simply reproduced.

For example, let

$$1.\ \mathit{Reverse}([]) = [], \quad 2.\ \mathit{Reverse}([a]) = [a], \quad 3.\ \mathit{Reverse}([b]) = [b],$$
$$4.\ \mathit{Reverse}([a,b]) = [b,a], \quad 5.\ \mathit{Reverse}([b,a]) = [a,b]$$

be some examples for the *Reverse*-function. The pattern of the initial rule is simply a variable X, since the example input terms have no common root symbol. Hence, the unique position at which the pattern contains a variable and the example inputs different constructors is the root position. The first example input consists of only the constant [] at the root position. All remaining example inputs have the constructor *cons* as root. I.e., two subsets are induced by the root position, one containing the first example, the other containing all remaining examples. The lggs of the example inputs of these two subsets are [] and $[X|Xs]$ respectively which are the patterns of the two successor rules.

Introducing Function Calls The second method to generate successor sets for an unfinished rule with pattern p^n for a target function F is to replace its rhs by a call to a defined function F', i.e. by a term $F'(R_1(p^n), \ldots, R_m(p^n))$. Each R_i denotes a new introduced defined (sub)function. This finishes the rule, since now the rhs does not longer contain variables not contained in the lhs. In order to get a rule leading to a *correct* hypothesis, for each example equation $F(i^n) = o$ of function F whose input i^n matches p^n with substitution σ must hold: $F'(R_1(p^n), \ldots, R_m(p^n))\sigma \xrightarrow{!} o$. This holds if for each output o an example equation $F'(i'_1, \ldots, i'_m) = o$ of function F' exists such that $R_i(p^n)\sigma \xrightarrow{!} i'_i$ for each R_i and i'_i. Thus, if we find example equations of F' with outputs o, then we abduce example equations $R_i(i^n) = i'_i$ for the new subfunctions R_i and induce them from these examples. Provided, the final hypothesis is correct for F' and all R_i then it is also correct for F.

In order to assure termination of the final hypothesis it must hold $i^{m\prime} < i^n$ according to any reduction order $<$ if the function call is recursive.[3]

[3] Assuring decreasing arguments is more complex if mutual recursion is allowed.

Introducing Subfunctions The last method to generate successor equations can be applied, if all outputs o of the inputs matching the pattern of the considered unfinished rule have the same constructor c as roots. Let c be of arity m then the rhs of the rule is replaced by the term $c(Sub_1(p^n), \ldots, Sub_m(p^n))$ where each Sub_i denotes a new introduced defined (sub)function. This finishes the rule since all variables from the new rhs are contained in the lhs. The examples for the new subfunctions are abduced from the examples of the current function as follows: If $o|_i$ are the ith subterms of the outputs o, then the equations $Sub_i(i^n) = o|_i$ are the example equations of the new subfunction Sub_i. Thus, correct rules for Sub_i compute the ith subterm of the outputs o such that the term $c(Sub_1(p^n), \ldots, Sub_m(p^n))$ normalizes to the outputs o.

A Remark on the Described Successor Functions As described in Sect. 4.2, (recursive) calls to defined functions specified by examples are only introduced at the root of a rhs (since such calls are introduced by replacing an unfinished rhs). Of course, generally, function calls can occur at any position in a rhs, compare for example the recursive definition for *Init* in Fig. 1. The reason why e.g. *Init* can be induced by our approach though function calls are only introduced at root positions is that deeper positions are (indirectly) considered as consequence of subprogram introduction as described in Sect. 4.2. Rhs root positions of such subprograms correspond to deeper positions of the rhs of the rule calling these subprograms.

5 Experimental Results

We have implemented a prototype of the described algorithm in the programming language Maude [13]. Maude is a reflective language which is based on equational and rewriting logic. Reflection means that Maude programs can deal with Maude programs as data. The implementation includes two extensions compared to the approach described in the previous section: First, example equations may contain variables such that the amount of example equations needed to specify a target function decreases. This is advantageous for the specifier as well as it leads to smaller induction times. Second, different variables within a pattern can be tested for equality This establishes—besides pattern refinement—a second form of case distinction.

In Tab. 1 we have listed experimental results for sample problems. The first column lists the names for the induced target functions, the second the names of additionally specified background functions, the third the number of given examples (for target functions), the fourth the number of automatically introduced recursive subfunctions, the fifth the maximal number of calls to defined functions within one rule, and the sixth the times in seconds consumed by the induction. Note that the example equations contain variables if possible (except for the *Add*-function); compare Fig. 1. The experiments were performed on a Pentium 4 with Linux and the Maude 2.3 interpreter.

target functions	bk funs	#expl	#subfuns	#funcalls	times
Length	/	3	0	1	.012
Last	/	3	0	1	.012
Odd	/	4	0	1	.012
ShiftL	/	4	0	1	.024
Reverse	*Snoc*	4	0	2	.024
Even, *Odd*	/	3, 3	0	1	.028
Mirror	/	4	0	2	.036
Take	/	6	0	1	.076
ShiftR	/	4	2	2	.092
DelZeros	/	7	0	1	.160
Insertion Sort	*Insert*	5	0	2	.160
PlayTennis	/	14	0	0	.260
Add	/	9	0	1	.264
Member	/	13	0	1	.523
Reverse	/	4	2	3	.790
Quick Sort	*Append*,P_1,P_2	6	0	5	63.271

Table 1. Some inferred functions

All induced programs compute the intended functions with more or less "natural" definitions. *Length*, *Last*, *Reverse*, and *Member* are the well known functions on lists. *Reverse* has been specified twice, first with *Snoc* as background knowledge which inserts an element at the end of a list and second without background knowledge. The second case (see Fig. 1 for given data and computed solution), is an example for the capability of automatic subfunction introduction and nested calls of defined functions. *Odd* is a predicate and true for odd natural numbers, false otherwise. The solution contains two base cases (one for 0, one for 1) and in the recursive case, the number is reduced by 2. In the case where both *Even* and *Odd* are specified as target functions, both functions of the solution contain one base case for 0 and a call to the other function reduced by 1 as the recursive case. I.e. the solution contains a mutual recursion. *ShiftL* shifts a list one position to the left and the first element becomes the last element of the result list, *ShiftR* does the opposite, i.e., shifts a list to the right such that the last element becomes the first one. The induced solution for *ShiftL* contains only the *ShiftL*-function itself and simply "bubbles" the first element position by position through the list, whereas the solution for *ShiftR* contains two automatically introduced subfunctions, namely again *Last* and *Init*, and conses the last element to the input list without the last element. *Mirror* mirrors a binary tree. *Take* keeps the first n elements of a list and "deletes" the remaining elements. This is an example for a function with two parameters where both parameters are reduced within the recursive call. *DelZeros* deletes all zeros from a list of natural numbers. The solution contains two recursive equations. One for the case that the first element is a zero, the second one for all other cases. *Insertion Sort* and *Quick Sort* respectively are the well known sort algorithms. The five respectively six well chosen examples as well as the additional exam-

ples to specify the background functions are the absolute minimum to generate correct solutions. The solution for *Insertion Sort* has been generated within a time that is not (much) higher as for the other problems, but when we gave a few more examples, the time to generate a solution explodes. The reason is, that all outputs of lists of the same length are equal such that many possibilities of matching the outputs in order to find recursive calls exist. The number of possible matches increases exponentially with more examples. The comparatively very high induction time for *Quick Sort* results from the many examples needed to specify the background functions and from the complex calling relation between the target function and the background functions. P_1 and P_2 are the functions computing the lists of smaller numbers and greater numbers respectively compared to the first element in the input list. For Add we have a similar problem. First of all, we have specified Add by ground equations such that more examples were needed as for a non-ground specification. Also for Add holds, that there are systematically equal outputs, since, e.g., $Add(2,2), Add(1,3)$ etc. are equal and due to commutativity. Finally, *PlayTennis* is an attribute vector concept learning example from Mitchell's machine learning text book [14]. The 14 training instances consist of four attributes. The five non-recursive rules learned by our approach are equivalent with the decision tree learned by ID3 which is shown on page 53 in the book. This is an example for the fact, that learning decision trees is a subproblem of inducing functional programs.

6 Conclusions and Further Research

We described a method to induce functional programs represented as confluent and terminating constructor systems. The presented methodology is inspired by classical and recent analytical approaches to the fast induction of functional programs. One goal was to overcome the drawback that "pure" analytical approaches does not facilitate the use of background knowledge and generally have relatively restricted hypothesis languages and on the other side to keep the analytical approach as far as possible in order to be able to induce more complex functions in a reasonable amount of time. This has been done by applying a search in a more comprehensive hypothesis space but where the successor functions are data-driven and not generate-and-test based, such that the number of successors is more restricted and the hypothesis spaced is searched in a controlled manner. Though the successor functions are data-driven, the search is complete and only favors hypotheses inducing fewer partitions but applies no further heuristics to estimate, how many partitions the final hypothesis will have. Developing such heuristics will be one of the further research topics.

References

1. Biermann, A.W., Guiho, G., Kodratoff, Y., eds.: Automatic Program Construction Techniques. Collier Macmillan (1984)

2. Flener, P., Yilmaz, S.: Inductive synthesis of recursive logic programs: Achievements and prospects. Journal of Logic Programming **41**(2–3) (1999) 141–195
3. Kitzelmann, E., Olsson, R., Schmid, U., eds.: Proceedings of the ICML 2005 Workshop Approaches and Applications of Inductive Programming. (2005)
4. Olsson, R.: Inductive functional programming using incremental program transformation. Artificial Intelligence **74**(1) (1995) 55–83
5. Summers, P.D.: A methodology for LISP program construction from examples. Journal ACM **24**(1) (1977) 162–175
6. Kodratoff, Y., J.Fargues: A sane algorithm for the synthesis of LISP functions from example problems: The Boyer and Moore algorithm. In: Proc. AISE Meeting Hambourg. (1978) 169–175
7. Jouannaud, J.P., Kodratoff, Y.: Characterization of a class of functions synthesized from examples by a summers like method using a 'B.M.W.' matching technique. In: Proc. International Joint Conference on Artificial Intelligence (IJCAI-79), Morgan Kaufmann (1979) 440–447
8. Kitzelmann, E., Schmid, U.: Inductive synthesis of functional programs: An explanation based generalization approach. Journal of Machine Learning Research **7** (2006) 429–454 Special topic on Approaches and Applications of Inductive Programming.
9. Flener, P.: Inductive logic program synthesis with DIALOGS. In Muggleton, S., ed.: Proceedings of ILP'96, Springer (1997) 175–198
10. Kitzelmann, E., Schmid, U.: Inducing constructor systems from example-terms by detecting syntactical regularities. Electronical Notes in Theoretical Computer Science **174**(1) (2007) 49–63
11. Malerba, D.: Learning recursive theories in the normal ILP setting. Fundamenta Informaticae **57**(1) (2003) 39–77
12. Plotkin, G.D.: A note on inductive generalization. In: Machine Intelligence. Volume 5. Edinburgh University Press (1969) 153–163
13. Clavel, M., Durán, F., Eker, S., Lincoln, P., Martí-Oliet, N., Meseguer, J., Talcott, C.: The maude 2.0 system. In Nieuwenhuis, R., ed.: Rewriting Techniques and Applications (RTA 2003). Number 2706 in Lecture Notes in Computer Science, Springer-Verlag (June 2003) 76–87
14. Mitchell, T.M.: Machine Learning. McGraw-Hill Higher Education (1997)