Available online at www.sciencedirect.com
ScienceDirect
ELSEVIER
Electronic Notes in Theoretical Computer Science 174 (2007) 49–63
Electronic Notes in Theoretical Computer Science
www.elsevier.com/locate/entcs

Inducing Constructor Systems from Example-Terms by Detecting Syntactical Regularities

Emanuel Kitzelmann[a,1,3] Ute Schmid[a,2,3]

[a] *Department of Information Systems and Applied Computer Science*
University of Bamberg
96045 Bamberg, Germany

Abstract

We present a technique for inducing functional programs from few, well chosen input/output-examples (I/O-examples). Potential applications for automatic program or algorithm induction are to enable end users to create their own simple programs, to assist professional programmers, or to automatically invent completely new and efficient algorithms. In our approach, functional programs are represented as constructor term rewriting systems (CSs) containing recursive rules. I/O-examples for a target function to be implemented are a set of pairs of terms $(F(i_i), o_i)$ meaning that $F(i_i)$—denoting application of function F to input i_i—is rewritten to o_i by a CS implementing the function F. Induction is based on detecting syntactic regularities between example terms. In this paper we present theoretical results and describe an algorithm for inducing CSs over arbitrary signatures/data types which consist of one function defined by an arbitrary number of rules with an arbitrary number of non-nested recursive calls in each rule. Moreover, we present empirical results based on a prototypical implementation.

Keywords: inductive program synthesis, rule-based programming, functional programming, constructor systems

1 Introduction

Automatic induction of recursive declarative programs from input/output-examples (I/O-examples) is an active area of research since the sixties (see [1] for classical methods, [3] for systems in the field of inductive logic programming, and [5] 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 [10]), programs of

[1] Email: emanuel.kitzelmann@wiai.uni-bamberg.de
[2] Email: ute.schmid@wiai.uni-bamberg.de
[3] We would like to thank the anonymous reviewers whose comments helped to improve the paper.

doi:10.1016/j.entcs.2006.11.015

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. 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. Moreover, the specification in terms of examples of the target function to be implemented can be extremely declarative, e.g., can consist of example inputs together with an evaluation function. On the other side, generate-and-test methods are search intensive and therefore time consuming. These characteristics qualify them for invention of new and efficient algorithms (cp. [10]). Analytical approaches have more restricted program classes since deriving programs by analyzing examples is more complicated than enumerating programs. Facilitating the usage of background knowledge is more complicated for the same reason. Moreover, analysis of examples is not possible for example specifications based on evaluation functions but requires input/output-examples (I/O-examples). On the other side, analysis minimizes search and makes these approaches fast. These characteristics qualify analytical approaches for end-user programming (see [12] for an example) or assisting systems. Since the technique presented in this paper is analytical, we focus to analytical approaches in the following.

One classical and influential analytical approach was developed by Summers [13], who put inductive synthesis on a firm theoretical foundation. Summers' system induces functional Lisp programs. It proceeds in two steps: In a first step, traces and predicates for distinguishing the inputs are calculated for each I/O-example. By integrating traces and predicates into a conditional expression a non-recursive program computing all I/O-examples is constructed as result of the first synthesis step. In a second step, regularities are searched for between the traces and predicates respectively. Found regularities are then inductively generalized and expressed in form of the resulting recursive program. Summers' method is able to induce one function definition whose body consists of a conditional for an arbitrary number of base cases and exactly one recursive case containing exactly one recursive call. Parameters are restricted to the data type S-expression (the general data type in Lisp) and the 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. An extension of Summers' method which relaxes the very simple program schema is the BMWk algorithm [7,4]. It is also restricted to parameters of type S-expression and to linear recursion. A reformulation and generalization of the BMWk algorithm in the framework of term rewriting systems has been described by LeBlanc in [8]. This generalization overcome the restriction to S-expressions as only usable data type. Moreover, it overcome the restriction to linear recursion but requires that the user provides the recursion scheme of a program to be induced, i.e., the patterns of the recursive rules and the arguments for the recursive calls. LeBlanc's generalization has not been implemented. A recent approach of functional program induction inspired by these classical ap-

proaches and also formulated within the term rewriting framework is described in [6]. Extensions to the previous methods are that subfunctions/subprograms and additionally needed parameters are inferred automatically and in a systematic way. All these functional analytical two-step methods are restricted to small fixed sets of primitive functions (basic constructors and selector functions for the respective data types and a predicate for testing whether a constructor is atomic, e.g., the empty list) of which the induced programs can be composed. That is, they cannot handle additionally provided problem-specific functions as background knowledge to induce more complex programs. Particularly, testing for atomic expressions as the only used predicate restricts the class of inducable programs to programs for *structural problems*. Reversing a list falls into this class for example, but not sorting a list, since for sorting a list a predicate for comparing two elements regarding an order is needed. Moreover, the described systems have in common that the provided I/O-examples may not arbitrarily be chosen from the graph of the target function but have to be the first k examples regarding the inductive structure of the underlying data type.

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 data types in the field of ILP (see [3]). Most of the specialized systems are analytical approaches. In contrast to the classical functional approaches, some of them can handle background knowledge; particularly, some of them are not restricted to structural problems. Two relatively recent ILP methods for induction of recursive logic programs are DIALOGS-II [14] and an (unimplemented) method described in [11] by Rao. Both methods allow arbitrary numbers of base cases and recursive cases for one relation and more than one recursive call in one body. Moreover, usage of background knowledge is facilitated by both systems. Both methods can infer additional predicates and parameters, yet this ability is restricted to particular schemas "hard-wired" in the induction system in case of DIALOGS-II. Moreover, the user has to choose the schema which shall be utilized. DIALOGS-II is restricted to some predefined data types like lists and natural numbers. Rao's method is restricted to decompose the inputs respecting their recursive structure, e.g., a list can only be decomposed into the first k elements and the rest list. In contrast, DIALOGS-II can, for example, decompose a list into two lists of equal length. The user has to choose from a set of predefined decompositions. Both methods are restricted to only one recursion parameter, i.e., one parameter have to decrease in each recursive call and is checked in the base cases.

The analytical and functional approach described in this paper represents I/O-examples as well as induced programs as constructor term rewriting systems (CSs) as a generic model of functional programs, instead of using a particular programming language. Compared to DIALOGS-II it is more general in that induced programs are not restricted to particular predefined data types, e.g., to lists or numbers, but I/O-examples can be defined over arbitrary finite signatures which are then adopted for the induced programs. Decompositions of inputs and argument terms for recursive

calls are inferred automatically, but with Rao's method it shares the restriction to decompositions respecting the internal inductive structure of the respective data type. It is more general than both other methods in that the number of recursion parameters is not restricted to one. In contrast to DIALOGS-II and Rao's method our approach is in the current state not able to use background knowledge nor is it able to induce additional parameters (which are not given in the I/O-examples) or subfunctions.

2 Preliminaries

We give an introduction to term rewriting systems and define some data types in this section.

2.1 Terms and Positions

The set of (finite) terms over a finite signature Σ and a countably infinite set of variables V disjoint from the function symbols in Σ is denoted by $T(\Sigma, V)$. Terms without variables are called *ground* terms. A term t is called *linear* iff each variable occurs only once in t. A term can be viewed as a finite, labelled, ordered tree as follows: (i) Each variable or constant corresponds to a tree consisting of only one node labelled by the variable or constant respectively and (ii) the term $f(t_1, \ldots, t_n)$ corresponds to the tree with the root node labelled by f and the trees corresponding to $t_1, \ldots, t_n$ as immediate subtrees in order from left to right. A *position* within a term t is a sequence of positive integers indicating a path from the root of the tree of t to one of its nodes. The position of the root node is the empty sequence, denoted by ϵ. The subtree/subterm at position u is written $t|_u$. If we want to state that a term t contains a subterm s at any position, we write $t = C[s]$. C is called *context*. For two terms t, s and a position u of t, $t[s]_u$ denotes the result of replacing the subterm of t at position u by the term s.

A *substitution* is a mapping from variables to terms, $\sigma : V \to T(\Sigma, V)$. A substitution $\sigma : V \to T(\Sigma, V)$ is extended to a mapping from $T(\Sigma, V)$ to $T(\Sigma, V)$ which is also denoted by σ and written in postfix notation; $t\sigma$ is the result of applying σ to all variables in t. A substitution which maps variables to variables only is called *variable renaming*. If $s = t\sigma$, then s is called an *instance* of t. We say that t *subsumes* s or that t is a *generalization* of s. We also say that s *matches* t by σ. Given two terms s_1, s_2 and a substitution σ such that $s_1\sigma = s_2\sigma$, then we say that s_1, s_2 *unify* and call σ *unifier* of s_1 and s_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 linear term t which subsumes all terms in S and which is itself subsumed by each other linear term subsuming all terms in S. The term t is called *least general linear generalization (lglg)* (of the terms in S).

A *reduction order* on $T(\Sigma, V)$ is a well-founded order $\prec_{ro}$ on $T(\Sigma, V)$ that is (i) closed under substitutions, i.e., if $t \prec_{ro} s$ and σ is an arbitrary substitution then $t\sigma \prec_{ro} s\sigma$, and (ii) closed under contexts, i.e., if $t \prec_{ro} s$ and C is an arbitrary

context then $C[t] \prec_{ro} C[s]$.

2.2 (Constructor) Term Rewriting Systems

A *term rewriting system (TRS)* is a pair (Σ, R) where R is a finite set of *rewrite rules* (or rules for short) $l \to r$ where $l, r \in T(\Sigma, V)$, $l \notin V$, and $Var(r) \subseteq Var(l)$. In the following we write only R for a TRS (Σ, R). For a rule $l \to r$, l is called *left-hand side (lhs)* and r is called *right-hand side (rhs)* of the rule. A *constructor system (CS)* is a TRS in which Σ can be partitioned into a set $\mathcal{F}$ of *defined function symbols* and a set $\mathcal{C}$ of *constructors*, such that the lhs of every rewrite rule has the form $F(t_1, \ldots, t_n)$ with $F \in \mathcal{F}$ and $t_1, \ldots, t_n \in T(\mathcal{C}, V)$. The *rewrite relation* $\to_R$ is defined as follows: A term t rewrites to s according to R, written $t \to_R s$ iff there exists a rule $l \to r$ in R, a substitution σ, and a context C such that $t = C[l\sigma]$ and $s = C[r\sigma]$. We call $l\sigma$ *redex* and $r\sigma$ *contractum*. The reflexive-transitive closure of the rewrite relation $\to_R$ is denoted by $\overset{*}{\to}_R$. If a term cannot be rewritten, then it is in *normal form*. A sequence of finitely or infinitely many rewrite steps $t_0 \to_R t_1 \to_r \cdots$ is called *derivation*. We call s *reduct* of t iff $t \overset{*}{\to}_R s$. If additionally s is in normal form, then s is called normal form of t, written $t \overset{!}{\to} s$. We say that t normalizes to s. A TRS is called *terminating* iff there is no infinite derivation, i.e., if each derivation leads to a normal form after finitely many rewrite steps. A TRS is called *confluent* iff every two reducts of one term have a common reduct. If a TRS is confluent, then every term has at most one normal form. A TRS is *left-linear* iff all its lhss are linear. A sufficient condition for confluence of a CS is that it is left-linear and no two of its lhss unify. A confluent CS constitutes a functional program: defined function symbols denote defined functions of the program. Constructors denote predefined functions which are used to compose the program. Consider a "program call" $F(t_1, \ldots, t_n)$ which normalizes to s. We call $t_1, \ldots, t_n$ *input* and s *output* of $F(t_1, \ldots, t_n)$ iff $t_1, \ldots, t_n \in T(\mathcal{C}, V)$ and $s \in T(\mathcal{C}, V)$ respectively.

2.3 Data-Type Definitions

The examples in this paper use the data types natural numbers, list, and binary tree. A natural number is either the constant zero, 0, or a term $s(x)$ denoting the successor of the natural number x. A list is either the empty list [] or a term $cons(x, xs)$ with x the first element and xs the rest list. We denote the list $cons(x, xs)$ by $[x|xs]$ and a list $cons(x_1, cons(x_2, \ldots, cons(x_m, xs) \cdots))$ by $[x_1, x_2, \ldots, x_m|xs]$. If $xs = []$ we write $[x_1, x_2, \ldots, x_m]$. A binary tree is either a value x expressed by the term $val(x)$ or a term $tree(l, val(x), r)$ (written $\langle l, val(x), r\rangle$) with l, r the left and right subtree respectively and $val(x)$ the value of the root node.

3 The Considered Class of Constructor Systems

We define the subclass of CSs which can be induced by our algorithm in terms of a schema.

3.1 Flat One-Function CSs

The class of constructor systems (CSs) which can be induced is characterized as follows: Each rule has the form

$$F(p_1, \ldots, p_n) \to t$$

for a fixed defined function symbol F, i.e., $\mathcal{F} = \{F\}$. Since t may contain the defined function symbol, rules can be recursive. For a recursive rule

$$F(p_1, \ldots, p_n) \to C[F(r_1, \ldots, r_n)]$$

holds (i) that C may contain (further) occurrences of the defined function symbol F, i.e., that a rule may contain an arbitrary number of recursive calls, and (ii) that $r_1, \ldots, r_n \in T(\mathcal{C}, V)$, i.e., that recursive calls are non-nested. We call the class of CSs matching this basic schema *flat one-function CSs*.

We call $p_1, \ldots, p_n$ in the lhs of a rule *pattern*, $F(r_1, \ldots, r_n)$ in the rhs *recursive call*, and $r_1, \ldots, r_n$ *recursion terms*. We denote a flat one-function CS by its unique defined function symbol F. We require that (i) $F(r)$ is smaller than $F(p)$ for any recursive call regarding a fixed reduction order $\prec_{ro}$, (ii) all rules are left-linear, and (iii) no two lhss unify. These conditions guarantee termination and confluence.

Remark 3.1 Note, that we do *not* require the set of patterns in a flat one-function CS to be complete (or exhaustive) in the sense, that all terms in $T(\mathcal{C}, V)$ are matched by at least one pattern. Thus, there might be normal forms for particular inputs which contain the defined function symbol. Completeness of induced flat one-function CSs depends on the given I/O-examples.

3.2 Examples for Flat One-Function CSs

The class of flat one-function CSs contains several standard functions for lists, e.g., *Head*, *Tail*, *Append*, *Length*, *Last* (returns the last element), *Init* (returns the given list without the last element), *Take* and *Drop* (keeping only the first n elements of a list and dropping the first n elements from a list respectively), *Zip* (takes two lists and returns a list of pairs of corresponding elements), *Sum* (takes a list of natural numbers and returns the sum of all contained numbers), and *Reverse* (reversing a list by using an accumulator variable). Examples of flat one-function CSs for functions on natural numbers are *Add*, *Sub*, and several predicates, e.g., *Odd*, *Even*, $=$, $\leq$. A particular subclass of flat one-function CSs are (non-recursive) *classifiers* on instance spaces defined by attribute vectors which are classically learned in the field of machine learning. We do not consider such classifiers in this paper since we are interested in inducing recursive programs on recursive data types. Figure 1 shows two further examples for the class of flat one-function CSs. *DelZeros* deletes all zeros from a list and *TreeRev* reverses a binary tree. *TreeRev* is an example for a tree-recursive flat one-function CS. In Section 5 we evaluate our induction algorithm empirically for some of the the mentioned examples and some additional examples.

$$
\begin{aligned}
DelZeros([]) &\rightarrow [] \\
DelZeros([0|xs]) &\rightarrow DelZeros(xs) \\
DelZeros([s(x)|xs]) &\rightarrow [s(x)|DelZeros(xs)]
\end{aligned}
$$

$$
\begin{aligned}
TreeRev(val(x)) &\rightarrow val(x) \\
TreeRev(\langle l, val(x), r\rangle) &\rightarrow \langle TreeRev(r), val(x), TreeRev(l)\rangle
\end{aligned}
$$

Fig. 1. Two example CSs, *DelZeros* and *TreeRev*

Examples for CSs *not* contained in the class of flat one-function CSs are *Mult*, *Member* (a predicate which checks whether a particular element is contained in a list), or sorting lists, because each of these functions needs subfunctions to be implemented, i.e., the respective CSs contain rules for more than one defined function symbol. E.g., a CS for *Mult* consists of rules for *Mult* and rules for *Add*, or a CS implementing quicksort consists of rules for the main function as well as rules for the subfunctions *Partition*, *Append*, and $\leq$. Of course one can consider such subfunctions as predefined and define their symbols to be constructors such that the CSs for the respective main function consist of only the main function. Then these CSs also fall in the class of flat one-function CSs but they cannot be induced from I/O-examples containing only the basic constructors introduced in Section 2.3 by the method presented here because our method cannot deal with background knowledge until yet. These CSs can only be induced if the provided example outputs are already composed of the constructors denoting the subfunctions. E.g., *Mult* could be induced from I/O-examples like $(Mult(s^2(0), s^3(0)),\ Add(Add(0, s^3(0)), s^3(0)))$ but not from I/O-examples like $(Mult(s^2(0), s^3(0)),\ s^6(0))$.

3.3 Regularities between Computations

Inputs, patterns, and recursion terms each are *lists* or *vectors* respectively of terms. For better readability, we denote a vector of terms $t_1, \ldots, t_n$ by t^n, i.e., we denote inputs, patterns, and recursion terms by i^n, p^n, and r^n, respectively. For a list of terms $t^n = t_1, \ldots, t_n$ and a substitution σ, we denote the instantiated list $t_1\sigma, \ldots, t_n\sigma$ by $t^n\sigma$. We transfer the terms *subsumes*, *matches*, *unifies*, and *least general linear generalization (lglg)* to lists of terms analogously. E.g., p^n subsumes i^n iff there exists a substitution σ such that $i^n = p^n\sigma$ (i^n matches p^n by σ). Or, for example, p^n is an lglg of a set of inputs $\{i^n, i'^n, i''^n, \ldots\}$ iff p_1 is an lglg of $\{i_1, i'_1, i''_1, \ldots\}$, p_2 is an lglg (containing other variables) of $\{i_2, i'_2, i''_2, \ldots\}$ and so on.

In tradition of Summers [13], we use relations which hold between recursively defined functions and computations processed by such functions to inductively infer the recursive definition from given computations which are assumed to be computations of a recursive target function. If $F(i^n)$ matches a recursive rule with lhs $F(p^n)$ by σ, i.e., $F(i^n) = F(p^n)\sigma$ then we call corresponding instances of the recursive

calls, $F(r^n)\sigma$, in the instantiated rhs $t\sigma$ of that rule *recursive calls of* $F(i^n)$. The subterm lists $r^n\sigma = i'^n$ of $t\sigma$ are again inputs to F and normalize to their outputs o'. The following theorem states that for an input i^n and the corresponding output o, the outputs o' of the recursive calls $F(i'^n)$ of $F(i^n)$ occur as subterms in o at the positions of the respective recursive calls. Essentially, induction of recursive rules in our approach is done by reverting the theorem:

Theorem 3.2 *Let F be a flat one-function CS, i^n an input, and o the corresponding output, i.e., $i_1, \ldots, i_n, o \in T(\mathcal{C}, V)$ and $F(i^n) \xrightarrow{!} o$. Let $F(p^n) \to t$ be the rule for which i^n matches p^n and let σ be the corresponding substitution, i.e., $i^n = p^n\sigma$. We assume that t contains at least one recursive call. Let $\{F(r_1^n), \ldots, F(r_k^n)\}$ $(k \geq 1)$ be all different recursive calls which occurs in t and $U_1, \ldots, U_k$ the sets of positions of the recursive calls such that $u \in U_j$ iff $t|_u = F(r_j^n)$ for all $j \in \{1, \ldots, k\}$. Let $F(r_j^n)\sigma \xrightarrow{!} o_j$ with $o_j \in T(\mathcal{C}, V)$ for all $j \in \{1, \ldots, k\}$. Then for all $j \in \{1, \ldots, k\}$ and any $u \in U_j$ holds $o|_u = o_j$.*

Proof. $F(i^n) = F(p^n)\sigma$ rewrites according to the rule $F(p^n) \to t$ in one step to $t\sigma$. In $t\sigma$ occur subterms $F(r_j^n)\sigma$ at positions U_j for $j \in \{1, \ldots, k\}$. These recursive calls are normalized to $o_1, \ldots, o_k$ per precondition. □

For an example, let F be the *DelZeros*-CS as shown in Figure 1. This CS takes only one parameter, i.e., we write i, p, and r instead of i^n, p^n, and r^n for inputs, patterns, and recursion terms, respectively. Let be $i = [3, 0, 1]$ then holds $o = [3, 1]$. The rule whose lhs is matched by the "program call" $DelZeros([3, 0, 1])$ is the third one: $DelZeros([s(x)|xs]) \to [s(x)|DelZeros(xs)]$. The corresponding substitution is $\sigma = \{x \mapsto 2, xs \mapsto [0, 1]\}$. The rule has only one recursive call $(k = 1)$, namely $F(r_1^n) = DelZeros(xs)$ at position 2 in the rhs, i.e., $U_1 = \{2\}$. The recursive call of $DelZeros([3, 0, 1])$ is $DelZeros(xs\sigma) = DelZeros([0, 1])$ at position 2 in the instantiated rhs. It normalizes to the output $o_1 = [1]$ which is the subterm of $o = [3, 1]$ at position 2.

4 Inducing Correct Flat 1-Function CSs

We start with basic definitions which we need throughout this section:

Definition 4.1 A set of *I/O-examples* for a function F from $T(\mathcal{C}, V)^n$ to $T(\mathcal{C}, V)$ is a CS F consisting of non-recursive rules such that $F(i^n) = o$ if $F(i^n) \to o$ $(o \in T(\mathcal{C}, V))$ is an I/O-example. We use an index and write F_E instead of simply F for the example CS if the context is ambiguous.

Definition 4.2 A confluent and terminating CS F consisting of rules $F(p^n) \to t$ is *consistent/complete/correct* w.r.t. a set of I/O-examples F_E iff

consistent: for each I/O-example $F(i^n) \to o$ holds: $F(i^n) \xrightarrow{!}_F o$ or $F(i^n) \xrightarrow{!}_F s$ for a term $s \notin T(\mathcal{C}, V)$.

complete: for each I/O-example $F(i^n) \to o$ holds: $F(i^n) \xrightarrow{!}_F s$ for a term $s \in T(\mathcal{C}, V)$.

correct: the CS is both consistent and complete.

Since the rules of a CS are induced independently, we need to define correctness of a single rule:

Definition 4.3 For a pattern p^n let F be a CS containing I/O-examples—i.e., non-recursive rules—whose inputs are subsumed by p^n. Let E be a subset of these I/O-examples and $C(E)$ the remaining rules, i.e., $C(E) = F \setminus E$. A rule ρ with pattern p^n is called *correct* w.r.t. E and $C(E)$ iff the CS which results from replacing E by ρ in F is correct w.r.t. E.

This definition includes the special case that F is a set of I/O-examples and E is the subset containing all I/O-examples whose inputs are subsumed by p^n. In this case the definition states that a rule ρ with pattern p^n is correct w.r.t. E and the remaining I/O-examples iff the CS resulting from replacing the set of I/O-examples E whose inputs match p^n by ρ is correct w.r.t. the replaced I/O-examples.

Induction of a correct CS is organized in two levels as follows: At the higher level, lhss of the rules of the CS to be induced are searched for. This is essentially a search for *patterns* because each pattern determines the lhs of a rule. At a second level, an rhs is computed for each lhs. If computation of a rhs succeeds for each found lhs, then the result is the completely induced CS. If computation of rhss fails for at least one lhs, then a new set of lhss is searched for. Computation of an rhs fails if and only if no rhs exists for the corresponding lhs such that the resulting rule is correct.

Generally there is an infinite number of CSs with different normalizing relations which are correct w.r.t. a set of I/O-examples because correctness w.r.t. I/O-examples makes no claim concerning all terms other than the example inputs. The induction algorithm returns only *one* CSs and therefore it is important to know, which of the correct CSs will be selected. Such a criterion is called *inductive bias*. Informally, the inductive bias of our algorithm can be described by three criteria (the stated order is relevant): An induced correct CS has (i) as few as possible rules, (ii) as specific as possible patterns, and (iii) as general as possible rhss. As few as possible rules means that there exists no other correct CS containing fewer rules. As specific as possible patterns means that each pattern p^n is the lglg of all example inputs i^n which it subsumes. As general as possible rhss means that if there are different possibilities for a particular position, then a variable from the pattern is preferred over a recursive call and a recursive call is preferred over a constructor symbol.

4.1 I/O-examples

Our algorithm is based on detecting regularities between I/O-examples. These regularities are—roughly—the relations between outputs as stated in Theorem 3.2. "Roughly" since—according to Theorem 3.2—the outputs $o_j, j \in \{1, \ldots, k\}$ are assumed to be equal to the subterms of o at positions U_j *up to variable renaming*. Because of this regularity-detection method, I/O-examples have to be well chosen.

More technically, the inputs have to be *recursively subsumed* w.r.t. a CS computing the target function:

Definition 4.4 Let F be a CS which is correct w.r.t. a set of I/O-examples F_E. The I/O-examples are called *recursively subsumed* w.r.t. F iff for all example inputs i^n which match the pattern of any recursive rule of F hold: Let $F(p^n) \rightarrow t$ be the recursive rule such that i^n matches p^n by the substitution σ. Then for each recursive call $F(r^n)$ in t the instantiation $r^n\sigma$ is, up to variable renaming, contained as an example input in F_E.

Since the induction algorithm is not able to use background knowledge nor is it able to automatically introduce additional parameters, the example outputs have to be composed of the constructors of which the resulting rhss are composed and the example inputs have to be inputs for all parameters.

4.2 Searching for Patterns

Before we describe the search for patterns, we state a few characteristics of the state space: If F is a CS induced from a set of I/O-examples F_E, then each example input is subsumed by a pattern of F and there does not exist a pattern which subsumes no example input. In order to reduce the search space, we only consider those patterns which are lglgs of all example inputs in F_E which they respectively subsume.

These three characteristics and the requirement for target CSs, that no two lhss unify, lead to the following characterization of the state space. For each considered set of patterns P holds:

- P subsumes all example inputs,
- no two patterns in P unify,
- each pattern in P subsumes at least one example input,
- each pattern in P is the lglg of all example inputs which it subsumes.

The search space is finite, if the number of I/O-examples is finite. In order to induce a CS with as few as possible rules, the state space will be ordered such that sets with fewer patterns are considered before sets with more patterns. With this order, the state space has exactly one (up to variable renaming) minimum, namely the set containing exactly one pattern—the lglg of all example inputs. This state is the initial state. And the state space has exactly one (up to variable renaming) maximum, namely the set containing each example input as its own pattern. This state is the last state considered for which computation of correct rules ever succeeds since simply the I/O-example itself is a rule which is correct (w.r.t. itself).

Now suppose a state P with an arbitrary number of patterns which comply with the stated conditions. If for at least one pattern a correct rule could not be computed, then successor states have to be computed. Let p^n be such a pattern for which no correct rule exists. Then the I/O-examples whose inputs are subsumed by p^n has to be partitioned into a minimum number of at least two subsets and p^n has to be replaced by the lglgs of the inputs of the respective subsets. We call the

new set of lglgs/patterns *most generally partitioning lglgs (mgpls)*.

This is done as follows: First a position u from $F(p^n)$ which is labeled by a variable in $F(p^n)$ and by a constructor in each subsumed example lhs is selected. Since $F(p^n)$ is the lglg of the subsumed example lhss it then holds that in at least two example inputs these constructors differ. Then respectively all example inputs with the *same* constructor at position u are taken into the same subset. That leads to a partition of the example inputs. Finally, for each subset the lglg is computed.

A successor state is a state in which all patterns for which no correct rule could be computed are replaced by corresponding sets of mgpls. Since the position which determines a partition is generally not unique, also the sets of mgpls are not unique. Thus, all combinations of replacing these patterns by mgpls are included as successor states.

For example, let

$$
\begin{array}{lll}
1. & \mathit{DelZeros}([]) & \rightarrow\ [] \\
2. & \mathit{DelZeros}([0]) & \rightarrow\ [] \\
3. & \mathit{DelZeros}([s(x)]) & \rightarrow\ [s(x)] \\
4. & \mathit{DelZeros}([0,0]) & \rightarrow\ [] \\
5. & \mathit{DelZeros}([s(x),0]) & \rightarrow\ [s(x)] \\
6. & \mathit{DelZeros}([s(x),s(y)]) & \rightarrow\ [s(x),s(y)]
\end{array}
$$

be I/O-examples for *DelZeros* (cp. Fig. 1). The initial state consists of the lglg of the six example inputs $[], [0], [s(x)], [0,0], [s(x),0], [s(x),s(y)]$ which is simply a single variable, q. Since there exists no correct rule for this pattern, successor sets of patterns has to be computed. Since the current pattern is a variable, the (unique) position in $\mathit{DelZeros}(q)$ determining the partition is position 1. The two constructors which occurs at this position in the example lhss are [] and *cons*. Hence, the examples are partitioned into two subsets containing the first example and all other examples respectively. The new patterns are [] and $[q|qs]$ since these are the lglgs of the first input and the other inputs respectively. For pattern [] a rule will be found (simply the first I/O-example) but for pattern $[q|qs]$ no rhs will be found. Thus, the remaining examples has to be partitioned again. Now there are *two* positions in $\mathit{DelZeros}([q|qs])$ which come into question, 1.1 and 1.2. Position 1.1 denotes the variable q in the current lhs and the constructors 0 and s in the remaining example inputs 2-6 respectively. It partitions the I/O-examples into one set consisting of I/O-examples 2 and 4 and another set consisting of I/O-examples 3, 5, and 6. This leads to two new patterns/lglgs, $[0|qs]$ and $[s(x)|qs]$. Position 1.2 denotes the variable qs in the current lhs and the constructors [] and *cons* in the remaining example inputs respectively. It partitions the I/O-examples into one set consisting of I/O-examples 2 and 3 and another set consisting of I/O-examples 4-6. This leads to two new patterns, $[q]$ and $[q, q']$. Thus, we get two successor states for the pattern state $\{[],\ [q|qs]\}$, namely first $\{[],\ [0|qs],\ [s(x)|qs]\}$ (cp. the

DelZeros-CS shown in Fig. 1) and second $\{[],\ [q],\ [q, q']\}$.

4.3 Computing Right Hand Sides

Given a set of patterns, for each lhs with pattern p^n a correct rhs has to be computed. The symbol at an arbitrary position in an rhs of a flat one-function CS F can be either

- a constructor from $\mathcal{C}$,
- the defined function symbol F (recursive call), or
- a variable contained in the pattern p^n of the corresponding lhs.

The following theorem states sufficient conditions for a recursive calls:

Theorem 4.5 *Let $F(i^n) \to o$ be an I/O-example and p^n a pattern which subsumes i^n with substitution σ. Let $F(i'^n) \to o'$ be a second I/O-example such that (i) $o|_u = o'\tau$ for some position u and variable renaming τ, (ii) $i'^n = r^n\theta$ for a list of terms r^n and some substitution θ for all variables contained in r^n, and (iii) $\theta\tau \subseteq \sigma$. Let F be a confluent and terminating CS which is correct w.r.t. $F(i'^n) \to o'$ and contains a rule $\rho : F(p^n) \to t[F(r^n)]_u$ such that the lhs $F(p^n)$ does not unify with another lhs in F. Then $F(i^n)$ normalizes to a term s with $s|_u = o|_u$, i.e., local correctness at position u of ρ w.r.t. the I/O-example $F(i^n) \to o$ and $F \setminus \{\rho\}$ is assured.*

Proof. $F(i^n) = F(p^n)\sigma$ per precondition rewrites in one step to $t[F(r^n)]_u\sigma = t\sigma[F(r^n)\sigma]_u$ according to rule ρ. According to (ii) and (iii) holds $t\sigma[F(r^n)\sigma]_u = t\sigma[F(r^n)\theta\tau]_u = t\sigma[F(i'^n)\tau]_u$. $F(i'^n)\tau$ normalizes to $o'\tau$ per precondition, i.e., $t\sigma[F(i'^n)\tau]_u$ normalizes to $t'[o'\tau]_u$ for some term t'. It holds $t'[o'\tau]_u = t'[o|_u]_u$ according to (i), i.e., $F(i^n)$ normalizes to $s = t'[o|_u]_u$ and it holds $s|_u = o|_u$. □

The theorem states conditions respecting *one* I/O-example $F(i^n) \to o$ whose input is subsumed by a pattern p^n of a rule to be constructed, which are sufficient to introduce a particular recursive call $F(r^n)$ at a particular position u. Of course, *all* I/O-examples whose inputs are subsumed by p^n has to fulfill the conditions of the theorem *with the same recursive call* $F(r^n)$.

The previous characterization of rhss and the theorem about recursive calls lead to the following general method for constructing an rhs of a correct rule given a pattern p^n and I/O-examples whose inputs match the pattern. The following three cases are considered in the stated order:

1. **Pattern variable:** All outputs can be produced by the same pattern variable: Then the rhs becomes this variable.
2. **Recursive call:** All outputs can be produced by the same recursive call $F(r^n)$: Then the rhs becomes $F(r^n)$.
3. **Constructor:** The roots of all outputs are the same constructor f with arity m: Then the rhs becomes the term $f(t_1, \ldots, t_m)$ where the t_i are constructed by considering these three cases for the subterms of the outputs at position i.

If no case is applicable then no correct rhs exists for this pattern and it has to be searched for a new set of patterns.

For an example, let us again consider the I/O-examples for *DelZeros*:

1. $DelZeros([]) \rightarrow []$, 2. $DelZeros([0]) \rightarrow []$,
3. $DelZeros([s(x)]) \rightarrow [s(x)]$, 4. $DelZeros([0,0]) \rightarrow []$,
5. $DelZeros([s(x),0]) \rightarrow [s(x)]$, 6. $DelZeros([s(x),s(y)]) \rightarrow [s(x),s(y)]$

Assumed, the pattern $[s(x)|qs]$ which subsumes the inputs 3, 5, and 6 with substitutions $\sigma_3 = \{x \mapsto x, qs \mapsto []\}$, $\sigma_5 = \{x \mapsto x, qs \mapsto [0]\}$, and $\sigma_6 = \{x \mapsto x, qs \mapsto [s(y)]\}$ respectively has been found. Since for all of the three considered I/O-examples the output differs from the respective instantiations of both pattern variables, case 1 fails. Also case 2 fails, because no recursive call can be found such that for all three I/O-examples the conditions from Theorem 4.5 are fulfilled. Case 3 succeeds because all three considered outputs have the same constructor symbol, *cons*, as root. Thus, the root of the rhs to be constructed becomes *cons* and the three cases are applied to positions 1 and 2 of the three considered example outputs. For position 1 this leads to the subterm $s(x)$ after two steps. For position 2 case 1 fails. But case 2 (recursive call) succeeds, because for the recursive call $DelZeros(qs)$ the conditions from Theorem 4.5 are fulfilled for all three subterms at position 2 of the three outputs: The subterm of output 3 at position 2 is []. For I/O-example 1 holds that its output equals this subterm with variable renaming $\tau = \emptyset$. The input of I/O-example 1 matches the recursive call $DelZeros(qs)$ by substitution $\theta = \{qs \mapsto []\}$ and it holds $\theta\tau \subseteq \sigma_3$. For I/O-example 5 the conditions are fulfilled with I/O-example 2 as second I/O-example, $\tau = \emptyset$, and $\theta = [0]$. For I/O-example 6 the conditions are fulfilled with I/O-example 3 as second I/O-example, $\tau = \{x \mapsto y\}$, and $\theta = [s(x)]$. The resulting rhs is $[s(x)|DelZeros(qs)]$.

5 Evaluation of the Approach

Though our approach essentially learns functional programs, it is straightforward to transform any induced flat one-function CS into a logic program, e.g., a prolog program. For example, consider the *DelZeros*-CS shown in Figure 1. An equivalent prolog program is:

$$
\begin{array}{lll}
DelZeros([],[]) & \leftarrow & \\
DelZeros([0|XS],Z) & \leftarrow & DelZeros(XS,Z) \\
DelZeros([s(X)|XS],[s(X)|Z]) & \leftarrow & DelZeros(XS,Z)
\end{array}
$$

We have implemented a prototype of the described algorithm in the programming language Maude [2]. 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.

function	#expl	#rules(#rec)	#rec. calls	#rec. params	times
Length	3	2(1)	1	1	.002
Last	3	2(1)	1	1	.003
IncList	3	2(1)	1	1	.003
Even	4	3(1)	1	1	.004
TreeRev	4	2(1)	2	1	.009
Add	9	2(1)	1	1	.022
DelZeros	6	3(2)	1	1	.055
$\leq$	9	3(1)	1	2	.081
Sum	13	3(2)	1	1	.127
Take	9	3(2)	1	2	.145
Zip	9	3(2)	1	2	.263
PlayTennis	14	5(0)	0	0	.679

Table 1
Some inferred functions

In Table 1 we have listed experimental results for sample problems. The first column lists the names for the induced functions, the second the number of given I/O-examples, the third the total number of induced rules and in parentheses the number of induced *recursive* rules, the fourth the maximal number of recursive calls within one rule, the fifth the number of recursion parameters, and the sixth the times in seconds consumed by the synthesis. The experiments were performed on a Pentium 4 with Linux and the program runs are interpreted with the Maude 2.2 interpreter.

The functions, except for *IncList* and *PlayTennis*, are described in Section 3.2. *IncList* applies the successor function, s, to each element of a given list. *PlayTennis* is an attribute vector concept learning example from Mitchell's machine learning text book [9]. 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. All induced programs compute the intended function.

6 Conclusions and Further Research

We described a method to induce a particular class of functional programs represented by confluent and terminating flat one-function CSs. The presented methodology is inspired by classical and recent analytical approaches to the induction of functional programs. The method is distinguished from most other methods in

its ability to induce programs over arbitrary data types and in that the induced programs can contain more than one recursion parameter. Until now, neither background knowledge can be used nor can additional subprograms or parameters not contained in the I/O-examples be induced. Yet techniques for automatic introduction of further subprograms as well as further parameters to the induced programs have been developed within the analytical approaches and should be applicable to our approach as well.

References

[1] Biermann, A. W., G. Guiho and Y. Kodratoff, editors, "Automatic Program Construction Techniques," Collier Macmillan, 1984.

[2] Clavel, M., F. Durán, S. Eker, P. Lincoln, N. Martí-Oliet, J. Meseguer and C. Talcott, *The maude 2.0 system*, in: R. Nieuwenhuis, editor, *Rewriting Techniques and Applications (RTA 2003)*, number 2706 in Lecture Notes in Computer Science (2003), pp. 76–87.

[3] Flener, P. and S. Yilmaz, *Inductive synthesis of recursive logic programs: Achievements and prospects*, Journal of Logic Programming **41** (1999), pp. 141–195.

[4] Jouannaud, J. P. and Y. Kodratoff, *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)* (1979), pp. 440–447.

[5] Kitzelmann, E., R. Olsson and U. Schmid, editors, "Proceedings of the ICML 2005 Workshop Approaches and Applications of Inductive Programming," 2005.

[6] Kitzelmann, E. and U. Schmid, *Inductive synthesis of functional programs: An explanation based generalization approach*, Journal of Machine Learning Research **7** (2006), pp. 429–454, special topic on Approaches and Applications of Inductive Programming.

[7] Kodratoff, Y. and 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, pp. 169–175.

[8] Le Blanc, G., *BMWk revisited: Generalization and formalization of an algorithm for detecting recursive relations in term sequences*, in: F. Bergadano and L. de Raedt, editors, *Machine Learning, Proc. of ECML-94* (1994), pp. 183–197.

[9] Mitchell, T. M., "Machine Learning," McGraw-Hill Higher Education, 1997.

[10] Olsson, R., *Inductive functional programming using incremental program transformation*, Artificial Intelligence **74** (1995), pp. 55–83.

[11] Rao, M. R. K. K., *Learning recursive prolog programs with local variables from examples*, in: *Proceedings of the ICML'05-workshop on Approaches and Applications of Inductive Programming*, 2005, pp. 51–57.

[12] Schmid, U. and J. Waltermann, *Automatic synthesis of XSL-transformations from example documents*, in: *Artificial Intelligence and Applications Proceedings (AIA 2004, Innsbruck, Austria, February 16-18)* (2004), pp. 252–257.

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

[14] Yilmaz, S., "Inductive Synthesis of Recursive Logic Programs," Master's thesis, Bilkent University (1997), http://user.it.uu.se/ pierref/pub/SerapMSc.ps.gz.