Journal of Machine Learning Research 7 (2006) 429–454 Submitted 7/05; Revised 1/06; Published 2/06

Inductive Synthesis of Functional Programs: An Explanation Based Generalization Approach

Emanuel Kitzelmann EMANUEL.KITZELMANN@WIAI.UNI-BAMBERG.DE
Ute Schmid UTE.SCHMID@WIAI.UNI-BAMBERG.DE
Department of Information Systems and Applied Computer Science
Otto-Friedrich-University
Bamberg, Germany

Editors: Roland Olsson and Leslie Pack Kaelbling

Abstract

We describe an approach to the inductive synthesis of recursive equations from input/output-examples which is based on the classical two-step approach to induction of functional Lisp programs of Summers (1977). In a first step, I/O-examples are rewritten to traces which explain the outputs given the respective inputs based on a datatype theory. These traces can be integrated into one conditional expression which represents a non-recursive program. In a second step, this initial program term is generalized into recursive equations by searching for syntactical regularities in the term. Our approach extends the classical work in several aspects. The most important extensions are that we are able to induce a *set* of recursive equations in one synthesizing step, the equations may contain more than one recursive call, and additionally needed parameters are automatically introduced.

Keywords: inductive program synthesis, inductive functional programming, explanation based generalization, recursive program schemes

1. Introduction

Automatic induction of recursive programs from input/output-examples (I/O-examples) is an active area of research since the sixties and of interest for AI research as well as for software engineering (Lowry and McCarthy, 1991; Flener and Partridge, 2001). In the seventies and eighties, there were several approaches to the synthesis of Lisp programs from examples or traces (see Biermann et al. 1984 for an overview). The most influential approach was developed by Summers (1977), who put inductive synthesis on a firm theoretical foundation.

Summers' early approach is an explanation based generalization (EBG) approach, thus it widely relies on algorithmic processes and only partially on search: In a first step, traces—steps of computations executed from a program to yield an output from a particular input—and predicates for distinguishing the inputs are calculated for each I/O-pair. Construction of traces, which are *terms* in the classical functional approaches, relies on knowledge of the inductive datatype of the inputs and outputs. That is, traces *explain* the outputs based on a theory of the used datatype given the respective inputs. The classical approaches for synthesizing Lisp-programs used the general Lisp datatype *S-expression*. By integrating traces and predicates into a conditional expression a non-recursive program explaining all I/O-examples is constructed as a result of the first synthesis step. In a second step, regular-

ities are searched for between the traces and predicates respectively. Found regularities are then inductively generalized and expressed in the form of the resulting recursive program.

The programs synthesized by Summers' system contain exactly one recursive function, possibly along with one constant term calling the recursive function. Furthermore, all synthesizable functions make use of a small fixed set of Lisp-primitives, particularly of exactly one predicate function, *atom*, which tests whether its argument is an atom, e.g., the empty list. The latter implies two things: First, that Summers' system is restricted to induce programs for *structural* problems on S-expressions. That means, that execution of induced programs depends only on the structure of the input S-expression, but never on the semantics of the atoms contained in it. For example, *reversing* a list is a structural problem, yet not *sorting* a list. The second implication is, that calculation of the traces is a deterministic and algorithmic process, i.e., does not rely on search and heuristics.

Due to only limited progress regarding the class of programs which could be inferred by functional synthesis, interest decreased in the mid-eighties. There was a renewed interest of inductive program synthesis in the field of inductive logic programming (ILP) (Flener and Yilmaz, 1999; Muggleton and De Raedt, 1994), in genetic programming and other forms of evolutionary computation (Olsson, 1995) which rely heavily on search.

We here present an EBG approach which is based on the methodologies proposed by Summers (1977). We regard the functional two-step approach as worthwhile for the following reasons: First, algebraic datatypes provide guidance in expressing the outputs in terms of the inputs as the first synthesis step. Second, it enables a seperate and thereby specialized handling of a knowledge dependent part and a purely syntactic driven part of program synthesis. Third, using both algebraic datatypes and seperating a knowledge-dependent from a syntactic driven part enables a more accurate use of search than in ILP or evolutionary programming. Fourth, the two-step approach using algebraic datatypes provides a systematic way to introduce auxiliary recursive equations if necessary.

Our approach extends Summers in several important aspects, such that we overcome fundamental restrictions of the classical approaches to induction of Lisp programs: First, we are able to induce a *set* of recursive equations in one synthesizing step, second, the equations may contain more than one recursive call, and third, additionally needed parameters are automatically introduced. Furthermore, our generalization step is domain-independent, in particular independent from a certain programming language. It takes as input a first-order term over an arbitrary signature and generalizes it to a recursive program scheme, that is, a set of recursive equations over that signature. Hence it can be used as a learning component in all domains which can represent their objects as recursive program schemes and provide a system for solving the first synthesis step. For example, we use the generalization algorithm for learning recursive control rules for AI planning problems (cp. Schmid and Wysotzki 2000; Wysotzki and Schmid 2001).

2. Overview Over the Approach

The three central objects dealt with by our system are (1) sets of *I/O-examples* specifying the algorithm to be induced, (2) *initial (program) terms* explaining the I/O-examples, and (3) *recursive program schemes (RPSs)* representing the induced algorithms. Their functional role in our two-step synthesis approach is shown in Figure 1.

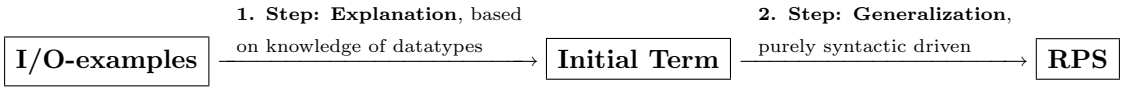

Figure 1: Two synthesis steps

2.1 First Synthesis Step: From I/O-examples to an Initial Term

An example for I/O-examples is given in Table 1. The examples specify the *lasts* function which takes a list of lists as input and yields a list of the last elements of the lists as output. In the first synthesis step, an initial term is constructed from these examples. An

$$\begin{aligned} [] &\mapsto [], \\ [[a]] &\mapsto [a], \\ [[a,b]] &\mapsto [b], \\ [[a,b,c],[d]] &\mapsto [c,d], \\ [[a,b,c,d],[e,f]] &\mapsto [d,f], \\ [[a],[b],[c]] &\mapsto [a,b,c] \end{aligned}$$

Table 1: I/O-examples for *lasts*

initial term is a term respecting an arbitrary first-order signature extended by the special constant symbol Ω, meaning the *undefined* value and directing generalization in the second synthesis step. Suitably interpreted, an initial term evaluates to the specified output when its variable is instantiated with a particular input of the example set and to *undefined* for all other inputs.

Table 2 gives an example of an initial term. It shows the result of applying the first synthesis step to the I/O-examples for the *lasts* function as shown in Table 1. *if* means the 3ary non-strict function which returns the value of its second parameter if its first parameter evaluates to *true* and otherwise returns the value of its third parameter; *empty* is a predicate which tests, whether its argument is the empty list; *head* and *tail* yield the first element and the rest of a list respectively; *cons* constructs a list from one element and a list; and [] denotes the empty list.

Calculation of initial terms relies on knowledge of the datatypes of the example inputs and outputs. For our exemplary *lasts* program inputs and outputs are lists. Lists are uniquely constructed by means of the empty list [] and the constructor *cons*. Furthermore, they are uniquely decomposed by the functions *head* and *tail*. That allows to calculate a unique term which expresses an example output in terms of the input. For example, consider the fourth I/O-example from Table 1: If x denotes the input $[[a,b,c],[d]]$, then the term $cons(head(tail(tail(head(x)))), head(tail(x)))$ expresses the specified output $[c,d]$ in terms of the input. Such traces are constructed for each I/O-pair. The overall concept for integrating the resulting traces into one initial term is to go through all traces in parallel position by position. If the same function symbol is contained at the current position in all traces, then it is introduced to the initial term at this position. If at least two traces differ at the current position, then an *if*-expression is introduced. Therefore a predicate function is calculated to discriminate the inputs according to the different traces. Construction of the initial term proceeds from the discriminated inputs and traces for the second and

```
if(empty(x), [],
 cons(
  head(
   if(empty(tail(head(x))), head(x),
    if(empty(tail(tail(head(x)))), tail(head(x)),
     if(empty(tail(tail(tail(head(x))))), tail(tail(head(x))),
      Ω)))),
  if(empty(tail(x)), [],
   cons(
    head(
     if(empty(tail(head(tail(x)))), head(tail(x)),
      Ω)),
    if(empty(tail(tail(x))), [],
     Ω)))))))
```

Table 2: Initial term for *lasts*

third branch of the *if*-tree respectively. We describe the calculation of initial terms from I/O-examples, i.e., the first synthesis step, in Section 4.

2.2 Second Synthesis Step: From Initial Terms to Recursive Equations

In the second synthesis step, initial ground terms are generalized to a recursive program scheme. Initial terms are considered as *(incomplete) unfoldings* of an RPS which is to be induced by generalization. An RPS is a set of recursive equations whose left-hand-sides consist of the names of the equations followed by their parameter lists and whose right-hand-sides consist of terms over the signature from the initial terms, the set of the equation names, and the parameters of the equations. One equation is distinguished to be the main one. An example is given in Table 3. This RPS, suitably interpreted, computes the *lasts* function as described above and specified by the examples in Table 1. It results from

$$lasts(x) = \text{if}(\text{empty}(x), [], \text{cons}(\text{head}(last(\text{head}(x))), lasts(\text{tail}(x))))$$

$$last(x) = \text{if}(\text{empty}(\text{tail}(x)), x, last(\text{tail}(x)))$$

Table 3: Recursive Program Scheme for *lasts*

applying the second synthesis step to the initial term shown in Table 2. Note that it is a *generalization* from the initial term in that it not merely computes the *lasts* function for the example inputs but for input-lists of arbitrary length containing lists of arbitrary length.

The second synthesis step does not depend on domain knowledge. The meaning of the function symbols is irrelevant, because the generalization is completely driven by detecting syntactical regularities in the initial terms. To understand the link between initial terms and RPSs induced from them, we consider the process of incrementally unfolding an RPS.

Unfolding of an RPS is a (non-deterministic and possibly infinite) rewriting process which starts with the instantiated head of the main equation of an RPS and which repeatedly rewrites a term by substituting any instantiated head of an equation in the term with either the equally instantiated body or with the special symbol Ω. Unfolding stops, when all heads of recursive equations in the term are rewritten to Ω, i.e., the term contains no rewritable head any more. Consider the *last* equation from the RPS shown in Table 3 and the initial instantiation $\{x \mapsto [a, b, c]\}$. We start with the instantiated head $last([a, b, c])$ and rewrite it to the term:

$$\text{if}(\text{empty}(\text{tail}([a, b, c])), [a, b, c], \text{last}(\text{tail}([a, b, c])))$$

This term contains the head of the *last* equation instantiated with $\{x \mapsto tail([a, b, c])\}$. When we rewrite this head again with the equally instantiated body we obtain:

$$\begin{array}{l}\text{if}(\text{empty}(\text{tail}([a, b, c])), [a, b, c],\\ \quad \text{if}(\text{empty}(\text{tail}(\text{tail}([a, b, c]))), \text{tail}([a, b, c]),\\ \quad\quad \text{last}(\text{tail}(\text{tail}([a, b, c])))\end{array}$$

This term now contains the head of the equation instantiated with $\{x \mapsto tail(tail([a, b, c]))\}$. We rewrite it once again with the instantiated body and then replace the head in the resulting term with Ω and obtain:

$$\begin{array}{l}\text{if}(\text{empty}(\text{tail}([a, b, c])), [a, b, c],\\ \quad \text{if}(\text{empty}(\text{tail}(\text{tail}([a, b, c]))), \text{tail}([a, b, c]),\\ \quad\quad \text{if}(\text{empty}(\text{tail}(\text{tail}(\text{tail}([a, b, c])))), \text{tail}(\text{tail}([a, b, c])), \Omega)))\end{array}$$

The resulting *finite* term of a *finite* unfolding process is also called *unfolding*. Unfoldings of RPSs contain regularities if the heads of the recursive equations are more than once rewritten with its bodies before they are rewritten with Ωs. The second synthesis step is based on detecting such regularities in the initial terms.

We describe the generalization of initial terms to RPSs in Section 3. The reason why we first describe the second synthesis step and only afterwards the first synthesis step is, that the latter is governed by the goal of constructing a term which can be generalized in the second step. Therefore, for understanding the first step, it is necessary to know the connection between initial terms and RPSs as established in the second step.

2.3 Characteristics and Limitations of the Approach

The overall objective of our approach is automatical induction of recursive functional programs from I/O-examples which are correct with respect to the functional behaviour desired by the user. Since the approach is based on finding differences between traces, i.e., analyzing one example in relation to the following example, the examples have to be the first k examples according to an ordering of the underlying data-type with the first example beeing the least complex instance for which the target recursive program is defined. This is in contrast to learning from a randomly chosen set of training data (according to some distribution) which is the common setting in most learning approaches, e.g., in all PAC-learning (Valiant, 1984) algorithms. Another implication of this generalization methodology is that very few

examples are sufficient. This is again in contrast to most learning settings, especially in contrast to the identification-in-the-limit setting (Gold, 1967). A third implication is that the examples have to be correct, i.e., the desired function has to be consistent with the examples. This is not a limitation in our view since we assume that the examples are given by some (end-user) programmer who knows the functional behaviour of the target program and thus can provide a few correct examples. A fourth implication of the example-driven approach is, that termination is assured for the induced programs. This is an important characteristic since in general, termination is not decidable. Our algorithms output a set of recursive equations which are consistent with the I/O-examples, i.e., which compute any specified example-output from the respective example-input.

There are restrictions regarding the programs which can be synthesized. The first step (see Section 2.1 for an overview and Section 4 for details) is restricted to structural problems, i.e., functions on lists may only depend on the list structure but not on the meaning of the items in the lists. The induced recursive equations stand in some call-relation. Due to the second synthesis step (see Section 2.2 for an overview and Section 3 for details), this relation is restricted to be *flat*, that is, recursive calls cannot be nested. Furthermore, the relation is non-mutual, i.e., if one equation calls a second one, then the second one cannot call the first one. Since the induction process relies on two successive synthesis steps, the overall restrictions are the sum of the restrictions of the first step and the restrictions of the second step. On the other hand, the two-step approach provides some modularity. Since the limitation to structural problems is a restriction of *only* the first step, it would be sufficient to only extend or substitute the first step to extend our approach to be capable of dealing with non-structural problems, e.g., sorting problems.

For experimental results, a discussion of the approach, and a comparison to other inductive programming systems see Sections 5 and 6.

3. Generalizing an Initial Term to an RPS

Since our generalization algorithm exploits the relation between an RPS and its unfoldings, in the following we will first introduce the basic terminology for terms, substitutions, and term rewriting as for example presented in Dershowitz and Jouanaud (1990). Then we will present definitions for RPSs and the relation between RPSs and their unfoldings. The set of all possible RPSs constitutes the hypothesis language for our induction algorithm. Some restrictions on this general hypothesis language are introduced and finally, the components of the generalization algorithm are described.

3.1 Preliminaries

We denote the set of natural numbers starting with 0 by $\mathbb{N}$ and the natural numbers greater than 0 by $\mathbb{N}_+$. A signature Σ is a set of (function) symbols with $\alpha : \Sigma \to \mathbb{N}$ giving the arity of a symbol. We write T_Σ for the set of ground terms, i.e., terms without variables, over Σ and $T_\Sigma(X)$ for the set of terms over Σ and a set of variables X. We write $T_{\Sigma,\Omega}$ for the set of ground terms—called partial ground terms—constructed over $\Sigma \cup \{\Omega\}$, where Ω is a special constant symbol denoting the *undefined* value. Furthermore, we write $T_{\Sigma,\Omega}(X)$ for the set of partial terms constructed over $\Sigma \cup \{\Omega\}$ and variables X. With $T^\infty_{\Sigma,\Omega}(X)$ we denote the set of inifinite partial terms over Σ and variables X. Over the sets $T_{\Sigma,\Omega}$, $T_{\Sigma,\Omega}(X)$ and $T^\infty_{\Sigma,\Omega}(X)$

a complete partial order (CPO) $\leq$ is defined by: a) $\Omega \leq t$ for all $t \in T_{\Sigma,\Omega}, T_{\Sigma,\Omega}(X), T^{\infty}_{\Sigma,\Omega}(X)$ and b) $f(t_1,\ldots,t_n) \leq f(t'_1,\ldots,t'_n)$ iff $t_i \leq t'_i$ for all $i \in [1;n]$.

Terms can uniquely be expressed as labeled trees: If a term is a constant symbol or a variable, then the corresponding tree consists of only one node labeled by the constant symbol or variable. If a term has the form $f(t_1,\ldots,t_n)$, then the root node of the corresponding tree is labeled with f and contains from left to right the subtrees corresponding to $t_1,\ldots,t_n$. We use the terms *tree* and *term* as synonyms. A *position* of a term/tree is a sequence of positive natural numbers, i.e., an element from $\mathbb{N}_+^*$. The set of positions of a term t, denoted $\mathbf{pos}(t)$, contains the empty sequence ϵ and the position iu, if the term has the form $t = f(t_1,\ldots,t_n)$ and u is a position from $\mathbf{pos}(t_i), i \in [1;n]$. Each position of a term uniquely denotes one subterm. We write $t|_u$ for denoting that subterm which is determined as follows: (a) $t|_\epsilon = t$, (b) if $t = f(t_1,\ldots,t_n)$ and u is a position in t_i, then $t|_{iu} = t_i|_u$, $i \in [1;n]$. E.g., for the term $f(x,y,g(h(a,p(s,t),b),z))$ the position 312 denotes the subterm $p(s,t)$ because it is the second subterm of the first subterm of the third subterm of the original term. We say that position u is smaller than position u', $u \leq u'$, if u is a prefix of u'. If u is a position of term t and $u' \leq u$, then u' is a position of t. For a term t and a position u, $\mathbf{node}(t,u)$ denotes the fixed symbol $f \in \Sigma$, if $t|_u = f(t_1,\ldots,t_n)$ or $t|_u = f$ respectively. The set of all positions at which a fixed symbol f appears in a term is denoted by $\mathbf{pos}(t,f)$. The replacement of a subterm $t|_u$ by a term s in a term t at position u is written as $t[u \leftarrow s]$. Let U denote a set of positions in a term t. Then $t[U \leftarrow s]$ denotes the replacement of all subterms $t|_u$ with $u \in U$ by s in t.

A *substitution* σ is a mapping from variables to terms. Substitutions are naturally continued to mappings from terms to terms by $\sigma(f(t_1,\ldots,t_n)) = f(\sigma(t_1),\ldots,\sigma(t_n))$. Substitutions are written in postfix notation, i.e., we write $t\sigma$ instead of $\sigma(t)$. Substitutions $\beta : X \to T_\Sigma$ from variables to *ground* terms are called (variable) *instantiations*. A term p is called *pattern* of a term t, iff $t = p\sigma$ for a substitution σ. A pattern p of a term t is called *trivial*, iff p is a variable and *non-trivial* otherwise. We write $t \leq_s p$ iff p is a pattern of t and $t <_s p$ iff additionally holds, that p and t can not be unified by variable renaming only.

A *term rewriting system (TRS)* over Σ and X is a set of pairs of terms $\mathcal{R} \subseteq T_\Sigma(X) \times T_\Sigma(X)$. The elements (l,r) of $\mathcal{R}$ are called rewrite rules and are written $l \to r$. A term t' can be derived in one rewrite step from a term t using $\mathcal{R}$ ($t \to_\mathcal{R} t'$), if there exists a position u in t, a rule $l \to r \in \mathcal{R}$, and a substitution $\sigma : X \to T_\Sigma(X)$, such that (a) $t|_u = l\sigma$ and (b) $t' = t[u \leftarrow r\sigma]$. $\mathcal{R}$ implies a rewrite relation $\to_\mathcal{R} \subseteq T_\Sigma(X) \times T_\Sigma(X)$ with $(t,t') \in \to_\mathcal{R}$ if $t \to_\mathcal{R} t'$.

3.2 Recursive Program Schemes

Definition 1 (Recursive Program Scheme) *Given a signature Σ, a set of* function variables $\Phi = \{G_1,\ldots,G_n\}$ *for a natural number $n > 0$ with $\Sigma \cap \Phi = \emptyset$ and arity $\alpha(G_i) > 0$ for all $i \in [1;n]$, a natural number $m \in [1;n]$, and a set of equations*

$$\mathcal{G} = \begin{array}{c} \{\ G_1(x_1,\ldots,x_{\alpha(G_1)}) = t_1, \\ \vdots \\ G_n(x_1,\ldots,x_{\alpha(G_n)}) = t_n\ \} \end{array}$$

where the t_i are terms with respect to the signature $\Sigma \cup \Phi$ and the variables $x_1, \ldots, x_{\alpha(G_i)}$, $\mathcal{S} = (\mathcal{G}, m)$ is an RPS. $G_m(x_1, \ldots, x_{\alpha(G_m)}) = t_m$ is called the main equation of $\mathcal{S}$.

The function variables in Φ are called *names* of the equations, the left-hand-sides are called *heads*, the right-hand-sides *bodies* of the equations. For the *lasts* RPS shown in Table 3 holds: $\Sigma = \{if, empty, cons, head, tail, []\}$, $\Phi = \{G_1, G_2\}$ with $G_1 = lasts$ and $G_2 = last$, and $m = 1$. $\mathcal{G}$ is the set of the two equations.

We can identify a TRS with an RPS $\mathcal{S} = (\mathcal{G}, m)$:

Definition 2 (TRS implied by an RPS) *Let $\mathcal{S} = (\mathcal{G}, m)$ be an RPS over Σ, Φ and X, and Ω the bottom symbol in $T_{\Sigma,\Omega}(X)$. The equations in $\mathcal{G}$ constitute rules $\mathcal{R}_\mathcal{S} = \{G_i(x_1, \ldots, x_{\alpha(G_i)}) \rightarrow t_i \mid i \in [1; n]\}$ of a term rewriting system. The system additionally contains rules $\mathcal{R}_\Omega = \{G_i(x_1, \ldots, x_{\alpha(G_i)}) \rightarrow \Omega \mid i \in [1; n],\ G_i$ is* recursive*$\}$.*

The standard interpretation of an RPS, called *free interpretation*, is defined as the supremum in $T^\infty_{\Sigma,\Omega}(X)$ of the set of all terms in $T_{\Sigma,\Omega}(X)$ which can be derived by the implied TRS from the head of the main equation. Two RPSs are called *equivalent*, iff they have the same free interpretation, i.e., if they compute the same function for every interpretation of the symbols in Σ. Terms in $T_{\Sigma,\Omega}$ which can be derived by the *instantiated* head of the main equation regarding some instantiation $\beta : X \rightarrow T_\Sigma$ are called *unfoldings* of an RPS relative to β. Note, that terms derived from RPSs are partial and do not contain function variables, i.e., all heads of the equations are eventually rewritten by Ωs.

The goal of the generalization step is to find an RPS which *explains* a set of initial terms, i.e., to find an RPS such that the initial terms are unfoldings of that RPS. We denote initial terms by $\bar{t}$ and a set of initial terms by $\mathcal{I}$. We liberalize $\mathcal{I}$ such that it may include *incomplete* unfoldings. Incomplete unfoldings are unfoldings, where some subtrees containing Ωs are replaced by Ωs.

We need to define four further concepts, namely *recursion positions* which are positions in the equation bodies where recursive calls appear, *substitution terms* which are the argument terms in recursive calls, *unfolding positions* which are positions in unfoldings at which the heads of the equations are rewritten with their bodies, and finally *parameter instantiations in unfoldings* which are subterms of unfoldings resulting from the initial parameter instantiation and the substitution terms:

Definition 3 (Recursion Positions and Substitution Terms) *Let $G(x_1, \ldots, x_{\alpha(G)}) = t$ with parameters $X = \{x_1, \ldots, x_{\alpha(G)}\}$ be a recursive equation. The set of* recursion positions *of G is given by $R = \mathbf{pos}(t, G)$. Each recursive call of G at position $r \in R$ in t implies substitutions $\sigma_r : X \rightarrow T_\Sigma(X) : x_j \mapsto t|_{rj}$ for all $j \in [1; \alpha(G)]$ for the parameters in X. We call the terms $t|_{rj}$* substitution terms *of G.*

For equation *lasts* of the *lasts* RPS (Table 3) holds $R = \{32\}$ and $x\,\sigma_{32} = tail(x)$. For equation *last* holds $R = \{3\}$ and $x\,\sigma_3 = tail(x)$.

Now consider an unfolding process of a recursive equation and the positions at which rewrite steps are applied in the intermediate terms. The first rewriting is applied at root-position ϵ, since we start with the instantiated head of the equation which is completely rewritten with the instantiated body. In the instantiated body, rewrites occur at recursion

positions R. Assume that on recursion position $r \in R$ the instance of the head is rewritten with an instance of the body. Then, relative to the resulting *subtree* at position r, rewrites occur again at recursion positions, e.g., at position $r' \in R$. Relative to the entire term these latter rewrites occur therefore at compositions of position r and recursion positions, e.g., at position rr' and so on. We call the infinite set of positions at which rewrites can occur in the intermediate terms within an unfolding of a recursive equation *unfolding positions*. They are determined by the recursion positions as follows:

Definition 4 (Unfolding Positions) *Let R be the recursion positions of a recursive equation G. The set of* unfolding positions *U of G is defined as the smallest set of positions which contain the position ϵ and, if $u \in U$ and $r \in R$, the position ur.*

The unfolding positions of equation *lasts* of the *lasts* RPS are $\{32, 3232, 323232, \ldots\}$.

Now we look at the variable instantiations occuring during unfolding a recursive equation. Recall the unfolding process of the *last* equation (see Table 3) described at the end of Section 2.2. The initial instantiation was $\beta_\epsilon = \beta = \{x \mapsto [a, b, c]\}$, thus in the body of the equation (replaced for the instantiated head as result of the first rewrite step), its variable is instantiated with this initial instantiation. Due to the substitution term $tail(x)$, the variable of the head in this body is instantiated with $\beta_3 = \sigma_3\,\beta_\epsilon = \{x \mapsto tail([a, b, c])\}$, i.e., the variable in the body replaced for this instantiated head is instantiated with $\sigma_3\,\beta_\epsilon$. A further rewriting step implies the instantiation $\beta_{33} = \sigma_3\,\sigma_3\,\beta_\epsilon = \sigma_3\,\beta_3 = \{x \mapsto tail(tail([a, b, c]))\}$ and so on. We index the instantiations occuring during unfolding with the unfolding positions at which the particular instantiated heads were placed. They are determined by the substitutions implied by recursive calls and an initial instantiation as follows:

Definition 5 (Instantiations in Unfoldings) *Let $G(x_1, \ldots, x_{\alpha(G)}) = t$ be a recursive equation with parameters $X = \{x_1, \ldots, x_{\alpha(G)}\}$, R and U the recursion positions and unfolding positions of G resp., σ_r the substitutions implied by the recursive call of G at position $r \in R$, and $\beta : X \to T_\Sigma$ an initial instantiation. Then a family of instantiations indexed over U is defined as $\beta_\epsilon = \beta$ and $\beta_{ur} = \sigma_r\,\beta_u$ for $u \in U, r \in R$.*

3.3 Restrictions and the Generalization Problem

An RPS which can be induced from initial terms is restricted in the following way: First, it contains no mutual recursive equations, second, there are no calls of recursive equations within calls of recursive equations (no nested recursive calls). The first restriction is not a semantical restriction, since each mutual recursive program can be transformed to an equivalent (regarding a particular algebra) non-mutual recursive program. Yet it is a *syntactical* restriction, since unfoldings of mutual RPSs can not be generalized using our approach. A restriction similar to the second one was stated by Rao (2004). He names TRSs complying with such a restriction *flat* TRSs.

Inferred RPSs conform to the following syntactical characteristics: First, all equations, potentially except of the main equation, are recursive. The main equation may be recursive as well, but, it is the only equation not required to be recursive. Second, inferred RPSs are minimal, in that (i) each equation is directly or indirectly (by means of other equations) called from the main equation, and (ii) no parameter of any equation can be omitted

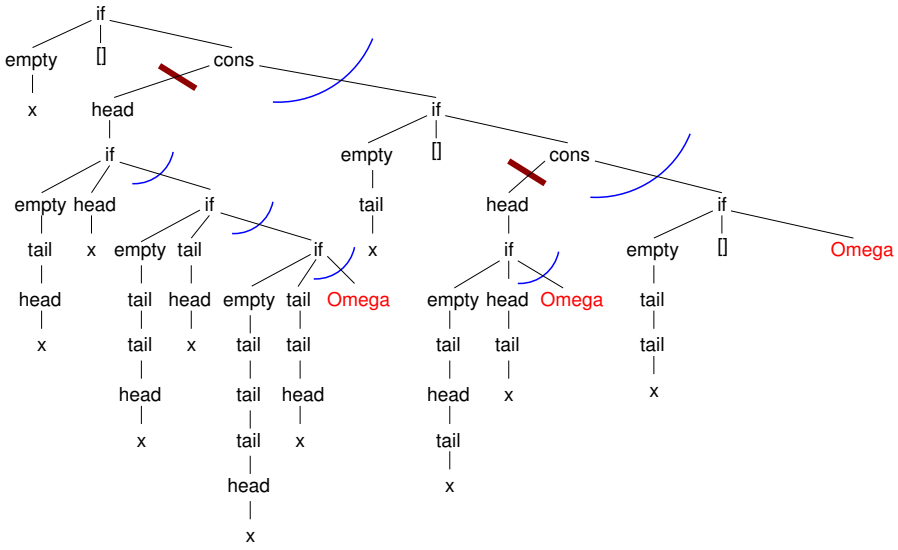

Figure 2: Initial Tree for *lasts*

without changing the free interpretation. RPSs complying with the stated restrictions and characteristics are called *minimal, non-mutual, flat recursive program schemes*.

There might be several RPSs which explain an initial term $\bar{t}$, but have different free interpretations. For example, Ω is an unfolding of *every* RPS with a recursive main equation. Therefore, an important question is which RPS will be induced. Summers (1977) required that recurrence relations hold at least over three succeeding traces and predicates to justify a generalization. A similar requirement would be that induced RPSs explain the initial terms *recurrently*, meaning that $\mathcal{I}$ contains at least one term $\bar{t}$ which can be derived from an unfolding process, in which each recursive equation had to be rewritten at least three times with its body. We use a slightly different requirement: One characteristic of minimal RPSs is, that if at least one substitution term is replaced by another, then the resulting RPS has a different free interpretation. We call this characteristic *substitution uniqueness*. Thus, it is sensible to require that induced RPSs are substitution unique regarding the initial terms, i.e., that *if* some substitution term is changed, then the resulting RPS *no longer* explains the initial terms. It holds, that a minimal RPS explains a set of initial trees recurrently, if it explains it substitution uniquely.

Thus the problem of generalizing a set of initial terms $\mathcal{I}$ to an RPS is to find an RPS which explains $\mathcal{I}$ and which is substitution unique regarding $\mathcal{I}$.

3.4 Solving the Generalization Problem

We will not state the generalization algorithm in detail in this section but we will describe the underlying concepts and the algorithm in a more informal manner. For this section and its subsections we use the term *body* of an equation for terms which are strictly speaking *incomplete* bodies: They contain only the name of the equation instead of complete recursive calls including substitution terms at recursion positions. For example, we refer to the term $\mathit{if}(\mathit{empty}(x), [], \mathit{cons}(\mathit{head}(\mathit{last}(\mathit{head}(x))), \mathit{lasts}))$ as the body for equation *lasts* of the *lasts*

RPS (see Table 3). The reason is, that we infer the complete body in two steps: First the term which we name body in this context, second the substitution terms for the recursive calls.

Generalization of a set of initial terms to an RPS is done in three successive steps, namely *segmentation of the terms*, *construction of equation bodies* and *calculation of substitution terms*. These three generalization steps are organized in a divide-and-conquer algorithm, where backtracking can occur to the divide-phase. *Segmentation* constitutes the divide-phase which proceeds top-down through the initial terms. Within this phase recursion positions (see Definition 3) and positions indicating further recursive equations are searched for each induced equation. The latter set of positions is called *subscheme positions* (see Definition 6 below). Found recursion positions imply unfolding positions (see Definition 4). As a result of the divide-phase the initial terms are divided into several parts by the subscheme positions, such that—roughly speaking—each particular part is assumed to be an unfolding of *one* recursive equation. Furthermore, the particular parts are segmented by the unfolding positions, such that—roughly speaking—each segment is assumed to be the result of *one* unfolding step of the respective recursive equation.

Consider the initial tree in Figure 2, it represents the initial term for *lasts*, shown in Table 2. The curved lines on the path to the rightmost Ω divide the tree into three segments which correspond to unfolding steps of the main equation, i.e., equation *lasts*. Note, that the rightmost segment is incomplete. The short broad lines denote two subtrees which are—except of their root *head*—unfoldings of the *last* equation. The curved lines within these subtrees divide each subtree into segments, such that each segment corresponds to one unfolding step of the *last* equation.

When the initial trees are segmented, calculation of equation bodies and of substitution terms follows within the conquer-phase. These two steps proceed bottom-up through the divided initial trees and reduce the trees during this process. The effect is, that bodies and substitution terms for each equation are calculated from trees which are unfoldings of *only* the currently induced equation and hence, each segment in these trees is an instantiation of the body of the currently induced equation. For example, for the *lasts* tree shown in Figure 2, a body and substitution terms are first calculated from the two subtrees, i.e., for the *last* equation. Since there are no further recursive equations called by the *last* equation—i.e., the segments of the two subtrees contain themselves no subtrees which are unfoldings of further equations—each segment is an instantiation of the body of the *last* equation. When this equation is completely inferred, the two subtrees are replaced by suitable instantiations of the head of the inferred *last* equation. The resulting reduced tree is an unfolding of merely *one* recursive equation, the *lasts* equation. The three segments in this reduced tree—indicated by the curved lines on the path to the rightmost Ω—are instantiations of the body of the searched for *lasts* equation. From this reduced tree, body and substitution terms for the *lasts* equation are induced and the RPS is completely induced.

Segmentations are searched for, whereas calculation of bodies and substitution terms are algorithmic. Construction of bodies always succeeds, whereas calculation of substitution terms—such that the inferred RPS explains the initial terms—may fail. Thus, an inferred RPS can be seen as the result of a search through a hypothesis space where the hypotheses are segmentations (divide-phase), and a *constructive* goal test, including construction of bodies and calculation of substitution terms (conquer-phase), which tests, whether the

completely inferred RPS explains the initial terms (and is substitution unique regarding them). In the following we describe each step in more detail:

3.4.1 Segmentation

When induction of an RPS from a set of initial trees $\mathcal{I}$ starts, the hypothesis is, that there exists an RPS with a recursive main equation which explains $\mathcal{I}$. First, recursion and subscheme positions for the hypothetical main equation G_m are searched for.

Definition 6 (Subscheme Positions) *Subscheme positions are all smallest positions in the body of a recursive equation G which denote subterms, in which calls of* further *recursive equations from the RPS appear, but* no recursive *call of equation G.*

E.g., the only subscheme position of equation *lasts* of the *lasts* RPS (Table 3) is $u = 31$. A priori, only particular positions from the initial trees come into question as recursion and subscheme positions, namely those which belong to a path leading from the root to an Ω. The reason is, that eventually *each* head of a recursive equation at any unfolding position in an intermediate term while unfolding this equation is rewritten with an Ω:

Lemma 7 (Recursion and Subscheme Positions imply Ωs) *Let $\bar{t} \in T_{\Sigma,\Omega}$ be an (incomplete) unfolding of an RPS $\mathcal{S} = (\mathcal{G}, m)$ with a recursive main equation G_m. Let R, U and S be the sets of recursion, unfolding and subscheme positions of G_m respectively. Then for all $u \in U \cap \mathbf{pos}(\bar{t})$ holds:*

1. $\mathbf{pos}(\bar{t}|_u, \Omega) \neq \emptyset$
2. $\forall s \in S$: *if* $us \in \mathbf{pos}(\bar{t})$ *then* $\mathbf{pos}(\bar{t}|_{us}, \Omega) \neq \emptyset$

It is not very difficult to see that this lemma holds. For a lack of space we do not give the proof here. It can be found in (Kitzelmann, 2003) where Lemma 7 and Lemma 9 are proven as one lemma. Knowing Lemma 7, before search starts, the initial trees can be reduced to their *skeletons* which are terms resulting from replacing subtrees without Ωs with variables.

Definition 8 (Skeleton) *The skeleton of a term $t \in T_{\Sigma,\Omega}(X)$, written* $\mathbf{skeleton}(t)$ *is the minimal pattern of t for which holds* $\mathbf{pos}(t, \Omega) = \mathbf{pos}(\mathbf{skeleton}(t), \Omega)$.

For example, consider the subtree indicated by the leftmost short broad line of the tree in Figure 2. Omitting the root *head*, it is an unfolding of the *last* equation of the *lasts* RPS shown in Table 3. Its skeleton is the *substantially* reduced term:

$$\mathit{if}(x_1, x_2, \mathit{if}(x_3, x_4, \mathit{if}(x_5, x_6, \Omega)))$$

Search for recursion and subscheme positions is done on the skeletons of the original initial trees. Thereby the hypothesis space is substantially narrowed without restricting the hypothesis language, since only those hypotheses are ruled out which are a priori known to fail the goal test.

Ωs are not only implied by recursion and subscheme positions, but also imply Ωs recursion and subscheme positions since Ωs in unfoldings result *only* from rewriting an instantiated head of a recursive equation in a term with an Ω:

Lemma 9 (Ωs imply recursion and subscheme positions) *Let $\bar{t} \in T_{\Sigma,\Omega}$ be an (incomplete) unfolding of an RPS $\mathcal{S} = (\mathcal{G}, m)$ with a recursive main equation G_m. Let R, U and S be the sets of recursion, unfolding and subscheme positions of G_m respectively. Then for all $v \in \mathbf{pos}(\bar{t}, \Omega)$ hold*

- *It exists an $u \in U \cap \mathbf{pos}(\bar{t}), r \in R$ with $u \leq v < ur$ or*
- *it exists an $u \in U \cap \mathbf{pos}(\bar{t}), s \in S$ with $us \leq v$.*

Proof: in (Kitzelmann, 2003).

From the definition of subscheme positions and the previous lemma follows, that subscheme positions are determined, if a set of recursion positions has been fixed. Lemma 7 restricts the set of positions which come into question as recursion and subscheme positions. Lemma 9 together with characteristics from subscheme positions suggests to organize the search as a search for recursion positions with a depending parallel calculation of subscheme positions. When hypothetical recursion, unfolding, and subscheme positions are determined they are checked regarding the labels in the initial trees on pathes leading to Ωs. The nodes between one unfolding position and its successors in unfoldings result from the same body (with different instantiations). Since variable instantiations only occur in subtrees at positions *not* belonging to pathes leading to Ωs, for each unfolding position the nodes between it and its successors *are necessarily equal*:

Lemma 10 (Valid Segmentation) *Let $\bar{t} \in T_{\Sigma,\Omega}$ be an unfolding of an RPS $\mathcal{S} = (\mathcal{G}, m)$ with a recursive main equation G_m. Then there exists a term $\check{t}_G \in T_{\Sigma,\Omega}(X)$ with $\mathbf{pos}(\check{t}_G, \Omega) = R \cup S$ such that for all $u \in U \cap \mathbf{pos}(\bar{t})$ hold: $\check{t}_G \leq_\Omega \bar{t}|_u$ where $\leq_\Omega$ is defined as (a) $\Omega \leq_\Omega t$ if $\mathbf{pos}(t, \Omega) \neq \emptyset$, (b) $x \leq_\Omega t$ if $x \in X$ and $\mathbf{pos}(t, \Omega) = \emptyset$, and (c) $f(t_1, \ldots, t_n) \leq_\Omega f(t'_1, \ldots, t'_n)$ if $t_i \leq_\Omega t'_i$ for all $i \in [1; n]$.*

Proof: in (Kitzelmann, 2003).

This lemma has to be slightly extended, if one allows for initial trees which are incomplete unfoldings. Lemma 10 states the requirements to assumed recursion and subscheme positions which can be assured at segmentation time. They are necessary for an RPS which explains the initial terms, yet not sufficient to assure, that an RPS complying with them exists which explains the initial trees. That is, later a backtrack can occur to search for other sets of recursion and subscheme positions. If found recursion and subscheme positions R and S comply with the stated requirements, we call the pair (R, S) a *valid segmentation*.

In our implemented system the search for recursion positions is organized as a greedy search through the space of sets of positions in the skeletons of the initial trees. When a valid segmentation has been found, compositions of unfolding and subscheme positions denote subtrees in the initial trees assumed to be unfoldings of further recursive equations. Segmentation proceeds recursively on each set of (sub)trees denoted by compositions of unfolding positions and one subscheme position $s \in S$. We denote such a set of initial (sub)trees $\mathcal{I}_s$.

3.4.2 Construction of Equation Bodies

Construction of each equation body starts with a set of initial trees $\mathcal{I}$ for which at segmentation time a valid segmentation (R, S) has been found, and an already inferred RPS for

each subscheme position $s \in S$ which explains the subtrees $\mathcal{I}_s$. These subtrees of the trees in $\mathcal{I}$ are replaced by the suitably instantiated heads or respectively bodies of the main equations of the already inferred RPSs. For example, consider the initial tree for *lasts* shown in Figure 2. When calculation of a body for the main equation *lasts* starts from this tree, an RPS containing only the *last* equation which explains all three subtrees indicated by the short broad lines has already been inferred. The initial tree is reduced by replacing these three subtrees by suitable instantiations of the head of the *last* equation. We denote the set of reduced initial trees also with $\mathcal{I}$ and its elements also with $\bar{t}$. By reducing the initial trees based on already inferred recursive equations, the problem of inducing a set of recursive equations is reduced to the problem of inducing merely *one* recursive equation (where the recursion positions are already known from segmentation).

An equation body is induced from the segments of an initial tree which is assumed to be an unfolding of one recursive equation.

Definition 11 (Segments) *Let $\bar{t}$ be an initial tree, R a set of (hypothetical) recursion positions and U the corresponding set of unfolding positions. The set of* complete segments *of $\bar{t}$ is defined as: $\{\bar{t}|_u[R \leftarrow G] \mid u \in U \cap \mathbf{pos}(\bar{t}), R \subset \bar{t}|_u\}$*

For example, consider the subtree indicated by the leftmost short broad line of the initial tree in Figure 2 without its root *head*. It is an unfolding of the *last* equation as stated in Table 3. When the only recursion position 3 has been found it can be splitted into three segments, indicated by the curved lines:

1. $\text{if}(\text{empty}(\text{tail}(\text{head}(x))), \text{head}(x), G)$
2. $\text{if}(\text{empty}(\text{tail}(\text{tail}(\text{head}(x)))), \text{tail}(\text{head}(x)), G)$
3. $\text{if}(\text{empty}(\text{tail}(\text{tail}(\text{tail}(\text{head}(x))))), \text{tail}(\text{tail}(\text{head}(x))), G)$

Expressed according to segments, the fact of a repetitive pattern between unfolding positions (see Lemma 10) becomes the fact, that the sequences of nodes between the root and each G are equal for each segment. Each segment is an instantiation of the body of the currently induced equation. In general, the body of an equation contains other nodes among those between its root and the recursive calls. These further nodes are also equal in each segment. Differences in segments of unfoldings of a recursive equation can *only* result from different instantiations of the variables of the body. Thus, for inducing the body of an equation from segments, we assume each position in the segments which is equally labeled in *all* segments as belonging to the body of the assumed equation, but each position which is variably labeled in at least two segments as belonging to the instantiation of a variable. This assumption can be seen as an inductive bias since it might occur, that also positions which are equal over all segments belong to a variable instantiation. Nevertheless it holds, that if an RPS exists which explains a set of initial trees, then there also exists an RPS which explains the initial trees and is constructed based on the stated assumption. Based on the stated assumption, the body of the equation to be induced is determined by the segments and defined as follows:

Definition 12 (Valid Body) *Given a set of reduced initial trees, the most specific maximal pattern of all segments of all the trees is called* valid body *and denoted $\hat{t}_G$.*

The maximal pattern of a set of terms can be calculated by first order anti-unification (Plotkin, 1969).

Calculating a valid body regarding the three segments enumerated above results in the term $\mathit{if}(\mathit{empty}(\mathit{tail}(x)), x, G)$. The *different* subterms of the segments are assumed to be instantiations of the parameters in the calculated valid body. Since each segment corresponds to one unique unfolding position, instantiations of parameters in unfoldings as defined in Definition 5 are now given. For example, from the three segments enumerated above we obtain:

1. $\beta_\epsilon(x) = \mathrm{head}(x)$
2. $\beta_3(x) = \mathrm{tail}(\mathrm{head}(x))$
3. $\beta_{33}(x) = \mathrm{tail}(\mathrm{tail}(\mathrm{head}(x)))$

3.4.3 Inducing Substitution Terms

Induction of substitution terms for a recursive equation starts on a set of reduced initial trees which are assumed to be unfoldings of one recursive equation, an already inferred (incomplete) equation body which contains only a G at recursion positions, and variable instantiations in unfoldings according to Definition 5. The goal is to complete each occurence of G to a recursive call including substitution terms for the parameters of the recursive equation.

The following lemma follows from Definition 5 and states characteristics of parameter instantiations in unfoldings more detailed. It characterizes the instantiations in unfoldings against the substitution terms of a recursive equation considering each single position in them.

Lemma 13 (Instantiations in Unfoldings) *Let $G(x_1, \ldots, x_{\alpha(G)}) = t$ be a recursive equation with parameters $X = \{x_1, \ldots, x_{\alpha(G)}\}$, recursion positions R and unfolding positions U, $\beta : X \to T_\Sigma$ an instantiation, σ_r substitution terms for each $r \in R$ and β_u instantiations as defined in Definition 5 for each $u \in U$. Then for all $i, j \in [1; \alpha(G)]$ and positions v hold:*

1. *If $(x_i\,\sigma_r)|_v = x_j$ then for all $u \in U$ hold $(x_i\beta_{ur})|_v = x_j\beta_u$.*
2. *If $(x_i\,\sigma_r)|_v = f((x_i\,\sigma_r)|_{v1}, \ldots, (x_i\,\sigma_r)|_{vn}), f \in \Sigma, \alpha(f) = n$ then for all $u \in U$ hold $\mathbf{node}(x_i\beta_{ur}, v) = f$.*

We can read the implications stated in the lemma in the inverted direction and thus we get almost immediately an algorithm to calculate the substitution terms of the searched for equation from the known instantiations in unfoldings.

One interesting case is the following: Suppose a recursive equation, in which at least one of its parameters *only* occurs within a recursive call in its body, for example the equation $G(x, y, z) = \mathit{if}(\mathit{zerop}(x), y, +(x, G(\mathit{prev}(x), z, \mathit{succ}(y))))$ in which this is the case for parameter z.[1] For such a variable no instantiations in unfoldings are given when induction of substitution terms starts. Also such variables are not contained in the (incomplete) valid

1. A practical example is the *tower-of-hanoi*-problem.

equation body. Our generalizer introduces them each time, when none of the both implications of Lemma 13 hold. Then it is assumed, that the currently induced substitution term contains such a "hidden" variable at the current position. Based on this assumption the instantiations in unfoldings of the hidden variable can be calculated and the inference of subtitution terms for it proceeds as described for the other parameters.

When substitution terms have been found, it has to be checked, whether they are substitution unique with regard to the reduced initial terms. This can be done for each substitution term that was found separately.

3.4.4 Inducing an RPS

We have to consider two further points: The first point is that segmentation presupposes the initial trees to be explainable by an RPS with a *recursive* main equation. Yet in Section 3.3 we characterized the inferable RPSs as liberal in this point, i.e., that also RPSs with a non-recursive main equation are inferable. In such a case, the initial trees contain a constant (not repetitive) part at the root such that no recursion positions can be found for these trees (as for example the three subtrees indicated by the short broad lines in Figure 2 which contain the constant root *head*). In this case, the root node of the trees is assumed to belong to the body of a non-recursive main equation and induction of RPSs recursively proceeds at each subtree of the root nodes.

The second point is that RPSs explaining the subtrees which are assumed to be unfoldings of further recursive equations at segmentation time are already inferred. Based on these already inferred RPSs, the initial trees are reduced and then a body and substitution terms are induced. Calculation of a body always succeeds, whereas calculation of substitution terms may fail. To deal with induction of RPSs explaining the subtrees as an independent problem requires, that *if* there exists a set of RPSs explaining the subtrees such that substitution terms can be calculated then substitution terms can be calculated for *any* set of RPSs explaining the subtrees.

Fortunately we could prove, that this requirement holds provided the main equation is constructed according to the "maximal-body" principle (see Definition 12). A proof sketch is as follows: Assume there are two different RPSs explaining the subtrees of a fixed subscheme position. Provided the main equations of the two RPSs are constructed according to the "maximal-body" principle, one can prove that the main equations of both RPSs have the same number of parameters with the same instantiations for explaining the subtrees (see Schmid, 2003, page 203, Theorem 7.3.3). Though the main equations of the RPSs might be different in their non-parameter positions, it is then assured that induction of the current equation will succeed for either both of the two different RPSs or for none of them but not for only one. The reason is that the possibly different non-parameter positions only affect the calculation of the body which always succeeds and that the critical point of inferring substitution terms is only affected by the parameters of the main equations of the RPSs and their instantiations.

4. Generating an Initial Term

Our theory and prototypical implementation for the first synthesis step uses the datatype *List*, defined as follows: The empty list [] is an (α-)list and if a is in element of type α and

l is an α-list, then $cons(a, l)$ is an α-list. Lists may contain lists, i.e., α may be of type *List* α'.

4.1 Characterization of the Approach

The constructed initial terms are composed from the list constructor functions $[], cons$, the functions for decomposing lists $head, tail$, the predicate *empty* testing for the empty list, one variable x, the 3ary (non-strict) conditional function *if* as control structure, and the bottom constant Ω meaning *undefined*. Similar to Summers (1977), the set of functions used in our term construction approach implies the restriction of induced programs to solve *structural* list programs. An extension to Summers is that we allow the example inputs to be *partially* ordered instead of only totally ordered. This is related to the extension of inducing *sets* of recursive equations as described in Section 3 instead of only one recursive equation.

We say that an initial term *explains* I/O-examples, if it evaluates to the specified output when applied to the respective input or to *undefined*. The goal of the first synthesis step is to construct an initial term which explains a set of I/O-examples and which can be explained by an RPS.

4.2 Basic Concepts

Definition 14 (Subexpressions) *The set of subexpressions of a list l is defined to be the smallest set which includes l itself and, if l has the form $cons(a, l')$, all subexpressions of a and of l'. If a is an atom, then a itself is its only subexpression.*

Since *head* and *tail*—which are defined by $head(cons(a, l)) = a$ and $tail(cons(a, l)) = l$—decompose lists uniquely, each subexpression can be associated with the unique term which computes the subexpression from the original list. E.g., consider the list $[[a], [b]]$. The set of all subexpressions together with their associated terms is: $\{x = [[a], [b]],\ head(x) = [a],\ tail(x) = [[b]],\ head(head(x)) = a,\ tail(head(x)) = [],\ head(tail(x)) = [b],\ tail(tail(x)) = [],\ head(head(tail(x))) = b,\ tail(head(tail(x))) = []\}$.

Since lists are uniquely constructed by the constructor functions $[]$ and *cons*, traces which compute the specified output can uniquely be constructed from the terms for the subexpressions of the respective input:

Definition 15 (Construction of Traces) *Let $i \mapsto o$ be an I/O-pair (i is a list). If o is a subexpression of i, then the trace is defined to be the term associated with o. Otherwise o has the form $cons(a, l)$. Let t and t' be the traces for the I/O-pairs $i \mapsto a$ and $i \mapsto l$ respectively. The trace for $i \mapsto o$ is defined to be the term $cons(t, t')$.*

For example, the trace for computing (the example-output) $[a, b]$ from (the example-input) $[[a], [b]]$ is the term $cons(head(head(x)), head(tail(x)))$.

Similar to Summers, we discriminate the inputs with respect to their structure, more precisely with regard to a partial order over them implied by their structural complexity. As stated above, we allow for arbitrarily nested lists as inputs. A partial order over such lists is given by: $[] \leq l$ for all lists l and $cons(a, l) \leq cons(a', l')$, iff $l \leq l'$ and, if a and a' are again lists, $a \leq a'$.

Consider any unfolding of an RPS. Generally it holds, that greater positions on a path leading to an Ω result from more rewritings of a head of a recursive equation with its body compared to some smaller position. In other words, the computation represented by a node at a greater position is one on a deeper recursion level than a computation represented by a smaller position. Since we use only the complexity of an input list as criterion whether the recursion stops or whether another call appears with the input decomposed in some way, deeper recursions result from more complex inputs in the induced programs.

4.3 Solving the Term Construction Problem

The overall concept of constructing the initial tree is to introduce the nodes from the traces position by position to the initial tree *as long as the traces are equal* and to introduce an *if*-expression as soon as at least two (sub)traces differ. The predicate in the *if*-expression divides the inputs into two sets. The "then"-subtree is recursively constructed from the input/trace-pairs whose inputs evaluate to *true* with the predicate and the "else"-subtree is recursively constructed from the other input/trace-pairs. Eventually only one single input/trace-pair remains when an *if*-expression is introduced. In this case an Ω indicating a recursive call on this path is introduced as leaf at the current position in the initial term and (this subtree of) the initial tree is finished. The reason for introducing an Ω in this case is, that we assume, that *if* the input/trace-set would contain a pair with a more complex input, than the respective trace would at some position differ from the remaining trace and thus it would imply an *if*-expression, i.e., a recursive call at some deeper position. Since we do not know the position at which this difference would occur, we can not use this single trace, but have to indicate a recursive call on this path by an Ω. Thus, for principal reasons, the constructed initial terms are undefined for the most complex inputs of the example set. There are two consequences of this particular loss of information in the initial terms compared to the I/O-examples. Since the following generalization step is based on the initial terms (1) the neccessary number of examples increases and (2) *if* the generalized program is incorrect it could especially be incorrect for the most complex examples. Thus consistence of the induced programs with respect to the I/O-examples is generally only assured for all examples except of the most complex ones.

We now consider the both cases that all roots of the traces are equal and that they differ respectively more detailed.

4.3.1 Equal Roots

Suppose all generated traces have the same root symbol. In this case, this symbol constitutes the root of the initial tree. Subsequently the sub(initial)trees are calculated through a recursive call to the algorithm. Suppose the initial tree has to explain the I/O-examples $\{[a] \mapsto a,\ [a,b] \mapsto b,\ [a,b,c] \mapsto c\}$. Calculating the traces and replacing them for the outputs yields the input/trace-set $\{[a] \mapsto head(x),\ [a,b] \mapsto head(tail(x)),\ [a,b,c] \mapsto head(tail(tail(x)))\}$. All three traces have the same root *head*, thus we construct the root of the initial tree with this symbol. The algorithm for constructing the initial subterm of the constructed root *head* now starts recusively on the set of input/trace-pairs where the traces are the subterms of the roots *head* from the three original traces, i.e., on the set $\{[a] \mapsto x,\ [a,b] \mapsto tail(x),\ [a,b,c] \mapsto tail(tail(x))\}$.

The traces from these new input/trace-set have different roots, that is, an *if*-expression is introduced as subtree of the constructed initial tree.

4.3.2 Introducing Control Structure

Suppose the traces (at least two of them) have different roots, as for example the traces of the second input/trace-set in the previous subsection. That means that the initial term has to apply different computations to the inputs corresponding to the different traces. This is done by introducing the conditional function *if*, i.e., the root of the initial term becomes the function symbol *if* and contains from left to right three subtrees: First, a predicate term with the predicate *empty* as root to distinguish between the inputs which have to be computed differently with regard to their complexity; second, a tree explaining all I/O-pairs whose inputs are evaluated to *true* from the predicate term; third, a tree explaining the remaining I/O-examples. It is presupposed, that all traces corresponding to inputs evaluating to *true* with the predicate are equal. These equal subtraces become the second subtree of the *if*-expression, i.e., they are evaluated, if an input evaluates to *true* with the predicate. That means that never an Ω occurs in a "then"-subtree of a constructed initial tree, i.e., that recursive calls in the induced RPSs may only occur in the "else"-subtrees. For the "else"-subtree the algorithm is recursively processed on all remaining input/trace-pairs.

For the predicate must hold that it evaluates to *true* for the least complex inputs because the "then"-subtree represents the termination of recursion whereas the "else"-subtree represents a further recursive call (for more complex inputs) of the induced program. An algorithm for calculating predicates evaluating to *true* for a particular expression and to *false* for any more complex expression can be found in (Smith, 1984, page 310). If, for example, the two input lists $[a, b]$ and $[a, b, c]$ shall be distinguished then the predicate is $empty(tail(tail(x)))$. For more complex data types as for example trees, or for nested lists, calculation of predicates might not be unique. Then a strategy for chosing a predicate has to be applied.

5. Experimental Results

We have implemented prototypes (without any thoughts about efficiency) for both described steps, construction of the initial tree and generalization to an RPS. The implementations are in Common-Lisp. In Table 4 we have listed experimental results for a few sample problems. Due to the restrictions of the first synthesis step all these induced programs deal with structural problems and are composed of only the primitive functions stated in Section 4.1. Many interesting programs, as for example *quicksort* or *towers-of-hanoi*, do not meet these restrictions and are not regarded. Due to the restriction of the second synthesis step all these programs contain no nested recursive calls. The first column lists the names for the induced functions, the second column lists the number of given I/O-pairs, the third column lists the total number of induced equations and in parentheses the number of induced *recursive* equations, and the fourth column lists the times consumed by the first step, the second step, and the total time respectively. The experiments were performed on a Pentium 4 with Linux and the program runs are interpreted with the *clisp* interpreter.

All induced programs compute the intended function. The number of given examples is in each case the minimal one. When given one example less, the system does *not* produce

function	#expl	#eqs(#rec)	times in sec
last	4	2(1)	.003 / .001 / .004
unpack	4	1(1)	.003 / .002 / .005
init	4	1(1)	.004 / .002 / .006
evenpos	7	2(1)	.01 / .004 / .014
switch	6	1(1)	.012 / .004 / .016
lasts	6	2(2)	.014 / .015 / .029
shift	6	3(2)	.015 / .033 / .048
mult-lasts	6	3(3)	.023 / .21 / .233
reverse	6	4(3)	.031 / .422 / .453
multi	12	5(5)	.114 / 6.96 / 7.074

Table 4: Some inferred functions

an *unintended* program, but produces *no* program. Indeed, an initial term is produced in such a case which is consistent with the example set, but no RPS is generalized, because it exists no RPS which explains the initial term *and is substitution unique with regard to it* (see Section 3.3).

last computes the last element of a list. The main equation is not recursive and only applies a *head* to the result of the induced recursive equation which computes a one element list containing the last element of the input list. *unpack* produces an output list, in which each element from the input list is encapsulated in a one element list, e.g., $unpack([a, b, c]) = [[a], [b], [c]]$. *unpack* is the classical example in (Summers, 1977). *init* returns the input list without the last element. *evenpos* computes a list containing each second element of the input list. The main equation is not recursive and only deals with the empty input list as special case. *switch* returns a list, in which each two successive elements of the input list are on switched positions, e.g., $switch([a, b, c, d, e]) = [b, a, d, c, e]$. *lasts* is the program described in Section 2. The given I/O-examples are those from Table 1. *shift* moves the last element of the input list to the front of the list. The main equation is not recursive and only deals with the empty list and a one-element-list as special cases. The two induced recursive equations compute the last element and the *init* of the input list respectively and are combined to compute the *shift* function. *mult-lasts* takes a list of lists as input just like *lasts*. It returns a list of the same structure as the input list where each inner list contains repeatedly the last element of the corresponding inner list from the input. For example, $mult\text{-}lasts([[a, b], [c, d, e], [f]]) = [[b, b], [e, e, e], [f]]$. All three induced equations are recursive. The third equation computes a one element list containing the last element of an input list. The second equation calls the third equation and returns a list of the same structure as a given input list where the elements of the input list are replaced by the last element. The first equation calls the second equation to compute the inner lists. *reverse* reverses a list. The induced program has an unusual form, nevertheless it is correct. Finally *multi* is a combination of *mult-lasts*, *unpack*, and *switch*. It takes a list of lists as input and applies *mult-lasts* to the first list, *unpack* to the second list, *switch* to the third list, and then again *mult-lasts* to the fourth list, *unpack* to the fifth list, *switch* to the sixth list and so on. *multi* is in one run induced from the examples shown in Table 5. The induced program is

$$
\begin{array}{rcl}
[] & \mapsto & [], \\
[[a]] & \mapsto & [[a]], \\
[[a,b]] & \mapsto & [[b,b]], \\
[[a,b,c],[d]] & \mapsto & [[c,c,c],[[d]]], \\
[[a,b,c,d],[e,f],[g]] & \mapsto & [[d,d,d,d],[[e],[f]],[g]], \\
[[a,b,c,d,e],[f,g,h],[i,j]] & \mapsto & [[e,e,e,e,e],[[f],[g],[h]],[j,i]], \\
[[a],[b,c,d,e],[f,g,h],[i]] & \mapsto & [[a],[[b],[c],[d],[e]],[g,f,h],[i]], \\
[[a],[b],[c,d,e,f],[g,h]] & \mapsto & [[a],[[b]],[d,c,f,e],[h]], \\
[[a],[b],[c,d,e,f,g],[h],[i]] & \mapsto & [[a],[[b]],[d,c,f,e,g],[h],[[i]]], \\
[[a],[b],[c,d,e,f,g,h],[i],[j,k],[l]] & \mapsto & [[a],[[b]],[d,c,f,e,h,g],[i],[[j],[k]],[l]], \\
[[a],[b],[c],[d],[e],[f,g]] & \mapsto & [[a],[[b]],[c],[d],[[e]],[g,f]], \\
[[a],[b],[c],[d],[e],[f],[g]] & \mapsto & [[a],[[b]],[c],[d],[[e]],[f],[g]]
\end{array}
$$

Table 5: I/O-examples for *multi*

shown in Table 6. Note that the names *multi, switch, unpack* etc. of the equations of the induced RPS are ex post introduced from us; the system introduces names $G_1, G_2, \ldots$.

$$
\begin{array}{rl}
multi(x) = & \mathrm{if}(\mathrm{empty}(x), [], \mathrm{cons}(multlasts(\mathrm{head}(x)), \\
& \quad \mathrm{if}(\mathrm{empty}(\mathrm{tail}(x)), [], \mathrm{cons}(unpack(\mathrm{head}(\mathrm{tail}(x))), \\
& \quad\quad \mathrm{if}(\mathrm{empty}(\mathrm{tail}(\mathrm{tail}(x))), [], \mathrm{cons}(switch(\mathrm{head}(\mathrm{tail}(\mathrm{tail}(x)))), \\
& \quad\quad\quad multi(\mathrm{tail}(\mathrm{tail}(\mathrm{tail}(x))))))))))
\end{array}
$$

$$
\begin{array}{rl}
switch(x) = & \mathrm{if}(\mathrm{empty}(\mathrm{tail}(x)), x, \mathrm{cons}(\mathrm{head}(\mathrm{tail}(x)), \mathrm{cons}(\mathrm{head}(x), \\
& \quad \mathrm{if}(\mathrm{empty}(\mathrm{tail}(\mathrm{tail}(x))), [], switch(\mathrm{tail}(\mathrm{tail}(x)))))))
\end{array}
$$

$$
unpack(x) = \mathrm{cons}(\mathrm{cons}(\mathrm{head}(x), []), \mathrm{if}(\mathrm{empty}(\mathrm{tail}(x)), [], unpack(\mathrm{tail}(x))))
$$

$$
multlasts(x) = \mathrm{if}(\mathrm{empty}(\mathrm{tail}(x)), x, \mathrm{cons}(\mathrm{head}(last(x)), multlasts(\mathrm{tail}(x))))
$$

$$
last(x) = \mathrm{if}(\mathrm{empty}(\mathrm{tail}(\mathrm{tail}(x))), \mathrm{tail}(x), last(\mathrm{tail}(x)))
$$

Table 6: Recursive Program Scheme for *multi*

Considering the times taken by the first and second synthesis step for the problems listed in Table 4 one finds (1) that they depend on the number of examples for the first step and on the number of recursive equations for the second step and (2) that the times taken from the second step increase faster than the times taken from the first step. A detailed analysis of the complexities of the two synthesis steps has still to be done. For some results regarding the second step see (Schmid, 2003, Section 7.4.1).

6. Comparison with Other Inductive Programming Systems

Inductive learning of programs is in general primarily known from the field of inductive logic programming (ILP), where the target language is relational descriptions in form of logic programs, e.g., Prolog programs. However the usual goal of ILP is learning *concepts* in form of a single, non-recursive predicate but not learning *recursive algorithms* with multiple interdependent predicates. Nevertheless there are ILP systems that have reasonable behaviour on inducing recursive logic programs, GOLEM (Muggleton and Feng, 1990) as an example. One interactive ILP system *specializing* in synthesizing recursive programs is DIALOGS (Flener, 1997). For a comparison of different ILP systems specializing in learning recursive predicates see (Flener and Yilmaz, 1999). More recent approaches to learn recursive logic programs are the approach of Rao and Sattar (2001) and the system ATRE (Malerba, 2003; Berardi et al., 2004). Two non-ILP systems for inducing recursive programs are the evolutionary computation system ADATE (Automatic Design of Algorithms Through Evolution) (Olsson, 1995) which induces functional programs in Standard ML and the Optimal Ordered Problem Solver (OOPS) (Schmidhuber, 2004). All these systems and approaches differ in their induction strategy, in the training data (many examples vs. few examples, only positive vs. both positive and negative examples, I/O-examples vs. example-inputs together with an evaluation function), in whether the induction relies on background knowledge, and in the limitations regarding inducable programs.

Our approach is different from most of the other approaches in that it is mostly analytical instead of search-based. In the following, we discuss this difference considering our system, ADATE, the well known ILP system FOIL (Quinlan, 1990) which was extended with concepts to learn recursive clauses (Cameron-Jones and Quinlan, 1993), and the Optimal Ordered Problem Solver. FOIL as well as ADATE and our system are capable of inducing more than one recursive function/clause in one run. FOIL needs a specification for *every* clause it shall induce, whereas ADATE and our system are capable of automatically introduce auxiliary recursive functions and thereby auxiliary parameters. E.g., one can give a specification of reversing a list to our system in terms of the I/O-examples $\{[] \mapsto [],\ [a] \mapsto [a],\ [a,b] \mapsto [b,a],\ [a,b,c] \mapsto [c,b,a],\ [a,b,c,d] \mapsto [d,c,b,a]\}$ and it automatically introduces an auxiliary function containing the second accumulating variable. When ADATE or our system outputs more than one recursive function these functions clearly are interdependent. In contrast, when different predicates to learn in one run are specified in FOIL, they are mostly learned independently one after another though foremost learned predicates can be used as background knowledge for the remaining predicates. FOIL has no knowledge of structured datatypes, e.g. lists, on its own and actually can handle only atoms. Thus lists have to be simulated with constants and one has to specify procedures for "composing" and "decomposing" such simulated lists as background knowldedge.

FOIL and ADATE directly search through a hypothesis space, whereas our system deterministically constructs an explanation of the I/O-examples in a first step and only then searches a hypothesis space for a generalization of the explanation. The main effect regarding this difference is that FOIL and ADATE can be given any background knowledge in terms of additional predicate specifications in the case of FOIL and predefined SML functions in the case of ADATE respectively. These predicates or functions respectively are then used in the synthesized programs. Since the branching factor in the search spaces

grows as this background knowledge increases, increasing background knowledge supposably tends to result in increasing run times. In contrast—though our generalization component is domain independent—, our system on the whole is restricted to background knowledge that admits an almost deterministic explanation of the I/O-examples. Therefore it cannot be given any predefined functions to be used in a synthesized program. Until now, synthesized programs can only be composed of the predefined functions stated in Section 4.1. Since the particular knowledge of datatypes admits deterministic explanations, it is used to *restrict* the hypothesis search space.

It would be interesting to compare the run times of FOIL, ADATE, and our system. However, since the systems have different restrictions, it is not trivial to find adequate and significant problems and specifications for a comparison. The restrictions of FOIL—no handling with structured datatypes and no automatic introduction of auxiliary predicates and variables—could be dealt with by simulating lists and by specifying all needed predicates. On the other side, the restrictions of our system—only particular primitive functions can be used in the synthesized programs—cannot be bypassed at present. For problems which need only few predicates/functions as background knowledge and contain only one recursive predicate/function as for example *last* or *member*, FOIL as well as our system take less than one second on a Pentium 4 with Linux. We have not measured the run times of the ADATE system for these simple problems, but on the web pages of the ADATE system[2] Roland Olsson reports on 570 seconds on a 200MHz PentiumPro for reversing a list.

Like FOIL and ADATE, the Optimal Ordered Problem Solver is based on a "generate-and-test" method. In (Schmidhuber, 2004), inducing a recursive program for *towers-of-hanoi* is reported. The induction takes a few days on a personal computer.

It is theoretically plausible as well as empirically evident that higher generality of inducable programs leads to higher computational effort of the program synthesizer. ADATE as well as OOPS are highly general program synthesizers with high run times. On the other extreme is our system with strong restrictions regarding synthesizable programs but much faster program inductions. An interesting question is, whether it could be possible to combine both approaches. We think that one approach to combine both methods could be to generate the traces and predicates with some "generate-and-test" method and then generalizing the integrated initial terms with our generalization algorithm. This would overcome the restrictions to structural problems as well as to restricted background knowledge which are implied by our analytical trace and predicate construction method. On the other hand, by keeping the construction and generalization of traces one would (1) presumably hold some advantage regarding run time compared to pure "generate-and-test" algorithms because searching for traces is less elaborate than searching for a recursive program and (2) hold the important point of constructing programs which are assured to terminate. Thus this could be a good compromise regarding the conflicting aspects of generality of the induced programs, computational effort of the induction algorithm, and assurance of termination for the induced programs.

2. http://www-ia.hiof.no/~rolando/

7. Conclusion and Further Research

We presented an EBG approach to inducing sets of recursive equations representing functional programs from I/O-examples. The underlying methodologies are inspired by classical approaches to induction of functional Lisp-programs, particularly by the approach of Summers (1977). The presented approach goes in three main aspects beyond Summers' approach: *Sets* of recursive equations can be induced at once instead of only one recursive equation, each equation may contain more than one recursive call, and additionally needed parameters are introduced systematically. We have implemented prototypes for both steps. The generalizer works domain-independent and all problems which comply to our general program scheme (Definition 1) with the restrictions described in Section 3.3 can be solved, whereas construction of initial terms as described in Section 4 relies on knowledge of datatypes.

We are investigating several extensions for the first synthesis step: First, we try to integrate knowledge about further datatypes such as trees and natural numbers. For example, we believe, that if we introduce *zero* and *succ*, denoting the natural number 0 and the successor function resp. as constructors for natural numbers, *prev* for "decomposing" natural numbers and the predicate *zerop* as bottom test on natural numbers, then it should be possible to induce a program returning the length of a list for example. Another extension will be to allow for more than one input parameter in the I/O-examples, such that *append* becomes inducable for example. A third extension should be the ability to use user-defined or in a previous step induced functions within an induction step.

Until now our approach suffers from the restriction to structural problems due to the principal approach to calculate traces deterministically without search in the first synthesis step. We work on overcoming this restriction, i.e., on extending the first synthesis step to the ability of dealing with problems which are not (only) structural, list sorting for example. A strong extension to the second step would be the ability to deal with nested recursive calls, yet this would imply a much more complex structural analysis on the initial terms.

Acknowledgments

We would like to acknowledge previous work from Martin Mühlpfordt and Fritz Wysotzki. Martin Mühlpfordt implemented the second synthesis step. We also like to thank three anonymous reviewers for their very helpful comments and suggestions for improving an earlier draft of this paper.

References

M. Berardi, A. Varlaro, and D. Malerba. On the effect of caching in recursive theory learning. In R. Camacho, R. D. King, and A. Srinivasan, editors, *Inductive Logic Programming: ILP 2004*, pages 44–62. Springer, 2004.

A. W. Biermann, G. Guiho, and Y. Kodratoff, editors. *Automatic Program Construction Techniques*. Collier Macmillan, 1984.

R. Mike Cameron-Jones and J. Ross Quinlan. Avoiding pitfalls when learning recursive theories. In *IJCAI*, pages 1050–1055. Morgan Kaufmann, 1993.

N. Dershowitz and J.-P. Jouanaud. Rewrite systems. In J. Leeuwen, editor, *Handbook of Theoretical Computer Science*, volume B. Elsevier, 1990.

P. Flener. Inductive logic program synthesis with DIALOGS. In S. Muggleton, editor, *Proceedings of ILP'96*, pages 175–198. Springer, 1997.

P. Flener and D. Partridge. Inductive programming. *Autom. Softw. Eng.*, 8(2):131–137, 2001.

P. Flener and S. Yilmaz. Inductive synthesis of recursive logic programs: Achievements and prospects. *Journal of Logic Programming*, 41(2–3):141–195, 1999.

E. Mark Gold. Language identification in the limit. *Information and Control*, 10(5):447–474, 1967.

E. Kitzelmann. Inductive functional program synthesis – a term-construction and folding approach. Master's thesis, Dept. of Computer Science, TU Berlin, 2003. http://www.cogsys.wiai.uni-bamberg.de/kitzelmann/documents/thesis.ps.

M. L. Lowry and R. D. McCarthy. *Autmatic Software Design*. MIT Press, Cambridge, Mass., 1991.

D. Malerba. Learning recursive theories in the normal ILP setting. *Fundamenta Informaticae*, 57(1):39–77, 2003.

S. Muggleton and L. De Raedt. Inductive logic programming: Theory and methods. *Journal of Logic Programming, Special Issue on 10 Years of Logic Programming*, 19-20:629–679, 1994.

S. H. Muggleton and C. Feng. Efficient induction of logic programs. In *Proceedings of the First Conference on Algorithmic Learning Theory*, pages 368–381, Tokyo, 1990. Ohmsha.

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

G. D. Plotkin. A note on inductive generalization. In *Machine Intelligence*, volume 5, pages 153–163. Edinburgh University Press, 1969.

J. Ross Quinlan. Learning logical definitions from relations. *Machine Learning*, 5:239–266, 1990.

M. R. K. Krishna Rao. Inductive inference of term rewriting systems from positive data. In *Algorithmic Learning Theory*, pages 69–82, 2004.

M. R. K. Krishna Rao and A. Sattar. Polynomial-time learnability of logic programs with local variables from entailment. *Theoretical Computer Science*, 268(2):179–198, 2001.

U. Schmid. *Inductive Synthesis of Functional Programs – Universal Planning, Folding of Finite Programs, and Schema Abstraction by Analogical Reasoning*. Springer, 2003.

U. Schmid and F. Wysotzki. Applying inductive programm synthesis to macro learning. In *Proc. 5th International Conference on Artificial Intelligence Planning and Scheduling (AIPS 2000)*, pages 371–378. AAAI Press, 2000.

J. Schmidhuber. Optimal ordered problem solver. *Machine Learning*, 54(3):211–254, 2004.

D. R. Smith. The synthesis of LISP programs from examples: A survery. In A. W. Biermann, G. Guiho, and Y. Kodratoff, editors, *Automatic Program Construction Techniques*, pages 307–324. Macmillan, 1984.

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

L. G. Valiant. A theory of the learnable. In *STOC '84: Proceedings of the sixteenth annual ACM symposium on Theory of computing*, pages 436–445, New York, NY, USA, 1984. ACM Press.

F. Wysotzki and U. Schmid. Synthesis of recursive programs from finite examples by detection of macro-functions. Technical Report 01-2, Dept. of Computer Science, TU Berlin, Germany, 2001.