Synthesis of Recursive Functions with Interdependent Parameters

Martin Mühlpfordt and Ute Schmid*

*Methods of Artificial Intelligence, Computer Science Department, Sekr. Fr 5–8, Technical University Berlin, Franklinstr. 28/29, D-10587 Berlin, Germany, Email: schmid@cs.tu-berlin.de

Abstract. We present a methodology for the inductive synthesis of recursive functions based upon the theoretical framework of context-free tree grammars. The synthesis task is splitted into two parts: First, a small set of positive input/output examples is transformed into a straight-forward ("initial") program by means of heuristic search; second, the straight-forward program is generalized to a recursive function. In this paper we concentrate on the second part of the synthesis task. That is, we deal with the problem of infering recurrence relations from straight-forward programs. We will describe our theoretical framework and propose an induction algorithm. Straight-forward programs are regarded as elements of some term algebra with unknown elements but with known rules for forming syntactically correct expressions. The synthesis task than is to fold such a term into a recursive program scheme (RPS). The algorithm works purely syntactically - without information about number and ordering of parameters of the desired program. In a cyclic pattern-matching process we identify the structure ("skeleton") of the program, its parameters and their substitutions. The skeleton is identified as constant subterm occuring in a regular way; subterms which change in a regular way are identified as parameters together with a substitution. We present a technique for detecting substitutions which are interdependent between parameters. Thereby, we can infer a greater class of recursive functions than standard generalization-to-n techniques.

Keywords. inductive program synthesis, grammatical inference, context free tree grammers

1 Introduction

Inductive synthesis of recursive programs from examples was a major research topic in the seventies and eighties. Typically, the proposed algorithms had their background in the domain of functional (LISP) programming and relied on small sets of positive input/output examples [13, 9, 2]. In the nineties inductive program synthesis has become of interest again in the context of inductive logic programming (ILP) [10, 11]. As in the classical approaches, our aim is to infer functional expressions (i.e. terms and not logical clauses) from small sets of positive input/output examples. But, in contrast to both the functional and ILP approach, our algorithm works independed of the syntax of some given program language (as LISP or PROLOG). Instead, our work is based upon term algebras [2] and recursive program schemes [4, 14]. Of course - by interpreting the symbols of a given term algebra with respect to some programming language - concrete programs can be generated.

In [12] we described the basic ideas of our methodology. We propose to split the program synthesis task in two distinct parts: In a first step, we construct straight-forward ("initial") programs, that is, non-recursive terms which transform the input part of an example into the desired output. In the second step, we generalize the straight-forward program to a recursive function. This idea was originally employed by Summers [13]. His algorithm THESIS workes on $cons$-expressions as input data which are transformed into the output by rewriting - with respect to a complete partial order over $conses$ - using a small set of primitive functions (namely car and cdr). In [12] we show that a wider class of input/output examples can be handled using a generic planning algorithm for constructing straight-forward programs. While some information about the semantics of the program domain is needed to construct straight-forward programs, generalization can be performed purely syntactically.

This second step - infering recurrence relations from straight-forward programs - is the topic of this paper. We extend the work in [12] in two ways: First, we present a formalization of our approach based upon the theoretical background of context-free tree grammers [5]. Second, we introduce a new technique for identifying parameters and their substitutions. Thereby, we are not only able to infer a variety of recursive structures (as tail recursion, linear recursion, tree recursion and combinations [12]) but addi-

tionally, we can deal with parameters which are substituted interdepended from one another. That is, we can infer functions like *mod(x, y) = if lt(x, y) then x else mod(minus(x, y), y)*.

In the following, we will first introduce our terminology and define the synthesis task. Then we present our method in detail. Afterwards we will illustrate the method and discuss its scope and efficiency. We finish with relations to other work and a short conclusion.

2 Induction of Recursive Program Schemes

2.1 Basic Terminology

First, we will introduce some concepts and notations which are used during the rest of the paper.

Terms and trees. A signature Σ is a finite set of function symbols of fixed arity $r(f), r : \Sigma \to \mathbb{N}$. With X we denote the set of variables, with $\mathcal{T}_\Sigma(X)$ the set of terms over Σ and X, and with $\mathcal{T}_\Sigma$ the set of all groundterms (terms without variables) over Σ. We use tree and term as synonyms.

A tree in $\mathcal{T}_\Sigma(X)$ will be denoted by $t(x_1, \ldots, x_p)$ in order to point out the variables, if unambiguous, we use the short hand vector notation $t(\vec{x})$. $t(t_1, \ldots, t_p)$ denotes the tree obtained by simultaneously substituting the terms t_i for each occurence of the variables x_i in $t(x_1, \ldots, x_p)$, $i = 1, \ldots, p$. The short hand vector notation $t(\vec{t})$ will be used. To point out which trees are being substituted for which variables, we will note $t(t_1/x_1, \ldots, t_p/x_p)$ resp. $t(\vec{t}/\vec{x})$. We call the terms t_i instantiations of the variables x_i. $var(t)$ is the set of all variables in the term t. We use the standard concepts for substitution and unification.

Subterms. A position in t is defined in the usual way as a sequence of natural numbers: (a) λ is a position in t; (b) if $t = f(t_1, \ldots, t_p)$, $f \in \Sigma$, and u is a position in t_i, then $i.u$ is a position in t.

A subterm of t at the postition u (denoted by t/u) is defined as: (a) $t/\lambda = t$; (b) if $t = f(t_1, \ldots, t_p)$, $f \in \Sigma$, and u is a position in t_i ($1 \leq i \leq p$), then $t/i.u := t_i/u$. For a term t and a position u in t the function $node(t, u)$ results in a pair (f, r) with $t/u = f(t_1, \ldots, t_r)$, t_i some terms from $\mathcal{T}_\Sigma(X)$, and $r = r(f)$.

A prefix of a tree $t \in \mathcal{T}_\Sigma(X)$ is a tree $p \in \mathcal{T}_\Sigma(\{y_1, \ldots, y_n\})$, with $\{y_1, \ldots, y_n\} \cap X = \emptyset$, and such there exists subtrees $t_1, \ldots, t_n$ of t with $t = p(t_1/y_1, \ldots, t_n/y_n)$. We write $p \leq t$ if p is a prefix of t. We write $p < t$, if p and t cannot be unified by renaming of variables only.

A segment $s \in \mathcal{T}_\Sigma(Y)$ of a term $p \in \mathcal{T}_\Sigma(X \cup Y)$ ($X \cap Y = \emptyset$) along the "recursion points" Y in a term $t \in \mathcal{T}_\Sigma$ at occurence w is defined as: (a) s is a segment of p in t at occurence λ iff $t = p(t\vec{x}/\vec{x}, ty_1/y_1, \ldots, ty_n/y_n)$ (i.e. $p \leq t$) and $s = p(t\vec{x}/\vec{x})$; (b) if $t = p(t\vec{x}/\vec{x}, ty_1/y_1, \ldots, ty_n/y_n)$, and s a segment of p in ty_i along Y at occurence w then s is a segment of p in t along Y at occurence $i.w$.

With $S(p, Y, t)$ we denote the set of all segments of p in t along Y, with $W(p, Y, t)$ the set of all occurences of p in t along Y. $s(w) : W(p, Y, t) \to S(p, Y, t)$ is the mapping between an occurence w and the corresponding segment.

Tree grammars. A regular tree grammar (RTG) $R = (N, \Sigma, P, s)$ consists of disjoint signatures N (nonterminals, all symbols of N with rank 0) and Σ (terminals), a finite rewrite system P over $N \cup \Sigma$, and a distinct constant symbol $s \in N$ (initial symbol). All rules in P are of the form $A \to t$, where $A \in N$ and $t \in \mathcal{T}_{N \cup \Sigma}$.

A context free tree grammar (CFTG) $C = (N, \Sigma, P, s)$ consists of disjoint signatures N (nonterminals with arity $r(A) \in \mathbb{N}$) and Σ (terminals), a finite rewrite system P over $N \cup \Sigma$, and a distinct constant symbol $s \in N$, $r(s) = 0$ (initial symbol). All rules in P are of the form $A(x_1, \ldots, x_n) \to t$, where $A \in N$, $r(A) = n$, $x_1, \ldots, x_n$ are pairwise different variables, and $t \in \mathcal{T}_{N \cup \Sigma}(\{x_1, \ldots, x_n\})$. For rules $A \to t_1$ and $A \to t_2$ we use the abbreviation $A \to t_1 \mid t_2$.

Starting with the initial symbol s, the nonterminals will be replaced by the right hand side of the appropriate rule whereby all variables are substituted by the corresponding terms. With $\xrightarrow{*}_P$ denoting the reflexive-transitive closure of the rewrite relation $\to_P$ generated by P, the language generated by a grammar $G = (N, \Sigma, P, s)$ is $\mathcal{L}(G) = \{t \in \mathcal{T}_\Sigma(X) \mid s \xrightarrow{*}_P t\}$.

2.2 Definition of Recursive Program Schemes

A recursive program scheme (RPS) on a signature Σ, a set of variables X, and a set of function variables Φ is a pair $\langle S, t_0 \rangle$, where $t_0 \in \mathcal{T}_{\Sigma \cup \Phi}(X)$ (main program) and S is a system of n equations (recursive subprograms): $S = \langle G_i(x_1^i, \ldots, x_{n_i}^i) = t_i, i = 1, \ldots, n \rangle$, with $G_i \in \Phi$, $x_j^i \in X$ for each $j = 1, \ldots, n_i$, and $t_i \in \mathcal{T}_{\Sigma \cup \Phi}(\{x_1^i, \ldots, x_{n_i}^i\})$ for all i.

Each RPS is associated with a CFTG $C_{\langle S, t_0 \rangle} = (\Phi \cup \{s\}, \Sigma \cup \{\Omega\}, P, s)$, $s \notin \Phi$, $\Omega \notin \Sigma$ (Ω as the bottom element); P is defined by: $P = \{s \to t_0 \mid \Omega\} \cup \{G_i(x_1^i, \ldots, x_{n_i}^i) \to t_i \mid \Omega\}$ with $G_i \in \Phi$ (cf. [5]).

With the CFTG $C_{\langle S, t_0 \rangle}$ we can now unfold a recursive scheme to terms in $\mathcal{T}_{\Sigma \cup \{\Omega\}}(X)$.

An interpretation i of the unfolded trees of an RPS $\langle S, t_0 \rangle$ is a homomorphism $\overline{i}$ in a Σ-algebra A, $\overline{i} : \mathcal{T}_\Sigma \to A$, defined over operation symbols in Σ and extended to terms. A valued interpretation is an interpretation together with a valuation $\alpha : X \to A$. $\overline{\alpha}$ denotes the extension of α to terms. In the following we will regard untyped structures only.

The initial programs, for which we want to infer an RPS, are interpreted and valuated terms in an unknown algebra. Because the algebra A is unknown, we regard an initial program *not* as element of A but as valued term of the term algebra over Σ and the valuation is $\alpha : X \to \mathcal{T}_\Sigma$ (that is, the variables are valuated by groundterms).

With $\mathcal{L}(C, \alpha)$ we denote the set of the valued terms of a grammar C: $\mathcal{L}(C, \alpha) = \{\overline{\alpha}(t) | t \in \mathcal{L}(C)\}$.

Example 1 (Unfolding an RPS). Consider the following RPS $\langle S, t_0 \rangle$ (the upper indices specify the arity of the symbol):
$\Sigma = \{z^0, pr^1, sc^1, eqz^1, m^2, g^3\}$, $X = \{x\}$, $\Phi = \{G^1\}$, $t_0 = G(x)$, $S : G(x) = g(eqz(x), sc(z), m(x, G(p(x))))$.

We can unfold it with the associated CFTG $C_{\langle S,t_0 \rangle} = (\Phi \cup \{s\}, \Sigma \cup \{\Omega\}, P, s)$, $P = \{s \to G(x) \mid \Omega,\quad G(x) \to g(eqz(x), sc(z), m(x, G(p(x)))) \mid \Omega\}$, and get for instance the following term:

$$
\begin{aligned}
t = & g(eqz(\quad x),\ sc(z), m(\quad x,\\
& g(eqz(\quad pr(x)),\ sc(z), m(\quad pr(x),\\
& g(eqz(pr(pr(x))), sc(z), m(pr(pr(x)), \Omega))))))
\end{aligned}
$$

This term can be interpreted in the algebra $\mathcal{N}at$ of the natural numbers with the additional sort $Boolean$ by $i : \{z \mapsto 0, pr(x) \mapsto p(x) =_{Def} pred(x), sc(x) \mapsto s(x) =_{Def} succ(x), m(x, y) \mapsto mult(x, y), eqz(x) \mapsto eq0(x) =_{Def}$ if $x = 0$ then $true$ else $false, g(x, y, z) \mapsto g(x, y, z) =_{Def}$ if x then y else $z\}$. Together with a valuation $\alpha(x) = s(s(0))$ we obtain the term $t_{Nat} = \overline{\alpha}(\overline{i}(t))$ with

$$
\begin{aligned}
& t_{Nat} = \\
& g(eq0(\quad s(s(0))),\ s(0), mult(\quad s(s(0)),\\
& g(eq0(\quad p(s(s(0)))),\ s(0), mult(\quad p(s(s(0))),\\
& g(eq0(p(p(s(s(0))))), s(0), mult(p(p(s(s(0)))), \Omega))))))
\end{aligned}
$$

which is an example for an initial program.

2.3 Limitations

We set the following restrictions to the final RPS $\langle S, t_0 \rangle$:

1. There is only one recursive subprogram G, i.e. $|S| = 1$.
2. All variables appear in the body t_1 of the recursive subprogram G (i.e. there is no variable just handed down) and each variable is used in at least one substitution per recursive call.
3. There is no recursive call in a substitution (like in the Ackermann function).
4. The term t_0 (the main program) consists only of the recursive subprogram call.

Additionaly, there are some limitations for inference:

5. Variables, which are used as constants (like n in the exponential function *f(n,m) = g(eq0(m),1, mult(n, f(n, p(m))))*) can not be identified as variables. They occur in the resulting RPS as constant parts in the subprogram body.
6. If the main program calls the subprogram with substitutions of variables (like t_0 = *G(succ(x))*), then they can not be separated from the valuation of the variables.

2.4 Synthesis problem

Given an initial program (like the one given in ex. 1) we want to infer an appropriate CFTG with the initial program as element of the language defined by it. That is, we regard program synthesis as special kind of grammatical inference.

Now we are ready to specify the synthesis problem:

Definition 1 (Synthesis problem). Let t_{init} be an initial program. Then

- a signatur Σ,
- a set of variables $X = \{x_1, \ldots, x_{n_V}\}$,
- a CFTG $C_{\langle S,t_0 \rangle} = (\{s, G\}, \Sigma \cup \{\Omega\}, P, s)$ corresponding to an RPS $\langle S, t_0 \rangle$ with $P = \{s \to G(x_1, \ldots, x_{n_V}) \mid \Omega,\quad G(x_1, \ldots, x_{n_V}) \to t_1 \mid \Omega\}, t_1 \in \mathcal{T}_{\Sigma \cup \{G\}}(X)$, and
  $t_1 = p(\vec{x},$
  $\quad G(ts_{11}(\vec{x}), \ldots, ts_{1n_V}(\vec{x}))/y_1,$
  $\quad \vdots$
  $\quad G(ts_{n_R1}(\vec{x}), \ldots, ts_{n_Rn_V}(\vec{x}))/y_{n_R})$
  $(p \in \mathcal{T}_{\Sigma \cup \{G\}}(X \cup Y))$ with $Y = \{y_1, \ldots, y_{n_R}\}$ indicating the n_R recursion points and where $\forall i = 1, \ldots, n_R : \forall j = 1, \ldots, n_V : ts_{ij} \in \mathcal{T}_\Sigma(X), \bigcup_j var(ts_{ij}) = X$, and
- a valuation $\alpha : X \to \mathcal{T}_\Sigma$

are to be infered, such that $t_{init} \in \mathcal{L}(C_{\langle S,t_0 \rangle}, \alpha)$ and the following constraints hold:

Recurrence: for each $r = 1, \ldots, n_R$: $|\{w | w \in W(p, Y, t_{init}), w = v.r\}| \geq 2$ (We need *(i)* a root segment as hypothesis for the recursive structure to be induced, *(ii)* at least one additional segment for each recursion point for validating the hypothesis and for building a hypothesis for the substitutions, and *(iii)* at least one further segment for each recursion point for validating the substitution hypothesis.)

Simplicity: there exists no RPS $\langle S', t'_0\rangle$, $|S'| = 1$, such that $L(\mathcal{C}_{\langle S,t_0\rangle}) \subset L(\mathcal{C}_{\langle S',t'_0\rangle})$.

3 Method

The inference process is driven by the syntactical structure of the initial tree. Given an initial tree and presupposed, that there exists a CFTG and a valuation which generates this initial tree, we perform the following steps:

1. We are looking for a set of recursion points in the initial tree, i.e. a minimal recursive term (section 3.1.1).
2. The minimal recursive term can then be extended to the subprogram body. Lemma 1 leads to a simple criterion for separating the body and the instantiations (see section 3.1.2).
   Given the validity of this hypothesis for the subprogram body, the remaining unexplained subtrees have to be instantiations of the variables.

For constructing the substitutions we have to find regularities between the instantiations (section 3.2).

1. First we have to check, that each instantiation is used in a regular way in each recursive call, and
2. that each instantiation is partial representable by at least one of the previous instantiations in a regular way (lemma 2).
3. Afterwards we search all regular relations between an instantiation and the previous instantiations and unifiy them to the final substitution rule.
4. All the variables with the same instantiation in all segments are assumed as identically (lemma 3).

3.1 Building a prefix generating RTG

Definition 2 (Prefix generating RTG). Let t_{init} be an initial tree, Σ the set of all symbols in t_{init} without Ω, and $Z = \{z_1, \ldots, z_{n_V}\}$, $Y = \{y_1, \ldots, y_{n_R}\}$ two disjoint set of variables (Y indicating the recursion points). An RTG $R = (\{G, s\}, \Sigma, P, s)$ with $P = \{s \to G \mid \Omega,\quad G \to t_1 \mid \Omega\}$, $t_1 = p(z_1, \ldots, z_{n_V}, G/y_1, \ldots, G/y_{n_R})$, $p \in \mathcal{T}_\Sigma(Z \cup Y)$, and $s \xrightarrow{*}_P t'_{init}$, $t'_{init} \leq t_{init}$ is a prefix generating RTG to t_{init}. During derivation the variables z_i of t_1 are renamed, such that all variables in the resulting term occur at one position only.

(Note, that all generated Ω's in the derived prefix t'_{init} must match an Ω in t_{init}. This restricts the choice of the recursion points.)

That is, we have to construct a term p with Y indicating the recursion points, which leads to a prefix generating RTG to t_{init} and therewith to a segmentation of t_{init}. Then we have to enlarge p to the subprogram body presupposing lemma 1.

Lemma 1. (Separability of program body and instantiations)
Let $\alpha : X \to \mathcal{T}_\Sigma$ be a valuation and $C_{\langle S,t_0\rangle}$ be a CFTG to an RPS $\langle S, t_0\rangle$ with $P = \{s \to G(ts_{01}(\vec{x}), \ldots, ts_{0n_V}(\vec{x})),\quad G(x_1, \ldots, x_n) \to t_1\}$,

$$t_1 = p(\vec{x},\ G(ts_{11}(\vec{x}), \ldots, ts_{1n_V}(\vec{x})),\ \vdots\ G(ts_{n_R1}(\vec{x}), \ldots, ts_{n_Rn_V}(\vec{x}))),$$

$ts_{ij} \in \mathcal{T}_\Sigma(X)$ for all $i = 0, \ldots, n_R$ and all $j = 1, \ldots, n_V$. Let $I \neq \emptyset$ be an index set, so that for each $i \in I$ there exists a prefix $p_i \in \mathcal{T}_\Sigma(Y) \setminus Y$ with $p_i \leq \overline{\alpha}(ts_{0i}(\vec{x}))$ and for each $j = 1, \ldots, n_R$: $p_i \leq ts_{ji}$.

Then there exists an RPS $\langle S', t'_0\rangle$ over X' with the associated CFTG $C_{\langle S',t'_0\rangle}$ and a valuation $\alpha' : X' \to \mathcal{T}_\Sigma$, such that $\mathcal{L}(C_{\langle S,t_0\rangle}, \alpha) = \mathcal{L}(C_{\langle S',t'_0\rangle}, \alpha')$ and for each $i = 1, \ldots, n'_V$ there exists no prefix $p_i \in \mathcal{T}_\Sigma(Y) \setminus Y$ with $p_i \leq \overline{\alpha'}(ts'_{0i}(\vec{x'}))$ and for each $j = 1, \ldots, n_R$: $p_i \leq ts'_{ji}$.

(Proof: by constructing the RPS $\langle S', t'_0\rangle$ and structural induction.)

If the potential recursion points and thereby a segmentation of the initial tree have been found, then the following propositions hold: (1) The maximum prefix over all segments has to be the body of the subprogram. (2) The remaining subtrees should be instantiations of the variables and must be explained by a valuation and some substitutions.

That means, the predefinition of the recursion points determines the subprogram body and the variables. Hence the overall strategy is backtracking over possible segmentations of the initial tree.

3.1.1 Building a segmentation The search for the appropriate recursion points has to be backtrackable. Foremost we are looking for a first potential recursion point u^r_1 (the lower index indicates the number of the recursion point) in the initial tree, i.e. a position u^r_1 in t_{init} with $u^r_1 \neq \lambda$, $node(t_{init}, u^r_1) = node(t_{init}, \lambda)$.

Then we build a term p consisting of all nodes between the root and the recursion point by applying a function $skeleton$ (defined below) and test it by building the prefix generating RTG (i.e. the RTG generates a term $t'_{init} \leq t_{init}$) for which the constraints given in definition 2 must hold.

Afterwards, we stepwise enlarge p by further recursion points u^r_i (marked in p by pairwise different variables $y_i \in Y$) until the associated prefix generating RTG generates all Ω's in t_{init} and hold the constraints. The skeleton $p \in \mathcal{T}_\Sigma(Z \cup Y)$ is a minimal prefix to t_{init} which contains the recursion points.

Definition 3 (Building the skeleton). Let t_{init} be the initial tree, p be a skeleton build over n_R recursion points u_i^r, $i = 1, \ldots, n_R$, which leads to a prefix generating RTG to t_{init}, $Y = \{y_1, \ldots, y_{n_R}\}$ the set of variables in p indicating the recursion points, Z the set of all other variables in p (initial values: $p = \bot$, $Y = \emptyset$, $Z = \emptyset$). Let u^r be a further potential recursion point. Then the new skeleton is build by applying the function $skeleton(t_{init}, u^r, p, Z, Y)$, which is defined (in a declarative style):

$skeleton(t, \lambda, \ \bot, Z, Y) = (y, Z, Y \cup \{y\}), y \notin Y$
$skeleton(t, \lambda, \ z, Z, Y) = (y, Z \setminus \{z\}, Y \cup \{y\}), y \notin Y$
$skeleton(t, i.u, \bot, Z, Y) =$
$(f(z_1, \ldots, z_{i-1}, t', z_i, \ldots, z_{r-1}), Z', Y')$ with
$(f, r) = node(t, \lambda)$
z_j pairwise different variables, $z_j \notin Z \cup Y, j = 1, \ldots, r-1$
$Z' = Z \cup \{z_1, \ldots, z_{r-1}\} \cup Z''$
$(t', Z'', Y') = skeleton(t/i.\lambda, u, \bot, Z, Y)$
$skeleton(t, i.u, t_s, Z, Y) =$
$(f(t_1, \ldots, t_i', \ldots, t_r), Z', Y')$ with
$t_s = f(t_1, \ldots, t_i, \ldots, t_r)$
if $t_i \in Z$ then $(t_i', Z', Y') = skeleton(t/i.\lambda, u, \bot, Z, Y)$
if $t_i \notin Z$ then $(t_i', Z', Y') = skeleton(t/i.\lambda, u, t_i, Z, Y)$.

3.1.2 Building the subprogram body We have now found a term p over Σ and $Y \cup Z$, such that the associated prefix generating RTG generates a term t'_{init} with $t'_{init} \leq t_{init}$, $t_{init} = t'_{init}(t_1/z_1, \ldots, t_n/z_n)$, $t_i \in \mathcal{T}_\Sigma$, and for each $r = 1, \ldots, n_R$ ($n_R = |Y|$) $|\{w | w \in W(p, Y, t_{init}), w = v.r\}| \geq 2$.

As a result of lemma 1 the body of the subprogram is the maximum prefix p_{max} over all segments in $S(p, Y, t_{init})$: $p_{max} \in \mathcal{T}_\Sigma(X \cup Y)$ with $p_{max} \leq s \ \ \forall s \in S(p, Y, t_{init})$ and $\neg\exists \ p'_{max} > p_{max}$ with $p'_{max} \leq s \ \ \forall s \in S(p, Y, t_{init})$.

During the construction of the prefix p_{max} we preserve the variables in Y. Because of our definition of the segmentation, p_{max} leads to a prefix generating RTG for t_{init}.

3.2 Building the CFTG

The term p_{max} with n_R recursion points marked by variables $y_1, \ldots, y_{n_R}$ is the hypothesis for the subprogram body. All the variables in $X = var(p_{max}) \setminus Y$ have to be (not necessary different) variables of the CFTG (that means in particular, that all the terms t_i with $s = p_{max}(t_1/x_1, \ldots, t_{n_V}/x_{n_V})$ for each $s \in S(p, Y, t_{init})$ have to be instantiations of the variables).

$\vec{v}^w$ denotes the instantiations of the variables $\vec{x}$ in the segment at occurance w: $\vec{v}^w = (t_1, \ldots, t_{n_V})$ with $s(w) = p_{max}(t_1/x_1, \ldots, t_{n_V}/x_{n_V})$ for each $w \in W(p_{max}, Y, t_{init})$.

We have to check, that all instantiations of an occurence is used in a recurrent way in all subordinated occurences:

Lemma 2 (Necessary conditions for variables). *Let t_{init} be an initial tree, p_{max} be the inferred subprogram body with Y indicating the recursion points, $var(p_{max}) = X \cup Y$, $X \cap Y = \emptyset$, $n_V = |X|$, $n_R = |Y|$, and $\vec{v}^w$ the instantiation of the variables X at occurrence $w \in W(p_{max}, Y, t_{init})$.*

Usage of all instantiations: *It must hold $\forall \ i \in \{1, \ldots, n_V\}$: $\forall \ r \in \{1, \ldots, n_R\} \ \ \exists \ k \in \{1, \ldots, n_V\}$ and $\exists t \in \mathcal{T}_\Sigma(Z \cup \{x_i\})$ where t is a minimal prefix, such that $\forall \ w.r \in W(p_{max}, Y, t_{init}) : t(v_i^w/x_i) \leq v_k^{w.r}$.*

Partial generating of all instantiations: *The sets of partial substitutions $PSubst_{kr}$ of variable x_k in the r-th recursive call constructed in the following way are not empty:*

1. *Set for each $k \in \{1, \ldots, n_V\}$ and for each $r \in \{1, \ldots, n_R\}$ $PSubst_{kr} = \emptyset$, $Z_{kr} = \emptyset$*
2. *Search for each $i \in \{1, \ldots, n_V\}$ and for each $r \in \{1, \ldots, n_R\}$ all k for which a minimal prefix $ps \in \mathcal{T}_\Sigma(Z \cup \{x_i\})$ exists ($X \cup Z \cup Z_{kr} = \emptyset$ and all variables occure at only one position), with $\forall \ w.r \in W(p_{max}, Y, t_{init}) : ps(v_i^w/x_i) \leq v_k^{w.r}$.*
   Set $PSubst_{kr} := PSubst_{kr} \cup \{ps\}$, $Z_{kr} := Z_{kr} \cup Z$.

If one of these conditions does not hold, then no adequate CFTG can be built on the basis of the term p_{max}, i.e. there must be backtracking over the segmentation construction.

(Proof: These are consequences of the restrictions, that each variable has to occure in the subprogram body and has to be used in at least one substitution in a recursive call.)

We can now construct the final substitutions by means of the sets of partial substitutions. Therefore a general unifier σ_{kr} for each set of partial substitutions $PSubst_{kr}$ is built (these unifiers exists, because all terms in $PSubst_{kr}$ are prefixes of the same term v_k^r). Variables are renamed only in variables from X. Formally for all $k \in \{1, \ldots, n_V\}$ and for all $r \in \{1, \ldots, n_R\}$ $ps_i\sigma_{kr} = ps_j\sigma_{kr}$ for all $ps_i, ps_j \in PSubst_{kr}$.

Lemma 3 (Reduction of variables). *Contains some σ_{k1} a renaming $\{x_i \leftarrow x_j\}$, $x_i, x_j \in X$, then the variable x_i in p_{max} can be replaced by x_j. The set of variables can be reduced to $X := X \setminus \{x_i\}$.*

(Proof: $\forall w \in W(p_{max}, Y, t_{init})$ holds: $v_i^w = v_j^w$ (because of the construction of $PSubst_{kr}$), i.e. the instantiations of x_i and x_j are identical in all segments.)

Now we know the number of necessary variables $n_V = |X|$ and thereby the rank of G.

The substitution ρ_{kr} of the variable x_k in the r-th recursive call can be built by $\rho_{kr} = \{x_k \leftarrow ts_{kr}\}$ with $ts_{kr} = ps_{kr}\sigma_{kr}$ and ps_{kr} some term of $PSubst_{kr}$.

The constructed substitutions ρ_{kr} substitute a variable by a term over some variables. Finally, we have to check whether a substitution contains a subterm, which is independent from the variables.

Lemma 4 (Incomplete substitutions). *Let the right hand side ts_{kr} of a substitution $\rho_{kr} = \{x_k \leftarrow ts_{kr}\}$ contain some variables $z_i \in Z$. Let I be the index set for these variables (i.e. $var(ts_{kr}) = var(ts_{kr}) \cap X \cup \{z_i | i \in I\}$).*

Exists for every z_i a groundterm $t_i \in \mathcal{T}_\Sigma$, such that $\forall w.r \in W(p_{max}, Y, t_{init})$ holds $v_k^{w.r} = ts_{kr}\{x_1 \leftarrow v_1^w, \ldots, x_{n_V} \leftarrow v_{n_V}^w\}\{y_i \leftarrow t_i \ | i \in I\}$ the substitution ρ_{kr} can be extended to $\rho_{kr} := \rho_{kr}\{y_i \leftarrow t_i | i \in I\}$.

Otherwise no adequate CFTG can be built on the basis of the term p_{max}, i.e. there must be backtracking over the segmentation building.

Lemma 5 (Valuation). *The valuation $\alpha : X \to \mathcal{T}_\Sigma$ is determined by the instantiations $\vec{v}^\lambda = (v_1^\lambda, \ldots, v_{n_V}^\lambda)$: $\alpha(x_i) = v_i^\lambda$, $i = 1, \ldots, n_V$.*

(Proof: This is a consequence of the limitation, that there is no distinction between valuation and substitution in the subprogram call in the main program.)

As result we have now the CFTG $C_{\langle S,t_0\rangle} = (\{s, G\}, \Sigma \cup \{\Omega\}, P, s)$ with $\vec{x} = (x_1, \ldots, x_{n_V})$ and

$$\begin{aligned} P = \{ s &\to G(\vec{x}) \mid \Omega, \\ G(\vec{x}) &\to p(\vec{x}, \\ &\quad G(ts_{11}(\vec{x}), \ldots, ts_{1n_V}(\vec{x}))/y_1, \\ &\quad \vdots \\ &\quad G(ts_{n_R1}(\vec{x}), \ldots, ts_{n_Rn_V}(\vec{x}))/y_{n_R}) \mid \Omega \} \end{aligned}$$

and the valuation α, such that $t_{init} \in \mathcal{L}(C_{\langle S,t_0\rangle}, \alpha)$. The described method guarantees that this is the simplest possible recurrent hypothesis for t_{init}.

4 Illustration and Evaluation of the Method

4.1 Example

Consider the initial tree t_{init} given in fig. 1. We obtain the signature $\Sigma = \{0, s, p, eq0, mult, g\}$.

(1) We find the first and only potential recursion point $u_1^r = 3.\lambda$ and thereby the minimal prefix $p = g(z_1, z_2, y_1)$. By means of the appropriate prefix generating RTG the following prefix of the initial tree $t'_{init} = g(z_1, z_2, g(z_3, z_4, g(\ldots\Omega)\ldots) \leq t_{init}$ can be derived.

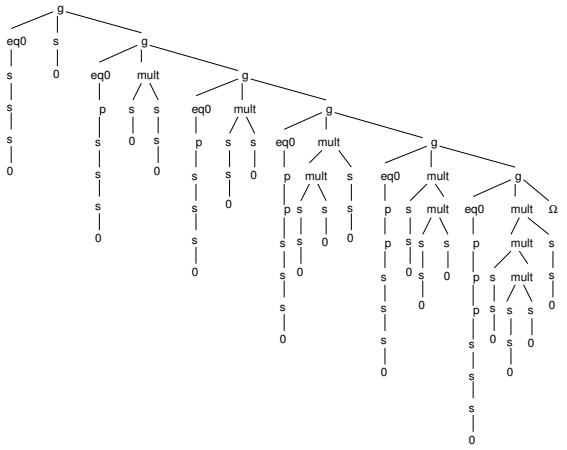

Figure 1 Example for an initial tree

(2) This results in the program body $p_{max} = g(eq0(x_1), x_2, y_1)$ with the following instantiations

w	v_1	v_2
λ	$s^3(0)$	$s(0)$
$1.\lambda$	$p(s^3(0))$	$mult(s(0), s^2(0))$
$1.1.\lambda$	$p(s^3(0))$	$mult(s^2(0), s(0))$
$1.1.1.\lambda$	$p^2(s^3(0))$	$mult(mult(s^2(0), s(0)), s^2(0))$
$1.1.1.1.\lambda$	$p^2(s^3(0))$	$mult(s^2(0), mult(s^2(0), s(0)))$
$1.1.1.1.1.\lambda$	$p^3(s^3(0))$	$mult(mult(s^2(0), mult(s^2(0), s(0))), s^2(0))$

(3) There can be detected no regularity for the first instantiation (the relation found between v_1^λ and $v_1^{1.\lambda}$ is $v_1^{1.\lambda} = p(v_1^\lambda)$ but $v_1^{1.1.\lambda} = p(s^3(0)) \neq p^2(s^3(0)))$. So another potential recursion point has to be found.

(1') We find the new recursion point $u_1^r = 3.3.\lambda$ with the minimal prefix $p = g(z_1, z_2, g(z_3, z_4, y_1))$.

(2') This results in the program body $p_{max} = g(eq0(x_1), x_2, g(eq0(p(x_3)), mult(x_4, x_5), y_1))$ with the instantiations given in table 1a.

(3') We detect the following regularities between the instantiations v_i^w and $v_k^{w.1}$:

i	k	t
1	1	$p(x_1)$
2	2	$mult(z_1, x_2)$
3	1	$p(x_3)$
4	2	$mult(z_1, x_4)$
5	5	x_5

(4, 5) We obtain the not empty sets of partial substitutions (representations of $v_k^{w.1}$ by v_i^w) with the unifiers σ_{k1} given in table 1b.

(6) There are the renamings $\{x_1 \leftarrow x_3\}$, $\{x_4 \leftarrow x_2\}$, so the set of variables can be reduced to $X = \{x_2, x_3, x_5\}$: $p_{max} = g(eq0(x_3), x_2, g(eq0(p(x_3)), mult(x_2, x_5), y_1))$ with the following substitutions: $\rho_{21} = \{x_2 \leftarrow mult(x_5, x_2)\}$, $\rho_{31} = \{x_3 \leftarrow p(x_3)\}$,

Table 1 Instantiations (a) and substitutions (b)

(a)

w	v_1	v_2	v_3	v_4	v_5
λ	$s^3(0)$	$s(0)$	$s^3(0)$	$s(0)$	$s^2(0)$
$1.\lambda$	$p(s^3(0))$	$mult(s^2(0), s(0))$	$p(s^3(0))$	$mult(s^2(0), s(0))$	$s^2(0)$
$1.1.\lambda$	$p^2(s^3(0))$	$mult(s^2(0), mult(s^2(0), s(0)))$	$p^2(s^3(0))$	$mult(s^2(0), mult(s^2(0), s(0)))$	$s^2(0)$

(b)

k	$PSubst_{k1}$	σ_{k1}
1	$\{p(x_1), p(x_3)\}$	$\{x_1 \leftarrow x_3\}$
2	$\{mult(x_5, z_1), mult(z_2, x_2), mult(z_3, x_4)\}$	$\{z_1 \leftarrow x_2, z_2 \leftarrow x_5, z_3 \leftarrow x_5, x_4 \leftarrow x_2\}$
3	$\{p(x_1), p(x_3)\}$	$\{x_1 \leftarrow x_3\}$
4	$\{mult(x_5, z_1), mult(z_2, x_2), mult(z_3, x_4)\}$	$\{z_1 \leftarrow x_2, z_2 \leftarrow x_5, z_3 \leftarrow x_5, x_4 \leftarrow x_2\}$
5	$\{x_5\}$	id

$\rho_{51} = \{x_5 \leftarrow x_5\}$. The valuation is thereby $\alpha : \{x_2 \mapsto s(0), x_3 \mapsto s^3(0), x_5 \mapsto s^2(0)\}$, and finally we get the CFTG $C_{(S,t_0)} = (\{s, G\}, \Sigma \cup \{\Omega\}, P, s)$ with

$$P = \{s \rightarrow G(x_2, x_3, x_5) \mid \Omega,$$
$$G(x_2, x_3, x_5) \rightarrow g(eq0(x_3), x_2, g(eq0(p(x_3)),$$
$$mult(x_2, x_5), G(mult(x_5, x_2), p(x_3), x_5))) \mid \Omega\}$$

4.2 Scope and efficiency of the method

4.2.1 Scope With the method described and illustrated above, the synthesis problem given in def. 1 can be solved within the limitations given in sec. 2.3. That is, we can deal with all (tail, linear, tree recursive) structures which can be generated by a *single* recursive function of arbitrary complexity.

The method can easily be extended to structures starting with a nonrecursive term (i.e. a more complex calling main program) as shown in [12] (see restriction 4 in sec. 2.3). Currently we are working on an extension of the method to the detection of sets of recursive functions (see restriction 1 in sec. 2.3). A problem which is probably beyond the scope of our method is the detection of indirect recursions (i.e. recursive functions which mutually call themselves).

We believe that the main advantage of our method is the technique for determining variables and substitutions. All parts of the initial program which are not constant over the different hypothetical recursion points are regarded as variables. In a second stage of inference, regularities and interdependences are identified and thereby a hypothesis about the number of variables, their valuation and the subsitution rule for the recursive call is obtained. The current restriction of our method to variables, which are not only passed through the recursive calls but have to be changed (in contrast to the start and goal peg in the Tower of Hanoi function, see table 2; see restriction 5 in sec. 2.3) can be overcome by a simple extension of our algorithm. The same is true for the restriction that all variables have to occure in each recursive call (cf. the recursive call for $y = 0$ in ackermann, see table 2; see restriction 2 in sec. 2.3). A problem for which we see no (simple) solution is the dealing with recursive substitutions (as in the Ackermann function, see table 2; see restriction 3 in sec. 2.3).

Table 2 Structures, which cannot be folded

```
G(n,a,b,c) = ; hanoi
  g(eq1(n), move(a,b),
    append(G(pred(n),a,c,b), move(a,b),
           G(pred(n),c,b,a)))
G(x,y) =     ; ackermann
  g(eq0(x), succ(y),
    g(eq0(y), G(pred(x),1),
      G(pred(x), G(x,pred(y)))))
```

Restriction 6 (see sec. 2.3) is inherent to our methodology: We want to fold RPSs from initial trees using it's structural characteristics only. Without additional information about number, sequence and names of parameters or - alternatively - about constant and operation symbols in the signature it is not possible to separate substitutions from valuations of variables.

4.2.2 Efficiency Our induction algorithm is a special kind of pattern-matcher for detecting regularities in labelled trees. The number of cycles for constructing a hypothesis is restricted by the size of the initial tree. Because we allow not only linear but also tree recursive structures the worst case effort is exponential.

The possible number of recursive calls (n_R) in the subprogram is restricted by the number of occurences of Ω (n_Ω) in the initial tree (see sec. 4.1.1): $n_R = \frac{n_\Omega - 1}{i} + 1$ with i as number of segments. That is, i has to be a whole-numbered divisor. Because in the worst case, $n_\Omega - 1$ is equivalent to some 2^x, the upper bound for the number of possible numbers of recursive calls is restricted by $|n_R| \leq log_2(n_\Omega - 1)$. The recurrence condition (see def. 1) requires for each recursion point at least three segments, i.e. $i \geq 3n_R$ and therefore $n_R \leq \sqrt{n_\Omega}$.

The number of valid hypotheses for the skeleton (n_s; cf. def. 3) is restricted by the maximum number d of occurences of the root label along a path in the initial tree. Each path from the root to an Ω can be partitioned in maximally $log_2\ d$ ways; therefore, $n_s \leq$

$(log_2\ d)^{\sqrt{n_\Omega}}$. This is also the maximal number for searching substitutions. Because we allow for interdependent substitutions the effort for checking all possible hypothesis is quadratical (see sec. 4.2).

Of course, our goal is to get initial trees with the minimal size for which the construction of a plausible folding hypothesis is possible as input. To detect the correct CFTG for an (already known) RPS, it is always enough to analyze its third unfolding.

5 Relations to Other Work

Generally, there is not much work done in the area of inductive program synthesis during the last years. Our work is in the tradition of Summers [13] and its extension to program schemes by Wysotzki [14]. Summers' synthesis theorem is restricted (1) to a single input parameter (a $cons$-expression) and (2) to linear recursions. For cases where no unique substitution can be found he introduces local variables (see also [7]). An extension of [13, 7] based upon the concept of term-rewriting was presented by Le Blanc [2]. His work is restricted to uninstantiated examples while we are using valuated terms.

In this aspect, our work is similar to some ILP approaches [10] where instantiated examples are used. Recently, in ILP the old topics and problems of inductive program synthesis has become rediscovered again [1, 6]. While these authors also work with small sets of only positive examples, typically they provide their system with information about the number and ordering of parameters of the desired function.

Finally, by using the framework of context-free tree grammers, our work has some relations to grammatical inference. To our knowledge there is no reported work about inductive inference of context-free tree-grammers. The approaches are typically restricted to regular grammers [3, 8].

6 Conclusion

We presented a generic method for the induction of recursive program schemes from interpreted and valuated terms. In contrast to classical approaches to inductive program synthesis and to ILP approaches we do not start with input/output examples but with initial programs which were constructed by planning. This separation of constructing non-recursive initial programs and of generalization has some advantages: We believe that building initial programs and generalization are based on different (cognitive) processes. Building initial programs from input/output examples inherently relies on domain knowledge (about data structures and operators) while generalization over terms can be performed purely syntactically by means of pattern matching. If the aim is to construct synthesis algorithms, which are able to deal with a larger class of recursive structures than the techniques at hand, it could also be useful to split the synthesis task into two problems with less complexity.

References

1. D.W. Aha, C.X. Ling, S. Matwin, and S. Lapointe. Learning singly-recursive relations from small datasets. In F. Bergadano, L. De Raedt, S. Matwin, and S. Muggleton, editors, *IJCAI93WS*, pages 47–58. MK, 1993.
2. G. Le Blanc. 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*, pages 183–197, 1994.
3. H. Boström. Theory-guided induction of logic programs by inference of regular languages. In *ICML*, pages 46–53, 1996.
4. B. Courcelle and M. Nivat. The algebraic semantics of recursive program schemes. In Winkowski, editor, *Math. Foundations of Computer Science*, volume 64 of *LNCS*, pages 16–30. Springer, 1978.
5. I. Guessarian. Program transformation and algebraic semantics. *Theoretical Comp. Sci.*, 9:39–65, 1979.
6. P. Idestam-Almquist. Efficient induction of recursive definitions by structural analysis of saturations. In L. De Raedt, editor, *Proc. 5th ILP, Leuven*, pages 77–94. Dept. of Computer Science, Leuven, 1995.
7. J. P. Jouannaud 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 *IJCAI*, pages 440–447, 1979.
8. T. Knuutila and M. Steinby. The inference of tree languages from finite samples: an algebraic approach. *Theoretical Computer Science*, 129(2):337–367, 1994.
9. Y. Kodratoff and J.Fargues. A sane algorithm for the synthesis of lisp functions from example problems: The boyer and moore algorithm. In *Proc. of the AISE Meeting Hambourg*, pages 169–175, 1978.
10. 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.
11. M. R. K. Krishna Rao. A class of Prolog programs inferable from positive data. In S. Arikawa and A. K. Sharma, editors, *Proc. of the 7th ALT, Sydney, Australia*, pages 273–284, Berlin, 1996. Springer.
12. U. Schmid and F. Wysotzki. Induction of recursive program schemes. In *Proceedings of the 10th European Conference on Machine Learning (ECML-98)*, pages 228–240. Springer, 1998.
13. P. D. Summers. A methodology for LISP program construction from examples. *Journal ACM*, 24(1):162–175, 1977.
14. F. Wysotzki. Representation and induction of infinite concepts and recursive action sequences. In *Proceedings of the 8th IJCAI*, Karlsruhe, 1983.