Folding of Finite Program Terms to Recursive Program Schemes

Emanuel Kitzelmann, Ute Schmid, Martin Mühlpfordt, and Fritz Wysotzki

Abstract—We present an approach to inductive synthesis of functional programs based on the detection of recurrence relations. A given term is considered as the kth unfolding of an unknown recursive program. If a recurrence relation can be identified in the term, it can be folded into a recursive program which (a) can reproduce the term and (b) generalizes over it. Our approach goes beyond Summers’ classical approach in several aspects: It is language independent and works for terms belonging to an arbitrary term algebra; it allows induction of sets of recursive equations which are in some arbitrary ‘calls’ relation; induced equations can be dependent on more than one input parameter and we can detect interdependencies of variable substitutions in recursive calls; the given input terms can represent incomplete unfoldings of an hypothetical recursive program.

Index Terms—Folding, Inductive Program Synthesis, Recursive Program Schemes.

I. INTRODUCTION

Automatic induction of programs from I/O examples is an active area of research since the sixties and of interest for AI research as well as for software engineering [1]. We present an approach which is based on the recurrence-detection method of Summers [2]. Induction of a recursive program is performed in two steps: First, input/output examples are rewritten into a finite program term, and second, the finite term is checked for recurrence. If a recurrence relation is found, the finite program is folded into a recursive function which generalizes over the given examples. The first step of Summers’ approach is knowledge dependent: In general, there are infinitely many possibilities to represent input/output examples as terms. Summers deals with that problem by restricting his approach to structural list problems. Alternatively, the finite program can be generated by the user [3] or constructed by AI planning [4].

In the following, we are only concerned with the second step of program synthesis – folding a finite program term in a recursive program. This corresponds to program synthesis from traces [5] and is an interesting problem in its own right. Providing a powerful approach to folding is crucial for developing synthesis tools for practical applications. Furthermore, because recursive programs correspond to a subset of context-free tree grammars, our approach to folding can be applied to artificial and natural grammar inference problems. Finally, combining program synthesis and AI planning makes it possible to infer general control policies for planning domains with recursive domains, such as the Tower of Hanoi [4].

Our approach extends Summers’ approach in several aspects:

First, it is language independent and works for terms belonging to an arbitrary term algebra, while Summers was restricted to Lisp programs; second, it allows induction of sets of recursive equations which are in some arbitrary ‘calls’ relation, while Summers was restricted to induction of a single, linear recursive equation; third, induced equations can be dependent on more than one input parameter and we can detect interdependencies of variable substitutions in recursive calls, while Summers was restricted to a single input list; finally, the given input terms can represent incomplete unfoldings of an hypothetical recursive program.

II. BASIC TERMINOLOGY

A. Terms and Term Rewriting

Terms. A signature $\Sigma$ is a set of (function) symbols with $\alpha : \Sigma \rightarrow N$ giving the arity of a symbol. With $X (X \cap \Sigma = \emptyset)$ we denote the set of variables, with $T_\Sigma (X)$ the terms over $\Sigma$, and with $T_\Sigma$ the ground terms (terms without variables). $\text{var}(t)$ is the set of all variables in term $t$. We use tree and term as synonyms. If $\{x_1, \ldots, x_n\}$ are the variables of a term $t$, then $t[x_1 \leftarrow t_1, \ldots, x_n \leftarrow t_n]$ or $\{t[1], \ldots, t[n]\}$ for short denotes the tree obtained by simultaneously substituting the terms $t_i$ for each occurrence of the variables $x_i$ in $t$, $i = 1, \ldots, n$.

A position in $t$ is defined in the usual way as a sequence of natural numbers: $(a)$ $\lambda$ is the root position of $t$, $(b)$ if $t = f(t_1, \ldots, t_n)$ and $u$ is a position in $t_1$ ($i = 1, \ldots, n$) then $i, u$ is a position in $t$. The composition $v \circ w$ of two positions is defined as $u, w$, if $v = u, \lambda$, the composition $v \circ k$ of a position $v$ and a natural number $k$ as $u, k, \lambda$, if $v = u, \lambda$. A partial order over positions is defined by $v \leq w$ if $v = w$ or it exists a position $u$ with $v \circ u = w$.

A subterm of a term $t$ at a position $u$ (written $t[u]$) is defined as: $(a)$ $t[\lambda] = t$, $(b)$ if $t = f(t_1, \ldots, t_n)$ and $u$ a position in $t_i$, then $t[i, u] = t[i, u, i = 1, \ldots, n]$. For a term $t$ and a position $u$, function $\text{node}(t, u)$ returns the fixed symbol $f \in \Sigma$, if $t[u] = f(t_1, \ldots, t_n)$. The set of all positions at which a fixed symbol $f$ appears in a term is denoted by $\text{pos}(t, f)$. With $\text{pos}(t)$ we refer to the set of all positions in term $t$. Obviously it holds that $v \in \text{pos}(t)$, if $w \in \text{pos}(t)$ and $v \leq w$. The replacement of a subterm $t[u]$ by a term $s$ in a term $t$ is written as $t[w \leftarrow s]$.

A term $p \in T_\Sigma\{(y_1, \ldots, y_n)\}$ is called first order pattern of a
term $t \in T_\Sigma(X)$. \((y_1, \ldots, y_n) \cap X = \emptyset\), if there exists subtrees $t_i$ such that $t = p[y_1 \leftarrow t_1, \ldots, y_n \leftarrow t_n]$, $i = 1, \ldots, n$. A pattern $p$ of a term $t$ is called trivial, if $p$ is a variable and non-trivial otherwise. We write $p \leq t$ if $p$ is a pattern of $t$ and $p < t$ if $p$ and $t$ cannot be unified by variable renaming only. $p$ is called maximal (first order) pattern of $t$, if $p \leq t$ and there exists no term $p'$ with $p < p'$ and $p' \leq t$.

Term rewriting. A term rewrite system over $\Sigma$ is a set of pairs of terms $R \subseteq T_\Sigma(X) \times T_\Sigma(X)$. The elements $(l, r)$ of $R$ are called rewrite rules and are written as $l \rightarrow r$. A term $t'$ can be derived in one rewrite step from a term $t$ using $R$ (i.e., $t \rightarrow t'$), if there exists a position $\alpha$ in $t$, a rule $l \rightarrow r \in R$, and a substitution $\sigma : X \rightarrow T_\Sigma(X)$, such that $t[\alpha \leftarrow \sigma(l)]$ and $t'[\alpha \leftarrow \sigma(r)]$. $R$ implies a rewrite relation $\rightarrow_R \subseteq T_\Sigma(X) \times T_\Sigma(X)$ with $(t, t') \in \rightarrow_R$ if $t \rightarrow t'$. The reflexive and transitive closure of $\rightarrow_R$ is denoted by $\Rightarrow_R$.

B. Recursive Program Schemes

Let $\Sigma$ be a signature and $\Phi = \{G_1, \ldots, G_n\}$ a set of function variables with $\Sigma \cap \Phi = \emptyset$ and arity $\alpha(G_i) = m_i > 0$. A recursive program scheme (RPS) $S$ on a signature $\Sigma$, variables $X$ and a set of function variables $\Phi = \{G_1, \ldots, G_n\}$ with $\Sigma \cap \Phi = \emptyset$ is a pair $(G, t_0)$ with a calling “main program” $t_0 \in T_{\Sigma\Phi}(X)$ and $G$ as a system of $n$ equations (recursive “subprograms”): $G = \langle G_1(x_1, \ldots, x_m) \rightarrow t_1 \mid i \in \{1, \ldots, n\}\rangle$ for the reflective and transitive closure of the rewrite relation implied by $\Rightarrow_{\Sigma \Phi}$. Applying this rules successively on a term $t_0$ is called unfolding of an RPS. In this paper we describe a method to fold such an unfolded term back to an RPS. We use the special symbol `$\Omega$ in an unfold term to indicate that a hypothetical unfolding process has stopped at this position, that is, a subprogram head has not been rewritten by the subprogram body, but by $\Omega$. All terms, in which all possible subprogram heads are replaced by $\Omega$, constitute the language of an RPS and correspond to initial program terms (Sect.III-A), which can be folded.

For a substitution with ground terms $\beta : X \rightarrow T_\Sigma$, called initial instantiation of $X$, the set of all terms $L(S, \beta) = \{t \mid t \in T_{\Sigma\Omega}(\beta), \beta(\Omega) \Rightarrow_{\Sigma\Omega} t\}$ is called the language generated by $S$ with initial instantiation $\beta$. For a particular equation $G_i \in \Phi$ the rewrite relation $\Rightarrow_{G_i\Omega}$ is implied by the rules $\Rightarrow_{G_i\Omega} = \{G_i(x_1, \ldots, x_m) \rightarrow t_i, G_i(x_1, \ldots, x_m) \rightarrow \Omega\}$. For an instantiation $\beta : \{x_1, \ldots, x_m\} \rightarrow T_\Sigma$ of parameters $G_i$, the language generated by subprogram $G_i$ is the set of all terms $L(G_i, \beta) = \{t \mid t \in T_{\Sigma\Omega}(\beta), \beta(G_i(x_1, \ldots, x_m)) \Rightarrow_{G_i\Omega} t\}$. If we start unfolding on an initialized subprogram and only apply the rewrite rule for this equation, then we receive the terms, which constitute the language of this equation.

Relations between subprograms. For an RPS $S$ let $H \notin \Phi$ be a function variable (for the unnamed main program). A relation $\text{call}_{RS} \subseteq \{H\} \times \Phi \times \Phi$ between subprograms and main program of $S$ is defined as: $\text{call}_{RS} = \{(H, G_j) \mid G_j \in \Phi, \text{pos}(t_0, G_j) \neq \emptyset\} \cup \{(G_i, G_j) \mid G_i, G_j \in \Phi, \text{pos}(t_0, G_j) \neq \emptyset\}$. The transitive closure $\text{call}_{RS}^+$ of $\text{call}_{RS}$ is the smallest set $\text{call}_{RS}^+ \subseteq \{H\} \cup \Phi \times \Phi$ for which holds: (a) $\text{call}_{RS} \subseteq \text{call}_{RS}^+$, (b) for all $P \in \{H\} \cup \Phi$ and $G_i, G_j \in \Phi$: If $P \text{call}_{RS}^+ G_i$ and $G_i \text{call}_{RS}^+ G_j$ then $P \text{call}_{RS}^+ G_j$.

For a recursive equation $G_i(x_1, \ldots, x_m) = t_i$ in $G$, the set of transitivity called (tc) equations $G_i$ with initial equation $G_i$ is defined as: $G_i \rightarrow_{G_i} = \{G_i(x_1, \ldots, x_m) \rightarrow t_i \mid G_i \text{call}_{RS}^+ G_j\}$. A set of tc equations for an initial equation $G_i$ is the set of all equations in an RPS called by $G_i$, directly or indirectly.

For a set of tc equations $G_i$, its initial equation $G_i$ and an instantiation $\beta : \{x_1, \ldots, x_m\} \rightarrow T_\Sigma$, the relations existing between a given RPS and its unfoldings For a set of tc equations $G_i$ with initial equation $G_i$ is that in case 2 every equations called by $G_i$ are unfolded.

C. Unfolding of RPSs

For a recursive equation $G_i(x_1, \ldots, x_m) = t_i$ with parameters $X_i = \{x_1, \ldots, x_m\}$ the set of recursion points is given by $\text{Rec}(t_i, G_i) \rightarrow \{i \mid \text{pos}(t_i, G_i)\}$.

Each recursive call of $G_i$ at position $u_r \in \text{Rec}(t_i, G_i)$ in $t_i$ implies substitutions $\sigma_r : X_i \rightarrow T_\Sigma$ of the parameters in $G_i$. The substitution terms, denoted by $\text{sub}(x_j, r)$, are defined by $\text{sub}(x_j, r) = \sigma_r(x_j)$ for all $x_j \in X_i$ and it holds $\text{sub}(x_j, r) = t_i[u_r \leftarrow \sigma(r)]$. Consider the equation $G(x) = \text{eq}(x, 0)$, $(x, G(\text{pred}(x)))$ for example. It holds $\text{Rec}(x) = \{3, 2, \lambda\}$ and $\text{sub}(x, 1) = \text{pred}(x)$. The set of unfolding points, denoted by $\text{Unf}$, is constructed over $\text{Rec}$ and inductively defined as the smallest set for which holds: (a) $\lambda \in \text{Unf}$, (b) if $u_{\text{Rec}} \in \text{Unf}$ and $u_{\text{Rec}} \in \text{Rec}$, then $u_{\text{Rec}} \neq u_{\text{Rec}} \in \text{Unf}$. In an unfolded term, unfolding points are the positions, at which rewrite steps occurred.

Unfolding points imply compositions of substitutions, denoted by $\text{sub}^+(x_j, u_{\text{Unf}})$, which are inductively defined by: (a) $\text{sub}^+(x_j, \lambda) = x_j$, (b) $\text{sub}^+(x_j, u_{\text{Rec}} \circ u_{\text{Rec}}) = \text{sub}(x_j, r) \{x_j \leftarrow \text{sub}^+(x_j, u_{\text{Rec}}), \ldots, x_m \leftarrow \text{sub}^+(x_m, u_{\text{Rec}})\}$, $j = 1, \ldots, m$. While substitution terms for a parameter give us its substitution in the next unfolding, compositions of substitutions for a parameter give us its substitution in the unfolding indexed by $u_{\text{Rec}}$. Unfoldings. For an initial instantiation $\beta : X_i \rightarrow T_\Sigma$, the instantiations of parameters in unfoldings of an equation $G_i$ are indexed by $u_{\text{Rec}}$ and defined as: $\beta_{u_{\text{Rec}}} = \beta_{\text{sub}}(x_j, u_{\text{Rec}})$. The set of all unfoldings $\mathcal{Y}_i$ of equation $G_i$ over instantiation $\beta$ is indexed by $u_{\text{Rec}}$ and defined as $\mathcal{Y}_i = \{\beta_{u_{\text{Rec}}}, u_{\text{Rec}} \in \text{Unf}\}$. The following lemma states the relation between the concepts introduced, that is, between an RPS and its unfoldings.

Lemma 1 (Inductive Structure of $\mathcal{L}$) Let $t \in \mathcal{L}(G_i, \beta)$ be an element of the language of $G_i$ over an initial instantiation $\beta : X_i \rightarrow T_\Sigma$. Then for all $u_{\text{Rec}} \in \text{Rec}(t) \cup \text{pos}(t)$ it holds: $t[u_{\text{Rec}}] = \Omega$ or $t[u_{\text{Rec}}] = \Omega \cup u_{\text{Rec}}$ (where $\text{Rec}(t) \rightarrow \Omega \cup u_{\text{Rec}} \rightarrow t[u_{\text{Rec}}] \rightarrow \Omega \cup u_{\text{Rec}}$). (Proof by induction over $u_{\text{Rec}}$.)

The relations existing between a given RPS and its unfoldings can be exploited for the reverse process – the induction of an unknown RPS from a given term which is considered as some unfolding of a set of recursive equations.
III. Induction of Recursive Program Schemes

A. Initial Programs

An initial program is a ground term \( t \in \mathcal{T}_{\Sigma,\Phi}(\Omega) \) which contains at least one \( \Omega \). A term \( t \) which contains function variables, i.e. \( t \in \mathcal{T}_{\Sigma,\Phi}(\Omega) \), is called reduced initial program. We use initial program and initial tree as synonyms. For terms which contain \( \Omega \), an order relation is defined by: (a) \( \Omega \leq \Omega \), if \( \text{pos}(t', \Omega) \neq \emptyset \), (b) \( x \leq \Omega \), if \( x \in X \) and \( \text{pos}(t', \Omega) = \emptyset \), (c) \( f(t_1, \ldots, t_n) \leq \Omega \) if \( f(t'_1, \ldots, t'_n) \), if \( \forall I \in \{1, \ldots, n\} \) it holds \( t'_I \leq \Omega \).

Initial programs correspond to the elements of the language of an RPS or a set of tc equations, whereas reduced initial programs correspond to elements of the language of a particular recursive equation. We say that an RPS \( S = (G, t_0) \) explains an initial program \( t_{\text{init}} \), if there exists an instantiation \( \beta : \text{var}(t_0) \rightarrow \mathcal{T}_\Sigma(\beta) \) of the parameters of the main program \( t_0 \) and a term \( t \in \mathcal{L}(\mathcal{S}, \beta) \), such that \( t_{\text{init}} \leq \Omega \), \( S \) is a recurrent explanation of \( t_{\text{init}} \), if furthermore exists a term \( t' \in \mathcal{L}(\mathcal{S}, \beta) \) which can be derived by at least two applications of rules \( \mathcal{R}_S \), such that \( t'_I \leq \Omega \). Analogously we say that a set of tc equations (or a particular recursive equation) explains a (reduced) initial program. An equation/RPS explains a set of initial programs \( t_{\text{init}} \), if it explains all terms \( t_{\text{init}} \in T_{\text{init}} \) and if there is a recurrent explanation for at least one of them.

B. Characteristics of Program Schemes

There are some restrictions of RPSs which can be folded using our approach:

No Nested Program Calls: Calls of recursive equations within substitution terms of another call of an equation are not allowed, that is \( \text{sub}(x, r) : X \times R \rightarrow \mathcal{T}_\Sigma(X) \), and not \( \text{sub}(x, r) : X \times R \rightarrow \mathcal{T}_{\Sigma,\Phi}(X) \) for each equation in \( G \).

No Mutual Recursion: There are no recursive equations \( G_i, G_j, i \neq j \), with \( G_i \text{ calls}^* G_j \) and \( G_j \text{ calls}^* G_i \), that is, the relation \( \text{calls}^* \) is antisymmetric.

The first restriction is semantical, that is, it reduces the class of calculable functions which can be folded. The second restriction is only syntactical since each pair of mutually recursive functions can be transformed into semantically equivalent functions which are not mutually recursive.

The appearance of RPSs which will be folded are characterized in the following way: For each program body \( t_i \) of an equation in \( G \) it holds

- \( \text{var}(t_i) = \{x_1, \ldots, x_{m_i}\} \), that is, all variables in the program head are used in the program body, and
- \( \text{pos}(t_i, G_i) \neq \emptyset \), that is, each equation is recursive.

These characteristics do not restrict the class of RPSs which can be folded.

Definition 1 (Substitution Uniqueness of an RPS) An RPS \( S = (G, t_0) \) over \( \Sigma \) and \( \Phi \), which explains recursively a set of initial trees \( T_{\text{init}} \) is called substitution unique wrt \( T_{\text{init}} \) if there exists no \( \Sigma \) over \( \Phi \) which explains \( T_{\text{init}} \) recursively and for which it holds: (a) \( t'_i = t_0 \), (b) for all \( G_i \in \Phi \) holds \( \text{pos}(t_i, G_i) = \text{pos}(t'_i, G_i) = U_{\text{rec}}(\Omega \rightarrow \emptyset) = t_i[t_i[U_{\text{rec}} \leftarrow \emptyset] \), and it exists an \( r \in R \) with \( \text{sub}(x_j, r) \neq \text{sub}(x_j, r) \).

Substitution uniqueness guarantees that it is not possible to replace a substitution term in \( S \), such that the resulting RPS \( S' \) still explains a given set of initial programs recursively. It can be shown that each RPS satisfying the given characteristics is substitution unique wrt the set of all terms which can be derived by it.

C. The Synthesis Problem

Now all preliminaries are given to state the synthesis problem:

Definition 2 (Synthesis Problem) Let \( T_{\text{init}} \subset \mathcal{T}_{\Sigma,\Phi}(\Omega) \) be a set of initial programs. The synthesis problem is to induce

a signature \( \Sigma' \),

a set of function variables \( \Phi = \{G_1, \ldots, G_n\} \),

an RPS \( S = (G, t_0) \) with a main program \( t_0 \in \mathcal{T}_{\Sigma,\Phi}(X) \) and a set of recursive equations \( G = \langle G_1(x_1, \ldots, x_{m_1}) = t_1, \ldots, G_n(x_1, \ldots, x_{m_n}) = t_n \rangle \) such that

- \( S \) recursively explains \( T_{\text{init}} \), and
- \( S \) is substitution unique (Def. 1).

IV. A Simple Example

We will illustrate our method, described in the following sections, by means of a (very) simple example. Consider the following RPS \( S = (G, t_0) \):

\[ G = \langle G(x) = \text{if} (\text{eq}(0(x), 1), \ast (x, G(\text{pred}(x))) \rangle, \ t_0 = G(x) \]

If the symbols are interpreted in the usual way, this RPS calculates the factorial function. An example for an initial program which can be generated by this RPS with initial instantiation \( \beta(x) = \text{succ}(\text{succ}(0)) \) is \( t_{\text{init}} = \text{if}(\text{eq}(0(\text{succ}(\text{succ}(0))), 1, \ast (\text{succ}(\text{succ}(0)), 0)), \text{pred}(\text{succ}(\text{succ}(0))))) \). This initial term might be an input term for our folding algorithm. In a first step (Sect.V-A), the recursion points (positions, at which a recursive call appeared) will be inferred. Only those positions are possible which lie on a path leading to an \( \Omega \) (only one in this example).

Furthermore, it must hold that the positions between the root and the recursion point reiterate itself up to arriving the \( \Omega \). This results in \( U_{\text{rec}} = \{3 \ 2 \ \lambda \} \) in this example. The minimal pattern which includes the mentioned symbols is \( t_{\text{init}} = \text{if}(x_1, x_2, \ast (x_3, \Omega)) \). The recursion point devives the initial term into three segments:

- \( \text{if}(\text{eq}(0(\text{succ}(\text{succ}(0))), 1, \ast (\text{succ}(\text{succ}(0)), 0)), \text{pred}(\text{succ}(\text{succ}(0))))) \),
- \( \text{if}(\text{eq}(0(\text{pred}(\text{succ}(\text{succ}(0))), 1, \ast (\text{pred}(\text{succ}(\text{succ}(0))), 0), \text{pred}(\text{succ}(\text{succ}(0))))) \),
- \( \text{if}(\text{eq}(0(\text{pred}(\text{pred}(\text{succ}(\text{succ}(0)))), 1, \ast (\text{pred}(\text{pred}(\text{succ}(\text{succ}(0)))), 0))), \text{pred}(\text{pred}(\text{succ}(\text{succ}(0))))) \)

Since there are no further \( \Omega \) in the initial tree, the searched for equation can only contain the found position as recursion point. In more complex RPSs it might happen that not all \( \Omega \) can be explained in the described way. In this case, we assume that another equation is called by the searched for equation. An RPS for the resulting subtrees will be induced seperately.

In a second step (Sect.V-B), the program body will be constructed by extending the term \( t_{\text{rec}} \) by all positions which remain constant over the segments. This method results in the term \( t_{\text{cl}} = \text{if}(\text{eq}(0(x_1), 1, \ast (x_1))) \). Identical subtrees in one segment are represented by the same variable (\( x_1 \) in our example).
At last (Sect. V-C), the substitution terms for the variables will be inferred. The subtrees which differ over the segments represented by the variables in $G_2$ are instantiations of the parameters in the recursive equation. For the single variable in our example we obtain the trees:

\[
\begin{align*}
\text{succ}(\text{succ}(0)) & \quad (1) \\
\text{pred}(\text{succ}(\text{succ}(0))) & \quad (2) \\
\text{pred}(\text{pred}(\text{succ}(\text{succ}(0)))) & \quad (3)
\end{align*}
\]

Now a recurrence relation will be searched such that the instantiations in one segment can be generated by instantiations in the preceding segment in a recurrent way. This results in the substitution term $\text{sub}(x_1, 1) = \text{pred}(x_1)$, since for the instantiations in the three segments it holds $(2) = \text{pred}(1)$, and $(3) = \text{pred}(2)$.

With our method we are able to fold initial trees for RPSs of substantially more complexity, namely with a constant part in the main programs, with more than one recursive equation, and with interdependent, switching and hidden variables in the equations. In the following, we present the theory and methodology formally, concluding each section with a theorem which backs up our approach.

V. INDUCING A SUBPROGRAM

A. Segmentation

When we use the segmentation, the hypothesis is that there exists a set of $t_c$ equations $G_i$ with initial equation $G_i(x_1, \ldots, x_m) = t_i$ which explains a set of given initial trees $T_{init}$. The goal is to find a set of possible recursion points $U_{rec}$ for such an equation $G_i$, which divides the initial trees into segments.

Each equation of an RPS occurs in some context. For instance, the main program might be a term $t_0 = G_1([1, 2, 3])$, that is, $G_1$ is called in an “empty” context. Within $G_1$, another equation $G_2$ might be called with in a term $t' = \text{cons}(42, G_2(x))$. In this case, $\text{cons}(42, \ldots)$ is the context of $G_2$. In the following, we call a recursive equation together with its context a sub-program. Consequently, for each recursive equation $G_i$, a set of hypothetical recursion points $U_{rec}$ and a (possibly empty) set of sub-scheme positions $U_{sub}$ must be determined.

Sub-scheme positions. The set of sub-scheme positions of an equation $G_i$ is defined as $U_{sub} = \{ u \circ k \in t_i \mid \exists u_{rec} \in U_{rec} : u < u_{rec}, \beta u_{rec} \in U_{rec} : u \circ k \leq u_{rec}, \text{exists } \rho \in \text{pos}(t_i, G_i) : i \neq j \text{ and } u \circ k \leq u_{rec}\}$. Sub-scheme positions in a program term indicate the call of another subprogram in the subterm at a sub-scheme position. Moreover it must hold, that the sub-scheme positions are the least positions in a program term at which a call of a subprogram is possible wrt the condition of no nested program calls.

The $\Omega$s in an initial tree give us a first restriction for the choice of possible recursion points, since each recursion point (and moreover each unfolding point constructed over the recursion points) and also each sub-scheme position composed to an unfolding point must lie on a path leading to an $\Omega$ in the initial tree. We say that an $\Omega$ is explained by the recursion points and sub-scheme positions. It must hold that all $\Omega$s in $T_{init}$ are explained in this way.

Method. The method is to search in the initial trees for possible recursion points. If a position was found, the set of sub-scheme positions is determined by the yet unexplained $\Omega$s. Such a pair $(U_{rec}, U_{sub})$ is called a valid hypothesis of recursion points and sub-scheme positions. Up to now, we only regarded the structure of initial trees. We now regard additionally the symbols (i.e., the node labels) which lie between the root of the initial trees and the so far inferred recursion points. These symbols imply a special minimal pattern of the body of the searched for subprogram, which is defined by means of the function $\text{skeleton}$.

Definition 3 (Skeleton) The skeleton of a term $t \in T_{init}(\Omega) \{X\}$, written $\text{skeleton}(t)$, is the minimal pattern of $t$ for which holds $\text{pos}(t, \Omega) = \text{pos}(\text{skeleton}(t), \Omega)$.

Obviously it holds: $\Omega \leq_0 \text{skeleton}(t)$. The mentioned pattern is named $\text{skeleton}$ and defined as $\text{skeleton}(t_{init}[U_{rec} \cup U_{sub} \leftarrow \Omega])$. As stated in the following definition, $\text{skeleton}$ must reiterate itself at each unfolding point constructed over the calculated recursion points, since each unfolding point indicates an unfolding of the same recursive equation $G_i$ (see Lemma 1).

Definition 4 (Valid Segmentation) Let $U_{unf}$ be the unfolding points constructed over a set of possible recursion points $U_{rec}$. The hypothesis $(U_{rec}, U_{sub})$ together with $\text{skeleton}$ is called a valid segmentation of an initial tree $t_{init}$ if for all $u_{unf} \in U_{unf} \cap \text{pos}(t_{init})$ holds: Given the set of all positions of $\Omega$ in $t_{init}$, which are above the recursion points as $U_{rec} = \{ u \mid \text{pos}(t_{init} | u, \Omega) \}, \exists u_{rec} \in U_{rec}$ with $u < u_{rec}$, it holds $\text{skeleton}[U_{rec} \leftarrow \Omega] \leq \Omega$. $U_{rec}$ with $\text{skeleton}$ is called a valid recurrent segmentation of $t_{init}$, if additionally holds that: $\exists u_{rec} \in U_{rec} : U_{rec} \subseteq \text{skeleton}[u_{rec}]$.

If a hypothesis can be verified by means of $\text{skeleton}$, then the trees will be searched for further possible recursion points, because the algorithm should detect a set of recursion points as large as possible. This sequence of steps for inferring a valid segmentation – (a) search for a further possible recursion point, (b) calculate the set of sub-scheme positions, (c) calculate $\text{skeleton}$ and verify the hypothesis – will be repeated, until no further possible recursion point can be found.

It is possible, that a valid segmentation doesn’t lead to an equation which explains the initial trees together with a set of $t_c$ equations. Therefore, the presented method results in a backtracking algorithm.

Theorem 1 (Segmentation Theorem) If a set of $t_c$ equations $G_i$ with initial equation $G_i(x_1, \ldots, x_m) = t_i$ with recursion points $U_{rec}$ and sub-scheme positions $U_{sub}$ explains an initial tree $t_{init}$ for an initial instantiation $\beta : \{x_1, \ldots, x_m\} \rightarrow T_\Sigma$, then $(U_{rec}, U_{sub})$ is a valid segmentation for $t_{init}$.

If $G_i$ explains a set of initial trees $T_{init}$, then $(U_{rec}, U_{sub})$ together with $\text{skeleton}$ is a valid segmentation for all trees in $T_{init}$ and a valid recurrent segmentation for at least one tree in $T_{init}$.

Proof: Let be $T_\Sigma$ the set of all unfoldings of equation $G_i$ and $u_{unf} \in U_{unf} \cap \text{pos}(t_{init})$ an arbitrary unfolding point in
If a valid recurrent segmentation \( (U_{rec}, U_{sub}) \) was found for a set of initial trees \( T_{ini} \) and if \( U_{sub} \neq \emptyset \), then induction of an RPS is performed recursively over the set of subtrees at the positions in \( U_{sub} \). Therefore, for each \( u \in U_{sub} \) a new set of initial trees will be constructed by including the subtrees at position \( u \) in each segment of each tree in \( T_{ini} \) to \( T_{ini}' \).

**B. Inducing a Program Body**

When we start with inferring the body of subprogram \( G_t \), a valid segmentation \( (U_{rec}, U_{sub}) \) is given for \( T_{ini} \). Moreover, for each sub-scheme position \( u \in U_{sub} \) a sub-scheme \( S^u = (G^u, \theta^u) \) of \( G_t \) which explains the subtrees in \( T_{ini}' \) is already induced. That allows us to fold (the inverse process of unfolding) the initial trees to reduced initial trees, denoted by \( T_{red} \), which can be explained by one recursive equation (the initial equation of the set of tc equations explaining \( T_{ini} \), Sect.II-C).

**Method.** A subprogram body is uniquely determined by a valid segmentation:

**Definition 5 (Valid Subprogram Body)** For a valid segmentation \( (U_{rec}, U_{sub}) \) of \( T_{ini} \) and the reduced initial trees \( T_{red} \), the term \( \xi_G \in \mathcal{T}_{\mathcal{D}, \phi \cup G}(X) \) is defined as the maximal pattern of all complete segments \( \{\xi_G\}_{\xi \in \mathcal{T}_{\mathcal{D}, \phi \cup G}(X)} \) and initial instantiations of \( X \) which explain the trees in each \( T_{ini} \). \( \xi \in \mathcal{T}_{\mathcal{D}, \phi \cup G}(X) \) is defined as the maximal pattern of all complete segments \( \{\xi_G\}_{\xi \in \mathcal{T}_{\mathcal{D}, \phi \cup G}(X)} \) and initial instantiations of \( X \) which explain the trees in each \( T_{ini} \).

A segment is complete, if it includes all recursion points in \( U_{rec} \).

The following lemma states that this method results in a valid solution.

**Lemma 2 (Maximization of the Body)** Let \( E \) be a finite index set with indices \( e \in E \). Let \( G(x_1, \ldots, x_n) = \xi_G \) be a recursive equation with \( X = \{x_1, \ldots, x_n\} \), \( \xi_G \in \mathcal{T}_{\mathcal{D}, \phi \cup G}(X) \) and initial instantiations \( \beta_e : X \rightarrow T_S \) for all \( e \in E \). Then it exists a recursive equation \( G'(x_1, \ldots, x_{n'}) = \xi_G \) with \( X' = \{x_1, \ldots, x_{n'}\} \), \( \xi_G' \in \mathcal{T}_{\mathcal{D}, \phi \cup G}(X') \), unfolding points \( U_{unf} \) and initial instantiations \( \beta_e' : X' \rightarrow T_S \) for all \( e \in E \), such that \( L(G, \beta_e) = L(G', \beta_e') \) for each \( e \in E \). Additionally, for each \( x \in X' \) it holds that the instantiations which can be generated by \( G' \) from \( \beta_e' \) are \( \xi_G(\beta_e, u_{\text{ini}})(x) \) for \( e \in E \), \( u_{\text{ini}} \in U_{unf} \) do not share a common not-trivial pattern.

**Proof:** It can be shown that substitution terms exist which generate the instantiations in the unfoldings of elements in \( L(G', \beta_e') \) from the initial instantiation \( \beta_e' \). The idea is to extend the body of equation \( G \) by the common pattern of the instantiations in the unfoldings of the elements of \( L(G, \beta_e) \). The initial instantiation will be reduced by this pattern. It must be considered, that variables can be interdependent.

The maximal pattern of a set of terms can be calculated by first order anti-unification. Only complete segments are considered. For incomplete segments, it is in general not possible to obtain a consistent introduction of variables during generalization. The variables in the resulting subprogram body represent that subtrees which differ over the segments of the initial trees. Identical subtrees are represented by the same variable.

**Equivalence of sub-schemes.** Because we handle induction of sub-schemes \( S^u \) as independent problem, we must ensure that if for each \( u \in U_{sub} \) exists a sub-scheme \( S^u \) which explains the trees in \( T_{ini} \) and it exists a recursive equation which explains the resulting reduced initial trees \( T_{red} \), then for arbitrary other sub-schemes \( S^w \) explaining the trees in each \( T_{ini}' \), it exists a recursive equation which explains the resulting reduced initial trees \( T_{red}' \). Such an “Equivalence of Sub-Schemes”-condition can be shown by considering the parameters with the initial instantiations of two different schemes which both explain the same initial trees. If they are constructed by maximizing the body, as described, it holds, that the parameters and instantiations are equal.

**Theorem 2 (Subprogram Body Theorem)** If it exists a set of tc equations \( G' \) with initial equation \( G'(x_1, \ldots, x_{n'}) = \xi_G \) with recursion points \( U_{rec} \) and sub-scheme positions \( U_{sub} \) which explains \( T_{ini} \), then it exists a set of tc equations \( G' \) with initial equation \( G(x_1, \ldots, x_{n}) = \xi_G \) which explains \( T_{ini}' \) such that it holds: \( \xi_G[U_{rec} \leftarrow G] = \xi_G \). (Proof follows from Def. 5, Lemma 2 and the fact, that two independently inferred sub-schemes are equivalent (see text).)

C. Inducing Substitution Terms

The subtrees of the trees in \( T_{red} \) which differ over the segments represent instantiations of the variables \( \text{var} (\xi_G) \). The goal is to infer a set of variables \( X \) with \( \text{var} (\xi_G) \subset X \) and substitution terms for each variable and each recursion point in \( U_{rec} \), such that the corresponding subtrees can be generated by an initial instantiation and compositions (Sect.II-C) of the inferred substitution terms. We are able to deal with quite complex substitution terms. Consider the following examples:

\[
\begin{align*}
\beta_1(x, y) &= y, \\
\beta_2(x, y) &= \text{eq}(x, y), \beta_3(x) &= \text{sub}(x, y)
\end{align*}
\]

A variable might be substituted by an operation involving other program parameters (4). Additionally, variables can switch there positions (given in the head of the equation) in the recursive call (5). Finally, there might be “hidden” variables which only occur within the recursive call (6). If you look at the body of \( f_3 \), variable \( x \) occurs only within the recursive call. The existence of such a variable cannot be detected when the program body is constructed by anti-unification but only a step later, when substitutions for the recursive call are inferred.

**Method.** The method for inferring substitution terms is based on the fact that instantiations of variables in an unfolding (resp. a segment) can be generated by instantiating the variables in the substitution terms with the instantiations of the preceding unfolding. \( \beta_{\text{ini}} \in U_{\text{ini}}(x) \) for arbitrary variable and recursion point index.
Moreover – starting with $u = \lambda$ – it holds that the substitution terms are inductively characterized by:

$$\forall x_k \in U_{\text{rec}}, \beta_{u_{\text{rec}}} = \beta_{u_{\text{rec}}} (x_k) \quad \forall x_k \in U_{\text{rec}}, \beta_{u_{\text{rec}}} (x_k) \quad \beta_{u_{\text{rec}}} = f \in \Sigma$$

$$\text{with arity } \alpha_f = n.$$  

This characterization can directly be transformed into a recursive algorithm which calculates the substitution terms, if we identify the instantiations in the unfoldings with the subtrees in $T_{\text{red}}$ which differ over the segments. If for a variable in $\text{var}(t_G)$ no substitution term can be calculated, because at a position $u$ it holds neither condition (a) nor (b), then it will be assumed that a "hidden" variable (that is $x \not\in \text{var}(t_G)$) is on this position in the substitution term. The set of variables $X$ will be extended by a new variable $x \not\in X$ and then reversed application of condition (a) yields the instantiations of the new variable in the unfoldings. The substitution terms for the new variable will be generated as described.

Incomplete unfoldings. Because we allow for incomplete unfoldings in the initial trees, it can occur that for a particular variable no represented subtree in an unfolding can be found. In this case, the instantiation of the variable is undefined in this unfolding and will be set to $\bot$. This can result in substitution terms which generate subtrees (representing instantiations) of the subtrees in the preceding unfolding (as described above), but don’t generate the same instantiations by applying the original definition (Sect.II-C). Furthermore, it can occur that a substitution term is not unique w.r.t Definition 1. By means of the following definition, the inferred substitution terms will be verified wrt the mentioned problems.

**Definition 6 (Valid Substitution Terms)** Let be $X_h$ a set of hidden variables with $X_h \cap \text{var}(t_G) = \emptyset$. $X = X_h \cup \text{var}(t_G)$ the set of inferred variables. The set of all substitution terms $\text{sub}(x, r), x \in X, r \in R$ is called valid, if it holds:

**Consistency:** $\forall t_e \in T_{\text{red}} \exists \beta_e : X \rightarrow T_{\Sigma}; \forall x_j \in \text{var}(t_G), u_{\text{rec}} \in U_{\text{rec}} \cap t, u \in \text{pos}(t_e, x_j) : t_e |_{u_{\text{rec}}} = \beta_e (\text{sub}^*(x_j, u_{\text{rec}})).$

**Uniqueness:** It not exists a variable $x \in X$ and a recursion point $u$, such that another substitution term $\text{sub}(x, r) \neq \text{sub}(x, r)$ which is consistent is.

**Minimality:** Let be $X_h \subseteq X_h$ a set of variables and $X' = X_h \cup \text{var}(t_G)$: It doesn’t exist a set of consistent and unique substitution terms for the variables $X'$.  

**Theorem 3 (Substitution Theorem)** Let $\text{sub} : X \times R \rightarrow T_{\Sigma}(X)$ with $X = \{x_1, \ldots, x_n\}$ be valid substitution terms. Let $G = \{x_1, \ldots, x_n\}$ be a recursive equation such that it holds: (a) $t_{\text{rec}} [U_{\text{rec}} \leftarrow G] = t_{\text{rec}}$ and (b) $\forall r \in R, x_j \in X : t_{\text{rec}} |_{u_{\text{rec}}} = \text{sub}(x_j, r)$. Then it holds: The set of tc equations $G = G^u \cup \{G(x_1, \ldots, x_n) = t_{\text{rec}}\}$ explains $T_{\text{red}}$ substitution unique.

**Proof:** From the condition of consistence in Definition 6 and Theorem 2 follows directly that equation $G$ explains $T_{\text{red}}$ constructed over $T_{\text{inv}}$ and the sub-schemes $S^n$. From the condition of uniqueness follows directly that this explanation is substitution unique (as postulated in Def. III-C). Additionally, with Lemma 1 follows that $G = G^u \cup \{G(x_1, \ldots, x_n) = t_{\text{rec}}\}$ explains $T_{\text{inv}}$ and from the substitution uniqueness of all sub-schemes $S^n$ and equation $G$ follows that the explanation of $T_{\text{inv}}$ by $G$ is substitution unique too.  

**VI. INDUCTING AN RPS**

If an RPS can be induced from a set of initial trees using the approach presented here, then induction can be seen as a proof of existence of a recursive explanation – given the restrictions presented in Sect.III-B.

**Theorem 4 (Existence of an RPS)** Let $T_{\text{inv}}$ be a set of initial trees indexed over $E$. $T_{\text{inv}}$ can be explained recursively by an RPS iff:

1. $T_{\text{inv}}$ can be recursively explained by a set of tc equations, or
2. $\forall e \in E : \exists f \in \Sigma$ with $\alpha(f) = n, n > 0$, and $\text{node}(t_e, \lambda) = f$, and $\forall t_k = \{t_e |_{k \lambda} | k = 1, \ldots, n\}$ it holds:

(a) $\forall t \in T^k : \text{pos}(t, \Omega) = \emptyset$, or
(b) $\forall t \in T^k : \text{pos}(t, \Omega) \neq \emptyset$ and it exists an RPS $S^k = (G^k, t_0)$ which recursively explains the trees in $T^k$.

(Proof in [6].)

To inductively construct an RPS from a set of initial trees, first a valid segmentation for the initial trees is searched-for. If one is found, body and substitutions are constructed as described above; if segmentation fails, recursively sub-schemes for the subtrees will be induced, and at the end, the final RPS will be constructed. A consequence of inferring sub-schemes separately is that subprograms are introduced locally with unique names. It can happen that two such subprograms are identical.

**VII. CONCLUSION**

We presented a new, powerful approach to folding of finite program terms, focussing on the formal framework. The synthesis algorithms and a variety of examples can be found in [6]. The induction method described in this paper is able to deal with all (tail, linear, tree recursive) structures which can be generated by a set of recursive functions of arbitrary complexity. We already demonstrated the applicability of our approach to control-rule learning for planning [4]. In future we plan to investigate applicability to grammar learning and to enduser programming.

**REFERENCES**


