Inductive Program Synthesis: From Theory to Application

Ute Schmid[*], **Emanuel Kitzelmann and Fritz Wysotzki**[**]

[*] Dept. of Mathematics/Computer Science, University of Osnabrück, D-49069 Osnabrück, Email: schmid@informatik.uni-osnabrueck.de

[**] Dept. of Electrical Eng. and Computer Science, Technical University Berlin, Franklinstr. 28, D-10587 Berlin, Email [jemanuel,wysotzki]@cs.tu-berlin.de

Abstract. We present an approach to folding of finite program terms based on the detection of recurrence relations in a single given term which is considered as the kth unfolding of an unknown recursive program. Our approach goes beyond Summers' classical approach of inductive program synthesis in several aspects and allows to deal with a larger class of programming problems. We show how inductive synthesis of recursive programs can be applied to control-rule learning and enduser programming: finite program terms are generated using domain-specific techniques and then presented as input to our folding algorithm. In the case of control-rule learning, the finite terms are obtained from a state-based planner and the resulting recursive functions represent knowledge how to guide plan construction. In the case of enduser programming, the user presents examples of the desired input/output relation, and a finite program, which transforms exactly the given IO pairs, is constructed by generate-and-test. The folded recursive program generalizes over the input domain and therefore can be applied to a larger class of problems.

Keywords. Inductive Program Synthesis, Folding, Recursive Program Schemes, Control Rule Learning, Enduser Programming

1 Introduction

Automatic induction of recursive programs from input/output examples is an active area of research since the sixties and of interest for AI research as well as for software engineering [LM91]: program synthesis from incomplete information is a challenging problem for machine learning algorithms, addressing discovery of generalized rules from observations. In the domain of knowledge-based software engineering, inductive synthesis is researched as a tool for knowledge inference in specific domains.

In the seventies and eighties, there were several approaches to the synthesis of Lisp programs from examples or traces (see [BGK84] for an overview). Due to only limited progress, interest decreased in the mid-eighties and research focussed on inductive logic programming (ILP) instead (see [FY99]). Although ILP proved to be a powerful approach to learning relational concepts, applications to learning of recursive clauses had only moderate success. Probably, one must accept the fact that *fully-automated* inductive program synthesis can only be applied to small programming problems and does not scale-up to synthesis of complex algorithms. Even the most succesful deductive system – the transformational system KIDS [Smi90] – depends on background knowledge in form of program schemes and needs user-interaction for selection of the appropriate scheme. Nevertheless, we belief that automatic inductive synthesis is still a problem of some interest to basic research – giving us insights in what class of programs can be learned from examples – and it might be one useful approach among others in tools to support enduser programming [SE99].

Our approach is based on the recurrence detection method of Summers [Sum77]. 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 [SE99] or constructed by AI planning [SW00]. In the following, we are mainly concerned with the second step of program synthesis – folding a finite program term in a recursive program. This corresponds to program synthesis from traces and is an interesting problem in its own right [Smi84]. Providing a powerful approach to folding is crucial for de-

I/O Examples:
$\{nil \rightarrow nil,\ (A) \rightarrow ((A)),\ (A\ B) \rightarrow ((A)\ (B)),\ (A\ B\ C) \rightarrow ((A)\ (B)\ (C))\}$

Derived Trace: ("initial program")

$$\begin{aligned} F_L(x) \leftarrow (&atom(x) \rightarrow && nil,\\ &atom(cdr(x)) \rightarrow && cons(x, nil),\\ &atom(cddr(x)) \rightarrow && cons(cons(car(x), nil),\\ & && cons(cdr(x), nil)),\\ &T \rightarrow cons(cons(car(x), nil), && cons(cons(cadr(x), nil),\\ & && cons(cddr(x), nil))))\end{aligned}$$

Recursive Generalization:

$$\begin{aligned} unpack(x) \leftarrow \quad & (atom(x) \rightarrow nil,\\ & \ T \rightarrow u(x))\\ u(x) \leftarrow \quad & (atom(cdr(x)) \rightarrow cons(x, nil),\\ & \ T \rightarrow cons(cons(car(x), nil), u(cdr(x))))\end{aligned}$$

Figure 1 Summers' **unpack** Example

velopping synthesis tools for practical applications.

In the next section we will give an informal review of Summers seminal work. Afterwards, we give an informal overview of our own approach which is presented in detail elsewhere [KSuW02, Sch01]. In the following two sections we show applications to control-rule learning and enduser programming and we conclude with a short evaluation and further work to be done.[1]

2 Detection of Recurrence Relations in Terms

Inductive program synthesis based on recurrence detection in finite terms was made popular by Summers [Sum77] who put inductive synthesis on a firm theoretical foundation. We first present an example (see Figure 1) to illustrate the general idea. We use the original Lisp notation of Summers: *atom(x)* tests whether x is atomar (here *nil*, representing the empty list) or a (non-empty) list; *car(x)* and *cdr(x)* return the head/tail of a non-empty list, combinations of these functions can be written abbreviated, such as *cadr(x)* for *car(cdr(x))*; *cons(x, y)* is the list constructor, adding x in front of list y.

Input is a small, positive set of examples for the desired input/output behavior of the Lisp program. The user is expected to present the *first* elements of the input parameter, e.g., a list with no, one, two, and three elements. Summers' approach is restricted to lists as input parameter. He uses the complete partial order over lists as knowledge for trace-generation. The derived trace is input for the folding algorithm. It represents how the first k inputs can be transformed into the output. Summers' approach is restricted to a small set of primitive functions and he considers structural list problems only, that is, problems which can be solved using knowledge about the *structure* of the input list but without knowledge about the *content* (e.g., the *reverse* function can be learned buut not the *member* function). Due to that restriction there exists a unique expression relating input and output for each I/O pair.

The folding algorithm is based on detection of recurrence relations using a pattern-matching approach. For the example in Figure 1 the following relations hold:

$$\begin{aligned} &f_1(x) = nil\\ &f_2(x) = cons(x, f_1(x))\\ &f_{k+1}(x) = cons(cons(car(x), nil), f_k(cdr(x))) \text{ for } k = 2, 3\end{aligned}$$

$$\begin{aligned} &p_1(x) = atom(x)\\ &p_{k+1}(x) = p_k(cdr(x)) \text{ for } k = 1, 2, 3.\end{aligned}$$

These relations are used to fold the trace into the recursive program in Figure 1.

Summers provided a synthesis theorem which can be seen as a justification why generalization of traces based on recurrence detection is legal. He exploits the relation between a given recursive function and its sequences of unfoldings as it is used in fixed point semantics [SS71]. The theorem represents the *converse* idea, that is, to find a recurrence relation characterization of a partial function, which is considered as the k-th unfolding of some unknown recursive function.

Theorem 1 (SUMMERS' BASIC SYNTHESIS THEOREM)
Given that each set of approximating functions $F_i(x)$ for a function $\mathbf{F}(x)$ is an ascending chain, then it holds that $\mathbf{F}(x) = \mathbf{sup}\{F_i(x)\}$, $i \rightarrow \infty$. If a set of examples defines $\mathbf{F}(x)$ with recurrence relations
$f_1(x), \ldots, f_n(x),\ f_{i+n}(x) = C(f_i(b(x)), x)$,
$p_1(x), \ldots, p_n(x), p_{i+n}(x) = p_i(b(x))$
für $i \geq 1$, then $\mathbf{F}(x)$ is equivalent to the following recursive program:

$$\begin{aligned} F(x) \leftarrow \quad & (p_1(x) \rightarrow f_1(x),\\ & \ \ldots\\ & \ p_n(x) \rightarrow f_n(x),\\ & \ T \rightarrow C(F(b(x)), x)).\end{aligned}$$

(proof, see [Sum77], pp. 168)

The recursive program scheme given in Summers' synthesis theorem contains a single recursive call, that is, the theorem holds only for *linear* recursion.

3 A Generalized Framework for Folding

Our approach extends Summers' approach in several aspects: first, it is language independent and works for terms belonging to an arbitrary term algebra [Wys83], while Summers was restricted to Lisp programs. This

[1]Please note that this paper is intended as an *overview* of our work, focussing on the underlying ideas. Therefore, we heavily refer to other papers of our group where the formal framework, technical details, and more complex examples are presented.

allows us to deal with a variety of problem domains, such as AI blocksworld problems or XSL terms, in addition to classical functional programming problems. 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. These extensions allow us to induce a larger class of programs corresponding to a more complex recursive program scheme than Summers' linear one. Finally, the given input terms can represent incomplete unfoldings of an hypothetical recursive program, which is important if the program terms to be folded are obtained by other sources, such as a user or an AI tool.

3.1 The Synthesis Problem

In general, our folding approach complies to the following recursive program scheme:

Definition 1 (Recursive Program Scheme) *Let Σ 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) $\mathcal{S}$ is a pair $(\mathcal{G}, t_0)$ with $t_0 \in \mathcal{T}_{\Sigma\cup\Phi}(X)$ and $\mathcal{G}$ as a system of n equations:*

$$\mathcal{G} = \begin{array}{ll} \langle G_1(x_1, \ldots, x_{m_1}) & = t_1, \\ \vdots & \\ G_n(x_1, \ldots, x_{m_n}) & = t_n \rangle \end{array}$$

with $t_i \in \mathcal{T}_{\Sigma\cup\Phi}(X), i = 1 \ldots n$.

This RPS characterizes the typical form of functional programs: we have an initial program call t_0 and a system of recursive equations $\mathcal{G}$. The initial program call is typically a grounded term, that is, variables occurring in a function head are instantiated with terms from $\mathcal{T}_\Sigma$. We will also use the notion of a "recursive subprogram", refering to a recursive equation together with the term in which context equation G_i is called.

In the following, we will see, that induction of such an RPS from some finite term ("initial program") involves the following steps:

1. Identification of recursion points, that is, positions in the term which correspond to the position of recursive calls in an equation.
2. Construction of the program body, that is, obtaining all parts of the term which are constant over the unfoldings of an recursive equation.
3. Identification of substitution terms, that is, obtaining the parameters of a recursive equation, their initial values, and their substitution in the recursive call.

As Summers, we can formulate a synthesis theorem which exploits the known relation between an RPS and its unfoldings for inductive sythesis; but, because our hypothesis language is more complex, we must provide a more general framework: an RPS can be considered as a term rewrite-system and the language of an RPS is the infinite set of its unfoldings [Sch01]:

Definition 2 (RPS as Rewrite System) *Let $\mathcal{S} = (\mathcal{G}, t_0)$ be an RPS and Ω a special symbol representing an undefined term. The equations in $\mathcal{G}$ constitute rules $\mathcal{R}_\mathcal{S}$ of a term rewrite system. The system additionally contains rules $\mathcal{R}_\Omega$:*

- $\mathcal{R}_\mathcal{S} = \{G_i(x_1, \ldots, x_{m_i}) \to t_i \mid i \in \{1, \ldots, n\}\}$,
- $\mathcal{R}_\Omega = \{G_i(x_1, \ldots, x_{m_i}) \to \Omega \mid i \in \{1, \ldots, n\}\}$.

We write $\xrightarrow{*}_{\Sigma,\Omega}$ for the reflexive and transitive closure of the rewrite relation implied by $\mathcal{R}_\mathcal{S} \cup \mathcal{R}_\Omega$.

Definition 3 (Language of an RPS) *Let $\mathcal{S} = (\mathcal{G}, t_0)$ be an RPS and $\beta : \mathbf{var}(t_0) \to \mathcal{T}_\Sigma$ an instantiation of the parameters in t_0. The set of all terms $\mathcal{L}(\mathcal{S}, \beta) = \{t \mid t \in \mathcal{T}_{\Sigma\cup\{\Omega\}}, \beta(t_0) \xrightarrow{*}_{\Sigma,\Omega} t\}$ is the language generated by the RPS with instantiation β. For a main program t_0 without variables, we can write $\mathcal{L}(\mathcal{S})$.*

We showed that the words belonging to the language of an RPS can be constructed inductively, that is, we can give a strong characterization of the relation between an RPS and its unfoldings which we can exploit for synthesis. Stated informally, an RPS *recursively explains* some finite term t_{init}, if there exists an unfolding t such that t_{init} is a prefix of t and an unfolding t', derived by at least two applications of all rewrite-rules $\mathcal{R}_\mathcal{S}$ such that t' is a prefix of t_{init} (see [KSuW02] for the relevant lemma).

There are some restrictions and characteristics of RPSs which can be folded using our approach. RPSs which can be folded are restricted such that no nested program calls and no mutual recursion is allowed. The first restriction is semantical, that is, the class of RPSs which can be folded is *smaller* than the class of calculable functions. 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. That means, an RPS with mutual recursions cannot be folded from an element of its language but can be transformed in a semantically equivalent RPS which can be folded.

Furthermore, our RPSs have the characteristics that all variables occuring in the head of an equation must occur in the body and that each equation in $\mathcal{G}$ is recursive. Both characteristics do not limit expressiveness: Variables which never are used in the body of a program do not contribute to the computation of a result and the language of the RPS. The second restriction ensures that induction of an RPS from a finite program term is really due to generalization (learning) and that the resulting RPS does not just represent the given finite program itself. Each RPS with some non-recursive equations can be transformed to a semantically and syntactically equivalent RPS with only recursive equations.

Crucial for our approach is, that an RPS folded from a finite term must be *minimal*, that is, it only contains recursive equations which are called at least once, each parameter in the head of an equation must be used at least once for instantiating a parameter in the body of the equation, and there exist no parameters with different names but always identical instantiation. Similar characteristics were formulated by [Le 94]. It is obvious that each RPS which does not fulfill these criteria can be transformed into one which does fulfill them by deleting unused equations and parameters and by unifying parameters with different names but identical instantiations.

To guarantee uniqueness for calculating the substitutions in a recursive equation for a given finite term, we introduce the notion of "substitution uniqueness", which states that it is not possible to replace a substitution term in an RPS $\mathcal{S}$ such that the resulting RPS $\mathcal{S}'$ still explains a given set of initial programs recursively [KSuW02].

Based on these basic considerations, we can state the synthesis problem. Note, that in general, we allow for induction of an RPS for a *set* of initial programs T_{init} rather than for a single initial program t_{init}.

Definition 4 (Synthesis Problem) *Let $T_{init} \subset \mathcal{T}_{\Sigma\cup\{\Omega\}}$ be a set of initial programs. The synthesis problem is to induce*

- *a signature Σ,*
- *a set of function variables $\Phi = \{G_1, \ldots, G_n\}$,*
- *a minimal RPS $\mathcal{S} = (\mathcal{G}, t_0)$ with a main program $t_0 \in \mathcal{T}_{\Sigma\cup\Phi}(X)$ and a set of recursive equations $\mathcal{G} = \langle G_1(x_1, \ldots, x_{m_1}) = t_1, \ldots, G_n(x_1, \ldots, x_{m_n}) = t_n\rangle$*

such that

- *$\mathcal{S}$ recursively explains T_{init}, and*
- *$\mathcal{S}$ is substitution unique.*

3.2 Solving the Synthesis Problem

As mentioned above, the task of a folding algorithm can be separated into three steps: For each given initial program, in a first step, a *segmentation* is constructed, which corresponds to the unfoldings of a hypothetical subprogram. As intermediate steps, first a set of hypothetical recursion points is identified and a skeleton of the body of the subprogram, which is searched, is constructed. If recursion points do not explain the complete initial tree, all unexplained subtrees are considered as a new set of initial trees for which the synthesis algorithm is called recursively and the subtrees are replaced by the name G_i of the to be inferred subprogram. If an initial tree cannot be segmented starting from the root, a constant initial part is identified and included in the body of the calling function – that is, in t_0 for top-level subprograms and in G_i if G_i calls G_j and G_j is the currently considered hypothetical subprogram.

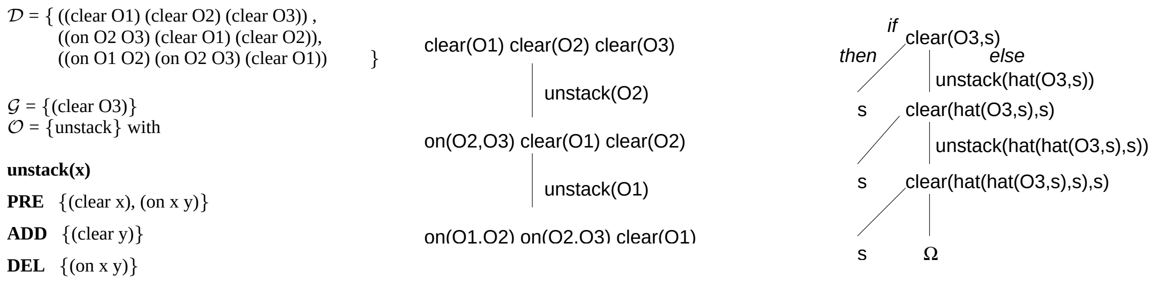
For a given segmentation, the *program body* is constructed by including all parts of initial trees which are constant over all segments into the subprogram body, that is, a maximal pattern is constructed. All remaining parts of the segments are considered as parameters and their substitutions. Accordingly, in a last step, a unique *substitution rule* must be calculated and as a consequence, the parameter instantiations of the function call are identified. The only backtrack-point for folding is calculating a valid segmentation for the initial trees.

For synthesis of an RPS, additionally, a main program must be induced. If a recursive subprogram explains initial trees from the root, the main program consists just of the call of this subprogram. If regularities are found only at lower levels of the initial tree, in a final step the subprogram must be embedded in its calling context.

The theoretical framework for our folding approach is presented in [KSuW02]. Additional details and the algorithmic realization is given in [Sch01]. The folding programm is implemented in Common Lisp, can be obtained from the authors. Obtaining segmentations is basically realized by breadth-first tree search over the initial programs. For maximization of the program body we use first order anti-unification [Plo69].

4 Learning Control-Rules for Planning

Control rule learning currently becomes a major interest in planning research [MG00, HSK00, SW00]: although a variety of efficient domain-independent planners have been developed in the nineties, for demanding real world applications it is necessary to guide


𝒟 = { ((clear O1) (clear O2) (clear O3)) ,
((on O2 O3) (clear O1) (clear O2)),
((on O1 O2) (on O2 O3) (clear O1)) }
𝒢 = {(clear O3)}
𝒪 = {unstack} with
unstack(x)
PRE {(clear x), (on x y)}
ADD {(clear y)}
DEL {(on x y)}
clear(O1) clear(O2) clear(O3)
unstack(O2)
on(O2,O3) clear(O1) clear(O2)
unstack(O1)
on(O1,O2) on(O2,O3) clear(O1)
if
clear(O3,s)
then
else
unstack(hat(O3,s))
s
clear(hat(O3,s),s)
unstack(hat(hat(O3,s),s))
s
clear(hat(hat(O3,s),s),s)
s
Ω

(a) (b) (c)

Figure 2 The **unstack** Problem *(a)*, its Linear Plan *(b)*, and the Resulting Finite Program *(c)*

search for (optimal) plans by exploiting knowledge about the structure of the planning domain. Learning control rules allows to gain the efficiency of domain-dependent planning without the need to hand-code such knowledge which is a time consuming and error-prone knowledge engineering task. Our functional approach to program synthesis is very well suited for control rule learning because in contrast to logic programs the control flow is represented explicitly. Furthermore, a lot of inductive logic programming systems do not provide an ordering for the induced clauses and literals. That is, evaluation of such clauses by a Prolog interpreter is not guaranteed to terminate with the desired result.

Our overall approach to control-rule learning is to first generate a plan for a predefined problem-domain, then rewrite the plan to a term, afterwards using this term as input to the folding algorithm and finally, save the induced RPS as domain specific control rule for the domain. We will illustrate this approach with a very simple blocksworld problem. More complex examples can be found in [SW00, Sch01, TSW01].

Input in the planning system is some planning problem for a domain. In our planner DPlan we use the problem domain description language PDDL for domain and problem specifications [Sch01]. In the example in Figure 2 the domain consists of a single operator *unstack* which is described by an precondition and its ADD-DEL effect, as usual. A concrete problem is specified by a planning *goal* and the number of objects. In contrast, to standard planning, we are not interested in a plan for a single initial state but for the set of *all* admissible states which can hold for the given objects. For illustrative reasons, we listed the set of states for the unstack example with three objects in Figure 2.a. DPlan constructs these states, starting with a complete description of a goal state by calculating the pre-images for each level. This corresponds to backwards planning with breath-first search, which is the typical strategy for universal planning [CRT98].

The resulting plan is represented in Figure 2.b. The topmost node represents a state fulfilling the planning goal. This state can be reached by unstacking object $O2$ in a state where $O2$ is stacked on $O3$, which itself can be reached by unstacking $O1$ in a state where $O1$, $O2$, and $O3$ are stacked.

A plan has a strong correspondence to a program trace: an input (initial state) is *transformed* into a desired output (goal state) by operator application. To construct a finite term, which can be used as input to our folder, the plan is rewritten in the following way: (1) introduce a situation variable s to hold the current state, (2) identify such literals in the state-descriptions which are responsible for the termination of operator-application and introduce them as conditions, and (3) introduce selector functions to refer to objects in a state-independ way.

A similar proposal was made by Manna and Waldinger [MW87]. The selector function for the *unstack* example is defined as

if not clear(x,s) then on(hat(x,s),x,s).

We can infer such functions automatically for a variety of planning domains [SW00, Sch01, WS01]. The finite program term for *unstack* is given in Figure 2.c.

It should be obvious that this term can be folded with the approach described above. The resulting RPS is:

$$\Sigma = \langle \textit{rec-unstack(o,s)} = \textit{if clear(o,s), s, unstack(hat(o,s), rec-unstack(hat(o,s), s))} \rangle$$

with t_0 = *unstack-rec(o,s)* for some constant o and some set of literals s.

This RPS can now included in the representation of the planning domain as additional knowledge. For a new planning problem with a goal corresponding to the pattern *clear(x)*, the sequence of *unstack* transformations can be generated by unfolding the RPS for an arbitrary number of objects.

5 Support for Enduser Programming

Dividing inductive program synthesis into two steps – construction of program traces and folding by recurrence detection – allows us to make program synthesis applicable to a variety of domains. The problem of constructing program traces must be solved for each domain separately, but the same folding-algorithm works domain-independently for all kinds of terms. In addition to control-rule learning, we have started to investigate support for enduser programming for transformation of XML documents.

More and more documents and applications are written in XML (Extensible Markup Language). The growing number of parsers for XML-documents allow to use them as input for many different applications – such as web-browsers, database servers, drawing programs, personal finance or news publishing programs [HM01]. XML is a *meta*-markup language, that is, there is no fixed set of predefined tags (like for the hypertext markup language HTML which is used for displaying documents in the web). Tag based languages describe content ("semantics") rather than form (layout) of data. For example `<h1>Introduction</h1>` in HTML marks "Introduction" as a top-level heading. How a specific content is displayed (e. g., that top-level headings have point-size 14, are centered and numbered consecutively) is described separately, for example by cascading stylesheets (CSS).

XSL (Extensible Stylesheet Language) is an XML application which was originally intended to transform XML-documents into a form which is viewable in web browsers (now XSL-FO). Note that XSL being an XML *application* means that this transformation language is itself realized within XML! We are especially interested is XSLT (XSL Transformations) – a general-purpose language for transforming one XML document into another one. XSLT can be used for transforming an XML document into a displayable document, for example, translating the tags of the source document into XHTML tags which can be interpreted by a web-browser. But in general, it can be used for arbitrary transformations. Because XSL Transformations are proven to be Turing complete, XSLT could be used as a programming language – with a somewhat inconvenient syntax.

Although XSLT is *not* intended as a general purpose programming language, transformations between XML documents can be quite complex. Some examples, based on questions posted to mailing lists, are:

- reversing strings for detecting and eliminating white-space substrings,
- padding space to text nodes to make them have specific length,
- grabbing specific elements occuring in nestings of arbitrary depth,
- case-conversions,
- introducing a new tag giving the total sales of an arbitrary number of items.

For all these examples, the suggested XSL Transformation can involve *recursive template applications*. (Note that some of these problems can also be solved with *for-each* operations.)

It can be assumed that the typical author of XML documents is not a fully educated programmer and that he/she therefore might experience some difficulty when trying to write XSL Transformations involving recursive processing. One possibility to support XML users – suggested by Becker [Bec00] – is, to extend XSLT by some classical loop-constructs and provide a compiler which transforms the loops into recursive template calls. This approach is based on the reasonable assumption that more computer users are familiar with loop-programming than with recursion. An alternative possibility is to let the user provide some examples of the wanted transformation and then generate the recursive transformation rule *automatically*. This second possibility is attractive for the (majority) of XML-users which are unfamiliar with programming!

Since each XML document is a *term* (a tree with a single root), our approach to program synthesis by folding of finite program terms seems well fitted for this domain. Currently, we are working on an XSL-synthesis system based on the following components[2]:

- The user specifies a "small" input XML document, say with three nested text-nodes, together with its desired transformation, that is, a "flattened" XML document with the same text-nodes on the same level.
- With a kind of genetic programming approach, an XSL-term is generated which transforms the input document into the output document in the desired way.
- This XSL-term is transformed into the standard term syntax used by our folder and it is tryed to fold the term, that is, to generalize its applicability to a XML document with an arbitrary number of nested text nodes using recursive template application. Finally, the output RPS is transformed back to XSL syntax.

[2]Diploma thesis by Jens Waltermann, University of Onsabrück

We hope that we can present positive results in the near future.

6 Conclusions

We presented an approach to inductive synthesis of *functional* programs, based on the classical two-step approach of Summers [Sum77]. This approach has several advantages: folding of terms can be performed on a strong theoretical background, relying on the relation between recursive programs and their unfoldings. In contrast to inductive approaches which rely on search in the hypothesis space, construction of an RPS is strongly guided by the structure of a given input term. Separating trace generation and generalization allows for applicability in diverse domains, as we have illustrated with two examples.

In the near future we plan to further investigate application of program synthesis to XSL transformations.

References

[Bec00] Oliver Becker. An XSLT loop compiler. http://www.informatik.hu-berlin.de/~obecker/XSLT/#loop-compiler, 7 2000.

[BGK84] A. W. Biermann, G. Guiho, and Y. Kodratoff, editors. *Automatic Program Construction Techniques*. Macmillan, New York, 1984.

[CRT98] A. Cimatti, M. Roveri, and P. Traverso. Automatic OBDD-based generation of universal plans in non-deterministic domains. In *Proc. 15th National Conference on Artificial Intelligence (AAAI-98)*, pages 875–881, Menlo Park, 1998. AAAI Press.

[FY99] P. Flener and S. Yilmaz. Inductive synthesis of recursive logic programs: Achievements and prospects. *Journal of Logic Programming*, 41(2–3):141–195, 1999.

[HM01] E. R. Harold and W. S. Means. *XML in a Nutshell*. O'Reilly, 2001.

[HSK00] Y. Huang, B. Selman, and H. Kautz. Learning declarative control rules for constraint-based planning. In *Proc. 17th International Conference on Machine Learning (ICML-2000)*, pages 415–422. Morgan Kaufmann, 2000.

[KSuW02] E. Kitzelmann, U. Schmid, M. Mühlpfordt, and F. Wysotzki. Inductive synthesis of functional programs. In J. Calmet, et al., editors, *Artificial Intelligence, Automated Reasoning, and Symbolic Computation (AISC'02)*, pages 26–37. Springer, LNAI 2385, 2002.

[Le 94] 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. Springer, 1994.

[LM91] M. L. Lowry and R. D. McCarthy, editors. *Automatic Software Design*. MIT Press, Cambridge, MA, 1991.

[MG00] M. Martín and H. Geffner. Learning generalized policies in planning using concept languages. In *Proc. 7th International Conference on Knowledge Representation and Reasoning (KR 2000)*, pages 667–677. Morgan Kaufmann, 2000.

[MW87] Z. Manna and R. Waldinger. How to clear a block: a theory of plans. *Journal of Automated Reasoning*, 3(4):343–378, 1987.

[Plo69] G. D. Plotkin. A note on inductive generalization. In *Machine Intelligence*, volume 5, pages 153–163. Edinburgh University Press, 1969.

[Sch01] Ute Schmid. Inductive synthesis of functional programs – Learning domain-specific control rules and abstract schemes. http://www.inf.uos.de/schmid/pub-ps/habil.ps.gz, Mai 2001. Habilitation Thesis, to be published with Springer LNAI.

[SE99] S. Schrödl and S. Edelkamp. Inferring flow of control in program synthesis by example. In *Proc. Annual German Conference on Artificial Intelligence (KI'99), Bonn*, LNAI, pages 171–182. Springer, 1999.

[Smi84] 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.

[Smi90] D. R. Smith. KIDS: A semiautomatic program development system. *IEEE Transactions on Software Engineering*, 16(9):1024–1043, 1990.

[SS71] D. Scott and Ch. Strachey. Towards a mathematical semantics for computer languages. In *Proc. 21st Symposium on Computers and Automata*, pages 19–46. Polytechnic Institute of Brooklyn, 1971.

[Sum77] P. D. Summers. A methodology for LISP program construction from examples. *Journal ACM*, 24(1):162–175, 1977.

[SW00] 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.

[TSW01] J. Toussaint, U. Schmid, and F. Wysotzki. Using recursive control rules in planning. In H. R. Arabnia, editor, *Proc. of ICAI'01*, pages 1012–1015. CSREA Press, 2001.

[WS01] 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.

[Wys83] F. Wysotzki. Representation and induction of infinite concepts and recursive action sequences. In *Proc. 8th International Joint Conference on Artificial Intelligence (IJCAI-83)*, pages 409–414. William Kaufmann, 1983.