

Automatic Synthesis of XSL-Transformations from Example Documents

Ute Schmid

Department of Mathematics/Computer Science
University of Osnabrück
49069 Osnabrück, Germany
schmid@informatik.uni-osnabrueck.de

Jens Waltermann

Department of Mathematics/Computer Science
University of Osnabrück
49069 Osnabrück, Germany
jwalterm@uni-osnabrueck.de

ABSTRACT

We present an application of inductive program synthesis to learning XSL transformations (XSLT) with recursive template application. Since XML and XSLT are term languages, we use an approach to inductive synthesis of functional programs. Synthesis is divided in two steps: First, a straight-forward, non-recursive program is generated which transforms a given set of input examples into the desired outputs; second, the straight-forward program is folded into a generalized recursive program based on recurrence detection. For our application, starting point is a single input/output example, that is, an XML document in its given form as input and in its desired form as output. A non-recursive XSLT is constructed via genetic programming. To apply our folding algorithm, the transformation is rewritten in standard term form. The resulting recursive XSLT is transformed back into XSL syntax.

KEY WORDS

Machine Learning, Inductive Program Synthesis, XML, Recursive Templates

1 Introduction

A growing number of documents and applications are written in XML (Extensible Markup Language). XML-parsers allow to use XML as input for many different applications – such as web-browsers, database servers, drawing programs, personal finance or news publishing programs. XML documents are *trees*. That is, there is a single root element and each element can have an arbitrary number of children.

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 in XSLT (XSL transformations) – a general-purpose language for transforming one XML document into another one. An XSLT tree corresponds to a nested program term, that is, XSLTs are functional programs. Transformations are defined in so-called templates which correspond to function definitions in a functional language. XSLT can be used for transform-

ing 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 can be used as a full functional programming language – with a somewhat inconvenient syntax [6].

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,
- grabbing specific elements in nestings of arbitrary depth,
- case-conversions,
- introducing a new tag giving the total sales of an arbitrary number of (nested) items.

For all these examples, the suggested XSL transformation involves *recursive template applications*.

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, e. g., by [10] – 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 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.

Program synthesis from I/O examples is a challenging problem for machine learning algorithms, addressing discovery of generalized rules from observations. In the seventies and eighties, there were several approaches to the synthesis of functional (Lisp) programs from examples or traces (see [1] for an overview). The functional approaches typically start with a small set of (positive) I/O examples and work in two distinct steps: First, I/O 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 is knowledge dependent: In general, there are infinitely many possibilities to represent I/O examples as terms. One possibility to deal with this problem is to restrict the program domain, e. g., to structural list problems [12]. Alternatively, the finite program can be generated by the user [5], constructed by AI planning [8], or with genetic programming as we will show below. The second step – folding – is a pattern matching task and can be performed for a large range of problems purely syntactically.

Due to only limited progress, interest in inductive program synthesis decreased in the mid-eighties and research focussed on inductive logic programming (ILP) instead [2]. 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. XSL transformations are simple helper programs which typically consist of a small number of templates. That is, complexity of realistic XSLTs is well in the scope of fully automated synthesis.

In the following, we present our approach to inductive synthesis of functional programs. Then we will give an overview of our approach to synthesis of XSL transformations with recursive template applications together with a running example. Afterwards, we will present our approach to generate a non-recursive XSLT for a given pair of I/O XML documents. We will describe how non-recursive XSLTs must be transformed for folding, and finally, we will evaluate the performance of our approach. We conclude with a short discussion and further work to be done.

2 Inductive Program Synthesis

2.1 Summers' Approach

Inductive program synthesis based on recurrence detection in finite terms was made popular by Summers [12] who put inductive synthesis on a firm theoretical foundation. We present an example (see fig. 1) to illustrate the general idea.

Input is a small, positive set of examples for the desired input/output behavior of the Lisp program. Summers approach is restricted to lists as input parameter. He uses the complete partial order over lists as knowledge for trace-generation. The user is expected to present the first elements of the input parameter. 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 (e. g., **unpack**, **reverse**) but without knowledge about the *content* (e. g., **member**, **sort**; [4]). Therefore, for

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")

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

Recursive Generalization:

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

Figure 1. Summers' **unpack** Example

each I/O pair there exists a unique expression relating input and output.

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:

$$\begin{array}{l} f_1(x) = nil, f_2(x) = cons(x, f_1(x)) \\ f_{k+1}(x) = cons(cons(car(x), nil), f_k(cdr(x))) \text{ for } k = 2, 3 \\ p_1(x) = atom(x), p_{k+1}(x) = p_k(cdr(x)) \text{ for } k = 1, 2, 3. \end{array}$$

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

[12] 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. 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.

2.2 Extension

Our own approach [9] 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, which is important if the program terms to be folded are obtained by other sources, such as a user or an AI tool.

The formal background, algorithms, and example applications for the folder are presented in detail in [9]. In

the following, we will present only the central ideas. In general, our folding approach complies to the following recursive program scheme:

Definition 1 (Recursive Program Scheme) Let Σ be a signature and $\Phi = \{G_1, \dots, G_n\}$ 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:

$$\mathcal{G} = \begin{cases} \langle G_1(x_1, \dots, x_{m_1}) & = t_1, \\ \vdots \\ \langle G_n(x_1, \dots, x_{m_n}) & = t_n \end{cases}$$

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

This scheme corresponds to a functional program with the G_i 's as recursive functions and t_0 as initial program call. Functions are constructed over a set of primitive functions Σ and given names from Φ .

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 occurring 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 where all equations are recursive.

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 [4]. 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.

Our approach is based on the notion of an RPS as a term rewriting system where each recursive call is replaced

either by the function body with the necessary parameter substitutions or rewriting is terminated – replacing the recursive call by Ω , a special symbol representing an undefined term. The rewrite system defines the languages of an RPS as the set of all possible unfoldings. We showed [9] 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} .

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.

Step 1 of the algorithm is the only backtrack point. This step is currently implemented such that searching for recurrence is guided by Ω leafs in the paths for efficiency. Consequently, initial programs must provide such Ω information, that is, have exactly the form of an unfolding of a recursive program.

3 Synthesis for XSL

The two-step approach to inductive functional program synthesis described above can be applied to learning XSL transformations with recursive template applications in the following way: In the first step, a non-recursive transformation (initial program) for a pair of XML documents is constructed by genetic programming. In the second step, this initial program is used as input to our folder. Since our folder works for a standard representation of terms and returns a program scheme corresponding to a function program (see def. 1), additionally transformation steps from XSL to terms and back are needed (see fig. 2).

An XSL transformation can be applied to an XML document by means of interpreters such as *Saxon* resulting in an output document. There are two possibilities to realize recursion in XSL: *Parametrized recursion* can work independent of an XML document and corresponds to recursion in functional programming, that is, a recursive template must have a terminating condition and a recursive call where the recursive parameter is changed thus that the termination condition can be reached. *Context-dependent recursion*, on the other hand, is governed by the tag structure of an XML document. There is no explicit manipulation of parameters. Termination occurs if the leaf nodes of the XML document are reached. Since our folding algorithm

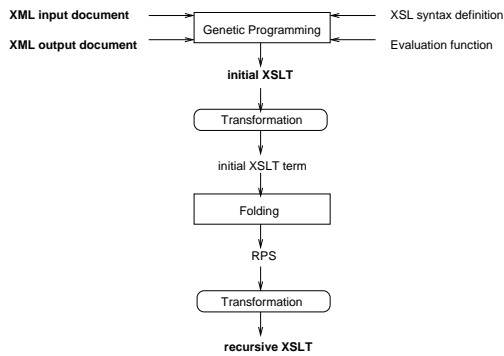


Figure 2. Synthesis of recursive XSLTs

(a) Input XML Document

```

<?xml version="1.0" encoding = "ISO-8859-1" ?>
<Tree><Tree><Tree><cont>(NLL)</cont></Tree>
  <cont>(NL)</cont>
  <Tree><cont>(NLR)</cont></Tree> </Tree>
  <cont>(N)</cont>
  <Tree><Tree><cont>(NRL)</cont></Tree>
  <cont>(NR)</cont>
  <Tree>
    <Tree>
      <cont>(NRRL)</cont></Tree>
      <cont>(NRR)</cont>
      <Tree><cont>(NRRR)</cont></Tree>
    </Tree>
  </Tree> </Tree> </Tree>
  
```

(b) Output XML Document

```

<?xml version="1.0" encoding="utf-8"?>
<order>(N) (NL) (NLL) (NLR) (NR) (NRL) (NRR) (NRRL) (NRRR)</order>
  
```

Figure 3. XML Documents

was designed to infer recursive functional programs, induction of parametrized recursion is straight-forward. In the following, we describe how to infer context-dependent recursions, which is the typical mode of recursion for XSLT.

In figure 3.a an XML document with nested tags is given. Its content may be needed in a flat structure for another application. Thus, a transformation into the XML document given in figure 3.b might be necessary. A non-recursive (initial) XSL transformation which can transform exactly the given input document is given in figure 4.a. To transform arbitrary documents with nested tags, a recursive generalization, such as given in figure 4.b is necessary. Note, that the recursive XSLT realizes a preorder tree traversal which is a generalization of the mailing list problem “grabbing specific elements” mentioned in section 1.

In the following section, we describe how the non-recursive XSL transformation (fig. 4.a) can be generated by a genetic programming approach.

4 Generating Initial Transformations

Genetic programming is typically applied to generate functional program terms from I/O examples [3, 7]. In general, new programs are generated by mutation, crossover, and reproduction and fitness is evaluated by the percentage of examples which are dealt with correctly by the gener-

(a) Non-recursive (initial) XSLT

```

01 <?xml version="1.0" encoding="utf-8" standalone="yes"?>
02 <xsl:stylesheet version="1.0"
03 xmlns:xsl="http://www.w3.org/1999/XSL/Transform">
04 <xsl:template match="/" name="root">
05 <order>
06 <xsl:for-each select="Tree">
07 <xsl:value-of select="cont"/><xsl:for-each select="Tree">
08 <xsl:value-of select="cont"/><xsl:for-each select="Tree">
09 <xsl:value-of select="cont"/><xsl:for-each select="Tree">
10 <xsl:value-of select="cont"/><xsl:for-each>
11 </xsl:for-each> </xsl:for-each> </xsl:for-each>
12 </order>
13 </xsl:template></xsl:stylesheet>
  
```

(b) Recursive XSLT

```

<?xml version="1.0" encoding="utf-8" standalone="yes"?>
<xsl:stylesheet version="1.0"
xmlns:xsl="http://www.w3.org/1999/XSL/Transform">
<xsl:template name="SUB1">
<xsl:for-each select="Tree">
<xsl:value-of select="cont"/>
<xsl:call-template name="SUB1"/>
</xsl:for-each></xsl:template>
<xsl:template name="Root" match="/">
<order>
<xsl:call-template name="SUB1"/>
</order></xsl:template></xsl:stylesheet>
  
```

Figure 4. XSL Transformations

ated program. We use a simplified approach, which can be viewed as a generate-and-test approach where program generation is constrained by certain characteristics: First, new XSLTs are generated by reproduction only, that is, they are extension of current programs; second, since we work on a single I/O pair, fitness is evaluated by characteristics of the generated program itself (see below).

The overall idea for generating initial XSL transformation is the following: We start with a single XSL stylesheet which contains default information in the header and a single template which matches the root of an XML document with an empty body (lines 1–4 and line 13 in fig. 4.a). A new *generation* is generated by inserting a new XSL operation at an arbitrary position within the given XSLT. Each member of a generation which generates a larger part of the desired XML output document than its parent is included in the successor *population*. Within a given generation, XSL transformations which do not generate a larger part of the desired output or which generate output not contained in the example document are extended by inserting further operations. Search for XSLTs within a generation is restricted by a parameter N for the search-depth which limits the number of operations which are inserted in current XSLTs. The algorithm terminates with a final population if no XSLT in this population can be extended further such that a larger part of the output XML can be generated.

For a given pair (I, O) of input/output XML documents, first, a complete set of possible XSL operations is generated in the following way:

Definition 2 (Basic Operations) A set $\theta(I, O)$ of basic XSL operations op can be generated from the input and output documents in the following way:

Table 1. Generation of Maximal XSL Stylesheets

- Generate an initial population P_0 , consisting of a single stylesheet S which generates exactly the root element of output document O .
- Until $P_{i+1} = P_i$:
 1. Initialize the successor population $P_{i+1} = \{\}$.
 2. Initialize the new generation $G_k^i = P_i$ with $k = 0$.
 3. For search depth $k = 1 \dots N$:
Calculate $G_{k+1}^i = \bigcup_{S \in G_k^i} E^k(S, \theta(I, O))$.
 - (a) If $S_c \in G_{k+1}^i$ is interpretable, matching, and padding wrt its parent $S_p \in P_i$ then $P_{i+1} = P_{i+1} \cup \{S_c\}$ and $G_{k+1}^i = G_{k+1}^i \setminus \{S_c\}$.
 - (b) If $S_c \in G_{k+1}^i$ is not interpretable or not matching then $G_{k+1}^i = G_{k+1}^i \setminus \{S_c\}$.

- Create value tags of the form `<xsl:value-of select="i" />`
- Create tree tags of the form `<xsl:for-each select="i" > ... </xsl:for-each>` with i as a tag name in I .
- Create tree tags `Tree(o)` of the form `<o> ... </o>` with o as a tag name in O .

For reproducing stylesheets, new operations from $\theta(I, O)$ are inserted at syntactically correct positions within tree tags.

Definition 3 (Extending Stylesheets) We write $E(S, op)$ to denote the set of stylesheets which results from extending a stylesheet S by inserting a single operation $op \in \theta(I, O)$ at different positions in S , and we write $E(S, \theta(I, O))$ to denote the union of $E(S, op)$ over all $op \in \theta(I, O)$. Finally, we write $E^k(S, \theta(I, O))$, $k \leq N$ to denote the stepwise extension of a stylesheet S by k operations.

If we apply an XSL transformation S to an XML document I we write $I \times S$. If interpretation fails the output is an empty document ϵ . For a stylesheet S and its extension $E(S, op)$ with the resulting documents $I \times S = D$ and $I \times E(S, op) = D'$ holds $D \in D'$ and we say D is contained in D' .

A given XSL stylesheet S can be characterized in the following way:

Definition 4 (Characteristics of S) A stylesheet S is called

interpretable: if $I \times S \neq \epsilon$,

matching: if $I \times S \in O$,

padding: if $I \times S \notin I \times S_p$ where S was generated from S_p ,

maximal: if S_c is padding and $I \times S_c \notin O \forall S_c \in E^k(S, \theta(I, O))$, $k = 1 \dots N$.

These characteristics are used as fitness function. Only interpretable, matching, and padding stylesheets are included in the successor population. The algorithm is given in table 1.

The algorithm is guaranteed to terminate because output XML documents are finite and because search-depth

N is a finite number. Of course, it can happen that the final population does not contain a stylesheet which generates the complete output document desired. Furthermore, different stylesheets might be constructed which generate the same output. Finally, it can happen that, although the output document can be generated, the structure of the stylesheet is not generalizable to recursion.

5 From XSL to RPS and Back

If we obtain a stylesheet such as given in figure 4.a by applying the algorithm given in table 1, in a next step, it must be transformed into standard term syntax as input into the folder.

Most transformation rules from XSL to standard terms and back are straight-forward. Details can be found in [13]. However, the initial XSLT must be extended somewhat due to the requirements of our folding algorithm: First, folding is defined for classical parameteric recursion, second, constructing the recursion hypothesis is guided by Ω leaves in the initial term (see sect. 2).

Since the generated XSLT operates on an XML document, the node which is currently considered during transformation can be viewed as context. We can include this “implicit” parameter of the transformation explicitly into the initial XSLT: For each operation (see def. 2) we include a formal parameter “*context*” and parameter substitution is introduced as a formal incrementation, guided by the tree structure of the XSLT.

Introducing Ω leaves needs a bit more effort: Each path of the XSL tree is checked for regularity, meaning that the path consists of a sequence of sub-paths which are identical with respect to sequence and position of nodes. If such a regularity is found, an Ω is introduced as a leaf node. The algorithm is a simplified variant of the first step of the folding algorithm (identification of recursion points, see sect. 2) and is given in detail in [13].

The transformed initial XSL term can then be presented to the folder. If the given term can be recursively generalized, an RPS is returned which subsequently is transformed back into XSL syntax. For our example, this is the program given in figure 3.b.

6 Performance Evaluation

All the transformation examples mentioned in section 1 rely on a complete traverse of an arbitrarily nested XML-tree. That is, our preorder example is crucial for realizing most kinds of complex XSL transformations. We made detailed performance measures for the preorder example discussed above and additionally for postorder traversal, using the same input XML (see fig. 3.a) but an output document with the content in a different (post-) order.

Transformation and folding can be realized with little effort. The bottleneck is the genetic algorithm. Therefore,

Table 2. Time effort for generating an initial XSLT

(a) Preorder			(b) Postorder			
#P	P	secs	N	#P	P	secs
1	1	2.179	2	2	1	5.458
2	2	3.284	3	5	17	237.855
3	4	9.203	4	5	33	1117.585
4	6	23.087	5	5	43	2674.809
5	6	26.520				
Σ		64.33				

we only report data for this part of synthesis.¹ In table 2.a, we give the population number, the number of stylesheets in a population and the time for generating each population in seconds. For the preorder traversal, a foldable initial XSLT could be found in somewhat over a minute. 851 *Saxon* transformations were calculated, that is, overall 851 stylesheets were applied to the input document and evaluated. From the 6 stylesheets in the final population only one contained regularities, such that the Ω nodes could be inserted and this stylesheet could be folded successfully.

For postorder traversal, the first foldable stylesheet can be generated in search-depth $N = 5$. The total generation time for search depths 2 to 5 are given in table 2.b. For search depth 5, 43783 *Saxon* transformations were calculated, from the 43 stylesheets in the final population, 14 contained regularities and 8 of them could be folded successfully.

Depending on the search-depth parameter N , the number of intermediate stylesheets in a generation grows heavily (step 3. in table 1). For this reason, space efficiency is more problematic than time efficiency. In our current implementation, we only save each population in a temporary file. A necessary extension to be able to deal with more complex problems is additionally to save the generations in files.

7 Conclusion

Inductive program synthesis – the classical functional approach as well as learning recursive Prolog programs with ILP – typically is considered as a challenging problem for basic research with little hope for practical applications. We showed how functional program synthesis can be applied to support end-user programming for XML applications. XSL transformations are small enough programs that they can be handled within the program synthesis framework. For our two step approach, folding of a given non-recursive XSLT into an XSLT with recursive template application presents no problem. The bottleneck is generating the initial XSLT which can only transform a given XML document into a desired output format.

We presented a genetic programming approach for generating such initial programs. Although this approach

¹The folder is implemented in Common Lisp. The genetic algorithm and the transformation are implemented in Java. The program was run on a system with a 2×1 GHz Pentium III SMP, 900 MB RAM under Linux with JDK 1.3.0.

is far from efficient, for a user working with XML but without experience in writing recursive algorithms it might be a useful tool. Performance could be increased by storing intermediate stylesheets in files and distribute reconstruction and evaluation over different computers working in parallel. Alternatively, the user might be asked to present a non-recursive XSLT, he/she might be supported in this process over a GUI with selectable XSLT operations. A similar approach was proposed by [11] for constructing imperative sorting programs.

Acknowledgements. Thanks to Holger Bringmann for encouraging us to apply inductive program synthesis to XSL.

References

- [1] A. W. Biermann, G. Guiho, and Y. Kodratoff, editors. *Automatic Program Construction Techniques*. Macmillan, New York, 1984.
- [2] P. Flener and S. Yilmaz. Inductive synthesis of recursive logic programs: Achievements and prospects. *Journal of Logic Programming*, 41(2–3):141–195, 1999.
- [3] J. Koza. *Genetic programming: On the programming of computers by means of natural selection*. MIT Press, Cambridge, MA, 1992.
- [4] 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, *ECML-94*, pages 183–197. Springer, 1994.
- [5] H. Lieberman. Tinker: A programming by demonstration system for beginning programmers. In A. Cypher, editor, *Watch What I Do: Programming by Demonstration*, chapter 2. MIT Press, Cambridge, MA, 1993.
- [6] D. Novatchev. The functional programming language XSLT – A proof through examples. <http://www.topxml.com/xsl/articles/fp>, November 2001.
- [7] Roland Olsson. Inductive functional programming using incremental program transformation. *Artificial Intelligence*, 74(1):55–83, March 1995.
- [8] U. Schmid and F. Wyszotki. Applying inductive program synthesis to macro learning. In *AIPS 2000*, pages 371–378. AAAI Press, 2000.
- [9] Ute Schmid. *Inductive Synthesis of Functional Programs – Learning Domain-Specific Control Rules and Abstract Schemes*, volume LNAI 2654. Springer, 2003.
- [10] A. T. Schreiner. XML – Architecture, tools, techniques. Scriptum, RIT, Rochester, NY, chap. 8, <http://www.cs.rit.edu/~ats/xml-2001-3/>, 2003.
- [11] 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.
- [12] P. D. Summers. A methodology for LISP program construction from examples. *Journal ACM*, 24(1):162–175, 1977.
- [13] Jens Waltermann. Induktive Synthese von rekursiven XSL Transformationen. Master's thesis, Department of Mathematics and Computer Science, University Osnabrück, 2003.