Induction of Functional Programs Based on Relations Between I/O-examples

Emanuel Kitzelmann and Ute Schmid

Bamberg University, Dept. of Information Systems and Applied Computer Science,
96045 Bamberg, Germany,
{emaunel.kitzelmann,ute.schmid}@wiai.uni-bamberg.de

1 Introduction

We present a technique for inducing functional programs from few, only positive, well chosen input/output-examples (I/O-examples). Inductive program synthesis systems (see [1] for classical systems, [2] for systems in the field of inductive logic programming) can be devided into two general approaches: (i) In the generate-and-test approach (e.g., ADATE [3]), programs of a defined class are enumerated heuristically and then tested against given examples. (ii) In the analytical approach (e.g., [4]), programs of a defined class are derived by analyzing given examples w.r.t. recurrent structures which are then generalized to a recursive function. Analytical approaches cover more restricted program classes but are faster than generate-and-test approaches. These characteristics qualify them for end-user programming or assisting systems whereas generate-and-test methods are more qualified for inventing new and efficient algorithms.

The analytical method presented here is in several aspects more general than other analytical techniques. E.g., (i) there is no restriction to a particular programming language nor to "hard-wired" data-types, (ii) tree-recursion is allowed, (iii) a function can be defined by any number of recursive cases and of base cases.

Programs are represented as (confluent and terminating) constructor term rewriting systems (CSs), i.e., as a set of (recursive) rewrite rules which corresponds to the equations defined within a functional program. A CS of the defined class (see Sec. 2) can be compiled to any functional programming language or to Prolog. For a rule $F(t_1, \ldots, t_n) \rightarrow r$, $F(t_1, \ldots, t_n)$ and r are called *left-hand side (lhs)* and *right-hand side (rhs)* resp., F is called *defined function*, and all other function symbols are called *constructors*. A term t *subsumes* a term s iff the variables in t can be substituted such that the result is s. We say that s is an *instance* of t and that t is a *generalization* of s. A CS computes an output (term) from a given input (term) by successively rewriting instantiations of lhss with instantiations of rhss in a term until there is no more intantiation of a lhs.

2 Inducing CSs from I/O-examples

The class of CSs which can be induced is characterized as follows: (i) A CS contains exactly one defined function F with exactly one parameter, called *pattern*,

(ii) each rhs contains an arbitrary number of occurences of the defined function symbol F (tree-recursion), (iii) recursive calls are not nested. We call the class of CSs matching this schema *flat one-function one-parameter CSs (flat-1-1 CSs)*. Examples and experimental results for inducing them are given in Section 3.

The induction is based on the insight that there are regularities between different I/O-pairs of a recursively defined function. For an example, consider the function *Unpack* which returns a list as output in which each element of the input list is encapsulated in a one element list. *Unpack* is computed by the following CS:

$$\begin{array}{lcl} \mathit{Unpack}([]) & \rightarrow & [] \\ \mathit{Unpack}([q|qs]) & \rightarrow & [[q]|\mathit{Unpack}(qs)] \end{array}$$

Now consider the input $[a, b, c]$ which is computed to the output $[[a], [b], [c]]$. $\mathit{Unpack}([a, b, c])$ leads to the recursive call $\mathit{Unpack}([b, c])$, i.e., $[b, c]$ is again an input and is computed to the output $[[b], [c]]$. We have derived a second I/O-pair and the regularity is that the output $[b, c]$ is the subterm of $[a, b, c]$ at the same position, 2, at which the recursive call in the rhs occured. This regularity—outputs of recursive calls are subterms of the output of the original call at those positions where the recursive calls occured—holds for all flat-1-1 CSs.

Induction of a consistent CS is organized in two levels as follows: At the higher level, patterns of rules are searched for. Each pattern determines the lhs of a rule. At a second level, an rhs is computed for each pattern.

Consider the three I/O-examples $[] \rightarrow [], [x] \rightarrow [[x]], [x, y] \rightarrow [[x], [y]]$ for *Unpack*. The search for patterns starts with one pattern, namely with the least general generalization (lgg) of all example inputs, i.e., with the most specific term subsuming all example inputs. The lgg of the example inputs for *Unpack* is simply a variable q. If a rhs is not found, then the lgg will be replaced by a minimum number of at least two patterns such that the new patterns (i) together subsume all example inputs, (ii) *partition* the inputs (into at least two disjoint subsets), and (iii) are lggs of the subsets of inputs which they respectively subsume. We call the new set of patterns *most generally partitioning lggs (mgpls)*. For the *Unpack*-example, the new patterns are $[]$ and $[q|qs]$. If no rhs will be found for at least one of them, then all unsucessful patterns are again replaced by a set of mgpls and so on. The search is organized such that a consistent CS with a minimal number of rules will be found.

For computing the rhs for a pattern p, the following three cases are checked in the stated order for each position u in order from left to right of all considered outputs in parallel: The subterm of the rhs at position u is

1. a variable contained in p (subterms of outputs are not considered any further),
2. a recursive call (subterms of outputs are not considered any further),
3. a term with a constructor as root.

There exists no other case, thus, if for any position none of the three cases succeed, then no rhs leading to a consistent CS exists and a new set of patterns has to be searched for.

function	#expls	#rules(#rec)	#rec. calls	times in sec
Length	3	2(1)	1	.002
Last	3	2(1)	1	.003
Init	3	2(1)	1	.003
Unpack	3	2(1)	1	.003
IncList	3	2(1)	1	.003
Even	4	3(1)	1	.004
TreeRev	4	2(1)	2	.008
Lasts	5	3(2)	1	.022
DelZeros	6	3(2)	1	.042
PlayTennis	14	5(0)	0	.679

Table 1. Some inferred functions

3 Experimental Results and Further Research

We have implemented the algorithm in the rewriting based language Maude [5]. In Table 1 we have listed experimental results for sample problems. The second column lists the number of given I/O-pairs, the third the total number of induced rules and in parentheses the number of induced *recursive* rules, the fourth the maximal number of recursive calls within one rule, and the fifth the synthesis time. The experiments were performed on a P4 with the Maude 2.2 interpreter.

All induced programs are correct. *Length*, *Last*, and *Init* are standard functions on lists, *Unpack* is described in Sec. 2, *IncList* applies the successor function to each number in a list, *Even* checks whether a natural number is even, *TreeRev* reverses a binary tree. An example for *Lasts* is $Lasts([[a,b],[c,d,e],[f]]) = [b,e,f]$. *DelZeros* deletes all 0s from a list. Finally, *PlayTennis* is an attribute vector concept learning example from Mitchell's machine learning text book [6].

Further research will include the integration of techniques for automatical introduction of subfunctions and additional parameters into induced programs. Such techniques have been invented for other analytical methods.

References

1. Biermann, A.W., Guiho, G., Kodratoff, Y., eds.: Automatic Program Construction Techniques. Collier Macmillan (1984)
2. Flener, P., Yilmaz, S.: Inductive synthesis of recursive logic programs: Achievements and prospects. Journal of Logic Programming **41**(2–3) (1999) 141–195
3. Olsson, R.: Inductive functional programming using incremental program transformation. Artificial Intelligence **74**(1) (1995) 55–83
4. Kitzelmann, E., Schmid, U.: Inductive synthesis of functional programs: An explanation based generalization approach. Journal of Machine Learning Research **7** (2006) 429–454 Special topic on Approaches and Applications of Inductive Programming.
5. Clavel, M., Durán, F., Eker, S., Lincoln, P., Martí-Oliet, N., Meseguer, J., Talcott, C.: The maude 2.0 system. In Nieuwenhuis, R., ed.: Rewriting Techniques and Applications (RTA 2003). Number 2706 in LNCS, Springer (2003) 76–87
6. Mitchell, T.M.: Machine Learning. McGraw-Hill Higher Education (1997)