Data-driven Induction of Functional Programs

Emanuel Kitzelmann[1]

Abstract. We present a new method and system, called IGOR2, for the induction of recursive functional programs from few non-recursive, possibly non-ground example equations describing a subset of the input-output behaviour of a function to be implemented.

1 Introduction

Classical attempts to construct functional LISP-programs from input/output-examples [10, 4] are *analytical*, i.e., a LISP-program belonging to a strongly restricted program class is algorithmically derived from examples. This is done by identifying repetitive syntactical patterns in traces. More recent approaches, e.g. [3, 7], *generate and test* programs until a program consistent with the examples is found. Theoretically, large program classes can be induced generate-and-test based. Yet although these latter systems use type information, some of them higher-order functions, and further techniques for pruning the search space, they strongly suffer from combinatorial explosion.

Also Inductive Logic Programming (ILP) [6] has originated some methods capable of inducing recursive programs on inductive types though ILP in general has a focus on classification. General purpose systems capable of recursive program induction like FOIL [9] are suitable to only a limited extent for program induction since they use greedy search methods with inappropriate heuristics. Special purpose systems [1] have problems similar to those described for functional approaches.

The IGOR2 [5] method described here combines classical analytical methods with an enumerative approach in order to put their relative strengths into effect. Induction is based on search in order to avoid strong a priori restrictions as imposed by purely analytical methods. But in contrast to the generate-and-test approach IGOR2 constructs successor programs during search using analytical methods. IGOR2 represents functional programs as sets of typed recursive first-order equations. The effect of constructing these equations analytically is that only equation sets entailing the example equations are enumerated. In contrast to greedy search methods, the search is complete—only programs known to be inconsistent are ruled out.

Compared to purely analytical systems, IGOR2 is a substantial extension since the class of inducible programs is much larger. E.g., all sample programs from [4, page 448] can be induced by IGOR2 but only a fraction of the sample problems in [5, Sect. 5] can be induced by the system described in in [4]. Compared to ILP systems capable of inducing recursive functions and recent enumerative functional methods like FOIL [9] and MAGICHASKELLER [3] IGOR2 mostly performs better regarding inducability of programs and/or induction times [2].

[1] University of Bamberg, Germany, email: emanuel.kitzelmann@uni-bamberg.de

2 General Method

Given a set E of example equations of the form $F(a) = r$ for any number of target functions F to be implemented as well as for already implemented *background functions* which may be used by the induced program IGOR2 returns a set of recursive equations P constituting a functional program which is *correct* w.r.t. the example equations in that it evaluates the left-hand sides (lhss) of the example equations to their right-hand sides (rhss). Even if example equations may contain variables, we call lhs arguments *example input* and rhss *example output* in the following.

There are infinitely many correct solutions P, one of them E itself. In order to select one or at least a finite subset of the possible solutions at all and a "good" solution in particular, IGOR2—like almost all inductive inference methods—is committed to a *preference bias*. IGOR2 prefers solutions P which partition the examples in fewer subsets, i.e., programs with fewer case distinctions. Case distinctions are realised by disjoint patterns in the equation lhss. This concept is known as *pattern matching* in functional programming. Additionally simple forms of conditions to restrict the applicability of an equation like equality of pattern variables are used but not described in this paper. The search for solutions is complete, i.e., programs with the *least number* of case distinctions are found. This preference bias assures that the recursive structure in the examples as well as the computability by predefined functions is best possible covered.

Example From appropriate type declarations and the examples[2]

$$\begin{aligned} &Rev([\,]) = [\,],\ Rev([X]) = [X],\\ &Rev([X,Y]) = [Y,X],\ Rev([X,Y,Z]) = [Z,Y,X],\\ &Rev([X,Y,Z,V]) = [V,Z,Y,X] \end{aligned} \tag{1}$$

and the background equations

$$\begin{aligned} &Last([X]) = X,\ Last([X,Y]) = Y,\\ &Last([X,Y,Z]) = Z,\ Last([X,Y,Z,V]) = V \end{aligned}$$

IGOR2 induces the following equations for Rev and an auxiliary function $Init$:

$$\begin{aligned} Rev([\,]) &= [\,]\\ Rev([X|Xs]) &= [Last([X|Xs])|Rev(Init([X|Xs]))]\\ Init([X]) &= [\,]\\ Init([X_1,X_2|Xs]) &= [X_1|Init([X_2|Xs])] \end{aligned}$$

The induction of a program is organised as a kind of best first search. During search, a hypothesis is a set of equations entailing the example equations and constituting a terminating program *but potentially with unbound variables in the rhss*, i.e., with variables in the

[2] We use a syntax for lists as known from PROLOG.

rhss not occurring in the lhss. We call such equations and hypotheses containing them *unfinished* equations and hypotheses. A goal state is reached, if at least one of the best—according to the preference bias described above—hypotheses is finished. Such a finished hypothesis is terminating by construction and since its equations entail the example equations, it is also correct.

The initial hypothesis is a program with one equation per target function, namely the *least general generalisation* [8] of the example equations. In most cases (e.g., for all recursive functions) one equation is not enough and the rhss remain unfinished. Then for one unfinished equation successors are computed which leads to new hypotheses. Now repeatedly unfinished equations of currently best hypotheses are replaced until a currently best hypothesis is finished.

3 Computing Successor Sets of Equations

Three operations are applied to compute successor equations: (i) Partitioning of the inputs by replacing the pattern p of the equation by a set of disjoint more specific patterns; (ii) replacing the rhs by a (recursive) call of a defined function; and (iii) replacing the rhs *subterms* in which unbound variables occur by calls to new subprograms.

3.0.1 Refining a Pattern

Computing a set of more specific patterns, case (i), in order to introduce a case distinction, is done as follows: A position in the pattern p with a variable resulting from generalising the corresponding subterms in the subsumed example inputs is identified. The inputs are partitioned such that those with the same symbol at this position belong to the same subset. This yields a partition of the example equations. Now for each subset a new initial hypothesis is computed, leading to a set of successor equations.

E.g., consider the examples (1) for Rev. The pattern of the initial equation is simply a single variable Q, since the example inputs have no common root symbol. The first example input consists of only the constant $[\,]$. All remaining example inputs have the list constructor $cons$ as root. I.e., two subsets are induced, one containing the first example, the other containing the remaining examples. The lggs of the example inputs of these two subsets are $[\,]$ and $[Q|Qs]$ resp. which are the (more specific) patterns of the two successor equations.

3.0.2 Introducing (Recursive) Function Calls and Help Functions

In cases (ii) and (iii) help functions are invented. This includes the generation of examples from which they are induced. For case (ii) this is done as follows: Function calls are introduced by matching the currently considered outputs, i.e., those outputs whose inputs match the pattern of the currently considered equation, with the outputs of any defined function. If all current outputs match, then the rhs of the current unfinished equation can be set to a call of the matched defined function. The argument of the call must map the currently considered inputs to the inputs of the matched defined function. For case (iii), the example inputs of the new defined function also equal the currently considered inputs. The outputs are the corresponding subterms of the currently considered outputs.

For an example of case (iii) consider the Rev examples except the first one as they have been put into one subset in the previous section. The initial equation for these is:

$$Rev([Q|Qs]) = [Q_2|Qs_2] \tag{2}$$

It is unfinished due two the two unbound variables in the rhs. Now the two unfinished subterms (consisting of exactly the two variables) are taken as new subproblems. This leads to two new example sets for two new help functions Sub_1 and Sub_2: $Sub_1([X]) = X,\ Sub_1([X,Y]) = Y,\ \ldots,\ Sub_2([X]) = [\,],\ Sub_2([X,Y]) = [X],\ \ldots$. The successor equation-set for the unfinished equation contains three equations determined as follows: The original unfinished equation (2) is replaced by the finished equation $Rev([Q|Qs]) = [Sub_1([Q|Qs] \mid Sub_2[Q|Qs]]$ and from the new example sets initial equations are derived.

Finally, as an example for case (ii), consider the examples for the help function Sub_2 and the unfinished initial equation:

$$Sub_2([Q|Qs] = Qs_2 \tag{3}$$

The example outputs, $[\,], [X], \ldots$ of Sub_2 match the example outputs for Rev. That is, the unfinished rhs Qs_2 can be replaced by a (recursive) call to the Rev-function. The argument of the call must map the inputs $[X], [X,Y], \ldots$ of Sub_2 to the corresponding inputs $[\,], [X], \ldots$ of Rev, i.e., a new help function, Sub_3 is needed. This leads to the new example set $Sub_3([X]) = [\,],\ Sub_3([X,Y] = [X],\ \ldots$ The successor equation-set for the unfinished equation (3) contains the finished equation $Sub_2([Q|Qs] = Rev(Sub_3([Q|Qs]))$ and the initial equation for Sub_3.

4 Conclusion and Future Research

IGOR2 integrates classical data-driven program induction techniques with search. Comparisons show that this approach is competitive with existing program induction methods regarding solvable problems and mostly solves problems faster [2]. In future work we will extend IGOR2 to higher-order functions such that well known higher-order functions like Map can be used in induced programs.

REFERENCES

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

[2] Martin Hofmann, Emanuel Kitzelmann, and Ute Schmid, 'Analysis and evaluation of inductive programming systems in a higher-order framework'. Submitted to ECML'08, `http://www.cogsys.wiai.uni-bamberg.de/publications/ecml08submission.pdf`, 2008.

[3] Susumu Katayama, 'Systematic search for lambda expressions', in *Revised Selected Papers from the Sixth Symposium on Trends in Functional Programming, TFP 2005*, ed., Marko C. J. D. van Eekelen, volume 6, pp. 111–126. Intellect, (2007).

[4] E. Kitzelmann and U. Schmid, 'Inductive synthesis of functional programs: An explanation based generalization approach', *Journal of Machine Learning Research*, **7**, 429–454, (2006).

[5] Emanuel Kitzelmann, 'Data-driven induction of recursive functions from input/output-examples', in *Proceedings of the ECML/PKDD 2007 Workshop on Approaches and Applications of Inductive Programming (AAIP'07)*, pp. 15–26, (2007).

[6] S. Muggleton and L. De Raedt, 'Inductive logic programming: Theory and methods', *Journal of Logic Programming, Special Issue on 10 Years of Logic Programming*, **19-20**, 629–679, (1994).

[7] Roland Olsson, 'Inductive functional programming using incremental program transformation', *Artificial Intelligence*, **74**(1), 55–83, (1995).

[8] G. D. Plotkin, 'A note on inductive generalization', in *Machine Intelligence*, volume 5, 153–163, Edinburgh University Press, (1969).

[9] J. R. Quinlan and R. M. Cameron-Jones, 'FOIL: A midterm report', in *Proceedings of the 6th European Conference on Machine Learning*, ed., P. Brazdil, LNCS, pp. 3–20, London, UK, (1993). Springer-Verlag.

[10] D.R. Smith, 'The synthesis of LISP programs from examples: A survery', in *Automatic Program Construction Techniques*, eds., A.W. Biermann, G. Guiho, and Y. Kodratoff, 307–324, Macmillan, (1984).