Inductive Programming – Example-driven construction of functional programs

Ute Schmid, Martin Hofmann, Emanuel Kitzelmann

We developed an efficient, analytical approach for learning recursive functional programs from examples. \textit{IGOR2} is a realization of this approach as constructor term rewriting system. We can show that our approach compares very well to other systems of inductive programming and we will discuss applications in the domains of cognitive modelling, end-user programming and automated software engineering.

1 Introduction

Inductive programming research addresses the problem of learning computer programs from incomplete specifications, typically samples of the desired input/output behaviour and possibly additional constraints (Biermann, Guiho, & Kodratoff, 1984; Flener & Schmid, to appear). Induced programs are typically in a declarative – functional or logic – programming language. As a special application of machine learning (Mitchell, 1997), inductive programming creates program hypotheses, that is, generalised, typically recursive, programs. In contrast to classification learning, program hypotheses must cover all given examples correctly since for programs it is expected that a desired input/output relation holds for all possible inputs. In contrast to deductive approaches to automated program construction, there is no guarantee that the induced program meets the intention of the user. Inductive programming approaches can be characterised by learning biases, especially by a language bias which restricts the syntactic form of programs which can be characterised by a program scheme and by a search bias which determines the sequence in which hypotheses are constructed. In contrast to deductive approaches, the barrier for application is lower since the system user is not required to become an expert in the domain of formal specifications. Therefore, inductive programming approaches are good candidates for the development of programming assistants (Flener & Partridge, 2001).

There are two general approaches to inductive programming: analytical and search-based methods. Search-based methods enumerate syntactically correct programs and test these programs against the given examples guided by some search strategy. The inductive logic programming (ILP) system \texttt{FOIL} constructs sets of Horn clauses by sequential covering, that is by a first construct then test approach. This approach was also applied to learning recursive rule sets (Quinlan & Cameron-Jones, 1995). The most successful search-based system is \texttt{ADATE} (Olsson, 1995) which constructs ML programs using evolutionary principles. The system \texttt{MAGICHASKELL} (Katayama, 2005) enumerates \texttt{HASKELL} programs with higher-order functions.

Analytical methods are guided by the structure underlying the given input/output examples. Beginning of research on inductive programming in the nineteen-seventies was concerned with such analytical strategies for learning \texttt{LISP} programs from small sets of positive input/output examples (Biermann et al., 1984). The most influential of the early systems is \texttt{THESYS} (Summers, 1977). It realized a two step approach to synthesise programs: In a first step, input/output examples were rewritten into traces, in a second step recurrent patterns were searched-for in the traces and the found regularities were generalised to a recursive function. \texttt{THESYS} was restricted to structural problems on lists such as unpacking elements or reverting. Due to that restriction rewriting of input/output examples could be realized with a deterministic algorithm where inputs where characterised by the list structure using \texttt{empty} as only predicate. Folding of traces was restricted to linear recursion based on Summers' synthesis theorem which stated that a Kleene sequence of traces can be interpreted as being generated by the \texttt{th} sequence of unfoldings of the searched-for linear recursive program.

For \texttt{THESYS} and all later analytical approaches it is enough to present a small set of only positive input/output examples. These examples must be the first representants of the underlying data-type of the input parameter. In contrast, in search-based approaches, an arbitrary set of positive examples can be presented. Typically, more examples than for analytical approaches are necessary. In addition negative examples can be used to eliminate unsuitable hypotheses.

An extension of \texttt{THESYS} to a larger class of recursive programs is \texttt{IGOR1} (Schmid & Wysotzki, 1998; Schmid, 2003; Kitzelmann & Schmid, 2006). Rewriting of input/output examples allowed nested expressions and folding was guided by a more general program scheme allowing for linear and tree recursions and especially for inducing sets instead of single recursive functions. Another analytical system in the context of ILP is \texttt{DIALOG} (Flener, 1996). Here synthesis relies on interaction with a user who selects a suitable program scheme for the given problem. The most recent analytical approach to inductive programming is \texttt{IGOR2} which will be presented in the following.

2 IGOR2

\texttt{IGOR2} (Kitzelmann & Schmid, 2007; Kitzelmann, 2008; Kitzelmann & Hofmann, 2008; Kitzelmann, 2009) is a new analytical approach to inductive programming which overcomes inherent limitations of systems in the \texttt{THESYS} tradition, includes powerful concepts introduced in inductive logic programming, and relates to the state of art in functional programming languages. The two step approach of \texttt{THESYS} results in an inherent bottleneck to program synthesis due to the first step – rewriting input/output examples to traces. A rewriting algorithm relies on a set of predefined predicates to discern different inputs and on a
set of predefined operators such as list constructors and deconstructors to express the construction of outputs from the inputs. Only if both sets are kept very small and if discerning of inputs is possible without nested conditions an efficient algorithm with unique results can be defined. In consequence, folding of traces into recursive programs is restricted to such simple trace structures also. To allow the generation of more complex traces, the rewriting algorithm needs to employ more complex pattern recognition abilities and partially has to rely on the same inference mechanisms which are used in the folding step of synthesis. Igor2 therefore relies on a single-step algorithm to generalise input/output examples to recursive programs.

The idea of allowing the use of background knowledge in learning as it was introduced in ILP (Muggleton & De Raedt, 1994) was also successfully applied in ILP approaches to inductive programming (Muggleton & Feng, 1990). More complex programs can be learned by providing information of functions which can be used in program construction. For example, quicksort can be learned by providing not only examples for sorting but also examples for partitioning lists and for appending lists. One drawback of search-based approaches is, however, that the background knowledge needs to be restricted to such functions which are necessary to induce the target program because otherwise there is combinatorial explosion. Igor2 allows the use of background knowledge but the introduction of such additional functions is restricted by matching against the given examples due to the analytical strategy used.

Contemporary functional languages such as ML or Haskell allow to define functions using patterns for case distinction and rely on the definition of data types. These characteristics are reflected in constructor term rewriting systems (Baader & Nipkow, 1998). Igor2 is defined in this framework and implemented in Maude (Clavel et al., 2003). Input for Igor2 is

- a set of non-recursive equations specifying the input/output examples of the target function,
- definitions of the data types for the target program (which can be very general, e.g. just sort, or more specific) by means of constructors,
- optional definitions of background knowledge, also in form of non-recursive equations.

The output of Igor2 is a program hypothesis in form of a set of recursive term rewriting rules. The returned hypothesis has the following guaranteed characteristics (Kitzelmann, 2009): All input/output examples are covered. The hypothesis is a recursive generalisation which is minimal with respect to the number of case distinctions, the number of rules and the syntactical complexity of rule bodies. The learned target function is guaranteed to terminate.

As Igor1, Igor2 can induce target functions together with additional functions necessary to solve the problem. For example, to construct a function for reversing lists, Igor2 induces the additional functions last and init without any hint from the system user. Such kind of function invention is the equivalent problem to necessary predicate invention (Flener & Yilmaz, 1999) as discussed in ILP. The specification and the induced solution of reverse is given in Figure 1.

The algorithm of Igor2 can be outlined as follows: First initial rules are constructed as least general generalisation (Plotkin, 1969) of the example equations with resulting patterns as generalisation of the example inputs and rule bodies as generalisation (**data type specification omitted**)

- `eq reverse([]) = []`
- `eq reverse(cons(X, [])) = cons(X, [])`
- `eq reverse(cons(X, cons(Y, []))) = cons(Y, cons(X, []))`
- `eq reverse(cons(X, cons(Y, cons(Z, [])))) = cons(Z, cons(Y, cons(X, [])))`
- `eq reverse(cons(X, cons(Y, cons(Z, cons(V, []))))) = cons(V, cons(Z, cons(Y, cons(X, []))))`

Figure 1: Induction of reverse by Igor2

of the example outputs with according parameter substitutions and empty condition. If the resulting rule bodies contain unbound variables, successor hypotheses are computed applying the following three methods: (1) Partitioning of the inputs by replacing one pattern by a set of disjoint more specific patterns or by adding a predicate to the condition. (2) Replacing the body by a (recursive) call of a defined function, where finding the argument of the function call is treated as a new induction problem. (3) Replacing the sub-terms in which unbound variables occur by a call to new sub-programs. In cases (2) and (3) auxiliary functions are invented, abducing input/output examples for them. The algorithm is described in more detail in Kitzelmann (2009) the system can be downloaded from the project website.

3 Comparison of IP Systems

To get closer insights in the strengths and weaknesses of Igor2 we characterised it in relation to other approaches to inductive programming with respect to: the algorithmic strategy, its scope and its efficiency (Hofmann, Kitzelmann, & Schmid, 2008, 2009). We used conditional higher-order term rewriting as a framework to characterise both logical and functional programs and characterised the currently available systems with respect to their language and search biases, the kind and number of examples necessary for induction, the possibility to make use of background knowledge and whether the systems are able to invent functions or predicates. Furthermore, we performed empirical evaluations of the different systems. A selection of results is given in Table 1.

As problems we have chosen such which are often presented in inductive programming papers and additionally such which highlight specific strengths and weaknesses: mult last replaces all elements with the last and shiftr makes a right-shift of all elements in a list. Therefore it is necessary to access the last element for further computations. Further functions are last which applies last on a list of lists, isort which is insertion-sort, and weave alternates elements from two lists into one. The
functions in odd/even are mutually recursive and need more than two rules; lasts, multlast, isort, and reverse suggest function invention, but only reverse needs necessary function invention to produce the desired program. lasts also splits up in more than two rules if no function invention is applied. To solve member pattern matching is required, because equality is not provided as predicate. The function weave is especially interesting, because it demands either for iterating over more than one argument resulting in more than one base case, or swapping the arguments at each recursive call.

Because FFOIL usually performs better with more examples, whereas MagicHaskeller and Igor II do better with less, each system got as many examples as necessary up to certain complexity, but then exhaustively, so no specific cherry-picking was allowed. For synthesising isort all systems had a function to insert into a sorted list, and the predicate < as background knowledge. For all systems except MagicHaskeller the definition of the background knowledge was extensional. Igor II was allowed to use variables. MagicHaskeller had paramorphic functions to iterate over a data type. Note that we did not test a system with a problem which it per se cannot solve due to its language bias. This is indicated with “—” instead of a runtime. A timeout after ten minutes is indicated with ⊥. Tests have been conducted on a Intel Dual Core 2.33 GHz with 4GB memory. 2.2, ADATE version 0.50 and MagicHaskeller 0.8.3-1. The input files for the presented and further problems can be obtained from the project web-page.

<table>
<thead>
<tr>
<th></th>
<th>ADATE</th>
<th>FFOIL</th>
<th>MagicH</th>
<th>Igor I</th>
<th>Igor II</th>
</tr>
</thead>
<tbody>
<tr>
<td>lasts</td>
<td>365.62</td>
<td>0.71</td>
<td>19.43</td>
<td>0.051</td>
<td>5.695</td>
</tr>
<tr>
<td>last</td>
<td>1.98</td>
<td>0.11</td>
<td>0.01</td>
<td>0.005</td>
<td>0.007</td>
</tr>
<tr>
<td>member</td>
<td>2.11</td>
<td>0.11</td>
<td>1.07</td>
<td>0.152</td>
<td>—</td>
</tr>
<tr>
<td>odd/even</td>
<td>—</td>
<td>&lt; 0.1</td>
<td>—</td>
<td>—</td>
<td>0.019</td>
</tr>
<tr>
<td>multlast</td>
<td>5.69</td>
<td>&lt; 0.1</td>
<td>0.30</td>
<td>0.331</td>
<td>0.023</td>
</tr>
<tr>
<td>isort</td>
<td>83.41</td>
<td>×</td>
<td>0.01</td>
<td>—</td>
<td>0.105</td>
</tr>
<tr>
<td>reverse</td>
<td>30.24</td>
<td>—</td>
<td>0.08</td>
<td>0.324</td>
<td>0.103</td>
</tr>
<tr>
<td>weave</td>
<td>27.11</td>
<td>0.2</td>
<td>⊥</td>
<td>0.001</td>
<td>0.022</td>
</tr>
<tr>
<td>shiftr</td>
<td>20.14</td>
<td>&lt; 0.1</td>
<td>157.32</td>
<td>0.041</td>
<td>0.127</td>
</tr>
</tbody>
</table>

— not tested × time out ⊥ wrong

5 Further Work

The empirical comparison showed that for the tested problems Igor2 performed very well with respect to synthesis time and scope. Since ADATE as evolutionary approach is in principle able to synthesise all possible programs but is comparatively slow to construct even small programs we currently investigate how analytical preprocessing of examples with Igor2 can speed-up synthesis time for ADATE with promising initial results (Crossley, Kitzelmann, Hofmann, & Schmid, 2009).

A current restriction of Igor2 is that it cannot induce rules with a call to another recursive function as outer function. This restriction is inherent to Igor2's strategy of abducting new input/output examples for function invention. For outer functions the input/output examples are the originally given ones. Therefore, we plan to extend Igor2 such that higher-order functions can be introduced in rule bodies. Since Maude does not provide higher-order concepts and for higher visibility of Igor2 in the functional programming community we currently provide a Haskell implementation.

Acknowledgement. The research project “Efficient Algorithms for Inductive Program Synthesis” is supported by DFG grant SCHM 1239/6-1 since October 2007.

References


Emanuel Kitzelmann


Contact

Prof. Dr. Ute Schmid, Dipl.-Inf. Emanuel Kitzelmann, Dipl.-Wirtsch.Inf (E.M.B.Sc.) Martin Hofmann Fakultät Wirtschaftsinformatik und Angewandte Informatik Otto-Friedrich-Universität Bamberg 96045 Bamberg Tel.: +49 (0)951-863-2861 Fax: +49 (0)951-863-2862 EMail: {schmid, kitzelmann, hofmann}@uni-bamberg.de Website: http://www.cogsys.wiai.uni-bamberg.de/effalip/
Martin Hofmann studied business information systems and business management at University of Bamberg and University of Wales, Swansea. He graduated 2007 and currently is research associate in the project “Efficient algorithms of inductive programming”.