*ECAI 2010*
*H. Coelho et al. (Eds.)*
*IOS Press, 2010*
*© 2010 The authors and IOS Press. All rights reserved.*
*doi:10.3233/978-1-60750-606-5-1063*

# Data-Driven Detection of Recursive Program Schemes

**Martin Hofmann and Ute Schmid** [1]

**Abstract.** We present an extension to a current approach to inductive programming (IGOR2), that is, learning (recursive) programs from incomplete specifications such as input/outout examples. IGOR2 uses an analytical, example-driven strategy for generalization. We extend the set of IGOR2's refinement operators by a further operator – identification of higher-order schemes – and can show that this extension does improve speed as well as scope.

## 1 Introduction

The automated synthesis of programs has been a research topic in AI and software engineering since about forty years. From the perspective of AI, this research investigates the abilities and programming and domain knowlesge which enables human problem solvers to construct program code fulfilling some desired input/output behavior. From the perspective of software engineering, this research could lead to the development of automated or semi-automated assistance tools for software developpers.

In principle, there are two major classes of approaches: (1) Deductive, knowledge-based approaches relying on complete specifications in some formal language and (2) inductive approaches relying on incomplete specifications, typically in the form of input/output (I/O) examples which illustrate the desired behavior of a program [8]. Our work is concerned with the latter approach – that is, *inductive programming* (IP). IP is a special branch of machine learning where inductive generalisation must cover *all* given I/O examples correctly.

Historically, IP started with two-step approaches to the synthesis of Lisp programs where a small set of positive examples was first transformed into program traces and these traces where generalized into recursive functions by some method of regularity detection [10]. Afterwards, some approaches of inductive logic programming (ILP) where applied to learning recursive sets of clauses [2]. Simultaneously, evolutionary approaches were applied to solving IP problems [6]. During the last decade, interest in IP decreased in AI and machine learning and at the same time, interest increased in the functional programming community [9].

There are two types of approaches tackling the IP problem, analytical, example-driven approaches and enumerating, generate-and-test approaches. The classical, two-step methods belong to the first type since recursive generalization is based on detecting regularities in the examples. Evolutionary approaches, such as ADATE belong to the second type since hypothetical programs are first generated and afterwards evaluated by testing their performance on the examples. In ILP examples for both types can be found. For example, the sequential covering algorithm of FOIL is a generate-and-test approach, while GOLEM works by generalizing over examples [4].

Our own approach, IGOR2 is an analytical approach to functional IP [5]. IGOR2ś scope of inducible programs and the time efficiency of the induction algorithm compares favorably with ILP and other approaches to inductive programming [4]. In the following, we present a current extension of IGOR2ś generalization strategy which is based on detection of higher-order program schemes. On the one hand, this extension can speed-up the synthesis process, on the other hand, it allows automated induction for a wider class of programs without further background knowledge. Using schemata to constrain the search space is a well established technique in many fields of AI research. However, reduction of search typically comes at the prize of reduced scope, since only such problems can be solved which are covered by the pre-specified schemas. In our approach, instantiation of a generic program schema is included as preferred strategy. If no schema can be matched, induction proceeds using the original, generic strategy for regularity detection.

## 2 The IP System IGOR II

IGOR II is an analytical, functional inductive programming system. Contemporary functional languages such as ML or HASKELL define functions using patterns for case distinction and rely on the definition of data types. These characteristics are reflected in constructor term rewriting systems [11]. IGOR II is defined in this framework and implemented in HASKELL. Its key features are termination by construction of both, its algorithm and the generated programs, handling of arbitrary user-defined data types, utilisation of arbitrary background knowledge, automatic invention of auxiliary functions as subprograms, learning complex call relationships (e.g. tree- and nested recursion), allowing for variables in the example equations, simultaneous induction of mutually recursive target functions, and the adhoc use of program schemes without user interaction.

Input for IGOR II consists of (a) a set of non-recursive equations specifying the I/O examples of the target function, (b) definitions of all used data types by means of constructors, and (c) optional definitions of background knowledge, also as non-recursive equations. The output of IGOR II is a program hypothesis in form of a set of recursive term rewriting rules with the following guaranteed characteristics [5]: All I/O examples are covered. The hypothesis is a recursive generalization which is minimal with respect to the number of case distinctions, the number of rules and the syntactical complexity of rule bodies.

The algorithm of IGOR II can be outlined as follows: First initial rules are constructed as least general generalisation [7] of the example equations with resulting patterns as generalisation of the example inputs and rule bodies as generalisation of the example outputs. If the resulting rule bodies contain unbound variables, successor hypotheses are computed applying all following three methods: (1) Partitioning of the inputs by replacing one pattern by a set of disjoint more

[1] Faculty Information Systems and Applied Computer Science, University of Bamberg, email: {martin.hofmann, ute.schmid}@uni-bamberg.de

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.

Recently,(4) a further method using program schemes was added. Contrary to many other systems such as MAGICHASKELLER, DIALOGS-II, or ADATE, which are also capable of dealing with schemes, no additional user knowledge is required. In *Constructive Algorithmics* [1], a subfield of functional programming, it is well known, that inductive data types have genuine universal properties. There structure induces morphisms, i.e. program schemes, *uniquely* defining mappings to any other type. Catamorphisms, for example, uniquely define a scheme for structural recursion over a given inductive type. We are able [3] to detect those properties in the provided examples and use them for synthesis, if applicable.

## 3 Empirical Analysis

To our knowledge is IGOR II the only system using recursion schemes in an analytical inductive way. Previous tests showed that it is faster or at least as fast as any other comparable analytical IP system . With all generate-and-test approaches it can compete w.r.t to time efficiency and also with most of them w.r.t. expressiveness [4]. Therefore, we tested our system only against its old implementation on a set of various example problems. The old version without schemes always applied all three operators at once. The new prefers hypotheses which use recursion schemes and only applies the other operators when the new one is not applicable.

All tests have been conducted under Ubuntu 7.10 on an Intel Dual Core 2.33 GHz with 4GB memory. For IGOR II the HASKELL-implementation version 0.7.1.2 has been used. The code of the system, as well as the example specifications and a batch file with all tests can be obtained from `http://www.cogsys.wiai.uni-bamberg.de/effalip/download.html`.

Table 1 shows the number of loops taken by the algorithm to find the correct solution for both, with templates and without. The row speedup is rounded to natural numbers. We did not include the runtimes, because the difference between both settings are not significant and a deviation within milliseconds is beyond an acceptable accuracy of measurement. To give a general impression of time efficiency, an overall runtime statistics is included.

**Table 1.** Algorithm loops needed for solution

| | `addN` | `allodd` | `and` | `drop` | `evens` | `fib,add` | `preorder` | `hanoi` | `lasts` | `lengths` | `mirror` | `odd/even` | `powset,++` | `reverse` | `rev,last` | `rev,++` | `sum` | `zeros` |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| without | 13 | ↻ | ↻ | ↻ | 23 | 173 | 5 | 5 | 5 | $1581^{\perp}$ | 4 | 2/2 | $76^{\perp}$ | 10 | 11 | 159 | 6 | 6 |
| with | 2 | 3 | 2 | 2 | 3 | 173 | 3 | 5 | 3 | 2 | 1 | 2/2 | 5 | 2 | 2 | 2 | 2 | 2 |
| speedup | 7 | – | – | – | 8 | 1 | 2 | 1 | 2 | 791 | 4 | 1/1 | 15 | 5 | 6 | 80 | 3 | 3 |

$\perp$ wrong solution, ↻ timeout

Runtime statistics in seconds
0.001 min, 139.761 max, 4.509 avg., 0.008 median, 24.295 std.dev.

We can conclude from the test setting that using catamorphisms as program schemes reduce the complexity of the search. Examples where IGOR II has been lost in search space (`allodd`, `evens`, `lengths`) are now solvables. The performance remains unchanged for problems where a catamorphism is not applicable (`hanoi`) or the background knowledge was suboptimal (`reverse,last`).

## 4 Conclusion

Contrary to previous approaches to incorporate program schemes, where either an (often very well) informed expert user has to provide a template in advance, or templates are used simply on suspicion, regardless whether they are target-aiming or not, we presented an approach to detect the applicability of a program scheme in the provided I/O examples. We utilise the universal property of catamorphisms introduce those schemes where appropriate and underpinned the benefits with an empirical analysis.

Ttheory about functional programming has already formalised many further morphisms with certain properties, which are worth to be looked at and tested for applicability. An idea could be to organise program schemes in a decision tree, where schemes in the same subtree share certain properties. By descending this tree, appropriate schemes can be selected from specific to general.

## REFERENCES

[1] Richard S. Bird and Oege De Moor. *Algebra of Programming*, volume 100 of *International Series in Computing Science*. Prentice Hall, 1997.

[2] P. Flener. *Logic Program Synthesis from Incomplete Information*. Kluwer Academic Press, Boston, 1995.

[3] Martin Hofmann. Data-driven detection of catamorphisms — towards prolem specific use of program schemes for inductive program synthesis. In *Proceedings of the* $11^{th}$ *Symposium on Trends in Functional Programming, Oklahoma City*, 2010.

[4] Martin Hofmann, Emanuel Kitzelmann, and Ute Schmid. Analysis and evaluation of inductive programming systems in a higher-order framework. In A. Dengel, K. Berns, T. M. Breuel, F. Bomarius, and T. R. Roth-Berghofer, editors, *German Conference on Artificial Intelligence (KI'08)*, volume 5243 of *LNAI*, pages 78–86. Springer-Verlag, 2008.

[5] Emanuel Kitzelmann. Analytical inductive functional programming. In M. Hanus, editor, *Proceedings of th 18th International Symposium on Logic-Based Program Synthesis and Transformation (LOPSTR 2008, Valencia, Spain)*, volume 5438 of *LNCS*, pages 87–102. Springer, 2008.

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

[7] G.D. Plotkin. A note on inductive generalisation. In B. Meltzer and D. Michie, editors, *Machine Intelligence 5*, pages 153–163. Edinburgh University Press, Edinburgh, 1969.

[8] Ute Schmid. *Inductive Synthesis of Functional Programs – Learning Domain-Specific Control Rules and Abstract Schemes*. Number 2654 in LNAI. Springer, Heidelberg, 2003.

[9] Ute Schmid, Emanuel Kitzelmann, and Rinus Plameijer, editors. *Approaches and Applications of Inductive Programming Third International Workshop, AAIP 2009, Edinburgh, UK, September 4, 2009, Revised Papers*, volume 5812 of *LNCS*, Heidelberg, 2010. Springer.

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

[11] Terese. *Term Rewriting Systems*, volume 55 of *Cambridge Tracts in Theoretical Computer Science*. Cambridge University Press, 2003.