Noname manuscript No.
(will be inserted by the editor)

A Combined Analytical and Search-Based Approach for the Inductive Synthesis of Functional Programs

Emanuel Kitzelmann

Abstract Inductive program synthesis addresses the problem of automatically generating (declarative) recursive programs from ambiguous specifications such as input/output examples. Potential applications range from software development to intelligent agents that learn in recursive domains. Current systems suffer from either strong restrictions regarding the form of inducible programs or from blind search in vast program spaces. The main contribution of my dissertation [4] is the algorithm IGOR2 for the induction of functional programs. It is based on search in program spaces but derives candidate programs directly from examples, rather than using them as test cases, and thereby prunes many programs. Experiments show promising results.

1 Introduction

Programs writing programs are an old vision of computer science and have important potential applications in software development, end-user programming and autonomous intelligent agents.

One direction of automated programming is *inductive program synthesis* or *inductive programming (IP)* where the problem specification, typically some examples of input/output behavior, is incomplete. IP is well-suited for end-user programming as end users are typically unfamiliar with formal specification. IP could fit smoothly into test-driven development by regarding test cases as incomplete specifications based on which prototypes are generated. In the field of intelligent agents, IP fits into the area of agents that learn from experience to improve on future problems. Unlike standard machine learning, IP focuses on *recursive* "models" learned from few consistent, meaningful, structured data (lists, trees, etc.) where also the function *value* is structured rather than merely a class or a vector of real numbers.

Existing IP systems can be classified according to two general approaches. *Analytical* methods detect recurrent patterns in I/O examples and generalize them to recursive (LISP) functions. This line of research was pursued until mid-1980s [9], experienced a short revival in *inductive logic programming*, and was resumed and combined with universal planning in [8]. Recent work in this tradition is [5]. Most recent systems are *generate-and-test* based [6,3], i.e., generate lots of candidate programs and test them against given examples. Analytical methods are faster since they do not rely on search in program spaces, but are strongly restricted: Programs can, e.g., be linear-recursive and built from constructors and selectors only; I/O examples need to be complete up to some complexity. On the other hand, searching in unrestricted program spaces is computationally demanding and heuristics are difficult to define since effects of program manipulations can hardly be predicted. IP is thus not yet applicable to real problems.

The main contribution of my dissertation is the algorithm IGOR2 to induce functional programs. It leverages complementary strengths of both approaches by combining search in fairly unrestricted program spaces with analytical techniques to generate candidates. Using search allows for relaxing the restrictions of analytical methods while data-driven program construction guides the search. IGOR2 can use background knowledge (BK) and automatically invents subfunctions. It finds complex recursion schemes like that of *Ackermann* or the mutual-recursive definition of odd/even. Found solutions are assured to correctly compute the examples.

International Computer Science Institute
1947 Center Street, Suite 600
Berkeley, CA 94704, USA
E-mail: emanuel@icsi.berkeley.edu

Listing 1 Examples for the Rocket planning problem

```
1 rocket( nil , s)              → move(s)
2 rocket((o1 nil ), s)          → unload(o1, move(load(o1, s)))
3 rocket((o1 o2 nil ), s) →
      unload(o1, unload(o2, move(load(o2, load(o1, s)))))
```

Listing 2 Strategy for rocket induced by IGOR2 in 0.012 sec

```
rocket( nil , s)      → move(s)
rocket((o os), s)     → unload(o, rocket(os, load(o, s)))
```

2 IGOR 2

IGOR2 induces recursive functional programs from sets of I/O examples for target functions and BK. Target functions are to be induced whereas background functions are assumed to be implemented already and may be used. I/O examples may contain (∀-quantified) variables such that one "I/O pair" represents all its instances. Induced programs as well as I/O pairs are represented as *constructor (term rewriting) systems (CSs)*. Informally, a CS is a set of equations, read from left to right as (rewrite) rules, over a first-order algebraic signature that is partitioned into *defined functions* (roots of the left-hand sides) and *constructors*. The argument-terms, built from constructors and variables, of the defined functions at the heads of the left-hand sides (LHSs) are called *pattern*. CSs are a basic form of functional programs, excluding higher-order functions.

Let's see how IGOR2 works by sketching the induction of a strategy for *rocket* [10], a simple benchmark problem in automated planning. The problem is to transport a number of objects from earth to moon where the rocket can only move in one direction. The solution is to load all objects, fly to the moon and unload the objects. The assumption now is that a planner already solved the problem for zero to two objects.[1] The problem instances and plans are then translated to example inputs and outputs for IGOR2 (Listing 1).[2] The objects are provided as a list (constructors nil, empty list, and an "empty syntax" constructor _ _ to "cons" an object to a list). The variable s denotes a *state*, similar to situation calculus. From the three examples, IGOR2 induces the recursive strategy as shown in Listing 2.

Since target concepts are specified ambiguously in inductive inference, the best an inductive reasoning system can do is to produce a (largely) consistent *hypothesis* and hence needs a criterion, an *inductive bias*, to chose one of the possible hypotheses. IGOR2's bias is to prefer fewer conditions, i.e., fewer patterns in the LHSs of a CS. Thus, IGOR2 starts with a hypothesis consisting of only one single rule (no conditions at all) per target function—the *least general generalization (LGG)* [7] of its examples. The LGG for the rocket examples is

```
rocket(os, s) → s'.
```

The variable os results from the different constructors nil and _ _ at the same position in the example rules. The variable s' results from the different symbols move and unload. This initial hypothesis is *unfinished* due to the variable s' that does not occur in the LHS such that the rule does not represent a function. Now IGOR2, in principle, applies three refinement operators in parallel to that initial rule: (i) It refines the pattern (LHS) by a *set of disjoint more specific patterns*; (ii) it considers unfinished *sub*terms of the RHS as new subproblems; (iii) it replaces the RHS by a *(recursive) function call*. These operations are data-driven so that many refinements can be ruled out a-priori. In particular, for our initial rocket rule, operators (ii) and (iii) do not apply: Since the RHS consists of a single variable, there are no proper subproblems to deal with. Further, since no example RHS subsumes another one, a recursive call of rocket as replacement for s' can be ruled out. It remains the first operator. IGOR2 detects that the pattern variable os results from the different constructors nil (1st example) and _ _ (2nd, 3rd example) and partitions the examples accordingly such that the first goes into one subset and the remaining two into a second one. The single initial rule is then replaced by initial rules of the two subsets, leading to the refined hypothesis:

```
rocket ( nil , s)     → move(s)
rocket ((o os), s)    → unload(o, s')
```

The second rule is unfinished again. Since the variable s' now occurs as a proper subterm in the RHS, it can be dealt with as a subproblem. Therefore, IGOR2 replaces s' by a call to a new subfunction sub,

```
rocket((o os), s) → unload(o, sub((o os), s)),
```

and takes as examples for it the appropriate subterms of the RHSs of the corresponding rocket examples:

```
sub((o1 nil ),s)    → move(load(o1,s))
sub((o1 o2 nil ),s) → unload(o2,move(load(o2,load(o1,s))))
```

The refinement step is finished by computing an initial rule for sub and adding it to the rocket hypothesis:

```
rocket( nil , s)      → move(s)
rocket((o os), s)     → unload(o, Sub((o os), s))
sub((o os), s)        → s'
```

The rule for sub remains unfinished. At this step, the RHSs of both sub examples are subsumed by an RHS

[1] The instance for zero objects is a bit artificial. We chose it to keep the example as simple as possible, but it may be skipped and replaced by the three objects instance.

[2] This is currently done by hand.

Listing 3 swap, induced from 6 examples in 6.82 sec

```
swap (x0 : x1 : xs, 0, 1)              → x1 : x0 : xs
swap (x0 : x1 : x2 : xs, 0, n+2)       →
     swap (x1 : swap (x0 : x2 : xs, 0, n+1), 0, 1)
swap (x0 : x1 : x2 : xs, n+1, m+2) →
     x0 : swap (x1 : x2 : xs, n, m+1)
```

of the rocket examples. Particularly, move(s) (RHS of 1st rocket example) subsumes move(load(o1, s)) (RHS of 1st sub example) by substitution {s ↦ load(o1, s)}; and unload(o1, move(load(o1, s))) (2nd rocket example) subsumes unload(o2, move(load(o2, load(o1, s)))) (2nd sub example) by {o1 ↦ o2, s ↦ load(o1, s)}. This indicates that the sub examples might be computed by calling rocket. IGOR2 now applies the found substitutions to the respective *LHSs* of the rocket examples and then tries to generate a term over the pattern variables that maps the LHSs of the sub examples to the substituted LHSs of the rocket examples. The found term,

```
(os, load(o, s)),
```

is taken as the argument of the rocket call:

```
rocket( nil , s)  → move(s)
rocket((o os), s) → unload(o, sub((o os), s))
sub((o os), s)    → rocket(os, load(o, s))
```

Since sub is not recursive, it can be eliminated by unfolding, leading to the solution stated in Listing 2.

3 Experiments

I implemented IGOR2 in the interpreted, rewriting based language MAUDE [1] and ran several experiments on an Intel Dual Core 2.33 GHz, 4GB RAM, Ubuntu machine.

IGOR2 induced correct definitions, partly by inventing recursive subfunctions, of several standard functions for natural numbers and lists, among them = and ≤, add and factorial (with mult as BK), reverse, take, drop, zip, and quicksort (with append and partition as BK). Also the *Ackermann* function was correctly induced (in 3 sec from 17 examples). Another successfully induced problem was swapping two elements in a list, indicated by their indices, e.g., swap([a,b,c,d], 2, 4) → [a,d,c,b]. It was specified by 6 examples (no BK), restricted to cases where the given indices occurred in the list, were different, and the first index was the smaller one. Listing 3 shows the induced solution.

IGOR2 also correctly induced functions on lists of lists, among them weave (Listing 4), which takes a list of lists and produces a (flat) list by taking, in rotation over the inner lists, one element after the other from the inner lists—the first element of the first list, then the first element of the second list etc. up to the first element of the last list, then the second element of the first list, the second element of the second list etc.

Listing 4 weave, induced from 11 examples in 33.35 sec; note the automatically invented recursive subfunction sub36 that drops the first element of the first list and rotates the lists

```
weave ( nil )              → []
weave ((x:xs) :: xss) → x : weave (sub36((x:xs) :: xss))
sub36 ((x:[]) :: nil )             → nil
sub36 ((x:[]) :: (y:ys) :: xss)    → (y:ys) :: xss
sub36 ((x:y:xs) :: nil )           → (y:xs) :: nil
sub36 ((x:y:xs) :: (y:ys) :: xss) →
        (y:ys) :: sub36 ((x:y:xs) :: xss)
```

Listing 5 Example of *Towers of Hanoi* for three discs

```
Hanoi(3, a, b, c, s) →
    move(1, a, c, move(2, b, c, move(1, b, a,
      move(3, a, c,
        move(1, c, b, move(2, a, b, move(1, a, c, s)))))))
```

Listing 6 Recursive solution of the *Towers of Hanoi*, induced from 3 examples in 0.08 sec

```
Hanoi (1, a, b, c, s)   → move (1, a, c, s)
Hanoi (n+1, a, b, c, s) → Hanoi (n, b, a, c,
                            move (n+1, a, c,
                              Hanoi (n, a, c, b, s)))
```

As a last example consider the famous *Towers of Hanoi*. The solution for three discs and the general recursive solution are illustrated in Figure 1. The recursive solution (Listing 6) was induced by IGOR2 from the solutions for one to three (Listing 5) discs.

Despite the remarkable positive results there are rather simple functions that could not be induced. Consider, e.g., multiplication of two natural numbers, represented by 0 and succ. The recursive case uses add:

```
mult (n+1, m) → add (mult (n, m), m)
```

If add is given as BK, IGOR2 could principally find the correct definition for mult yet failed to find it within three minutes since there is too less structure in the I/O examples to effectively prune the program space. If add is *not* given, then mult is not in the considered program space at all because the particular form of the rule does not allow for an automatic invention of add. This is because IGOR2 can currently only invent subfunctions that either compute *proper subterms* of an output or *arguments* of a (recursive) call of another function. Since add is called at the root of the RHS of mult, neither case applies. This restriction is due to the analytical construction of rules.

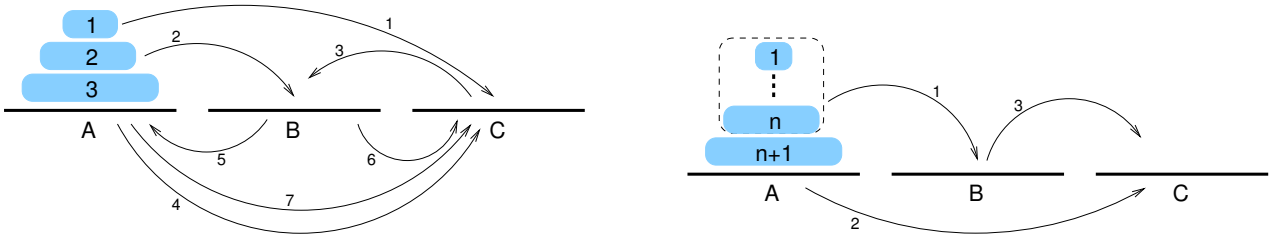

Fig. 1 The *Towers of Hanoi*: The solution for three discs, corresponding to Listing 5, at the left-hand side (the arrows represent moves of single discs) and the general (recursive) solution, corresponding to Listing 6, at the right-hand side (arrows 1 and 3 represent recursive calls)

4 Conclusions and Future Work

IP is a challenging research field with various important potential applications. IGOR2 is an attempt to relax the overly strong restrictions of analytical methods to a large extent without reverting to generate-and-test. Experiments show positive results. The reviewed, successfully induced problems are out of scope of former analytical systems and to the best of my knowledge, no other work reports on the successful induction of similar complex recursive functions in similar reasonable time. However, the synthesis operators still rule out certain program forms and if the I/O examples do not contain sufficient structure then the breadth-first search of IGOR2 becomes intractable. Future work must improve the search operators and the general search strategy.

One serious disadvantage of analytical techniques including IGOR2 is the requirement for sets of I/O examples that are complete up to some complexity. This often compels the programmer/specifier to think about missing I/O pairs even if a meaningful I/O pair, that would suffice for a generate-and-test method, is already provided. On the logical side, this problem could be tackled by introducing and reasoning with $\exists$-quantified variables in example outputs. To generally become more robust with regard to missing or erroneous information, as generally present in real-world AI problems, probabilistic reasoning need to be integrated into analytic IP.

An extension of IGOR2 that has already been worked on by Martin Hofmann (e.g. [2]) is the integration of higher-order functions like map/filter/reduce. Those functions encapsulate useful, terminating recursion patterns such that function definitions become more compact and hence are easier to find. Moreover, they allow for better analysis of the class of inducible functions.

References

1. Clavel, M., Durán, F., Eker, S., Lincoln, P., Martí-Oliet, N., Meseguer, J., Talcott, C.: The maude 2.0 system. In: Rewriting Techniques and Applications (RTA'03), *LNCS*, vol. 2706, pp. 76–87. Springer (2003)
2. Hofmann, M.: Data-driven detection of catamorphisms: Towards problem specific use of program schemes for inductive program synthesis. In: 22nd Symposium on Implementation and Application of Functional Languages (IFL) (2010)
3. Katayama, S.: Systematic search for lambda expressions. In: Selected Papers from the 6th Symposium on Trends in Functional Programming, pp. 111–126. Intellect (2007)
4. Kitzelmann, E.: A combined analytical and search-based approach to the inductive synthesis of functional programs. Ph.D. thesis, Fakultät Wirtschaftsinformatik und Angewandte Informatik, Otto-Friedrich Universität Bamberg (2010). URL `http://www.opus-bayern.de/uni-bamberg/volltexte/2010/280/`
5. Kitzelmann, E., Schmid, U.: Inductive synthesis of functional programs: An explanation based generalization approach. Journal of Machine Learning Research **7**, 429–454 (2006)
6. Olsson, J.R.: Inductive functional programming using incremental program transformation. Artificial Intelligence **74**(1), 55–83 (1995)
7. Plotkin, G.D.: A note on inductive generalization. Machine Intelligence **5**, 153–163 (1970)
8. Schmid, U.: Inductive Synthesis of Functional Programs: Universal Planning, Folding of Finite Programs, and Schema Abstraction by Analogical Reasoning. LNAI. Springer, Berlin; New York (2003)
9. Smith, D.R.: The synthesis of LISP programs from examples: A survey. In: Automatic Program Construction Techniques, pp. 307–324. Macmillan (1984)
10. Veloso, M.M., Carbonell, J.G.: Derivational analogy in prodigy: Automating case acquisition, storage, and utilization. Machine Learning **10**, 249–278 (1993)

Emanuel Kitzelmann studied computer science in Passau and at TU Berlin. He worked as "wissenschaftlicher Mitarbeiter" at the University of Bamberg from where he received his doctoral degree. His dissertation was supervised by Prof. Dr. Ute Schmid. Currently he is a DAAD sponsored research fellow at the International Computer Science Institute (ICSI) in Berkeley, USA. He now works on learning structured and concurrent policies in a decision theoretic setting and on applying inductive programming to learning hierarchical task networks.