Two New Operators for IGOR2 to Increase Synthesis Efficieny

Emanuel Kitzelmann

International Computer Science Institute
Berkeley, USA
emanuel@icsi.berkeley.edu

Abstract. Inductive program synthesis addresses the problem of automatically generating computer programs from incomplete specifications such as input/output examples. Potential applications range from automated software development to end-user programming to autonomous intelligent agents that learn from experience or observation. We present a recent version of the domain-independent algorithm IGOR2 for the inductive synthesis of recursive functional programs, represented as rewriting rules. IGOR2 combines classical *analytical* methods, that detect recursion by matching I/O examples, with *search* in program spaces as applied by recent generate-and-test methods; thereby widening the class of programs that are synthesizable in reasonable time. In particular, we present two recent improvements over an earlier IGOR2 version which significantly increase the efficiency of the synthesis. Functions that were not inducible in several minutes are now induced in several seconds. It has already been shown that an earlier version of IGOR2 outperforms other recent systems on several problems. In the empirical evaluation here, we show the significance of the improved synthesis operators by means of more complex problems, most of which were not tractable for IGOR2 until now.

1 Introduction

Inductive program synthesis or *inductive programming (IP)* means the automated synthesis of programs where the problem specification, typically some examples of input/output behavior, is *incomplete*. IP has important application fields. For a recent example in end-user programming, see [4]. In [15] it is shown how IP can be used to model the cognitive capability of learning productive problem-solving knowledge in recursive domains. E.g., the general recursive strategy to solve *Towers of Hanoi* for arbitrary numbers of discs could be learned from solution traces for 1–3 discs.

IP and *supervised machine learning* have in common that a general concept or model is learned from I/O examples. However, unlike standard supervised learning [5], where a learned model maps objects to qualitative or quantitative values (*classification* and *regression*, resp.) and the models are non-recursive, in the case of IP, induced programs are typically recursive and not only is the

(input) data structured (lists, trees etc.) but in general also the output. Further, we assume that examples are noise-free and require hypotheses to be consistent. Since recursion is a strong pattern, few data often suffices to learn the correct recursive function.

In this paper we are especially concerned with the synthesis of *recursive, functional* programs, represented as a special kind of term rewriting systems over first-order algebraic signatures. We describe a recent version of the IGOR2 algorithm that leverages complementary strengths of two approaches to IP: *Analytical* methods detect recurrent patterns in I/O examples and generalize them to recursive functions [16, 9]. This is efficient but suffers from strong restrictions regarding the form of inducible programs and requires I/O examples that are complete up to some complexity (e.g., an input-list for each number of elements up to some maximum must be specified). *Generate-and-test* based systems [12, 7] generate lots of candidate programs and test them against given examples or evaluation functions. They overcome the strong restrictions of the analytical approach but suffer from unconstrained search in vast program spaces. IGOR2 combines search in fairly unrestricted program spaces with analytical techniques to generate candidates and thereby widens the class of programs that are synthesizable in reasonable time. IGOR2 is able to use background knowledge (BK) and automatically invents recursive subfunctions. It finds complex recursion schemes like that of *Ackermann* or the mutual-recursive definition of `odd/even` and is applicable in different domains.

In [8], a preliminary IGOR2 version is described. We here review the general algorithm and then focus on the synthesis operators. In particular, we discuss some shortcomings and present extended versions that lead to a much more efficient synthesis and a wider class of tractable problems. The remainder of the paper is organized as follows: Section 2 introduces the IGOR2 algorithm including its original synthesis operators. In Section 3 we discuss two synthesis operators more detailed and describe improved versions of them. In Section 4 we empirically evaluate the proposed new operators. Section 5 discusses some related work and in Section 6 we conclude.

2 The Igor2 Algorithm

We call functions that are to be synthesized *target functions*. Functions that are assumed to be implemented already and can be used are called *background functions*.

2.1 Representation Language

We briefly review basic term rewriting concepts as described, e.g., in [1].

IGOR2 specifications of target and background functions as well as induced definitions of target functions are represented as *orthogonal* (see below) *constructor (term rewriting) systems (CSs)*. A CS is a set of (term rewrite) rules over a first-order algebraic signature (function symbols) and a set of variables,

where the signature is partitioned into *defined functions* and *constructors* and where each rule has the form

$$f(p_1, \ldots, p_n) \to t\,.$$

The symbol f is a defined function, the p_i are built from constructors and variables, and all variables in the right-hand side (RHS) t must also occur in the left-hand side (LHS) $f(p_1, \ldots, p_n)$ ($Var(t) \subseteq Var(f(p_1, \ldots, p_n))$). The argument constructor terms p_i are called *pattern*. We denote sequences of terms like $p_1, \ldots, p_n$ by $\boldsymbol{p}$.

A CS is called *orthogonal* if its LHSs are linear, i.e., each variable occurs at most once in one and the same LHS, and pairwise non-unifying. Two terms are non-unifying if there is no substitution σ of variables by terms such that the terms become equal if σ is applied to both of them. Orthogonal CSs are a basic form of functional programs, excluding higher-order functions. Evaluation of an (input) term s is done by repeatedly matching subterms of it with LHSs of the CS—leading to substitutions σ of the pattern variables—and replacing the subterms by the respective RHSs with variables substituted according to σ. Orthogonality assures that if an evaluation terminates, i.e., reaches a *normal form*, then this normal form is unique. Hence orthogonal CSs denote (deterministic) functions. The (non-unifying) patterns of different LHSs for the same defined function act (i) as *conditions* to evaluate inputs of particular different forms differently and (ii) *decompose* a matching term into subterms. This concept is called *pattern matching* in declarative programming.

2.2 The Inductive Synthesis Problem

Specifications of target and background functions are orthogonal CSs with the restriction that the RHSs are built from constructors (and variables) only and hence are in normal form. *Ground* (no variables) specification rules denote I/O examples whereas specification rules containing variables represent sets of I/O examples given by all their ground instances. The inductive synthesis problem is defined as follows:

Definition 1 (Induction problem). *Let Φ and B be two specifications with disjoint sets of defined functions, $\mathcal{D}_\Phi \cap \mathcal{D}_B = \emptyset$, called* target functions *and* background functions*, respectively. Find a CS P with defined functions $\mathcal{D}_P$, such that*

1. *P is orthogonal,*
2. *P does not (re)define background functions,*
3. *for each $f(\boldsymbol{i}) \to o \in \Phi$, $P \cup B$ evaluates $f(\boldsymbol{i})$ to o.*

We use the *Rocket* problem [17], a simple benchmark problem in automated planning, as a running example. 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

Listing 1.1. Examples for the Rocket 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 1.2. Strategy for rocket induced by IGOR2

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

now is that a planner (or an expert) 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 1.1). The objects are provided as a list (constructors nil , empty list, and an infix constructor _:_ to "cons" an object, 1st argument, to a list, 2nd argument). The variable s denotes a *state*, similar to situation calculus [10]. From the three examples, IGOR2 induces the recursive strategy as shown in Listing 1.2.

2.3 General Search Strategy and Preference Bias

The induction of a solution CS is organized as a uniform cost search in spaces of orthogonal CSs, where the definition of CSs is relaxed in that $Var(r) \subseteq Var(l)$ need *not* be satisfied for all rules $l \to r$. We refer to CSs not satisfying this property, the respective rules, and their RHSs as *unfinished* and to CSs, rules, RHSs that satisfy it as *finished*. Either case might be meant in the following if we just say CS, rule etc. Unfinished CSs lead to non-unique normal forms and hence do not encode (deterministic) functions. Purpose of the synthesis operators is to transform an unfinished CS into a finished one. IGOR2's refinement operators, described in the next section, assure that *all* constructed candidate CSs P satisfy Def. 1. Hence each *finished* CS is a solution.

The cost of a candidate CS is defined as the number of disjoint patterns in it, hence CSs that correctly compute the examples based on fewer case distinctions have lower cost and are preferred. The initial candidate CS consists of one single initial rule (see below) for each target function.

2.4 Initial Candidate Rules and CSs

As initial hypothesis for a set of specification rules, IGOR2 takes their *least general generalization (LGG)* [13]. That basically means that if all rules have the same symbol at a particular position, it is kept for that position in the LGG,

[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.

and if the symbols differ, a variable is introduced, where it is assured that the same variable is introduced at different positions, if the corresponding subterms in the example rules are the same at both positions. The LGG for the rocket examples (Listing 1.1) 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 rule is *unfinished* due to the variable s', which does not occur in the LHS. Hence it will be refined.

2.5 Synthesis Operators

In general, if a candidate CS P is chosen during the search, one of its unfinished rules r is selected to be refined. One refinement consists of a set s of successor rules. IGOR2 applies three operators independently to an unfinished rule r to compute refinements: (i) It splits r into sets of at least two new initial rules with disjoint patterns that are more specific than the pattern of r; (ii) it considers unfinished *sub*terms of the RHS of r as new subproblems; (iii) it replaces the RHS of r by (recursive) function calls. Assume a CS P is chosen and let $r \in P$ be the selected unfinished rule. Applying the refinement operators results in a finite (possibly empty) set $\{s_1, \ldots, s_n\}$ of successor-rule sets s_i. For each s_i a successor candidate CS P_i is generated by $P_i = (P \setminus \{r\}) \cup s_i$.

Rule Splitting. Consider the example rules in Listing 1.1 and the corresponding initial rule, rocket(os, s) → s'. The pattern variable os results from the different constructors nil (1st example) and _:_ (2nd, 3rd example). We call a position that denotes a *variable* in the LHS of an initial rule and *constructors* in the LHSs of the corresponding example rules a *pivot position*. Now the splitting operator χ_{split} partitions the examples according to the different constructors at the pivot position. In our example, the first example rule goes into one subset and the remaining two into a second one. The refinement of the unfinished initial rule then consists of a set of new initial rules, one for each subset of the generated partition. In our example:

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

Since the new initial rules always contain the different constructors at the pivot position in their LHSs, they are non-unifying. Since χ_{split} increases the number of disjoint patterns in a CS, it increases the cost of a candidate.

If more than one pivot position exists, this probably leads to different partitions and different refinements, all of which are returned by χ_{split}. In Section 3.1 we discuss this operator more detailed and propose a variant that (i) makes larger refinement steps and (ii) is deterministic. This leads to a more efficient synthesis process as empirically shown in Section 4.

Dealing with Unfinished Subterms Separately. Consider the initial rule $\mathsf{rocket}((\mathsf{o}{:}\mathsf{os}),\ \mathsf{s}) \rightarrow \mathsf{unload}(\mathsf{o},\ \mathsf{s}')$ for example rules 2, 3 that resulted from splitting the original initial rule and which is unfinished due to variable s'. Since s' occurs as a proper subterm in the RHS, it can be dealt with as a subproblem. Therefore, the subproblem operator χ_{sub} replaces s' by a call to a new subfunction sub,

$$\mathsf{rocket}((\mathsf{o} : \mathsf{os}),\ \mathsf{s}) \rightarrow \mathsf{unload}(\mathsf{o},\ \mathsf{sub}((\mathsf{o} : \mathsf{os}),\ \mathsf{s})),$$

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

$$\begin{array}{ll} \mathsf{sub}((\mathsf{o1} :\ \mathsf{nil}),\ \mathsf{s}) & \rightarrow\ \mathsf{move}(\mathsf{load}(\mathsf{o1},\ \mathsf{s})) \\ \mathsf{sub}((\mathsf{o1} :\ \mathsf{o2} :\ \mathsf{nil}),\ \mathsf{s}) & \rightarrow\ \mathsf{unload}(\mathsf{o2},\ \mathsf{move}(\mathsf{load}(\mathsf{o2},\ \mathsf{load}(\mathsf{o1},\ \mathsf{s})))) \end{array}$$

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

$$\begin{array}{ll} \mathsf{rocket}(\mathsf{nil},\ \mathsf{s}) & \rightarrow\ \mathsf{move}(\mathsf{s}) \\ \mathsf{rocket}((\mathsf{o} : \mathsf{os}),\ \mathsf{s}) & \rightarrow\ \mathsf{unload}(\mathsf{o},\ \mathsf{sub}((\mathsf{o} : \mathsf{os}),\ \mathsf{s})) \\ \mathsf{sub}((\mathsf{o} : \mathsf{os}),\ \mathsf{s}) & \rightarrow\ \mathsf{s}' \end{array}$$

The operator χ_{sub} is deterministic and only defined if the RHS of the selected rule is rooted by a constructor. Even though χ_{sub} adds new rules to a candidate, it does not increase its cost because the added rules always only contain patterns already present in the candidate CS.

Introducing (Recursive) Function Calls. Introducing (recursive) function calls with appropriate arguments is the most complex operation. For an unfinished rule $f(\boldsymbol{p}) \rightarrow t$, the function call operator χ_{call} produces refinements of the form $f(\boldsymbol{p}) \rightarrow f'(g_1(\boldsymbol{p}), \ldots, g_n(\boldsymbol{p}))$, where f' is some already defined function (a target-, background- or previously introduced subfunction; probably $f = f'$) and the g_i are new defined functions to be induced subsequently. The idea behind the g_i as arguments (instead of just constructor terms over pattern variables) is that the arguments in the call of f' possibly need to be computed by another, possibly new and/or recursive, subfunction. As an example consider the *Quicksort* algorithm where the arguments of the two recursive calls are sublists of smaller and greater elements w.r.t. a pivot element. These sublists themselves are computed by recursive partitioning functions.

The operator χ_{call} is based on matching the example outputs of the current unfinished rule for f with outputs belonging to (other) target or background functions f' and then computing arguments that appropriately map the inputs covered by the current rule for f to the corresponding inputs of f'.

Consider the unfinished initial rule for sub as introduced in the previous step: $\mathsf{sub}((\mathsf{o} : \mathsf{os}),\ \mathsf{s}) \rightarrow \mathsf{s}'$. The RHSs of both example rules of sub are subsumed by RHSs of the rocket examples. Particularly, $\mathsf{move}(\mathsf{s})$ (the RHS of the 1st rocket example) subsumes $\mathsf{move}(\mathsf{load}(\mathsf{o1},\ \mathsf{s}))$ (the RHS of the 1st sub example) by substitution $\sigma_1 = \{\mathsf{s} \mapsto \mathsf{load}(\mathsf{o1},\ \mathsf{s})\}$; and $\mathsf{unload}(\mathsf{o1},\ \mathsf{move}(\mathsf{load}(\mathsf{o1},\ \mathsf{s})))$ (2nd rocket example) subsumes $\mathsf{unload}(\mathsf{o2},\ \mathsf{move}(\mathsf{load}(\mathsf{o2},\ \mathsf{load}(\mathsf{o1},\ \mathsf{s}))))$ (2nd sub example) by

substitution $\sigma_2 = \{\mathsf{o1} \mapsto \mathsf{o2},\ \mathsf{s} \mapsto \mathsf{load(o1,\ s)}\}$. This indicates that the sub examples can be computed by calling rocket and the unfinished sub rule is refined to

$$\mathsf{sub((o\ :\ os),\ s)\ \rightarrow\ rocket(g1((o\ :\ os),\ s),\ g2((o\ :\ os),\ s))}\ .$$

It remains to derive example rules for the new subfunctions g1 and g2 and computing initial rules for them. The example inputs are the same as for sub because g1 and g2 are called with the same inputs as sub, due to the same arguments o : os, s. The functions g1, g2 need to map these inputs to the correct inputs of rocket. Therefore, χ_{call} applies the substitutions σ_1, σ_2 to the *LHSs* of the rocket example rules and takes the appropriate subterms as outputs for g1, g2:

$$\begin{array}{llll} \mathsf{g1((o1\ :\ nil),\ s)} & \rightarrow\ \mathsf{nil} & \mathsf{g2((o1\ :\ nil),\ s)} & \rightarrow\ \mathsf{load(o1,\ s)} \\ \mathsf{g1((o1\ :\ o2\ :\ nil),\ s)} & \rightarrow\ \mathsf{o2\ :\ nil} & \mathsf{g2((o1\ :\ o2\ :\ nil),\ s)} & \rightarrow\ \mathsf{load(o1,\ s)} \end{array}$$

The initial rules (LGGs) for g1 and g2 obtained from their example rules are finished so that the following finished CS has been achieved as solution:

$$\begin{array}{ll} \mathsf{rocket(\ nil\ ,\ s)} & \rightarrow\ \mathsf{move(s)} \\ \mathsf{rocket((o\ :\ os),\ s)} & \rightarrow\ \mathsf{unload(o,\ sub((o\ :\ os),\ s))} \\ \mathsf{sub((o\ :\ os),\ s)} & \rightarrow\ \mathsf{rocket(g1((o\ :\ os),\ s),\ g2((o\ :\ os),\ s))} \\ \mathsf{g1((o\ :\ os),\ s)} & \rightarrow\ \mathsf{os} \\ \mathsf{g2((o\ :\ os),\ s)} & \rightarrow\ \mathsf{load(o,\ s)} \end{array}$$

Neither sub nor g1, g2 are recursive. Hence they can be eliminated by unfolding, leading to the solution in Listing 1.2.

Like χ_{sub}, also χ_{call} does not increase the cost of the candidate because the added rules do not introduce additional patterns. To assure termination of IGOR2, the maximal depth of nested function calls, i.e., the maximal number of χ_{call} applications, is bounded by the user.

3 Discussion and Improvements

In this section we identify certain shortcomings of the splitting and the function call operators χ_{split} and χ_{call} and propose variants that circumvent the problems.

3.1 Rapid Rule-Splitting

Consider the *Ackermann* function, defined as a CS with constructors 0 (zero), S_ (successor) and variables m, n:

$$\begin{array}{ll} \mathsf{Ack\ (0,\ n)} & \rightarrow\ \mathsf{S\ n} \\ \mathsf{Ack\ (S\ m,\ 0)} & \rightarrow\ \mathsf{Ack\ (m,\ S\ 0)} \\ \mathsf{Ack\ (S\ m,\ S\ n)} & \rightarrow\ \mathsf{Ack\ (m,\ A\ (S\ m,\ n))} \end{array}$$

Given some I/O examples where all the four cases of zero and non-zero inputs for both arguments are covered, the initial unfinished rule would be $\mathsf{Ack(m,n)} \rightarrow\ \mathsf{(S\ x)}$, featuring two pivot positions that correspond to the variables m, n in the LHS. χ_{split} would thus introduce two successor candidates, each specializing one of

the two pattern variables to the two cases zero and non-zero. W.l.o.g., let P denote one of them. P is unfinished again and does not contain the pattern of the *third* rule of the *Ackermann* CS because in that rule, *both* pattern components are non-variables. Thus, a further application of χ_{split} to P, leading, say, to P', where the cost of P' is increased compared to P, would be necessary. However, the subprogram and the function call operators would also be applicable to P *without* increasing its cost. Hence, before P' is considered again, all possible sequences of χ_{sub} and χ_{call} applications to P would be tried.

The idea of an improved version of χ_{split} is to *combine* all possible splitting refinements—if more than one pivot position and hence more than one splitting exists—into one single splitting. Instead of computing a separate partition for each pivot position, we compute only *one* partition based on all combinations of different constructors at all pivot positions. In the case of the *Ackermann* function, instead of two refinements with two successor rules each, we then get *one* refinement with *four* successor rules, covering all the four combinations for zero and non-zero inputs for the two arguments: Ack(0, 0), Ack(0, S n), Ack(S m, 0), Ack(S m, S n). These patterns cover all patterns of the actual definition of the *Ackermann* function such that subsequent applications of χ_{sub} and χ_{call} take place in a search subspace which contains the solution. Since this rule splitting variant achieves the result of several applications of χ_{split} in one step, we call it *rapid rule-splitting* and denote it by χ_{rsplit}. The solution for the Ackermann function that is induced with rapid rule-splitting enabled, is:

```
Ack (0, 0)      → S 0
Ack (0, S n)    → S S n
Ack (S m, 0)    → Ack (m, S 0)
Ack (S m, S n)  → Ack (m, Ack (S m, n))
```

A minor drawback of rapid rule-splitting is its potential "over-specialization" as in the case of the *Ackermann* function (four induced rules instead of the sufficient three rules). This is not a problem as long as enough examples are provided. If, however, only few I/O examples are provided, then rapid rule-splitting might prevent a correct generalization since too few I/O examples might remain for each rule.

3.2 Simple Function Calls

Consider the following example rules, specifying the last function that returns the last element of a list (x,y,z denote variables):

```
1 last (x : nil)           → x
2 last (x : y : nil)       → y
3 last (x : y : z : nil)   → z
4 last (x : y : z : v : nil) → v
```

Further assume rule-splitting had already taken place so that the intermediate, unfinished, candidate CS is:

```
last (x : nil)     → x
last (x : y : xs)  → q
```

The second unfinished rule covers example rules $2, 3, 4$. Now assume we apply χ_{call} to introduce a recursive call of the form $\mathsf{last}\,(\mathsf{x} : \mathsf{xs}) \rightarrow \mathsf{last}\,(\mathsf{g}(\mathsf{x} : \mathsf{xs}))$. This is possible since each RHS of rules $2, 3, 4$ matches with another RHS. Actually, since all RHSs are variables, each RHS matches each other. Not all of these matchings are considered because, to assure termination of the induced program, the argument of the call must be decreased. Hence for each example i only matchings to examples $j < i$ are considered. One single refinement according to χ_{call} is then determined by one particular mapping of each RHS of rules $2, 3, 4$ to another one, satisfying the ordering constraint. In our case these are $1 \times 2 \times 3 = 6$ possibilities, hence χ_{call} would result in 6 successor candidates. In general, the more example rules are given, the more different matchings are possible and the more successor candidates are introduced by χ_{call}.

However, in the case of last, the argument of the recursive call need not be computed by an own function but is a constructor term, namely the tail of the input list: $\mathsf{last}\,(\mathsf{x} : \mathsf{y} : \mathsf{xs}) \rightarrow \mathsf{last}\,(\mathsf{y} : \mathsf{xs})$. If only candidates with a constructor term as argument are considered, the correct solution is *the only possible one*.

Therefore, we developed an additional operator, χ_{scall}, called simple-call operator, that finds refinements of the form $f(\boldsymbol{p}) \rightarrow f'(\boldsymbol{p'})$, where $\boldsymbol{p'}$ is a constructor term over variables from $\boldsymbol{p}$. Instead of matching specified outputs, χ_{scall} basically works by enumerating constructor terms as arguments up to a certain size and testing each one against the examples.

We did not completely *replace* χ_{call} by χ_{scall} but we always first apply χ_{scall} and only if it returns an empty refinement set—indicating, that a constructor argument is not sufficient to find a consistent function call—the original call operator χ_{call} is applied. The general idea is thus to first check for the few potential simple solutions and only if none exists, search for more complicated solutions.

4 Experiments

We implemented the extended synthesis operators on top of a preliminary IGOR2 version [8] that is implemented in the interpreted, rewriting based language MAUDE [2]. We use the symbol IGOR2*pre* to exclusively denote this version. To empirically evaluate the extensions, we applied IGOR2 to several non-trivial recursive problems. Each problem was tested with three IGOR2 configurations: (i) with IGOR2*pre*, (ii) with the simple-call operator χ_{scall} added as described in Sec. 3.2, and (iii) with χ_{scall} added and additionally with χ_{split} replaced by χ_{rsplit} (rapid rule-splitting), described in Sec. 3.1. The experiments were run on an Intel Core i5 2.53 GHz, 64Bit Linux machine. We gave each tested IGOR2 configuration 2 minutes synthesis time maximum per problem. Tab. 1 shows the results.

The *Blocksworld* problem tower is taken from [15] where IGOR2*pre* had been applied in the domain of cognitive modeling. tower denotes the recursive problem of building a tower of any number of blocks from initial configurations in the *Blocksworld*. The concept CB of how to clear a block (i.e., putting all blocks above

Table 1. Results

Problems	IGOR2 versions; times in sec.		
	pre	$+\chi_{\text{scall}}$	$+\chi_{\text{scall}} + \chi_{\text{rsplit}}$
tower w/ CB, isTower	0.90	0.94	0.89
lasts	38.37	0.43	$0.54^{\perp}$
lasts w/ last	⊘	0.23	0.23
drop	⊘	9.87	0.05
swap	⊘	⊘	1.15
ack	⊘	⊘	2.22
weave	⊘	⊘	30.01
oddslist	⊘	⊘	⊘

⊘: Timeout after 2 minutes; $\perp$: one case overly special

pre: IGOR2*pre*; $+\chi_{\text{scall}}$: χ_{call} only applied if χ_{scall} fails
$+\chi_{\text{scall}} + \chi_{\text{rsplit}}$: like $+\chi_{\text{scall}}$ and χ_{split} replaced by χ_{rsplit}

it to the table) was given as background knowledge as well as a predicate isTower to test if a certain tower is already present. We used the original examples. Since this problem is highly structured and the examples were well-chosen, IGOR2*pre* could tackle it well and the extensions have no impact.

The problems lasts and oddslist are taken from [6] where IGOR2*pre* has been compared with other recent IP systems on some list-processing problems. It was shown that IGOR2, pursuing a combined analytic and search-based approach, (i) could correctly induce more problems than the recent analytic system IGOR1 [9] and *inductive logic programming* systems like FOIL [14] and (ii) outperformed recent generate-and-test based functional IP systems [12, 7] on several tested problems. lasts takes a list of lists and returns a flat list of their last elements. The predicate oddslist takes a list of natural numbers, encoded as Peano numbers by 0 and succ, and returns *true* or *false* depending on whether all elements are odd. No background knowledge was provided, so the IP systems had to invent subfunctions last and odd or equivalent ways to compute the inherent subproblems.

It is well-known in AI that background knowledge generally can help to find solutions for complex problems, but also that *irrelevant* information can hamper finding a solution. An odd thing with IGOR2*pre* is that even *relevant* background knowledge may lead to increased synthesis time. This can be observed in the case of lasts which we tested (i) w/o background knowledge and (ii) with last as background knowledge. IGOR2*pre* could not find a solution in 2 minutes if last was provided. In contrast, IGOR2 with the additional χ_{scall} operator could, as one should expect, profit from the relevant background knowledge. The additional use of rapid rule-splitting had no further impact. We further observe that rapid rule-splitting did not completely generalize to the intended function in case of lasts w/o background knowledge. This is an example for that rapid rule-splitting might need additional examples to generalize well (cp. Sec. 3.1). The predicate

Listing 1.3. swap, induced from 6 examples

```
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)
```

Listing 1.4. weave, induced from 11 expls; 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)
```

oddslist could not be synthesized by any version within the allowed 2 minutes. Boolean-valued functions are generally hard for IGOR2 because of the missing structure in the outputs (which are just *true* or *false* in this case).

The function drop drops the first n elements from a list. We made this problem challenging for IGOR2*pre* by including the case where n is greater than the number of elements in the list in which case the empty list shall be returned. This leads to several I/O examples where the output is just the empty list, posing a problem for IGOR2*pre* because this causes many possible matchings of outputs. Since the solution does not contain nested functions calls, χ_{scall} quickly found a solution.

Finally, the *Ackermann* function ack and the functions weave and swap all are more complex than the former functions in terms of syntactical size, recursion structure, and/or number of parameters that are substituted in the recursive calls. They were neither solvable by IGOR2*pre* nor by just adding χ_{scall}. Yet with rapid rule-splitting enabled, many small candidates, mostly non-solutions, are pruned so that all three functions could be induced. swap swaps two elements in a list, indicated by their indices, e.g., swap([a,b,c,d], 2, 4) → [a,d,c,b]. It was restricted to cases where the given indices occurred in the list, were different, and the first index was the smaller one. Listing 1.3 shows the induced solution. weave 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.[2] Listing 1.4 shows the induced solution.

5 Related Research

Two recent functional IP systems are ADATE [12] and MAGICHASKELLER [7]. Both pursue a generate-and-test approach, i.e., use examples as test-cases, in-

[2] This is a generalized version of the weave function as tested in [6].

stead of directly deriving candidates from them as IGOR2 does. One advantage of this approach is that it is more robust w.r.t. the selection of and noise in problem specifications. However, they often need much more time to synthesize a solution [6]. In logic programming, the IP system DIALOGS [3] is closest to IGOR2. It is interactive and uses algorithm schemas like *divide-and-conquer*. Recently, domain-specific IP methods are again studied; e.g., [4] describes an algorithm to interactively synthesize string-processing programs in spreadsheets, and [11] describes a system to learn recursive hierarchical task networks in automated planning.

6 Conclusions and Future Work

IP is a challenging field with various important applications and much room for further improvement. We presented IGOR2, a competitive IP system that draws from different existing approaches to approach the practical tractability of relevant problems. We described improvements of two synthesis operators and empirically showed their significance w.r.t. efficient synthesis of non-trivial programs. It is worth noting that the efficiency-gain is *not* based on making the search less complete. The only drawback is that more examples might be needed in case of rapid rule-splitting. Despite the remarkable results, there is still much room for improvement. The current synthesis operators rule out certain program forms and if the examples do not contain sufficient structure, the BF search becomes intractable.

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 w.r.t. missing or erroneous information, as generally present in the real-world, probabilistic reasoning must be integrated into analytic IP.

Currently, we work on applying IGOR2 to learning hierarchical task networks in automated planning.

Acknowledgments

While writing this paper, the author was funded by the German Academic Exchange Service (DAAD) within the FIT program.

References

1. Baader, F., Nipkow, T.: Term Rewriting and All That. Cambridge University Press (1999)

2. 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)
3. Flener, P.: Inductive logic program synthesis with DIALOGS. In: 6th International Workshop on Inductive Logic Programming, (ILP'96), Selected Papers. LNCS, vol. 1314, pp. 175–198 (1997)
4. Gulwani, S.: Automating string processing in spreadsheets using input-output examples. In: 38th SIGACT-SIGPLAN Symposium on Principles of Programming Languages. ACM (2011)
5. Hastie, T., Tibshirani, R., Friedman, J.: The Elements of Statistical Learning. Springer-Verlag, 2nd edn. (2009)
6. Hofmann, M., Kitzelmann, E., Schmid, U.: A unifying framework for analysis and evaluation of inductive programming systems. In: Artificial General Intelligence (AGI'09). pp. 55–60. Atlantis Press (2009)
7. Katayama, S.: Systematic search for lambda expressions. In: 6th Symposium on Trends in Functional Programming, selected Papers. pp. 111–126. Intellect (2007)
8. Kitzelmann, E.: Analytical inductive functional programming. In: 18th International Symposium on Logic-Based Program Synthesis and Transformation, Revised Selected Papers. LNCS, vol. 5438, pp. 87–102. Springer-Verlag (2009)
9. Kitzelmann, E., Schmid, U.: Inductive synthesis of functional programs: An explanation based generalization approach. Journal of Machine Learning Research 7, 429–454 (2006)
10. McCarthy, J., Hayes, P.J.: Some philosophical problems from the standpoint of artificial intelligence. In: Machine Intelligence, vol. 4, pp. 463–502. Edinburgh University Press (1969)
11. Nejati, N., Langley, P., Konik, T.: Learning hierarchical task networks by observation. In: 23rd International Conference on Machine Learning. pp. 665–672. ACM (2006)
12. Olsson, J.R.: Inductive functional programming using incremental program transformation. Artificial Intelligence 74(1), 55–83 (1995)
13. Plotkin, G.D.: A note on inductive generalization. Machine Intelligence 5, 153–163 (1970)
14. Quinlan, J.R., Cameron-Jones, R.M.: FOIL: A midterm report. In: Proceedings of the 6th European Conference on Machine Learning. pp. 3–20. LNCS, Springer-Verlag (1993)
15. Schmid, U., Kitzelmann, E.: Inductive rule learning on the knowledge level. Cognitive Systems Research (2010)
16. Smith, D.R.: The synthesis of LISP programs from examples: A survey. In: Automatic Program Construction Techniques, pp. 307–324. Macmillan (1984)
17. Veloso, M.M., Carbonell, J.G.: Derivational analogy in prodigy: Automating case acquisition, storage, and utilization. Machine Learning 10, 249–278 (1993)