

# Analytical Inductive Functional Programming\*

Emanuel Kitzelmann

University of Bamberg, 96045 Bamberg, Germany

[emanuel.kitzelmann@uni-bamberg.de](mailto:emanuel.kitzelmann@uni-bamberg.de)

<http://www.uni-bamberg.de/kogsys/members/kitzelmann/>

**Abstract.** We describe a new method to induce functional programs from small sets of non-recursive equations representing a subset of their input-output behaviour. Classical attempts to construct functional LISP programs from input/output-examples are *analytical*, i.e., a LISP program belonging to a strongly restricted program class is algorithmically derived from examples. More recent approaches enumerate candidate programs and only *test* them against the examples until a program which correctly computes the examples is found. Theoretically, large program classes can be induced generate-and-test based, yet this approach suffers from combinatorial explosion. We propose a combination of search and analytical techniques. The method described in this paper is search based in order to avoid strong a-priori restrictions as imposed by the classical analytical approach. Yet candidate programs are computed based on analytical techniques from the examples instead of being generated independently from the examples. A prototypical implementation shows first that programs are inducible which are not in scope of classical purely analytical techniques and second that the induction times are shorter than in recent generate-and-test based methods.

## 1 Introduction

Synthesis of recursive declarative programs from input/output-examples (I/O-examples) is an active area of research since the seventies, though it has always been only a niche within other research fields such as machine learning, inductive logic programming (ILP) or functional programming.

The basic approach to inductive inference is to simply enumerate concepts—programs in this case—of a defined class until one is found which is consistent with the examples. When new examples appear and some of them contradict the current program then enumeration continues until a program consistent with the extended set of examples is found. Gold [1] stated this inductive inference model precisely and reported first results. Due to combinatorial explosion, this general enumerative approach is too expensive for practical use.

Summers [2] developed a completely different method to induce functional LISP programs. He showed, first, that under certain conditions traces—expressions

---

\* Research was supported by the German Research Community (DFG), grant SCHM 1239/6-1.

computing a single input or a finite set of inputs correctly—can be computed from I/O-examples without search and, second, that for certain classes of LISP programs even a recursive program can be constructed without search in a program space from traces. It suffices to search for repetitive patterns in the traces from which a recursive function definition can be derived. Several variants and extensions of Summers method have been proposed. An early overview is given in [3], a recent extension is described in [4]. We call this approach *analytical*. However, due to strong constraints imposed to the forms of I/O-examples and inducible programs in order to avoid search, only relatively simple functions can be induced.

Recently, some papers on the induction of functional programs have been published in the functional programming community [5,6]. They all take a generate-and-test based approach. A further enumerative functional system is ADATE [7]. Though these systems use type information, some of them higher-order functions, and further more or less sophisticated techniques for pruning the search space, they strongly suffer from combinatorial explosion. Again only comparatively small problems can be solved with acceptable resources and in reasonable time.

Also ILP [8] has originated some methods intended for, or respectively capable of, inducing recursive programs on inductive datatypes [9,10,11,12], though ILP in general has a focus on classification and concept learning. In ILP one distinguishes top-down systems starting with an overly general theory and stepwise specialising it, e.g., FOIL [10], and bottom-up systems which search in the converse direction, e.g., GOLEM [9]. Typically, top-down systems are generate-and-test based whereas bottom-up systems contain analytical elements such as least generalisations [13]. Most often, clauses are constructed one after the other, called the *covering approach*, and single clauses are constructed by a greedy search. This might be adequate for classifier induction but has its drawbacks for inducing algorithmic concepts including recursion. Moreover, preference biases such as the *foil-gain* and also the most often used generality model  $\theta$ -*subsumption* are inadequate for the induction of recursive programs. This led to the development of ILP methods specialised for program induction, partly based on *inverse implication* instead of  $\theta$ -subsumption, as surveyed in [12]. Especially the idea of *sub-unification* [14] is similar to the analytical approach of inducing functional programs. The problems are similar to those of the functional approaches described above.

In this paper we suggest a particular combination of the analytical and the enumerative approach in order to put their relative strengths into effect and to repress their relative weaknesses. The general idea is to base the induction on a search in order to avoid strong a-priori restrictions on I/O-examples and inducible programs. But in contrast to the generate-and-test approach we construct successor programs during search by using analytical techniques similar to those proposed by Summers, instead of generating them independently from the examples. We represent functional programs as sets of recursive first-order equations. Adopting machine learning terminology, we call such generated equation sets *hypotheses*. The effect of constructing hypotheses w.r.t. example equations is that only hypotheses entailing the example equations are enumerated. In

contrast to greedy search methods, the search is still complete—only programs known to be incorrect w.r.t. the I/O-examples are ruled out. Besides narrowing the search space, analytical hypotheses construction has the advantage that the constructed hypotheses need not to be evaluated on the I/O-examples since they are correct by construction.

The rest of the paper is organised as follows: In the following section we shortly review Summers method to derive recursive function definitions from recurrence relations found in a trace. In Section 3 we introduce some basic terminology. In Section 4 we present a strong generalisation of Summers theorem. In Section 5 we describe an algorithm based on the generalised synthesis theorem. Section 6 shows results for some sample problems, and in Section 7, we conclude.

## 2 Summers Synthesis Theorem

Summers showed how linear-recursive LISP-programs can be derived from a finite set of I/O-examples  $\{e_1, \dots, e_k\}$  without search. The method consists of two steps: First, a trace in form of a McCarthy-Conditional with predicates and program fragments for each I/O-pair

$$F(x) = (p_1(x) \rightarrow f_1(x), \dots, p_{k-1}(x) \rightarrow f_{k-1}(x), T \rightarrow f_k(x))$$

is constructed. Second, recurrences between predicates  $p_i(x)$  and program fragments  $f_i(x)$  are identified which are used to construct a recursive program.

*Example 1.* Consider the following I/O-examples for computing the initial sequence of a list:  
 $\langle\langle(A), ()\rangle\rangle, \langle\langle(A, B), (A)\rangle\rangle, \langle\langle(A, B, C), (A, B)\rangle\rangle, \langle\langle(A, B, C, D), (A, B, C)\rangle\rangle$ . From these, the following trace will be derived:

$$\begin{aligned} F(x) = & (atom(cdr(x)) \rightarrow nil \\ & atom(cddr(x)) \rightarrow cons(car(x), nil) \\ & atom(cddd(x)) \rightarrow cons(car(x), cons(cadr(x), nil)) \\ & T \rightarrow cons(car(x), cons(cadr(x), cons(caddr(x), nil)))) \end{aligned}$$

The following recurrences are identified:

$$\begin{aligned} p_{i+1}(x) &= p_i(cdr(x)) \quad \text{for } i = 1, 2 \\ f_{i+1}(x) &= cons(car(x), f_i(cdr(x))) \quad \text{for } i = 1, 2, 3 \end{aligned}$$

The recurrences are inductively generalised to  $i \in \mathbb{N}_+$  and the following recursive LISP-program results:

$$\begin{aligned} F(x) &= (atom(cdr(x)) \rightarrow nil \\ & \quad T \rightarrow F'(x)) \\ F'(x) &= (atom(cddr(x)) \rightarrow cons(car(x), nil) \\ & \quad T \rightarrow cons(car(x), F'(cdr(x)))) \end{aligned}$$

The first step, construction of traces, is based on strong constraints: First, traces are composed from a small fixed set of operation symbols, namely *cons*, *car*, *cdr* to uniquely compose and decompose S-expressions, the McCarthy-Conditional, and the predicate *atom* to check atomicity of S-expressions; second, each atom occurring in an example output must also occur in the corresponding example input; third, each atom may not occur more than once in an input; and fourth, the example inputs must be totally ordered according to a particular order over S-expressions. These restrictions, especially the first and the last one, are far too strong for practical use. In practice, programs make use of functions predefined in some library. This is not possible due to the first restriction. Moreover, comparing values regarding equality or order is not possible due to *atom* as the only allowed predicate. The last restriction forces the induced programs to be linear recursive. All classical methods surveyed in [3] as well as the recent powerful extension of Summers described in [4] suffer from these restrictions.

Our method described in Section 5 is not based on this first restrictive step of the classical methods and therefore we do not describe it here.

Now suppose that fragments and predicates for examples  $\{e_1, \dots, e_k\}$  can be expressed as recurrence relations with  $n$  being a fixed number  $> 0$ .

$$\begin{aligned} f_1(x), \dots, f_n(x), f_{i+n}(x) &= a(f_i(b(x)), x), \\ p_1(x), \dots, p_n(x), p_{i+n}(x) &= p_i(b(x)) \quad \text{for } 1 < i \leq k - n - 1 \end{aligned} \quad (1)$$

If  $k - 1 > 2n$  then we *inductively infer* that the recurrence relations hold for all  $i > 1$ . Given such a generalised recurrence relation, more and more predicates and program fragments can be derived yielding a chain of approximations to the function specified by the generalised recurrence relations. The function itself is defined to be the supremum of that chain.

**Theorem 1 (Basic Synthesis).** *The function specified by the generalisation of recurrence relations as defined in (1) is computed by the recursive function*

$$F(x) = (p_1(x) \rightarrow f_1(x), \dots, p_n(x) \rightarrow f_n(x), T \rightarrow a(F(b(x)), x))$$

The theorem is proved in [2]. It is easily extended to cases where the smallest index for the repetitive patterns is greater than 1 as in example 1 and for different basic functions  $b$  for fragments and predicates.

*Remark.* Note that the recurrence detection method finds differences *between* fragments and predicates of respectively two I/O-examples. Therefore, the provided examples need to be complete in the sense that if one I/O-pair with an input  $i$  of a particular complexity is given then all I/O-pairs with inputs of the intended domain smaller than  $i$  according to the order over S-expressions also must be given. For example, if a function for computing lists of arbitrary length including the empty list shall be induced and one provides an example for an input list of length 4 then also examples for input lists of length 3, 2, 1, and 0 must be given. In [3], also methods to find recurrences *within* a program fragment of an I/O-pair are surveyed. For this methods theoretically one I/O-pair suffices to induce a recursive function. Yet such methods are again based on enumerative search in program space.

### 3 Equations and Term Rewriting

We shortly introduce concepts on terms and term rewriting here as it is described, e.g., in [15].

We represent I/O-examples and functional programs as sets of equations (pairs of terms) over a first-order signature  $\Sigma$ . The variables of a term  $t$  are denoted by  $Var(t)$ . Signatures are many-sorted, i.e., terms are typed.

Each equation defining (amongst other equations) a function  $F$  has a left-hand side (lhs) of the form  $F(p_1, \dots, p_n)$  where neither  $F$  nor the name of any other of the defined functions occur in the  $p_i$ . Thus, the symbols in  $\Sigma$  are divided into two disjoint subsets  $\mathcal{D}$  of *defined function symbols* and  $\mathcal{C}$  of (type) *constructors*. In the following we assume that all considered sets of equations have this form.

Terms without defined function symbols are called *constructor terms*. The constructor terms  $p_i$  in the lhs of the equations for a defined function  $F$  may contain variables and are called *pattern*. This corresponds to the concept of pattern matching in functional programming languages and is our only form of case distinction. Each variable in the right-hand side (rhs) of an equation must occur in the lhs, i.e., in the pattern. We say that rhs variables must be *bound* (by the lhs).

In order to evaluate a function defined by equations we read the equations as *simplification (or rewrite) rules* from left to right as known from functional programming, i.e., as a *term rewriting system (TRS)*. TRSs whose lhs have defined function symbols as roots and constructor terms as arguments, i.e., whose lhs have the described pattern-matching form, are called *constructor (term rewriting) systems (CSs)*. We use the terms *equation* and *rule* as well as *equation set* and *CS* interchangeably throughout the paper, depending on the context. Evaluating an  $n$ -ary function  $F$  for an input  $i_1, \dots, i_n$  consists of repeatedly rewriting the term  $F(i_1, \dots, i_n)$  w.r.t. the rewrite relation implied by the CS until the term is in *normal form*, i.e., cannot be further rewritten. A sequence of (in)finitely many rewrite steps  $t_0 \rightarrow_R t_1 \rightarrow_R \dots$  is called *derivation*. If a derivation starts with term  $t$  and results in a normal form  $s$ , then  $s$  is called normal form of  $t$ , written  $t \xrightarrow{!} s$ . We say that  $t$  normalises to  $s$ . In order to define a *function* on a domain (a set of ground terms) by a CS, no two derivations starting with the same ground term may lead to different normal forms, i.e., normal forms must be unique. A sufficient condition for this is that no two lhs of a CS unify; this is a sufficient condition for a CS to be *confluent*. A CS is *terminating* if each possible derivation reaches a normal form in finitely many rewrite steps. A sufficient condition for termination is that the arguments/inputs of recursive calls strictly decrease within each derivation and w.r.t. a well founded order.

A *substitution*  $\sigma$  is a mapping from variables to terms and is uniquely extended to a mapping from terms to terms which is also denoted by  $\sigma$  and written in postfix notation;  $t\sigma$  is the result of applying  $\sigma$  to all variables in term  $t$ . If  $s = t\sigma$ , then  $t$  is called *generalisation* of  $s$  and we say that  $t$  *subsumes*  $s$  and that  $s$  *matches*  $t$  by  $\sigma$ . Given two terms  $t_1, t_2$  and a substitution  $\sigma$  such that  $t_1\sigma = t_2\sigma$ , then we say that  $t_1, t_2$  *unify*. Given a set of terms,  $S = \{s, s', s'', \dots\}$ , then there exists a term  $t$  which subsumes all terms in  $S$  and which is itself

subsumed by each other term subsuming all terms in  $S$ . The term  $t$  is called *least general generalisation (lgg)* of the terms in  $S$  [13]. We also need lgg's of *equations*. Therefore, consider an equation as a term with the equal sign as root symbol and the lhs and rhs as first and second subterm respectively. Then the lgg of a set of equations is the equation represented by the lgg of the terms representing the set of equations.

Each “node” in a term  $t$  has a unique position  $u$  denoted by  $t|_u$ .

## 4 A Generalised Synthesis Theorem

We call equations representing I/O-examples *example equations*. They have constructor terms as rhss, i.e., they are not allowed to contain function calls. Although example equations may contain variables, we call their lhss (*example inputs*) and their rhss (*example outputs*).

**Theorem 2.** *Let  $E$  be a set of example equations,  $F$  a defined function symbol occurring in  $E$ ,  $p$  a pattern for  $F$ , and  $E_{F,p} \subseteq E$  the example equations for  $F$  whose lhss match  $F(p)$  with substitutions  $\sigma$ .*

*If it exist a context  $C$ , a defined function symbol  $F'$  (possibly  $F' = F$ ) occurring in  $E$ , and a further defined function  $G$  such that for every  $F(i) = o \in E_{F,p}$  exist an equation  $F'(i') = o' \in E$  and a substitution  $\tau$  with*

$$o = C\sigma[o'\tau] \quad \text{and} \quad G(i) \stackrel{!}{\mapsto} i'\tau$$

*then*

$$(E \setminus E_{F,p}) \cup \{F(p) = C[F'(G(p))]\} \models E$$

*Proof.* Obviously it suffices to proof that  $E_{F,p}$  is modelled. Let  $F(i) = o$  be an equation in  $E_{F,p}$  and  $\sigma$  the substitution such that  $F(i) = F(p)\sigma$ . Then

$$F(i) = C[F'(G(p))]\sigma = C\sigma[F'(i'\tau)] = C\sigma[o'\tau] = o$$

□

The theorem states conditions under which example equations whose inputs match a lhs  $F(p)$  can be replaced by one single equation with this lhs and a rhs containing recursive calls or calls to other defined functions  $F'$ . The theorem generalises Summers' theorem and its extensions (cp. [3]) in several aspects: The example inputs need not be totally ordered, differences between example outputs are not restricted to the function to be induced but may include other user-defined functions, so called *background functions*. The number of recursive calls or calls of other defined functions is not limited.

As in Summers recurrence detection method, (recursive) function calls are hypothesised by finding recurrent patterns *between* equations and not within equations. Therefore, the requirement for complete sets of example equations applies to our generalisation as well.

Note that the theorem is restricted to unary functions  $F$ . This is easily extended to functions  $F$  with arity  $> 1$ .

## 5 The Igor2-Algorithm

In the following we write vectors  $t_1, \dots, t_n$  of terms abbreviated as  $\mathbf{t}$ .

Given a set of example equations  $E$  for any number of defined function symbols and background equations  $B$  describing the input-output behaviour of further user-defined functions on a suitable subset of their domains with constructor rhss each, the IGOR2 algorithm returns a set of equations  $P$  constituting a confluent constructor system which is *correct* in the following sense:

$$F(\mathbf{i}) \xrightarrow{!}_{P \cup B} o \quad \text{for all } F(\mathbf{i}) = o \in E \quad (2)$$

The constructor system constituted by the induced equations together with the background equations rewrites each example input to its example output. The functions to be induced are called *target functions*.

### 5.1 Signature and Form of Induced Equations

Let  $\Sigma$  be the signature of the example- and the background equations respectively. Then the signature of an induced program is  $\Sigma \cup \mathcal{D}_A$ .  $\mathcal{D}_A$  is a set of defined function symbols not contained in the example- and background equations and not declared in their signatures. These defined function symbols are names of auxiliary, possibly recursive, functions dynamically introduced by IGOR2. This corresponds to predicate invention in inductive logic programming.

Auxiliary functions are restricted in two aspects: First, the type of an auxiliary function is identical with the type of the function calling it. That is, auxiliary functions have the same types as the target functions. IGOR2 cannot automatically infer auxiliary parameters as, e.g., the accumulator parameter in the efficient implementation of *Reverse*. Second, auxiliary function symbols cannot occur at the root of the rhs of another function calling it. Such restrictions are called *language bias*.

### 5.2 Overview over the Algorithm

Obviously, there are infinitely many correct solutions  $P$  in the sense of (2), one of them  $E$  itself. In order to select one or at least a finite subset of the possible solutions at all and particularly to select “good” solutions, IGOR2—like almost all inductive inference methods—is committed to a *preference bias*. IGOR2 prefers solutions  $P$  whose patterns partition the example inputs in fewer subsets. The search for solutions is complete, i.e., solutions inducing the *least number* of subsets are found. Spoken in programming terminology: A functional program, fitting the language bias described above, with the least number of case distinctions correctly computing all specified examples is returned. This assures that the recursive structure in the examples as well as the computability through predefined functions is best possible covered in the induced program.

*Example 2.* From the type declarations

$$\begin{aligned} \text{nil} & : \rightarrow \text{List} \\ \text{cons} & : \text{Elem List} \rightarrow \text{List} \\ \text{Reverse} & : \text{List} \rightarrow \text{List} \\ \text{Last} & : \text{List} \rightarrow \text{Elem} \end{aligned}$$

the example equations<sup>1</sup>

$$\begin{aligned} 1. \text{ Reverse}([\ ]) & = [\ ] \\ 2. \text{ Reverse}([X]) & = [X] \\ 3. \text{ Reverse}([X, Y]) & = [Y, X] \\ 4. \text{ Reverse}([X, Y, Z]) & = [Z, Y, X] \\ 5. \text{ Reverse}([X, Y, Z, V]) & = [V, Z, Y, X] \end{aligned}$$

and the background equations

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

IGOR2 induces the following equations for *Reverse* and an auxiliary function *Init*<sup>2</sup>:

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

The induction of a terminating, confluent, correct CS is organised as a best first search, see Algorithm 1.

During search, a hypothesis is a set of equations entailing the example equations and constituting a terminating and confluent CS *but potentially with unbound variables in the rhss*. 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, i.e., does not contain unfinished equations. Such a finished hypothesis is terminating and confluent by construction and since its equations entail the example equations, it is also correct.

W.r.t. the described preference bias and in order to get a complete hypothesis w.r.t. the examples, the initial hypothesis is a CS with one rule per target function such that its pattern subsumes all example inputs. In most cases (e.g., for all recursive functions) one rule is not enough and the rhss will remain unfinished. Then for one of the unfinished rules successors will be computed which

<sup>1</sup> We use a syntax for lists as known from PROLOG.

<sup>2</sup> If no background equations are provided, then IGOR2 also induces *Last* as auxiliary function.

**Algorithm 1.** The general IGOR2 algorithm

---

```

 $h \leftarrow$  the initial hypothesis (one rule per target function)
 $H \leftarrow \{h\}$ 
while  $h$  unfinished do
   $r \leftarrow$  an unfinished rule of  $h$ 
   $S \leftarrow$  all successor rule sets of  $r$ 
  foreach  $h \in H$  with  $r \in h$  do
    remove  $h$  from  $H$ 
    foreach successor set  $s \in S$  do
      add  $h$  with  $r$  replaced by  $s$  to  $H$ 
   $h \leftarrow$  a hypothesis from  $H$  with the minimal number of case distinctions
return  $H$ 

```

---

leads to one or more hypotheses. Now repeatedly unfinished rules of currently best hypotheses are replaced until a currently best hypothesis is finished. Since one and the same rule may be member of different hypotheses, the successor rules originate successors of *all* hypotheses containing this rule. Hence, in each induction step several hypotheses are processed.

### 5.3 Initial Rules

Given a set of example equations for one target function, the initial rule is constructed by antiunifying [13] all example equations. This leads to the lgg of the example equations. In particular, the pattern of the initial rule is the most specific term subsuming all example inputs. Considering only lggs of example inputs as patterns narrows the search space. It does not constrain completeness of the search as shown by the following theorem.

**Theorem 3.** *Let  $R$  be a CS with non-unifying but possibly not most specific patterns which is correct regarding a set of example equations. Then there exists a CS  $R'$  such that  $R'$  contains exactly one lhs  $F(\mathbf{p}')$  for each lhs  $F(\mathbf{p})$  in  $R$ , each  $\mathbf{p}'$  being the lgg of all example inputs matching  $\mathbf{p}$ , and  $R$  and  $R'$  compute the same normal form for each example lhs.*

*Proof.* It suffices to show (i) that if a pattern  $\mathbf{p}$  of a rule  $r$  is replaced by the lgg of the example inputs matching  $\mathbf{p}$ , then also the rhs of  $r$  can be replaced such that the rewrite relation remains the same for the example lhss matching  $F(\mathbf{p})$ , and (ii) that if the rhs of  $r$  contains a call to a defined function then each instance of this call regarding the example inputs matching  $\mathbf{p}$  is also an example lhs (subsumed by  $F(\mathbf{p})$  or another lhs and, hence, also subsumed by lhss constituting lggs of example lhss). Proving these two conditions suffices because (i) assures equal rewrite relations for the example lhss and (ii) assures that each resulting term of one rewrite step which is not in normal form regarding  $R$  is also not in normal form regarding  $R'$ .

The second condition is assured if the example equations are complete. To show the first condition let  $F(\mathbf{p})$  be a lhs in  $R$  such that  $\mathbf{p}$  is *not* the lgg of the example inputs matching it. Then there exists a position  $u$  with  $\mathbf{p}|_u = X$ ,  $X \in \text{Var}(\mathbf{p})$  and  $\mathbf{p}'|_u = s \neq X$  if  $\mathbf{p}'$  is the lgg of the example inputs matching  $\mathbf{p}$ .

First assume that  $X$  does not occur in the rhs of the rule  $r$  with pattern  $\mathbf{p}$ , then replacing  $X$  by  $s$  in  $\mathbf{p}$  does not change the rewrite relation of  $r$  for the example lhss because still all example lhss are subsumed and are rewritten to the same term as before. Now assume that  $X$  occurs in the rhs of  $r$ . Then the rewrite relation of  $r$  for the example lhss remains the same if  $X$  is replaced by  $s$  in  $\mathbf{p}$  as well as in the rhs.  $\square$

#### 5.4 Processing Unfinished Rules

This section describes the three methods for replacing unfinished rules by successor rules. All three methods are applied to a chosen unfinished rule in parallel. The first method, *splitting rules by pattern refinement*, replaces an unfinished rule with pattern  $\mathbf{p}$  by at least two new rules with more specific patterns in order to establish a case distinction. The second method, *introducing subfunctions*, generates new induction problems, i.e., new example equations, for those subterms of an unfinished rhs containing unbound variables. The third method, *introducing function calls*, implements the principle described in Sec. 4 in order to introduce recursive calls or calls to other defined functions. Other defined functions can be further target functions or background knowledge functions. Finding the arguments of such a function call is considered as new induction problem. The last two methods implement the capability to automatically find auxiliary subfunctions. We denote the set of example equations whose inputs match the lhs  $F(\mathbf{p})$  of an unfinished rule by  $E_{F,\mathbf{p}}$ .

**Splitting Rules by Pattern Refinement.** The first method for generating successors of an unfinished rule is to replace its pattern  $\mathbf{p}$  by a set of more specific patterns, such that the new patterns induce a partition of the example inputs in  $E_{F,\mathbf{p}}$ . This results in a *set* of new rules replacing the original rule and—from a programming point of view—establishes a case distinction. It has to be assured that no two of the new patterns unify. This is done as follows: First a position  $u$  is chosen at which a variable stands in  $\mathbf{p}$  and constructors stands in the subsumed example inputs. Since  $\mathbf{p}$  is the lgg of the inputs, at least two inputs have different constructor symbols at position  $u$ . Then respectively all example inputs with the *same* constructor at position  $u$  are taken into the same subset. This leads to a partition of the example equations. Finally, for each subset of equations the lgg is computed leading to a set of initial rules with refined patterns.

Possibly, different positions of variables in pattern  $\mathbf{p}$  lead to different partitions. Then all partitions are generated. Since specialised patterns subsume fewer inputs, the number of unbound variables in the initial rhss (non-strictly) decreases with each refinement step. Eventually, if no correct hypothesis with fewer case distinctions exists, each example input is subsumed by itself such that the example equations are simply reproduced.

*Example 3.* Reconsider the example equations for *Reverse* in Example 2. The pattern of the initial rule is simply a variable  $Q$ , since the example inputs have no common root symbol. Hence, the unique position at which the pattern contains

a variable and the example inputs different constructors is the root position. The first example input consists of only the constant  $[]$  at the root position. All remaining example inputs have the constructor *cons* at that position. I.e., two subsets are induced, one containing the first example equation, the other containing all remaining example equations. The lggs of the example lhss of these two subsets are  $Reverse([])$  and  $Reverse([Q|Qs])$  resp. which are the lhss of the two successor rules.

**Introducing Subfunctions.** The second method to generate successor equations can be applied, if all example equations  $F(\mathbf{i}) = o \in E_{F,\mathbf{p}}$  have the same constructor symbol  $c$  as root of their rhss  $o$ . Let  $c$  be of arity  $m$  then the rhs of the unfinished rule is replaced by the term  $c(S_1(\mathbf{p}), \dots, S_m(\mathbf{p}))$  where each  $S_i$  denotes a new defined (sub)function. This finishes the rule since all variables from the new rhs are contained in the lhs. Examples for the new subfunctions are abduced from the examples of the current function as follows: If the  $o|_j$ ,  $j \in \{1, \dots, m\}$  denote the  $j$ th subterms of the example rhss  $o$ , then the equations  $S_j(\mathbf{i}) = o|_j$  are the example equations of the new subfunction  $S_j$ . Thus, correct rules for  $S_j$  compute the  $j$ th subterms of the rhss  $o$  such that the term  $c(S_1(\mathbf{p}), \dots, S_m(\mathbf{p}))$  normalises to the rhss  $o$ .

*Example 4.* Reconsider the *Reverse* examples except the first one as they have been put into one subset in Example 3. The initial equation for this subset is:

$$Reverse([Q|Qs]) = [Q_2|Qs_2] \quad (3)$$

It is unfinished due to 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  $S_1$  and  $S_2$ :  $S_1([X]) = X$ ,  $S_1([X, Y]) = Y$ ,  $\dots$ ,  $S_2([X]) = []$ ,  $S_2([X, Y]) = [X]$ ,  $\dots$ . The successor equation set for the unfinished equation contains three equations determined as follows: The original unfinished equation (3) is replaced by the finished equation  $Reverse([Q|Qs]) = [S_1([Q|Qs] | S_2[Q|Qs])]$  and from the new example sets for  $S_1$  and  $S_2$  initial equations are derived.

**Introducing Function Calls.** The last method to generate successor sets for an unfinished rule with pattern  $\mathbf{p}$  for a target function  $F$  is to replace its rhs by a call to a defined function  $F'$ , i.e. by a term  $F'(G_1(\mathbf{p}), \dots, G_k(\mathbf{p}))$  (if  $F'$  is of arity  $k$ ). Each  $G_j$ ,  $j \in \{1, \dots, k\}$  denotes a new introduced defined (sub)function. This finishes the rule, since now the rhs does not longer contain variables not contained in the lhs. In order to get a rule leading to a *correct* hypothesis, for each example equation  $F(\mathbf{i}) = o$  of function  $F$  whose input  $\mathbf{i}$  matches  $\mathbf{p}$  with substitution  $\sigma$  must hold:  $F'(G_1(\mathbf{p}), \dots, G_k(\mathbf{p}))\sigma \xrightarrow{!} o$ . This holds if for each example output  $o$  an example equation  $F'(\mathbf{i}') = o'$  of function  $F'$  exists such that  $o = o'\tau$  for a substitution  $\tau$  and  $G_j(\mathbf{p})\sigma \xrightarrow{!} i'_j\tau$  for each  $G_j$  and  $i'_j$ . Thus, if we find such example equations of  $F'$ , then we abduce example equations  $G_j(\mathbf{i}) = i'_j\tau$  for the new subfunctions  $G_j$  and induce them from these examples.

Provided, the final hypothesis is correct for  $F'$  and all  $G_j$  then it is also correct for  $F$ . In order to assure termination of the final hypothesis it must hold  $i' < i$  according to any reduction order  $<$  if the function call is recursive.<sup>3</sup>

*Example 5.* Consider the examples for the help function  $S_2$  from Example 4 and the corresponding unfinished initial equation:

$$S_2([Q|Qs]) = Qs_2 \quad (4)$$

The example outputs,  $[], [X], \dots$  of  $S_2$  match the example outputs for *Reverse*. That is, the unfinished rhs  $Qs_2$  can be replaced by a (recursive) call to the *Reverse*-function. The argument of the call must map the inputs  $[X], [X, Y], \dots$  of  $S_2$  to the corresponding inputs  $[], [X], \dots$  of *Reverse*, i.e., a new help function  $G$  with example equations  $G([X]) = [], G([X, Y]) = [X], \dots$  will be introduced. The successor equation set for the unfinished equation (4) contains the finished equation  $S_2([Q|Qs]) = \text{Reverse}(G([Q|Qs]))$  and the initial equation for  $G$ .

Note that in further steps the example outputs for  $S_1$  from Example 4 would match the example outputs of the predefined *Last*-function (Example 2) and  $G$  from Example 5 becomes the invented *Init*-function of Example 2.

## 6 Experiments

We have implemented a prototype of IGOR2 in the reflective term-rewriting based programming language Maude [16]. The implementation includes an extension compared to the approach described in the previous section. Different variables within a pattern can be tested for equality. This establishes—besides pattern refinement—a second form of case distinction.

In Tab. 1 we have listed experimental results for sample problems. The first column lists the names for the induced target functions, the second the names of additionally specified background functions, the third the number of given examples (for target functions), the fourth the number of automatically introduced recursive subfunctions, the fifth the maximal number of calls to defined functions within one rule, and the sixth the times in seconds consumed by the induction. Note that the example equations contain variables if possible (except for the *Add*-function). The experiments were performed on a Pentium 4 with Linux and the Maude 2.3 interpreter.

All induced programs compute the intended functions with more or less “natural” definitions. *Length*, *Last*, *Reverse*, and *Member* are the well known functions on lists. *Reverse* has been specified twice, first with *Snoc* as background knowledge which inserts an element at the end of a list and second without background knowledge. The second case (see Example 2 for given data and computed solution), is an example for the capability of automatic subfunction introduction and nested calls of defined functions. *Odd* is a predicate and true for odd natural

<sup>3</sup> Assuring decreasing arguments is more complex if mutual recursion is allowed.

**Table 1.** Some inferred functions

target functions	bk funs	#expl	#sub	#calls	times
<i>Length</i>	/	3	0	1	.012
<i>Last</i>	/	3	0	1	.012
<i>Odd</i>	/	4	0	1	.012
<i>ShiftL</i>	/	4	0	1	.024
<i>Reverse</i>	<i>Snoc</i>	4	0	2	.024
<i>Even, Odd</i>	/	3, 3	0	1	.028
<i>Mirror</i>	/	4	0	2	.036
<i>Take</i>	/	6	0	1	.076
<i>ShiftR</i>	/	4	2	2	.092
<i>DelZeros</i>	/	7	0	1	.160
<i>InsertionSort</i>	<i>Insert</i>	5	0	2	.160
<i>PlayTennis</i>	/	14	0	0	.260
<i>Add</i>	/	9	0	1	.264
<i>Member</i>	/	13	0	1	.523
<i>Reverse</i>	/	4	2	3	.790
<i>QuickSort</i>	<i>Append</i> , <i>P<sub>1</sub></i> , <i>P<sub>2</sub></i>	6	0	5	63.271

numbers, false otherwise. The solution contains two base cases (one for 0, one for 1) and in the recursive case, the number is reduced by 2. In the case where both *Even* and *Odd* are specified as target functions, both functions of the solution contain one base case for 0 and a call to the other function reduced by 1 as the recursive case. I.e. the solution contains a mutual recursion. *ShiftL* shifts a list one position to the left and the first element becomes the last element of the result list, *ShiftR* does the opposite, i.e., shifts a list to the right such that the last element becomes the first one. The induced solution for *ShiftL* contains only the *ShiftL*-function itself and simply “bubbles” the first element position by position through the list, whereas the solution for *ShiftR* contains two automatically introduced subfunctions, namely again *Last* and *Init*, and conses the last element to the input list without the last element. *Mirror* mirrors a binary tree. *Take* keeps the first  $n$  elements of a list and “deletes” the remaining elements. This is an example for a function with two parameters where both parameters are reduced within the recursive call. *DelZeros* deletes all zeros from a list of natural numbers. The solution contains two recursive equations. One for the case that the first element is a zero, the second one for all other cases. *InsertionSort* and *QuickSort* respectively are the well known sort algorithms. The five respectively six well chosen examples as well as the additional examples to specify the background functions are the absolute minimum to generate correct solutions. The solution for *InsertionSort* has been generated within a time that is not (much) higher as for the other problems, but when we gave a few more examples, the time to generate a solution explodes. The reason is, that all outputs of lists of the same length are equal such that many possibilities of matching the outputs in order to find recursive calls exist. The number of possible matches increases

exponentially with more examples. The comparatively very high induction time for *QuickSort* results from the many examples needed to specify the background functions and from the complex calling relation between the target function and the background functions.  $P_1$  and  $P_2$  are the functions computing the lists of smaller numbers and greater numbers respectively compared to the first element in the input list. For *Add* we have a similar problem. First of all, we have specified *Add* by ground equations such that more examples were needed as for a non-ground specification. Also for *Add* holds, that there are systematically equal outputs, since, e.g.,  $Add(2, 2)$ ,  $Add(1, 3)$  etc. are equal and due to commutativity. Finally, *PlayTennis* is an attribute vector concept learning example from Mitchell's machine learning text book [17]. The 14 training instances consist of four attributes. The five non-recursive rules learned by our approach are equivalent with the decision tree learned by ID3 which is shown on page 53 in the book. This is an example for the fact, that learning decision trees is a subproblem of inducing functional programs.

To get an idea of which problems cannot be solved consider again *QuickSort* and *InsertionSort*. If we would not have provided *Insert* and *Append* as background functions respectively then the algorithms could not have been induced. The reason is that *Insert* and *Append* occur at the root of the expressions for the recursive cases respectively. But, as mentioned in Sect. 5.1, such functions cannot be invented automatically.

In [18] we compare the performance of IGOR2 with the systems GOLEM and MAGICHASKELLER on a set of sample problems. IGOR2 finds correct solutions for more problems than the other systems and the induction times are better than those needed by MAGICHASKELLER.

## 7 Conclusion

We described a new method, called IGOR2, to induce functional programs represented as confluent and terminating constructor systems. IGOR2 is inspired by classical and recent analytical approaches to the fast induction of functional programs. One goal was to overcome the drawback that "pure" analytical approaches do not facilitate the use of background knowledge and have strongly restricted hypothesis languages and on the other side to keep the analytical approach as far as possible in order to be able to induce more complex functions in a reasonable amount of time. This has been done by applying a search in a more comprehensive hypothesis space but where the successor functions are data-driven and not generate-and-test based, such that the number of successors is more restricted and the hypothesis space is searched in a controlled manner. The search is complete and only favours hypotheses inducing fewer partitions.

Compared to classical "pure" analytical systems, IGOR2 is a substantial extension since the class of inducible programs is much larger. E.g., all of the sample programs from [4, page 448] can be induced by IGOR2 but only a fraction of the problems in Section 6 can be induced by the system described in [4]. Compared to ILP systems capable of inducing recursive functions and recent enumerative

functional methods like GOLEM, FOIL, and MAGICHASKELLER IGOR2 mostly performs better regarding inducibility of programs and/or induction times.

We see several ways to improve IGOR2, partly by methods existing in one or the other current system. Some of the enumerative functional systems apply methods to detect semantical equivalent programs in order to prune the search space. These methods could also be integrated into IGOR2. MAGICHASKELLER is one of the few systems using higher-order functions like *Map*. Many implicitly recursive functions can be expressed without explicit recursion by using such paramorphisms leading to more compact programs which are faster to induce. Currently we further generalise our synthesis theorem in order to analytically find calls of such paramorphisms provided as background knowledge. Finally we look at possible heuristics to make the complete search more efficient.

The IGOR2 implementation as well as sample problem specifications can be obtained at <http://www.cogsys.wiai.uni-bamberg.de/effalip/download.html>.

## References

1. Gold, E.M.: Language identification in the limit. *Information and Control* 10(5), 447–474 (1967)
2. Summers, P.D.: A methodology for LISP program construction from examples. *Journal of the ACM* 24(1), 161–175 (1977)
3. Smith, D.R.: The synthesis of LISP programs from examples: A survey. In: Biermann, A.W., Guiho, G., Kodratoff, Y. (eds.) *Automatic Program Construction Techniques*, pp. 307–324. Macmillan, Basingstoke (1984)
4. Kitzelmann, E., Schmid, U.: Inductive synthesis of functional programs: An explanation based generalization approach. *Journal of Machine Learning Research* 7, 429–454 (2006)
5. Katayama, S.: Systematic search for lambda expressions. In: van Eekelen, M.C.J.D. (ed.) *Revised Selected Papers from the Sixth Symposium on Trends in Functional Programming, TFP 2005*, Intellect, vol. 6, pp. 111–126 (2007)
6. Koopman, P.W.M., Plasmeijer, R.: Systematic synthesis of functions. In: Nilsson, H. (ed.) *Revised Selected Papers from the Seventh Symposium on Trends in Functional Programming, TFP 2006*, Intellect, vol. 7, pp. 35–54 (2007)
7. Olsson, J.R.: Inductive functional programming using incremental program transformation. *Artificial Intelligence* 74(1), 55–83 (1995)
8. Muggleton, S.H., Raedt, L.D.: Inductive logic programming: Theory and methods. *Journal of Logic Programming* 19-20, 629–679 (1994)
9. Muggleton, S.H., Feng, C.: Efficient induction of logic programs. In: *Proceedings of the First Conference on Algorithmic Learning Theory, Tokyo, Japan, Ohmsha*, pp. 368–381 (1990)
10. Quinlan, J.R., Cameron-Jones, R.M.: FOIL: A midterm report. In: Brazdil, P. (ed.) *Proceedings of the 6th European Conference on Machine Learning. LNCS*, pp. 3–20. Springer, London (1993)
11. Bergadano, F., Gunetti, D.: *Inductive Logic Programming: From Machine Learning to Software Engineering*. MIT Press, Cambridge (1995)
12. Flener, P., Yilmaz, S.: Inductive synthesis of recursive logic programs: Achievements and prospects. *The Journal of Logic Programming* 41(2-3), 141–195 (1999)

13. Plotkin, G.D.: A note on inductive generalization. *Machine Intelligence* 5, 153–163 (1970)
14. Lapointe, S., Matwin, S.: Sub-unification: a tool for efficient induction of recursive programs. In: *ML 1992: Proceedings of the Ninth International Workshop on Machine Learning*, pp. 273–281. Morgan Kaufmann Publishers Inc., San Francisco (1992)
15. Terese: *Term Rewriting Systems*. Cambridge Tracts in Theoretical Computer Science, vol. 55. Cambridge University Press, Cambridge (2003)
16. Clavel, M., Durán, F., Eker, S., Lincoln, P., Martí-Oliet, N., Meseguer, J., Talcott, C.: The maude 2.0 system. In: Nieuwenhuis, R. (ed.) *RTA 2003*. LNCS, vol. 2706, pp. 76–87. Springer, Heidelberg (2003)
17. Mitchell, T.M.: *Machine Learning*. McGraw-Hill Higher Education, New York (1997)
18. Hofmann, M., Kitzelmann, E., Schmid, U.: Analysis and evaluation of inductive programming systems in a higher-order framework. In: Dengel, A.R., Berns, K., Breuel, T.M., Bomarius, F., Roth-Berghofer, T.R. (eds.) *KI 2008*. LNCS, vol. 5243, pp. 78–86. Springer, Heidelberg (2008)