Inductive Programming: A Survey of Program Synthesis Techniques

Emanuel Kitzelmann

Cognitive Systems Group, University of Bamberg
emanuel.kitzelmann@uni-bamberg.de

Abstract. Inductive programming (IP)—the use of inductive reasoning methods for programming, algorithm design, and software development—is a currently emerging research field. A major subfield is inductive program synthesis, the (semi-)automatic construction of programs from exemplary behavior. Inductive program synthesis is not a unified research field until today but scattered over several different established research fields such as machine learning, inductive logic programming, genetic programming, and functional programming. This impedes an exchange of theory and techniques and, as a consequence, a progress of inductive programming. In this paper we survey theoretical results and methods of inductive program synthesis that have been developed in different research fields until today.

1 Introduction

Inductive programming (IP) is an emerging field, comprising research on inductive reasoning theory and methods for computer programming, algorithm design, and software development. In this sense, albeit with different accentuation, the term has been used by Partridge [1], by Flener and Partridge [2], within the workshops on "Approaches and Applications of Inductive Programming", and within the ICML'06 tutorial on "Automatic Inductive Programming".

IP has intersections with machine learning, artificial intelligence, programming, software engineering, and algorithms research. Nevertheless, it goes beyond each of these fields in one or the other aspect and therefore is a research field in its own right, intrinsically.

It goes beyond classical machine learning in that the focus lies on learning general programs including loops and recursion, instead of merely (mostly non-recursive) models or classifiers in restricted representational frameworks, such as decision trees or neural networks.

In classical software engineering and algorithm design, a *deductive*—reasoning from the general to the specific—view of software development is predominant. One aspires a general problem description as starting point from which a program or algorithm is developed as a particular solution. Methods based on deductive reasoning exist to partly automatize the programming and verification process—such as automatic code generation from UML diagrams, (deductive) program synthesis to generate algorithmic parts, program transformation and refactoring

U. Schmid, E. Kitzelmann, and R. Plasmeijer (Eds.): AAIP 2009, LNCS 5812, pp. 50–73, 2010.

to optimize programs, and theorem proving, model checking, and static analysis to verify programs. To emphasize this common deductive foundation one might speak of *deductive programming* to subsume established software development methods.

Inductive programming, on the other side, aims at developing methods based on *inductive*—from the specific to the general—reasoning (not to be confused with mathematical or structural induction) to assist in programming, algorithm design, and the development of software. Starting point for IP methods is specific data of a problem—use cases, test cases, desirable (and undesirable) behavior of a software, input/output examples (I/O-examples) of a function or a module interface, computation traces of a program for particular inputs and so forth. Such descriptions of a problem are known to be incomplete. Inductive methods produce a *generalization* of such an incomplete specification by identifying general patterns in the data. The result might be again a—more complete—specification or an actual implementation of a function, a module, or (other parts of) a program.

Inductive reasoning is per se unsound. Inductively obtained conclusions are *hypotheses* and incapable of proof regarding their premises. This is, perhaps, the most severe objection against IP. What is the use of methods whose results cannot be proven correct and possibly deviate from what was intended? However, if the data at hand is representative then it is likely that identified patterns actually hold in the general case and that, indeed, the induced result meets the general problem. On the other side, *all* software development necessarily makes a transition from a first *informal* and often incomplete problem description by the user or customer to a complete and ideally formal specification. This transition is (i) also incapable of formal proof and (ii) possibly based on—non-systematic, inexplicit—generalization. Also, IP should not be understood as a replacement for deductive methods but as an addition. IP may be used in different ways: to generate candidate solutions subject to further inspection, in combination with deductive methods to tackle a problem from the general description *as well as* from concrete (counter-)instances, to systematize occurring generalizations, or to check the representativeness of example cases provided by the user. Some problems, especially many problems in the field of artificial intelligence, elude a complete specification at all, e.g., face recognition. This factum is known as *knowledge-acquisition bottleneck*. Overall, there is no reason why systematically incorporating existing or easily formulated data by inductive methods should not improve efficiency and even validity of software development.

One important aspect of IP is the inductive synthesis of actual, executable *programs* including recursion or loops. Except to professional software development, possible application fields of the (semi-)automatic induction of programs from exemplary behavior are end-user programming and learning of recursive policies [3] in intelligent agents. Research on *inductive program synthesis (IPS)* started in the seventies. However, it has, since then, always been only a niche in several different research fields and communities such as artificial intelligence, machine learning, inductive logic programming (ILP), genetic programming, and

functional programming. Until today, there is no uniform body of theory and methods. This fragmentation over different communities impedes exchange of results and may lead to redundancies. The problem is all the more profound as only few people and groups at all are working on IPS worldwide.

This paper surveys theoretical results and IPS methods that have been developed in different research fields until today. We grouped the work into three blocks: First the classical, analytic data-driven induction of LISP programs as invented by Summers [4] and its generalizations (Section 3), second ILP (Section 4), and third several generate-and-test based approaches to the induction of functional programs (Section 5). In Section 6 we state some conclusions and ideas of further research. As general preliminaries, we informally introduce some common IPS concepts in the following section.

This survey is quite comprehensive, yet not complete and covers functional generate-and-test methods less detailed than the other two areas. This is due to limited space in combination with the author's areas of expertise and shall not be interpreted as a measure of quality. We hope that it will be a useful resource for all people interested in IP.

2 Basic Inductive Programming Concepts

IPS aims at constructing a computer program or algorithm from a *(known-to-be-)incomplete specification* of a function to be implemented, called *target function*. Incomplete means, that the target function is not specified on its whole domain but only on (small) parts of it. Typically, an incomplete specification consists of a subset of the graph of the function: *input/output examples (I/O-examples)*. Variables may be allowed in I/O-examples and also more expressive formalisms have been used to specify the target function.

An induced program contains function *primitives*, predefined functions known to the IPS system. Primitives may be fixed within the IPS system or dynamically be given as an extra, problem-specific, input. Dynamically provided primitives are called *background knowledge*.

Example 1. Suppose the following I/O-examples on lists (whatever the list elements $A, x, y, z, 1, 2, 3, 5$ stand for; constants, variables, or compound objects), are provided: $(A) \mapsto ()$, $(x, y, z) \mapsto (x, y)$, $(3, 5, 2, 1) \mapsto (3, 5, 2)$. Given the common list constructors/destructors `nil`, `cons`, `head`, `tail`, the predicate `empty` to test for the empty list, and the `if-then-else`-conditional as primitives, an IPS system might return the following implementation of the *Init*-function returning the input list without its last element:

```
F(x) = if empty(tail(x)) then nil
       else cons(head(x), F(tail(x))) .
```

Given a particular set of primitives, some target function may not be representable by only one recursive function definition such that a non-specified recursive *subfunction* needs to be introduced; this is called *(necessary) predicate invention* in ILP.

IPS is commonly regarded as a *search problem*. In general, the problem space consists of the representable programs as nodes and instances of the operators of the IPS system to transform one program into another as arcs. Due to *underspecification* in IP, typically infinitely many (semantically) different programs meet the specification. Hence, one needs criteria to choose between them. Such criteria are called *inductive bias* [5]. Two kinds of inductive bias exist: If an IPS system can only generate a certain proper subset of all (computable) functions of some domain, either because its language is restricted or because its operators are not able to reach each program, this constitutes a *restriction bias*. The order in which the problem space is explored and hence the ordering of solutions is the *preference bias*; it can be modelled as probability distribution over the program space.

3 The Analytical Functional Approach

A first systematic attempt to IPS was made by Summers [4]. He noticed that under particular restrictions regarding allowed primitives, program schema, and choice of I/O-examples, a recursive Lisp program can be computed from I/O-examples without search in program space. His insights originated some further research.

3.1 Summers' Pioneering Work

Summers' approach to induce recursive Lisp functions from I/O-examples includes two steps: First, a so-called *program fragment*, an expression of one variable and the allowed primitives, is derived for each I/O-pair such that applied to the input, evaluates to the specified output. Furthermore, predicates are derived to distinguish between example inputs. Integrated into a McCarthy conditional, these predicate/fragment pairs build a non-recursive program computing the I/O-examples and is considered as a first approximation to the target function. In a second step, recurrent relations between predicates and fragments each are identified and a recursive program generalizing them is derived.

Example inputs and outputs are *S-expressions*, the fundamental data structure of the Lisp language [6]. We define the set of *subexpressions* of an S-expression to consist of the S-expression itself and, if it is non-atomic, of all subexpressions of both its components.

The programs constructed by Summers' technique use the Lisp primitives *cons*, *car*, *cdr*, *nil*, *atom*, and T, the last denoting the truth value *true*. Particularly, no other predicates than *atom* and T (e.g., *eq* for testing equality of S-expressions), and no atoms except for *nil* are used. This choice of primitives is not arbitrary but crucial for Summers' methodology of deriving programs from examples without search. The McCarthy conditional and recursion are used as control structure. Allowing *atom* and T as only predicates and *nil* as only

atom in outputs means that the atoms in the I/O-examples, except for *nil*, are actually considered as *variables*. Renaming them does not change the meaning. This implies that any semantic information must be expressed by the *structure* of the S-expression.

1. Step: Initial Non-recursive Approximation. Given a set of k I/O-examples, $\{\langle i_1, o_1\rangle, \ldots, \langle i_k, o_k\rangle\}$, a program fragment $f_j(x), j \in \{1, \ldots, k\}$, composed of *cons*, *car*, and *cdr* is derived for each I/O-pair, which evaluates to the output when applied to the input: $f_j(i_j) = o_j$.

S-expressions are uniquely constructed by *cons* and destructed by *car* and *cdr*. We call *car*-*cdr* compositions *basic functions* (cp. [7]). Together with the following two conditions, this allows for determining unique program fragments. (i) Each atom may occur only once in each input. (ii) Each atom, except for *nil*, occurring in an output must also occur in the corresponding input. Due to the first condition, each subexpression occurs exactly once in an S-expression such that subexpressions are denoted by unique basic functions.

Deriving a program fragment works as follows: All subexpressions of an input, together with their unique basic functions, are enumerated. Then the output is rewritten by composing the basic functions from the input subexpressions with *cons* and *nil*.

Example 2. Consider the I/O-pair $((a\,.\,b)\,.\,(c\,.\,d)) \mapsto ((d\,.\,c)\,.\,(a\,.\,b))$. The input contains the following subexpressions, paired with the corresponding unique basic functions:

$$\langle((a\,.\,b)\,.\,(c\,.\,d)), I\rangle\,, \quad \langle(a\,.\,b), car\rangle\,, \quad \langle(c\,.\,d), cdr\rangle\,,$$
$$\langle a, caar\rangle\,, \quad \langle b, cdar\rangle\,, \quad \langle c, cadr\rangle\,, \quad \langle d, cddr\rangle\,.$$

Since the example output is neither a subexpression of the input nor *nil*, the program fragment becomes a *cons* of the fragments for the *car*- and the *cdr*-component, respectively, of the output. The *car*-part, $(d\,.\,c)$, again becomes a *cons*, namely of the basic functions for d: *cddr*, and c: *cadr*. The *cdr*-part, $(a\,.\,b)$, *is* a subexpression of the input, its basic function is *car*. With variable x denoting the input, the fragment for this I/O-example is thus:

$$cons(cons(cddr(x), cadr(x)), car(x))$$

Next, predicates $p_j(x), j = 1, \ldots, k$ must be determined. In order to get the correct program fragment f_j be evaluated for each input i_j, all predicates $p_{j'}(i_j), 1 \leq j' < j$ (positioned before p_j in the conditional) must evaluate to *false* and $p_j(i_j)$ to *true*. Predicates fulfilling this condition exist if the example inputs form a chain.

We do not describe the algorithm here. Both algorithms, for computing fragments and predicates, can be found in [7]. Figure 1 shows an example for the first step.

I/O-examples:

$$\begin{aligned}(a) &\mapsto nil,\\ (a,b) &\mapsto (a),\\ (a,b,c) &\mapsto (a,b),\\ (a,b,c,d) &\mapsto (a,b,c),\\ (a,b,c,d,e) &\mapsto (a,b,c,d)\,.\end{aligned}$$

First approximation:

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

Fig. 1. I/O-examples and the corresponding first approximation

2. Step: Recurrence Relations. The basic idea in Summers' generalization method is this: The fragments are assumed to be the actual computations carried out by a *recursive* program for the intended function. Hence fragments of greater inputs must comprise fragments of lesser inputs as subterms, with a suitable substitution of the variable x and in a recurrent form along the set of fragments. The same holds analogously for the predicates. Summers calls this relation between fragments and predicates *differences*.

As a preliminary for the following, we need to define the concept of a *context*. A *(one-hole) context* $C[\,]$ is a term including exactly one occurrence of the distinguished symbol $\Box$. $C[s]$ denotes the result of replacing the $\Box$ by the (sub)term s in $C[\,]$.

Definition 1. *A* difference *exists between two terms (fragments or predicates)* t, t' *iff* $t' = C[t\sigma]$ *for some context* $C[\,]$ *and substitution* σ*.*

If we have $k+1$ I/O-examples, we only consider the first k fragment/predicate pairs because the last predicate is always 'T', such that no sensible difference can be derived for it.

Example 3. The following differences, written as recurrence relations ($2 \leq i \leq 3$), can be identified in the first $k = 4$ fragments/predicates of the program of Figure 1.

$$\begin{aligned}p_1(x) &= atom(cdr(x)) & f_1(x) &= nil\\ p_2(x) &= atom(cddr(x)) & f_2(x) &= cons(car(x), nil)\\ p_{i+1}(x) &= p_i(cdr(x)) & f_{i+1}(x) &= cons(car(x), f_i(cdr(x)))\end{aligned}$$

In the general case, we have (for k fragments/predicates):

$$\begin{aligned}
&j-1 \text{ "constant" fragments (as derived from the examples):} \quad f_1, \ldots, f_{j-1},\\
&\text{further } n \text{ constant base cases:} \quad f_j, \ldots, f_{j+n-1},\\
&\text{finally, remaining } k-(j+n-1) \text{ cases recurring to} \qquad (1)\\
&\text{previous cases:} \quad f_{i+n} = C[f_i\sigma_1] \quad \text{for } i = j, \ldots, k-n\,;\\
&\text{analogously for predicates:} \quad p_1,\ \ldots,\ p_{j-1},\ p_j,\ \ldots,\ p_{j+n-1},\ p_{i+n} = p_i(\sigma_2)\ .
\end{aligned}$$

Index j denotes the first predicate/fragment pair which recurs in some following predicate/fragment pair (the first base case). The precedent $j-1$ predicate/fragment pairs do not recur. n is the interval of the recurrence. For Example 3 we have $j = 2$ and $n = 1$.

Inductive Inference. If $k - j \geq 2n$ then we *inductively infer* that the recurrence relations hold for all $i \geq j$.

In Example 3 we have $k - j = 2 \geq 2 = 2n$ and hence induce that the relations hold for all $i \geq 2$.

The generalized recurrence relations may be used to compute new approximations of the assumed target function. The mth *approximating function*, $m \geq j$, is defined as

$$F_m(x) = (p_1(x) \to f_1(x), \ldots, p_m(x) \to f_m(x), T \to \omega)$$

where the p_i, f_i with $j < i \leq m$ are defined in terms of the generalized recurrence relations and where ω means *undefined*. Consider the following *complete partial order* over partial functions, which is well known from denotational semantics:

$$F(x) \leq_F G(x) \text{ iff } F(x) = G(x) \text{ for all } x \in Dom(F)\,.$$

Regarding this order, the set of approximating functions builds a chain. The assumed target function $\mathbf{F}$ is defined as the supremum of this chain.

Now the hypothesized target function is defined, in terms of recurrence relations. In his synthesis theorem and its corollaries, Summers shows how a function defined this way can be expressed by a recursive program.[1]

Theorem 1 ([4]). *If* $\mathbf{F}$ *is defined in terms of recurrence relations as in* (1) *for* $j \leq i \in \mathbb{N}$ *then the following recursive program is identical to* $\mathbf{F}$*:*

$$\begin{aligned}
F(x) &= (p_1(x) \to f_1(x), \ldots, p_{j-1}(x) \to f_{j-1}(x),\\
&\qquad T \to G(x))\\
G(x) &= (p_j(x) \to f_j(x), \ldots, p_{j+n-1}(x) \to f_{j+n-1}(x),\\
&\qquad T \to C[G(\sigma(x))])\,.
\end{aligned}$$

[1] This works, in a sense, reverse to interpreting a recursively expressed function by the partial function given as the fixpoint of the functional of the recursive definition. In the latter case we have a recursive program and want to have the particular partial function computed by it—here we have a partial function and want to have a recursive program computing it.

Example 4. The recurrence relations from Example 3 with $i \geq 2$ define the function **F** to be the *Init*-function. According to the synthesis theorem, the resulting program is:

$$\begin{aligned} F(x) &= (atom(cdr(x)) \rightarrow nil, T \rightarrow G(x)) \\ G(x) &= (atom(cddr(x)) \rightarrow cons(car(x), nil), \\ &\qquad T \rightarrow cons(car(x), G(cdr(x)))) \,. \end{aligned}$$

Introducing Additional Variables. It may happen that no recurrent differences can be found between a chain of fragments and/or predicates. In this case, the fragments/predicates may be generalized by replacing some common subterm by an additional variable. In the generalized fragment/predicate chain recurrent differences possibly exist.

3.2 Early Variants and Extensions

Two early extensions are described. A broader survey of these and other early extensions can be found in [7].

BMWk—Extended Forms of Recurrences. In Summers' approach, the condition for deriving a recursive function from detected differences is that the differences hold—starting from an initial index j and for a particular interval n—recurrently along fragments and predicates with a constant context $C[\,]$ and a constant substitution σ for x. The BMWk[2] algorithm [8] generalizes these conditions by allowing for contexts and substitutions that are different in each difference. Then a found sequence of differences originates a sequence of contexts and substitutions each. Both sequences are considered as fragments of new *subfunctions*. The BMWk algorithm is then recursively applied to these new fragment sequences, hence features the automatic introduction of (necessary) subfunctions.

Furthermore, Summers' ad-hoc method to introduce additional variables is systematized by computing *least general generalization (lgg)* [9] of successive fragments.

Biermann et al—Pruning Enumerative Search Based on Recurrences within Single Traces. Summers objective was to avoid search and to justify the synthesis by an explicit inductive inference step and a subsequent proven-to-be-correct program construction step. This could be achieved by a restricted program schema and the requirement of a well chosen set of I/O-examples.

On the contrary, Biermann's approach [10] is to employ traces (fragments) to speed up an exhaustive enumeration of a well-defined program class, the so-called *regular* Lisp *programs*. Biermann's objectives regarding the synthesis were

1. *convergence* to the class of regular Lisp programs,
2. convergence on the basis of *minimal input information*,
3. robust behavior on different inputs.

[2] This abbreviates *Boyer-Moore-Wegbreit-Kodratoff*.

Particularly 2 and 3 are contradictory to the recurrence detection method—by 2 Biermann means that no synthesis method exists which is able to synthesize every regular Lisp program from fewer examples and by 2 he means that examples may be chosen randomly.

3.3 From Lisp to Term Rewriting Systems

At the beginning of Section 3.1 we stated the Lisp primitives as used in programs induced by Summers' method (as well as by BMWk and Biermann's method). This selection is crucial for the first step, the deterministic construction of first approximations, yet not for the generalization step. Indeed, the latter is independent from particular primitives, it rather relies on matching (sub)terms over arbitrary first-order signatures. Two recent systems inspired by Summers' recurrence detection method use *term rewriting systems* over first-order signatures to represent programs. Special types of TRSs can be regarded as (idealized) functional programs.

A *term rewriting system (TRS)* is a set of directed equations or *(rewrite) rules*. A rule is a pair of first-order terms $\langle l, r\rangle$, written $l \to r$. The term l is called *left-hand side (lhs)*, r is called *right-hand side (rhs)* of the rule.

We get an *instance* of a rule by applying a substitution σ to it: $l\sigma \to r\sigma$. The instantiated lhs $l\sigma$ is called *redex* (*red*ucible *ex*pression). *Contracting* a redex means replacing it by its rhs. A *rewrite step* consists of contracting a redex within an arbitrary context: $C[l\sigma] \to C[r\sigma]$. The *one-step rewrite relation* $\to$ of a rule is defined by the set of its rewrite steps. The *one-step rewrite relation* $\to_R$ of a TRS R is the union of the one-step rewrite relations of its single rules. The *rewrite relation* of a TRS R, $\overset{*}{\to}_R$, is the reflexive transitive closure of $\to_R$.

Igor1—Inducing Recursive Program Schemes. The system Igor1 [11] induces *recursive program schemes (RPSs)*. An RPS is a special form of TRS: The signature is divided into two disjoint subsets $\mathcal{F}$ and $\mathcal{G}$, called *unknown* and *basic* functions, respectively; rules have the form $F(x_1, \ldots, x_n) \to t$ where $F \in \mathcal{F}$ and the x_i are variables, and there is exactly one rule for each $F \in \mathcal{F}$.

Igor1's program schema is more general than Summers' in that recursive subfunctions are found automatically with the restriction that (recursive) calls of defined functions may not be nested in the rhss of the equations. Furthermore, additional parameters are introduced systematically.

(Mutually) recursive RPSs do not terminate. Their standard interpretation is the infinite term defined as the limit $\lim_{n\to\infty, F(\boldsymbol{x}) \overset{n}{\to} t} t$ where F denotes the main rule of the RPS. One gets finite approximations by replacing infinite subterms by the special symbol Ω, meaning *undefined*. Certainly, such an infinite tree and its approximations contain recurrent patterns because they are generated by *repeatedly* replacing instances of lhss of the rules by instances of rhss. Igor1 takes a finite approximation of some (hypothetical) infinite tree as input, discovers the recurrent patterns in it, and builds, based on these recurrences, an RPS R such that the input is a finite approximation of the infinite tree of R.

Example 5. For a simple example without subfunctions (the *Init* function again), consider the finite approximation of some unknown infinite term:

$$\begin{array}{l} \textbf{\textit{if}}(atom(cdr(\ \ x\ \ )), nil, \\ \quad cons(car(\ \ x\ \ ), \\ \quad\quad \textbf{\textit{if}}(atom(cdr(\ \ cdr(x)\ \ )), nil, \\ \quad\quad\quad cons(car(\ \ cdr(x)\ \ ), \\ \quad\quad\quad\quad \textbf{\textit{if}}(atom(cdr(\ \ cdr(cdr(x))\ \ )), nil, \\ \quad\quad\quad\quad\quad cons(car(\ \ cdr(cdr(x))\ \ ), \\ \quad\quad\quad\quad\quad\quad \Omega)))))) . \end{array}$$

At the path from the root to Ω, where the latter denotes the unknown infinite subterm of the infinite target term and hence, which has been generated by an unknown recursive RPS, we find a recurring sequence of *if*-*cons* pairs. This leads to the hypothesis that a replacement of the lhs of a recursive rule by its rhs has taken place at the *if*-positions. The term is divided at these positions leading to three segments (assume, the break-positions are replaced by Ω). An approximation of the assumed rhs is computed as the lgg of the segments: $if(atom(cdr(x)), nil, cons(car(x), \Omega))$.

The Ω denotes the still unknown recursive call. The non-equal parts of the segments, which are replaced by the variable x in the lgg, are highlighted by extra horizontal space in the term. These parts must have been generated by the substitution $\{x \leftarrow cdr(x)\}$ in the recursive call. Denoting the induced function by F, it is now correctly defined as

$$F(x) \rightarrow if(atom(cdr(x)), nil, cons(car(x), F(cdr(x)))) .$$

Different methods to construct a finite approximation as first synthesis step have been proposed. In [11], an extension of Summers' first step is described. Examples need not be linearly ordered and nested `if-then-else`-conditionals are used instead of the McCarthy conditional. In [3], *universal planning* is proposed as first step.

3.4 Igor2—Combining Search and Analytical Techniques

All methods based on Summers' seminal work described so far suffer from strong restrictions regarding their general program schemas, the commitment to a small fixed set of primitives, and, at least the early methods, to the requirement of linearly ordered I/O-examples.

The system IGOR2 [12] aims to overcome these restrictions, but not at the price of falling back to generate-and-test search (cp. Section 5). IGOR2 conducts a search in program space, but the transformation operators are data-driven and use techniques such as matching and least generalizations, similar to the methods described so far. In contrast to generate-and-test search, only programs being

correct with respect to the I/O-examples in a particular sense (but possibly unfinished) are generated. This narrows the search tree and makes testing of generated programs unnecessary.

Programs (as well as I/O-examples and background knowledge) are represented as *constructor (term rewriting) systems (CSs)*. CSs can be regarded as an extension of RPSs: The function sets $\mathcal{F}$ and $\mathcal{G}$ are called *defined functions* and *constructors*, respectively. The arguments of a defined function symbol in a lhs need not be variables but may be terms composed of constructors and variables and there may be several rules for one defined function. This extension corresponds to the concept of *pattern matching* in functional programming. One consequence of the CS representation is that I/O-examples themselves already constitute "programs", CSs. Hence, rewriting outputs into fragments to get a first approximation (Section 3.1) is not necessary anymore.

IGOR2 is able to construct complex recursive CSs containing several base- and (mutually) recursive rules, automatically identified and introduced recursive subfunctions, and complex compositions of function calls. Several interdependent functions can be induced in one run. In addition to I/O-examples, background knowledge may be provided.

3.5 Discussion

Summers' important insights were first, how the algebraic properties of data-structures can be exploited to construct program fragments and predicates without search and second, that fragments (and predicates) for different I/O-pairs belonging to one recursively defined function share recurrent patterns that can be used to identify the recursive definition. Obviously, it is necessary for recurrence detection that I/O-examples are not randomly chosen but that they consist of the first $k \in \mathbb{N}$ examples regarding the underlying order on S-expressions, i.e., that they are *complete* up to some level.

If the general schema of inducible functions becomes more complex, e.g., if subfunctions can be found automatically, and/or if background knowledge is allowed, then search is needed. IGOR2 shows that Summers' ideas for generalization can be integrated into search operators.

Search is also needed if the goal is to induce programs based on minimal sets of randomly chosen examples. In this case, the recurrence detection method cannot be applied. Biermann's method shows that it is possible for particular program classes to use fragments as generated in Summers' first step to constrain an exhaustive search in program space.

4 Inductive Logic Programming

Inductive Logic Programming (ILP) [13,14] is a branch of machine learning [5]—intensional concept descriptions are learned from (counter-)examples, called *positive and negative examples*. The specificity of ILP is its basis in computational logic: First-order clausal logic is used as uniform language for hypotheses,

examples, and background knowledge, semantics of ILP is based on entailment, and inductive learning techniques are derived by inverting deduction.

Horn clause logic together with resolution constitutes the (Turing-complete) programming language PROLOG. Program synthesis is therefore principally within the scope of ILP and has been regarded as one application field of ILP [13]. One of the first ILP systems, MIS [15], is an automatic programming/debugging system. Today, ILP is concerned with (relational) data-mining and knowledge discovery and program synthesis does not play a role anymore.

4.1 Preliminaries

An *atom* is a predicate symbol applied to arguments, a literal is an atom or negated atom. A *clause* is a (possible empty) disjunction of literals, a *Horn clause* is a clause with at most one positive literal, a *definite clause* is a clause with exactly one positive literal. A *definite program* is a finite set of definite clauses. A definite clause C consisting of the positive literal A and the negative literals $\neg B_1, \ldots, \neg B_n$ is equivalent to $B_1 \wedge \ldots \wedge B_n \rightarrow A$, written $A \leftarrow B_1, \ldots, B_n$.

4.2 Overview

In the definite setting, hypotheses and background knowledge are definite programs, examples are ground atoms. The following two definitions state the ILP problem with respect to the so-called *normal semantics*.[3]

Definition 2. *Let Π be a definite program and E^+, E^- be positive and negative examples. Π is*

complete *with respect to E^+ iff $\Pi \models E^+$,*
consistent *with respect to E^- iff $\Pi \not\models e$ for every $e \in E^-$,*
correct *with respect to E^+ and E^- iff it is complete with respect to E^+ and consistent with respect to E^-.*

Definition 3. *Given*

- *a set of possible hypotheses (definite programs) $\mathcal{H}$,*
- *positive and negative examples E^+, E^-,*
- *consistent background knowledge B (i.e., $B \not\models e$ for every $e \in E^-$) such that $B \not\models E^+$,*

find a hypothesis $H \in \mathcal{H}$ such that $H \cup B$ is correct with respect to E^+ and E^-.

Entailment ($\models$) is undecidable in general and for Horn clauses, definite programs, and between definite programs and single atoms in particular. Thus, in practice, different decidable (and preferably also efficiently computable) relations, which

[3] There is also a *non-monotonic* setting in ILP where hypotheses need not entail positive examples but only state true properties. This is useful for *data mining* or *knowledge discovery* but not for program synthesis, so we do not consider it here.

are sound but more or less incomplete, are used. We say that a hypothesis *covers* an example if it can be proven true from the background knowledge and the hypothesis. That is, a hypothesis is regarded correct if it, together with the background knowledge, covers all positive and no negative examples. Two commonly used notions are:

Extensional coverage. Given a clause $C = A \leftarrow B_1, \ldots, B_n$, a finite set of ground atoms B as background knowledge, positive examples E^+, and an example e, C *extensionally covers* e iff there exists a substitution θ such that $A\theta = e$ and $\{B_1, \ldots, B_n\}\theta \subseteq B \cup E^+$.

Intensional coverage. Given a hypothesis H, background knowledge B, and an example e, $H \cup B$ *intensionally covers* e iff e can be proven true from $H \cup B$ by applying some terminating theorem proving technique, e.g., depth-bounded SLD-resolution.

Example 6. As an example for extensional coverage, suppose $B = \emptyset$ and $E^+ = \{\ Init([c],[\,]), Init([b,c],[b]), Init([a,b,c],[a,b])\ \}$. The recursive clause $Init([X|Xs],[X|Ys]) \leftarrow Init[Xs, Ys]$ extensionally covers the positive example $Init([b,c],[b])$ with $\theta = \{X \leftarrow b, Xs \leftarrow [c], Ys \leftarrow [\,]\}$.

Both extensional and intensional coverage are sound. Extensional coverage is more efficient but less complete. As an example for the latter, suppose the positive example $Init([c],[\,])$ is missing in E^+ in Example 6. Then the stated recursive clause together with the base clause $Init([X],[\,])$ still intensionally covers $e = Init([b,c],[b])$ yet the recursive clause *does not* extensionally cover e anymore. Obviously, extensional coverage requires that examples (and background knowledge) are complete up to some complexity (cp Section 3.5). Another problem with extensional coverage is that if two clauses each do not cover a negative example, both together possibly do.

Extensional and intensional coverage are closely related to the general ILP algorithm (Algorithm 1) and the covering algorithm 2 as well as to the generality models θ-subsumption and entailment as described below (Section 4.3), respectively.

ILP is considered as a search problem. Typically, the search operators to compute new candidate programs are based on the dual notions of *generalization* and *specialization* of programs or clauses.

Definition 4. *A program Π is* more general than *a program Φ iff $\Pi \models \Phi$. Φ is said to be* more specific than *Π.*

This structure of the program space provides a way for pruning. If a program is not consistent then all generalizations are also not consistent and therefore need not be considered. This dually holds for non-completeness and specializations. Algorithm 1 shows a generic ILP algorithm. Most ILP systems are instances of it.

A common instance is the *covering algorithm* (Algorithm 2). The individual clauses of a program are generated independently one after the other. Hence, the problem space is not the program space (*sets* of clauses) but the clause space (*single* clauses). This leads to a more efficient search.

Algorithm 1. A generic ILP algorithm.

Input: B, E^+, E^-
Output: A definite program H such that $H \cup B$ is correct with respect to E^+ and E^-
Start with some initial (possibly empty) hypothesis H
repeat
  if *$H \cup B$ is not consistent* **then** specialize H
  if *$H \cup B$ is not complete* **then** generalize H
until *$H \cup B$ is correct with respect to E^+ and E^-*
return H

Algorithm 2. The covering (typically interpreted extensionally) algorithm.

Input and **Output** as in Algorithm 1
Start with the empty hypothesis $H = \emptyset$
repeat
  Add a clause C not covering any $e \in E^-$ to H
  Remove all $e \in E^+$ covered by C from E^+
until $E^+ = \emptyset$
return H

Entailment ($\models$) as well as θ-subsumption (Section 4.3) are *quasi-orders* on sets of definite programs and clauses, respectively. We associate "more general" with "greater". The operators carrying out specialization and generalization are called *refinement operators*. They map clauses to sets of (refined) clauses or programs to sets of (refined) programs. Most ILP systems explore the problem space mainly in one direction, either from general to specific (*top-down*) or the other way round (*bottom-up*). The three well-known systems FOIL [16] (top-down), GOLEM [17] (bottom-up), and PROGOL [18] (mixed) are instantiations of the covering algorithm.

Example 7. For an example of the covering algorithm, let B and E^+ be as in Example 6 and E^- all remaining instantiations for the "inputs" $[c], [b, c], [a, b, c]$, e.g., $Init([b, c], [c])$. Let us assume that a (base-)clause $Init([X], [\,])$ is already inferred and added and hence, the covered example $Init([c], [\,])$ is deleted from E^+. Assume, our instantiation of the covering algorithm is a top-down algorithm. This means, each clause is found by starting with a (too) general clause and successively specializing it until no negative examples are covered anymore. Let us start with the clause $Init([X|Xs], Ys) \leftarrow$. It covers all remaining positive but also all corresponding negative examples; it is too general. Applying the substitution $\{Ys \leftarrow [X|Ys]\}$ specializes it to $Init([X|Xs], [X|Ys]) \leftarrow$. This excludes some negative examples (e.g., $Init([b, c], [c])$). Adding the literal $Init(Xs, Ys)$ to the body again specializes the clause to $Init([X|Xs], [X|Ys]) \leftarrow Init(Xs, Ys)$. All

remaining positive examples are still covered but no negative example is covered anymore. Hence, the clause is added and the algorithm returns the two inferred clauses as solution.

Both specializations were refinements under θ-subsumption (Section 4.3, "Refinement Operators").

4.3 Generality Models and Refinement Operators

Instead of entailment ($\models$), θ-subsumption is often used in ILP as generality model. It is incomplete with respect to $\models$ but decidable, simple to implement, and efficiently computable. If we have background knowledge B, then we are not simply interested in whether a clause C is more general than a clause D but in whether C together with B is more general than D (together with B). This is captured by the notions of *relative* (to background knowledge) entailment respectively θ-subsumption.

Refinement under (Relative) θ-subsumption

Definition 5. *Let C and D be clauses and B a set of clauses.*

C θ-subsumes D, written $C \succeq D$, iff there exists a substitution θ such that $C\theta \subseteq D$.

C θ-subsumes D relative to B, written $C \succeq_B D$, if $B \models C\theta \rightarrow D$ for a substitution θ.

A Horn clause language quasi-ordered by θ-subsumption with an additional bottom element is a lattice. This does not generally hold for relative subsumption. Least upper bounds are called *least general generalizations (lgg)* [9]. Lggs and greatest lower bounds are computable and hence may be used for generalization and specialization, though they do not properly fit into our general notion of refinement operators because they neither map single clauses to sets of clauses nor single programs to sets of programs.

A useful restriction is to let background knowledge be a finite set of ground literals. In this case, lggs exist under subsumption relative to B and can be reduced to (non-relative) lggs. The bottom-up system GOLEM uses this scenario.

In general, (relative) θ-subsumption is sound but not complete. If $C \succeq D$ ($C \succeq_B D$) then $C \models D$ ($C \cup B \models D$) but not vice versa. For a counter-example of completeness let $C = P(f(X)) \leftarrow P(X)$ and $D = P(f(f(X))) \leftarrow P(X)$ then $C \models D$[4] but $C \not\succeq D$. As the example indicates, the incompleteness is due to *recursive* rules and therefore especially critical for *program synthesis*.

Refinement Operators. A specialization operator refines a clause by

- applying a substitution for a single variable or
- adding one most general literal.

A generalization operator uses inverse operations.

Application of these operators is quite common in ILP, e.g., in the systems MIS, FOIL, GOLEM, and PROGOL.

[4] D is simply the result of self-resolving C.

Refinement under (Relative) Entailment. Due to the incompleteness of θ-subsumption regarding recursive clauses, refinement under (relative) entailment has been studied. *Relative* entailment is defined as follows:

Definition 6. *Let C and D be clauses and B a finite set of clauses. Then C* entails *D relative to B, denoted $C \models_B D$, if $\{C\} \cup B \models D$.*

Neither lggs nor greatest specializations exist in general for Horn clause languages ordered by (relative) entailment.

Refinement Operators. Roughly speaking, entailment is equivalent to resolution plus θ-subsumption. This leads to specialization operators under (relative) entailment. Objects of refinement under entailment are not single clauses but *sets* of clauses, i.e., programs. A specialization operator under entailment refines a definite program by

- Adding a resolvent of two clauses or
- adding the result of applying the θ-subsumption specialization operator to a clause or
- deleting a clause.

4.4 Automatic Programming Systems

The three general-purpose systems FOIL, GOLEM, and PROGOL are successful in learning non-recursive concepts from large data sets, yet have problems to learn recursive programs: Due to their use of the covering approach (extensional coverage), they need complete example sets and background knowledge to induce recursive programs. Since they (at least FOIL and GOLEM) explore (i) only the θ-subsumption lattice of clauses and (ii) do this greedily, correct clauses may be passed. Finally, their objective functions in the search for clauses is to cover as many as possible positive examples. Yet base clauses typically cover only few examples such that these systems often fail to induce correct base cases.

Hence ILP systems especially designed to learn *recursive* programs have been developed. They address different issues: Handling of random examples, predicate invention, usage of general programming knowledge, and usage of problem-dependent knowledge of the user, which goes beyond examples. A comprehensive survey of automatic programming ILP systems can be found in [19].

Inverting entailment by structural analysis. Several systems—CRUSTACEAN [20], CLAM [21], TIM [22], MRI [23]—address the issue of inducing *recursive* programs from *random* examples by inverting entailment based on structural analysis, similar to Section 3, instead of searching in the θ-subsumption lattice. These systems also have similar restrictions regarding the general schema of learnable programs. However, some of them can use background knowledge; MRI can find more than one recursive clause.

Top-down induction of recursive programs. Top-down systems can principally—even if they explore the θ-subsumption clause-lattice only—generate arbitrary (in particular all recursive) Horn clauses.[5] Thus, if a top-down covering system would use intensional instead of extensional coverage, it could principally induce recursive programs from *random* examples. Certainly, this would require to find clauses in a particular order—base clauses first, then recursive clauses, only depending on base clauses and themselves, then recursive clauses, only depending on base clauses, the previously generated recursive clauses, and themselves, and so on. This excludes programs with mutually interdepending clauses. The system SMART [24] is based on these ideas. It induces programs consisting of one base clause and one recursive clause. Several techniques to sensibly prune the search space allows for a more exhaustive search than the greedy search applied by FOIL, such that the incompleteness issue of θ-subsumption-based search is weaken.

The system FILP [25] is a covering top-down system that induces *functional* predicates only, i.e., predicates with distinguished input- and output parameters, such that for each binding of the input parameters exactly one binding of the output parameters exists. This makes negative examples unnecessary. FILP can induce multiple interdependent predicates/functions where each may consist of several base- and recursive clauses. Hence, intensional coverage is not assured to work. FILP starts with a few randomly chosen examples and tries to use intensional covering as far as possible. If, during the intensional proof of some example, an instance of the input parameters of some predicate appears for which an output is neither given by an example nor can be derived intensionally, then FILP queries for this "missing" example and thereby completes the example set as far as needed.

Using programming knowledge. Flener argued, in several papers, for the use of program schemas that capture general program design knowledge like divide-and-conquer, generate-and-test, global-search etc., and has implemented this in several systems. He distinguishes between schema-*based* systems inducing programs of a system-inherent schema only and schema-*guided* systems, which take schemas as dynamic, problem-dependent, additional input and thus are more flexible. Flener's DIALOGS [26] system uses schemas and strong queries to restrict the search space and thereby is able to efficiently induce comparatively complex programs including predicate invention.

Jorge and Brazdil have—besides for *clause structure grammars* defining a program class and thus similar to schemas as dynamic language-bias—argued for so called *algorithm sketches*. An algorithm sketch is problem-dependent algorithm knowledge about the target function and provided by the user in addition to examples. This idea is implemented in their SKIL and SKILIT systems [27].

[5] Hence, although θ-subsumption is incomplete with respect to entailment due to recursive clauses, every clause, in particular the recursive clauses, can be generated by refinement based on θ-subsumption—if one searches top-down starting from the empty clause or some other clause general enough to θ-subsume the desired clauses.

4.5 Discussion

Compared to the classical approaches in Section3 (except for IGOR2), ILP has broadened the class of inducible relations by allowing for background knowledge, using particular search methods and other techniques (Section 4.4).

Shapiro [15] and Muggleton and De Raedt [13] argued for clausal logic as universal language in favor to other universal formalisms such as Turing machines or LISP. Their arguments are: (i) Syntax and semantics are closely and in a natural way related. Hence if a logic program makes errors, it is possible to identify the erroneous clause. Furthermore, there are simple and efficient operations to manipulate a logic program with predictable semantic effects (cp. Section 4.3). Both is not possible for, say, Turing machines. (ii) It suffices to focus on the logic of the program, control is left to the interpreter. In particular, logic programs (and clauses) are *sets* of clauses (and literals), order does not matter.

The first argument carries over to other declarative formalisms such as equational logic, term rewriting, and functional logic programming (FLIP [28] is an IPS system in this formalism). The second argument also carries over to some extent, declarative programming all in all shifts the focus off control and to logic. Yet in this generality it only holds for non-recursive programs or ideal, non-practical, interpreters. For the efficient interpretation of recursive programs however, order of clauses in a program and order of literals in a clause matters. Hence we think that declarative, (clausal- and/or equational-)logic-based formalisms are principally equally well suited for IPS.

Logic programs represent general relations. (Partial) functions are special relations—their domains are distinguished into source and target (or: a functional relation has input and output parameters) and they are single-valued (each instantiation of the input parameters implies a unique instantiation of the output parameters). Regarding functional- and logic programming, there is another difference: Functional programs are typically typed, i.e., their domain is partitioned and inputs and outputs of each function must belong to specified subsets, whereas logic programs are typically untyped. Interestingly, all three "restrictions" of functions compared to relations have been shown to be advantageous from a learnable point of view in ILP. The general reason is that they restrict the problem space such that search becomes more efficient and fewer examples are needed to describe the intended function. In particular, no negative examples are needed since they are implicitly given by the positive ones.

ILP is built around the natural generality structure of the problem space. Regarding *functional* relations, we observe an "oddity" of this structure. For definite programs, "more general", with respect to the minimal Herbrand model, means "more atoms". If the relation is a *function*, an additional ground atom must have a different instantiation of the input parameters compared to all other included atoms. Thus, "more general" in the case of definite programs representing functions reduces to "greater domain". In other words: *All functions with the same domain are incomparable with respect to generality.* Since most often one is interested in total functions, generality actually provides *no structure at all* of the space of possible solutions.

5 Functional Generate-and-Test Approaches

The functional IPS methods in this third block have in common that their search is generate-and-test based. I/O-examples are not used as a means to *construct* programs but only to *test* generated programs.

5.1 Genetic Programming

Genetic programming (GP) [29], like other forms of *evolutionary algorithms* is inspired by biological evolution. GP systems maintain *populations* of candidate solutions, get new ones by stochastical methods like *reproduction*, *mutation*, *recombination/crossover*, and *selection*, and thereby try to maximize *fitness*. Evolutionary search can be useful when the problem space is too broad to conduct an exhaustive search and simultaneously nothing or few is known about the *fitness landscape*, i.e., when it is not possible to construct sensible heuristics. The randomness of the search cares for a widespread exploration of the problem space which is guided by the fitness measure. On the other side, this "chaotic" search in a space with unknown properties makes it difficult to give any guaranties regarding solutions and leads to only approximated solutions. A GP problem is specified by *fitness cases* (e.g., example inputs of the target function), a fitness function, and primitives to be used in evolved expressions. There are no predefined goal criteria or preference biases in GP systems. The search is completely guided by the fitness function that is to be maximized.

Data structures and recursion do not play a predominant role in GP. A typical evolved program is an arithmetic expression or a propositional formula. Koza and his colleagues [30] integrated recursion into GP. One of the major issues is the handling of non-terminating programs. As a generate-and-test approach, GP relies on testing evolved candidate programs against the given examples. If non-termination may appear then a runtime limit is applied. This raises two problems if non-terminating programs are frequently generated: (i) The difficulty of assigning a fitness value to an aborted program and (ii) the runtime uselessly consumed by evaluating non-terminating programs. Wong and Mun [31] deal with this problem by a meta-learning approach to decrease the possibility of evolving non-terminating programs.

Others try to avoid non-termination completely: In her system PolyGP [32], Yu integrates *implicit recursion* through the use of user-provided higher-order functions. Kahrs [33] evolves *primitive recursive* functions over the natural numbers. Binard and Felty [34] evolve programs in System F, a typed lambda calculus where only total recursive functions are expressible. The primitive recursive functions are contained as proper subclass.

Hamel and Shen [35] have developed a method lying in the intersection of ILP, GP and algebraic specification. They evolve (recursive) algebraic specifications, i.e., equational theories over many-sorted signatures, using GP search methods. Instead of providing a fitness function, a target theory is, as in ILP, specified by positive and negative facts—ground equations in this case. Additionally, a background theory may be provided. The fitness function to be maximized is

derived from such a specification. Candidate theories satisfying more positive facts, excluding more negative facts, and being of smaller syntactical complexity are preferred.

5.2 ADATE

The ADATE system [36], to our knowledge the most powerful inductive programming system regarding inducible programs, is an evolutionary system in that it maintains a population of programs and performs a greedy search guided by a fitness function. Yet unlike GP, it is especially designed to evolve *recursive* programs and applies sophisticated program transformation operators, search strategy, and program evaluation functions to this end.

Programs are represented in ADATE-ML, a subset of Standard ML. Programs are rated according to a user-provided output evaluation function, user provided preference biases, and syntactical and computational complexity.

5.3 Systematic Enumeration of Programs

Two further recent methods, MagicHaskeller [37] and the software testing system G∀ST [38] essentially systematically enumerate programs of a certain class.

MagicHaskeller uses higher-order functions as background knowledge. Katayama argues that by using higher-order functions, programs can be represented in a compact form and by using strong typing, the problem space is narrowed such that a simple brute-force enumeration of programs could make sense. He furthermore considers MagicHaskeller as a base-line which could be used to evaluate the performance of more sophisticated methods. As a first result, Katayama compares MagicHaskeller and PolyGP for the problems *Nth*, *Length*, and *Map*, and states that PolyGP, in contrast to MagicHaskeller, needs different higher-order functions for each of these problems, needs several runs to find a solution, needs additional parameters to be set, and, nevertheless, consumes more time to induce a solution.

5.4 Discussion

One general advantage of generate-and-test methods is their greater flexibility, in at least to aspects: First regarding the problem space—there are no *principle* difficulties in enumerating even very complex programs. Second regarding the form of the incomplete specification. Whereas the search operators of an analytical technique depend on the specification (e.g., I/O-examples) such that different forms of specification need different search operator techniques, the search is more independent from the specification in generate-and-test methods such that more expressive forms of specification can easily be integrated. In particular, fitness functions in GP or the objective function in ADATE are more expressive than I/O-examples since no fixed outputs need to be provided but general *properties* to be satisfied by computed outputs can be specified.

The disadvantage of generate-and-test methods is that they generally generate far more candidate programs until a solution is found and hence need much more time than data-driven methods to induce programs of equal size. Several analytical and generate-and-test systems have been compared empirically in [39]. A further problem is non-termination. As generated programs need to be tested against the provided examples, non-termination is a serious issue. Higher-order functions or formalisms that a-priori only include total functions are helpful to circumvent this problem.

6 Conclusions and Further Research

In the previous sections, we described several approaches and systems to the inductive synthesis of functional and logic programs and discussed pros and cons and relations between them.

One obvious dimension to classify them is the way of how example data is used: As basis to construct candidate solutions (Section 3) or to test and evaluate independently generated candidates (Section 5). (In ILP, both approaches are found.) The analytical approach tends to be faster because many representable programs are a priori excluded from being generated. On the other side, since it strongly depends on the data and the language bias, it is much less robust and flexible regarding the whole problem specification including types of data, preference-, and language biases. Besides further developing both general approaches separately, we think that examining ways to combine them could be useful to achieve a satisfiable combination of robustness, flexibility, expressiveness, and efficiency. Our system IGOR2 and the well-known ILP system PROGOL indicate the potential of such an integration.

One important topic, that certainly has not received sufficient attention in the context of inductive program synthesis, is learning theory, including models of learning and criteria to evaluate candidate programs. PAC-learning, the predominant learning model in machine learning, is well-suited for restricted representation languages and noisy data, hence approximate solutions. Yet in program synthesis, we have rich representation languages, often assume error-free examples, and want have programs that *exactly* compute an intended function or relation. Moreover, efficiency, not only of the induction process, but of the induced program, becomes an important issue. Muggleton's *U-learning* model[6] captures these needs and is probably a good model or initial point to develop learning models for inductive program synthesis.

There has certainly been significant progress since the beginnings in the seventies. Yet inductive program synthesis still is not yet in a status to be applied to real problems. We think that it is now time for a more target-oriented approach. This does not mean to replacing general approaches by problem-dependent ad hoc techniques. We rather think that identifying and promoting specific application fields and domains could help to spark broader interest to the topic as well

[6] The 'U' stands for 'universal'.

as to sensibly identify strengths and weaknesses of existing methods, to extend them and to identify possibilities to integrate them in a useful way.

In the context of software engineering, we think that *test-driven development (TDD)* would be a good starting point to bring IPS to application. The paradigm requires preparing tests "(incompletely) defining" a function *before* coding it. Hence, IPS could smoothly fit in here. Moreover, TDD typically features a strong modularization such that only small entities need to be synthesized.

Within algorithms research, one could try to find (classes) of problems for which "better" than currently known algorithms are expected to exist and to apply IPS methods to them.

References

1. Partridge, D.: The case for inductive programming. Computer 30(1), 36–41 (1997)
2. Flener, P., Partridge, D.: Inductive programming. Automated Software Engineering 8(2), 131–137 (2001)
3. Schmid, U.: Inductive Synthesis of Functional Programs: Universal Planning, Folding of Finite Programs, and Schema Abstraction by Analogical Reasoning. LNCS (LNAI), vol. 2654. Springer, Heidelberg (2003)
4. Summers, P.D.: A methodology for LISP program construction from examples. Journal of the ACM 24(1), 161–175 (1977)
5. Mitchell, T.M.: Machine Learning. McGraw-Hill, New York (1997)
6. McCarthy, J.: Recursive functions of symbolic expressions and their computation by machine, part i. Communications of the ACM 3(4), 184–195 (1960)
7. Smith, D.R.: The synthesis of LISP programs from examples: A survey. In: Biermann, A., Guiho, G., Kodratoff, Y. (eds.) Automatic Program Construction Techniques, pp. 307–324. Macmillan, Basingstoke (1984)
8. Jouannaud, J.P., Kodratoff, Y.: Program synthesis from examples of behavior. In: Biermann, A.W., Guiho, G. (eds.) Computer Program Synthesis Methodologies, pp. 213–250. D. Reidel Publ. Co. (1983)
9. Plotkin, G.D.: A note on inductive generalization. Machine Intelligence 5, 153–163 (1970)
10. Biermann, A.W.: The inference of regular LISP programs from examples. IEEE Transactions on Systems, Man and Cybernetics 8(8), 585–600 (1978)
11. Kitzelmann, E., Schmid, U.: Inductive synthesis of functional programs: An explanation based generalization approach. Journal of Machine Learning Research 7, 429–454 (2006)
12. Kitzelmann, E.: Analytical inductive functional programming. In: Hanus, M. (ed.) Logic-Based Program Synthesis and Transformation. LNCS, vol. 5438, pp. 87–102. Springer, Heidelberg (2009)
13. Muggleton, S.H., De Raedt, L.: Inductive logic programming: Theory and methods. Journal of Logic Programming 19, 20, 629–679 (1994)
14. Nienhuys-Cheng, S.-H., de Wolf, R.: Foundations of Inductive Logic Programming. LNCS (LNAI), vol. 1228. Springer, Heidelberg (1997)
15. Shapiro, E.Y.: Algorithmic Program Debugging. MIT Press, Cambridge (1983)
16. Quinlan, J.R., Cameron-Jones, R.M.: FOIL: A midterm report. In: Brazdil, P.B. (ed.) ECML 1993. LNCS, vol. 667, pp. 3–20. Springer, Heidelberg (1993)

17. Muggleton, S.H., Feng, C.: Efficient induction of logic programs. In: Proceedings of the First Conference on Algorithmic Learning Theory, Ohmsha, pp. 368–381 (1990)
18. Muggleton, S.H.: Inverse entailment and progol. New Generation Computing 13, 245–286 (1995)
19. Flener, P., Yilmaz, S.: Inductive synthesis of recursive logic programs: Achievements and prospects. The Journal of Logic Programming 41(2-3), 141–195 (1999)
20. Aha, D.W., Lapointe, S., Ling, C.X., Matwin, S.: Inverting implication with small training sets. In: Bergadano, F., De Raedt, L. (eds.) ECML 1994. LNCS, vol. 784, pp. 29–48. Springer, Heidelberg (1994)
21. Rios, R., Matwin, S.: Efficient induction of recursive prolog definitions. In: McCalla, G.I. (ed.) Canadian AI 1996. LNCS, vol. 1081, pp. 240–248. Springer, Heidelberg (1996)
22. Idestam-Almquist, P.: Efficient induction of recursive definitions by structural analysis of saturations. In: Advances in Inductive Logic Programming. IOS Press, Amsterdam (1996)
23. Furusawa, M., Inuzuka, N., Seki, H., Itoh, H.: Induction of logic programs with more than one recursive clause by analyzing saturations. In: Džeroski, S., Lavrač, N. (eds.) ILP 1997. LNCS, vol. 1297, pp. 165–172. Springer, Heidelberg (1997)
24. Mofizur, C.R., Numao, M.: Top-down induction of recursive programs from small number of sparse examples. In: Advances in Inductive Logic Programming. IOS Press, Amsterdam (1996)
25. Bergadano, F., Gunetti, D.: Inductive Logic Programming: From Machine Learning to Software Engineering. MIT Press, Cambridge (1995)
26. Flener, P.: Inductive logic program synthesis with DIALOGS. In: ILP 1996. LNCS, vol. 1314, pp. 175–198. Springer, Heidelberg (1997)
27. Jorge, A.M.G.: Iterative Induction of Logic Programs. PhD thesis, Departamento de Ciência de Computadores, Universidade do Porto (1998)
28. Ferri-Ramírez, C., Hernández-Orallo, J., Ramírez-Quintana, M.: Incremental learning of functional logic programs. In: Kuchen, H., Ueda, K. (eds.) FLOPS 2001. LNCS, vol. 2024, pp. 233–247. Springer, Heidelberg (2001)
29. Koza, J.R.: Genetic Programming: On the Programming of Computers by Means of Natural Selection. MIT Press, Cambridge (1992)
30. Koza, J.R., Andre, D., Bennett, F.H., Keane, M.A.: Genetic Programming III: Darwinian Invention & Problem Solving. Morgan Kaufmann, San Francisco (1999)
31. Wong, M., Mun, T.: Evolving recursive programs by using adaptive grammar based genetic programming. Genetic Programming and Evolvable Machines 6(4), 421–455 (2005)
32. Yu, T.: Hierarchical processing for evolving recursive and modular programs using higher-order functions and lambda abstraction. Genetic Programming and Evolvable Machines 2(4), 345–380 (2001)
33. Kahrs, S.: Genetic programming with primitive recursion. In: Proceedings of the 8th annual Conference on Genetic and Evolutionary Computation (GECCO 2006), pp. 941–942. ACM, New York (2006)
34. Binard, F., Felty, A.: Genetic programming with polymorphic types and higher-order functions. In: Proceedings of the 10th annual Conference on Genetic and Evolutionary Computation (GECCO 2008), pp. 1187–1194. ACM Press, New York (2008)
35. Hamel, L., Shen, C.: An inductive programming approach to algebraic specification. In: Proceedings of the 2nd Workshop on Approaches and Applications of Inductive Programming (AAIP 2007), pp. 3–14 (2007)

36. Olsson, J.R.: Inductive functional programming using incremental program transformation. Artificial Intelligence 74(1), 55–83 (1995)
37. 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, vol. 6, pp. 111–126. Intellect (2007)
38. Koopman, P., Alimarine, A., Tretmans, J., Plasmeijer, R.: GAST: Generic automated software testing. In: Peña, R., Arts, T. (eds.) IFL 2002. LNCS, vol. 2670. Springer, Heidelberg (2003)
39. Hofmann, M., Kitzelmann, E., Schmid, U.: A unifying framework for analysis and evaluation of inductive programming systems. In: Proceedings of the Second Conference on Artificial General Intelligence, Atlantis, pp. 55–60 (2009)