ILP and Automatic Programming: Towards Three Approaches

Pierre Flener, Bilkent University, Ankara, Turkey [*]
Luboš Popelínský, Masaryk University, Brno, Czech Republic [†]
Olga Štěpánková, CTU Prague, Czech Republic [‡]

Abstract

The prospects of inductive logic programming (ILP) with respect to automatic programming (program synthesis) are discussed. We argue that logic program synthesis from incomplete information is but a niche of ILP, and study consequences of this statement. Then, three approaches are described: schema-driven synthesis of logic programs from incomplete specifications, the role of transformation techniques in ILP, and interactive assumption-based inductive learning.

1 Introduction

Automatic programming (or: program synthesis) [6, 1] is one of the lines of research directed towards the design and implementation of software meeting given requirements. The adjective "automatic" does not necessarily imply a fully automatic synthesis of programs. Any degree of automation is acceptable, as we would like to engage computers in the process of programming. The current research may be split into at least two streams, unfortunately with too little overlap: deductive synthesis from (assumed-to-be) complete specifications (by axioms) and inductive synthesis from (known-to-be) incomplete specifications (e.g. by examples). In this work, we are addressing the latter stream. However, we realize that a cooperation between those streams is necessary to solve real problems of software engineering, and that this would be advantageous for both of them. For more information on the need for cross-fertilization and cooperation between deductive and inductive synthesis, see [17].

At present, more attention is paid to the employment of ILP techniques in automatic programming [2, 4, 20, 8]. The goal of the paper is to show what makes inductive synthesis of logic programs specific, how it differs from concept learning as well as to analyze consequences of this difference. It implies the necessity to focus on exploitation of programmer's knowledge (like schemata), on program transformation techniques, and on human–computer interaction.
□

The organisation of the paper is following. In the Section 2., we analyze the use of inductive reasoning in logic program synthesis. The relation between ILP and logic program synthesis is discussed. The problems with inductive synthesis are shown and a solution is proposed. In the Section 3., a framework of a schema-driven synthesis of logic programs from incomplete specifications is given and results reached by *SYNAPSE* synthesizer are discussed. The schema-driven synthesis seems to be overcomming the most of existing ILP synthesizers by its capability to learn more complex predicates, even if some of background knowledge predicates are unknown. Section 4. is focused on a role of tranformation techniques in a process of program construction. Existence of powerfull transformation techniques could allow us to focus on a synthesis of simple programs, which can be later transformed into more efficient ones. Section 5. starts with a role of human–computer interaction. We elaborate a new approach to model-based learning – assumption-based interactive learning – which tries to find a minimal completion of an incomplete learning set leading to a correct program synthesis. In the Section 6., *WiM*, an assumption-based learning system is introduced. Results reached with *WiM* in learning simple list processing predicates are described and compared with those of *FILP* [4], CRUSTACEAN [2] and BMWk [23].

In the following section, we analyze the use of inductive reasoning in logic program synthesis and

[*]Department of Computer Engineering and Information Science, Faculty of Engineering, Bilkent University, TR-06533 Bilkent, Ankara, Turkey; pf@bilkent.edu.tr
[†]Faculty of Informatics, Masaryk University, Burešova 20, CZ–602 00 Brno, CZ; popel@fi.muni.cz
[‡]Faculty of Electrical Engineering, CTU, Technická 2, CZ-166 27 Praha 6, CZ; step@lab.felk.cvut.cz

a relation between concept learning and program synthesis. Under the term **teacher** we understand a human specifying a program not by explicit programming, and the term **learner** designates a program learning a logic program from such information provided by a teacher.

2 On the Use of Inductive Reasoning in Program Synthesis

ILP may be divided [11] into Empirical ILP (heuristic-based learning of a single concept from many examples) and Interactive ILP (algorithmic and oracle-based learning of many concepts from a few examples). An important distinction needs to be done here. Program synthesis from examples is but a niche (albeit a significant one) of ILP. Deductive approaches to (logic) program synthesis from complete specifications are traditionally only interested in synthesizing programs that actually perform some "computations", that is they incrementally "compute" some output(s) from some input(s) through looping behavior such as iteration or recursion. This includes divide-and-conquer programs, and global/local search programs [30]. Other program classes are straight-line programs (with no loops at all), which are virtually identical to their complete specifications and thus not subject to deductive synthesis, and generate-and-test programs. Although they might perform "computations" (in the above sense) during their individual generate and/or test phases, they don't perform overall "computations" and are thus generally considered as starting points (specifications) to deductive approaches to program synthesis/transformation, whose goal it is then precisely to distil a divide-and-conquer or global/local search program therefrom.

Regarding inductive approaches to (logic) program synthesis from incomplete specifications now, the same objectives (should) hold: divide-and-conquer and global/local search programs are the challenging program classes we (should) aim at. Straight-line programs solve "mere" data-classification problems and can be learned by exploring pre-enumerated search spaces, which is traditionally rather the realm of empirical machine learning. Generate-and-test programs should not really be a target class, unless there is a hint by the specifier for their synthesis. Such a hint could be some meta-knowledge giving e.g. the upper limit for the considered complexity of the concepts used in the generate and test phases of the program. Otherwise, generators and testers must be either invented ad hoc from the incomplete specifications (examples, ...), which seems even more complicated than synthesis of looping programs because the generators and testers are unrelated to each other, or re-used from a knowledge base. The considered knowledge base must be really large in order to cope with all problems (if the knowledge base is kept small in order to bias the learner/synthesizer, then there is no point in automated learning/synthesis, because the teacher/specifier already knows the target description/program).

In the sequel, by "programs" we mean recursive concept descriptions, and by "identification procedures" we mean non-recursive concept descriptions. Other differences between ILP in general and inductive program synthesis in particular are summarized in the following table:

	ILP	Inductive Program Synthesis
Class of hypotheses	any	recursive programs
Specifying agent	human or machine	human
Intended concept	sometimes unknown	always known
Consistency of examples	any attitude	assumed consistent
Number of examples	any	a few
Number of predicates in examples	at least 1	exactly 1
Rules of inductive inference	selective and constructive	necessarily constructive
Correctness of hypotheses	any attitude	total correctness is crucial
Existence of hypothesis schemas	hardly any	yes, many
Number of correct hypotheses	usually only a few	always many

Table 1: Inductive Program Synthesis as a Niche of Inductive Logic Programming

The central column of the table shows the spectrum of situations covered by ILP research, but it doesn't mean to imply that all learners do cover, or should cover, this full spectrum. The right-hand column however shows the most realistic situation for inductive program synthesis, that is a situation that should be covered by every synthesizer. Let's have a look now at these two columns.

In ILP, the agent who provides the examples can be either a human being or some automated device (such as a robot, a satellite, a catheter, ...). It is possible for this agent not to know the intended concept, which means that it may give examples that are not consistent with the intended concept, or that it may give wrong answers to queries from the learner. Examples can be given in any amounts: Empirical ILP expects numerous examples, while Interactive ILP expects only a few examples and often constructs its own examples so as to submit them to the teacher. Examples may involve more than one predicate-symbol: the instance "Tweety" of the concept "canary" could yield the example

$$mouth(tweety, beak) \wedge legs(tweety, 2) \wedge skin(tweety, feathers)$$
$$\wedge\, utterance(tweety, sings) \wedge color(tweety, yellow) \wedge \ldots$$

which involves many predicate-symbols, but not a $canary/1$ predicate-symbol. The used rules of inductive inference can be either selective (only the predicate-symbols of the premise may appear in the conclusion) or constructive (the conclusion "invents" new predicate-symbols). Selective rules are often sufficient to learn concepts, such as "canary", from multi-predicate examples. There are many learning situations where an approximately correct concept description is sufficient, whereas in other situations a totally correct description is hoped for. Schemata (template concept descriptions) are a well-known means of syntactic bias to reduce the size of learning search spaces. However, if one doesn't somehow know in advance whether the concept to be learned is "computational" (it has a program as a concept description) or not (it doesn't have a program as a concept description), then it is hard to use/find an adequate schema. Moreover, non-"computational" concepts tend to have either application-specific schemata (and then arises the question as to their acquisition) or general schemata such as

$$P(X, Y) \Leftarrow Generate(X, Y) \wedge Test(Y)$$

which spell out the entire search space of logic programs and thus don't reduce its size. For general concepts, there are usually only a few correct hypotheses: for instance, there is probably only one correct definition of the "canary" concept, in any given context.

But in inductive program synthesis, the most realistic setting is where the specifier is a human being who knows the intended concept and who is assumed to provide only examples that are consistent with that intended concept. "Knowing a concept" means that one can act as a decision procedure for answering membership queries for that concept [3], but it doesn't necessarily imply his/her ability to actually write that decision procedure. Such a specifier cannot be expected to be willing to give more than just a few examples. Examples only involve one predicate-symbol, namely the one for which a program is to be synthesized: for instance, an example of an integer-list sorting program could be $sort([2, 1, 3], [1, 2, 3])$. The used rules of inductive inference thus necessarily include constructive rules, as programs usually use other programs than just themselves. Total correctness of the synthesized program w.r.t. the intended concept is crucial in inductive synthesis. Programs are highly structured, complex entities that are usually designed according to some strategy (such as Divide-And-Conquer, Global/Local Search, etc): program synthesis can thus be effectively guided by a program schema that reflects some design strategy. The existence of many such schemata and the existence of many choice-points within these strategies entail the existence of many correct programs for a given "computational" concept. For instance, integer-list sorting can be implemented by Insertion-Sort, Merge-Sort, Quicksort programs, and many more.

So there is a dream of actually synthesizing programs from specifications by examples. Since many intentions are covered by an infinity of examples, finite specifications by examples cannot faithfully formalize such intentions, and the synthesizer needs to extrapolate the full intentions from the examples. This is necessarily done by unsound (or rather: not-guaranteed-to-be-sound) reasoning, such as induction or abduction.

2.1 Approaches to Inductive Synthesis

In the early 1970s, some researchers investigated how to synthesize programs from traces of sample executions thereof. However, traces are very procedural specifications, and constructing a trace means knowing the program, which rather defeats the purpose of synthesis. Regarding inductive synthesis from examples, there are basically two approaches [15] [1]:

1. *Trace-based Synthesis:* positive examples are first "explained" by means of traces (that fit some predetermined program schema), and a program is then obtained by generalizing these traces, using the above-mentioned techniques of inductive synthesis from traces. Sample works are surveyed by D.R. Smith [28]. This research was a precursor to the EBL/EBG research of Machine Learning.
2. *Model-based Synthesis:* a logic program is "debugged" w.r.t. positive and negative examples until its least Herbrand model coincides with the intentions. This is the ILP approach. Sample works are those of E.Y. Shapiro [27], and many others are compiled by Muggleton [25].

Historically speaking, both approaches barely overlap in time: trace-based synthesis research took place in the mid and late 1970s, whereas model-based synthesis research is ongoing ever since the early 1980s. Indeed, in the late 1970s, trace-based synthesis research hit a wall and partly declared defeat considering that the found techniques didn't seem to scale up to realistic problems. But then, E.Y. Shapiro [27] and others published their first experiments with model-based approaches, and model-based synthesis took over, not only for inductive program synthesis, but for inductive concept learning in general.

"Linguistically" speaking, both approaches also barely overlap: trace-based synthesis was pursued by the Functional Programming community, whereas model-based learning is being investigated by the Logic Programming community. Revivals of trace-based synthesis in the Logic Programming community have been suggested by Hagiya [18] and Flener [15, 16].

2.2 The Problems with Inductive Synthesis

It is illusory to hope that very general learning techniques carry over without major efficiency problems to particular tasks such as inductive program synthesis: since synthesis is akin to compilation, this illusion amounts to looking for a universal programming language.

Some good ideas of trace-based synthesis (such as schema-guidance) haven't received much attention by model-based learning research. Indeed, as seen above, for general concepts, there are hardly any schemas that wouldn't spell out the entire search space. Now, for the particular task of model-based synthesis, there **is** room for schemas: program schemas significantly reduce the search space, they bring "discipline" into an otherwise possibly anarchic debugging process, and they convey part of the program design knowledge.

There is a fundamental difference between a teacher/learner relationship and a specifier/synthesizer relationship. A teacher usually is expected to know **how** to compute/identify the concept s/he is teaching to the learner, whereas a specifier usually only knows **what** the concept is about, the determination of **how** to compute it being precisely the task of the synthesizer. So a teacher can guide a learner who is "on the wrong track", but a specifier usually can't. A teacher can, right before the learning session, set the learner "on the right track" by providing carefully chosen examples and/or background knowledge, but a specifier often can't. For instance, most ILP systems can learn the Quicksort program from examples of *sort*/2 plus logic procedures for *partition*/3 and *append*/3 as background knowledge. But this amounts to a "specification of *quicksort*/2", which is a valid objective for a teacher, but not for a specifier: one specifies *sort*/2, a problem, not *quicksort*/2, a solution! We really wonder about the efficiency of model-based learners in a true specifier/synthesizer setting, where **a lot of** relevant **and** irrelevant background knowledge is provided. A solution to the ensuing inefficiency would be structured background knowledge, such as classifying the *partition*/3 procedure as a useful instance of the induction-parameter-decomposition placeholder of a Divide-and-Conquer program schema.

The second problem with inductive synthesis is that examples alone are too weak a specification approach. When examples are generated by some automated device, one can hardly expect more information, but human example generators should be able to provide more information. This is crucial because of the abundance of negative learnability results from examples alone. As conveyed by Table 1, in program synthesis, we usually have the setting of a human specifier who knows the intended relation. So s/he probably knows quite a bit more about that relation, but can't express it by examples alone. For instance, it is unrealistic that somebody would want a sorting program (for integer lists, say) and not know the reason why $[2, 1]$ is sorted into $[1, 2]$ rather than into $[2, 1]$. The reason is of course that $1 < 2$, but the problem here is that the $<$ /2 predicate-symbol cannot be used in the examples. More generally, the problem is about the lack of provision of domain knowledge to the synthesizer and has been perceived a while ago. Various proposed solutions are type declarations for the parameters [27], type assertions about the intended relation [14], properties of the intended relation [15, 16], integrity constraints about a set of intended relations [11], and bias (all knowledge potentially useful for narrowing the search space), as generally used in ILP. Assertions/properties/integrity constraints/... should only be an additional source of potentially incomplete information about the intended concept. Otherwise, that is if this information is known/required to be complete, a deduction-based synthesizer would be more appropriate, as the examples could then be safely ignored. This is a problem with some of the proposed solutions [12]. Of course, if someone wants to give complete knowledge about the intentions, then the synthesizer should be able to handle it.

Inductive synthesis researchers are fully aware of the limitations of their research. They view it as just the provision of components and tools for software engineering environments. In the synthesizer-as-a-workbench-of-powerful-mini-synthesizers approach advocated by A.W. Biermann [5] and by schema-guided synthesis researchers such as D.R. Smith [29, 30], there is a place for inductive synthesizers, because certain classes of programs can be reliably synthesized with little effort from a few examples. □

Now we present an overview of the *SYNAPSE* synthesizer (for a full description see [15, 16]) , which stepwisely synthesizes divide-and-conquer logic programs from examples and properties.

3 Stepwise, Schema-guided Synthesis of Logic Programs: *SYNAPSE*

In [15, 16], a strategy for stepwise synthesis of logic programs from specifications by examples and properties is presented. It is part of an attempt at an automation of the logic programming methodology of Deville [13]. The synthesis process is guided by a divide-and-conquer schema, features non-incremental presentation of examples, and is interactive. The system is both inductive and deductive: it starts with inductive reasoning from the examples, and then performs deductive reasoning from the properties whenever appropriate.

Examples are ground facts. Only positive examples are used. Properties are definite clauses. In practice, it turns out that providing such properties is quite straightforward. It must be stressed here that the synthesis mechanism was designed to work even (or rather: especially) in the absence of recursive properties: recursion discovery and introduction is a major challenge in synthesis!

Let $compress(L, C)$ hold iff compact-list C is the compression of list L. This can be illustrated by the following example:

$$compress([a, a, b, b, a, c, c, c], [a, 2, b, 2, a, 1, c, 3])$$

We use this predicate for a description of the synthesis mechanism. The eight ground examples:

$$\begin{array}{l}
compress([], []) \\
compress([a], [a, 1]) \\
compress([b, b], [b, 2]) \\
compress([c, d], [c, 1, d, 1]) \\
compress([e, e, e], [e, 3]) \\
compress([f, f, g], [f, 2, g, 1]) \\
compress([h, i, i], [h, 1, i, 2]) \\
compress([j, k, l], [j, 1, k, 1, l, 1])
\end{array}$$

and the three properties:

$$\begin{array}{l}
true \Rightarrow compress([X], [X, 1]) \\
X = Y \Rightarrow compress([X, Y], [X, 2]) \\
X \neq Y \Rightarrow compress([X, Y], [X, 1, Y, 1])
\end{array}$$

are provided by the specifier.

Schemata are template programs with fixed control flows (e.g. divide-and-conquer, generate-and-test, global/local search, etc.). A possible divide-and-conquer schema for a binary predicate is as follows:

$$\begin{array}{llll}
R(X, Y) \Leftrightarrow & Minimal(X) & \wedge Solve(X, Y) \\
& \vee & \\
& \vee (1 \leq k \leq c) & NonMinimal(X) \\
& & \wedge Decompose(X, HX, TX) \\
& & \wedge Discriminate_k(HX, TX, Y) \\
& & \wedge R(TX, TY) \\
& & \wedge Process_k(HX, HY) \\
& & \wedge Compose_k(HY, TY, Y)
\end{array}$$

If X is minimal, e.g. an empty list or the integer zero, then usually Y is very easy to find. Otherwise, X is decomposed into a series HX of heads of X and a series TX of tails of X. Tails are of the same type as X, but smaller than X, e.g. a shorter list. The tails TX recursively yield tails TY of Y, by $R(TX, TY)$. The heads HY of Y are processed from the HX, and Y is composed from its heads HY and tails TY. If there is more than one way of composing Y, than discriminants are needed.

The synthesis consists of two phases of a total of eight steps that follow this divide-and-conquer schema.

- Expansion phase
  1. Syntactic creation of a first approximation
  2. Synthesis of $Minimal$ and $NonMinimal$
  3. Synthesis of $Decompose$
  4. Syntactic insertion of the recursive atoms
- Reduction phase
  5. Synthesis of $Solve$
  6. Synthesis of the $Process_k$ and $Compose_k$

7. Synthesis of the $Discriminate_k$
8. Syntactic generalization

The expansion phase corresponds to the trace-generation phase of trace-based synthesis (see section 2.1), because the given examples are "explained" in terms of divide-and-conquer traces. The reduction phase corresponds to the trace-generalization phase of trace-based synthesis, because the obtained traces are "folded" together by a recurrence relation detection mechanism.

For the *compress*/2 problem, synthesis proceeds as follows:

Expansion Phase

At Step 1, the examples are simply rewritten as follows:

$$\begin{aligned} compress(L,C) \Leftrightarrow\ & L = [] \wedge C = [] \\ & \vee L = [a] \wedge C = [a,1] \\ & \vee \ldots \\ & \vee L = [f,f,g] \wedge C = [f,2,g,1] \\ & \vee \ldots \end{aligned}$$

At Step 2, L is non-deterministically chosen as the induction parameter. Then, $L = []$ is tried as the instance of $Minimal$ and $L = [\_|\_]$ as the instance of $NonMinimal$, both instances being extracted from a knowledge base.
At Step 3, $L = [HL|TL]$ is tried as the instance of $Decompose$, by extraction from a knowledge base. The values of HL and TL can then be computed for each disjunct of the current program:

$$\begin{aligned} compress(L,C) \Leftrightarrow\ & L = []\ \wedge L = [] \wedge C = [] \\ & \vee L = [\_|\_] \wedge L = [HL|TL] \wedge L = [a] \wedge C = [a,1] \\ & \qquad \wedge HL = a \wedge TL = [] \\ & \vee \ldots \\ & \vee L = [\_|\_] \wedge L = [HL|TL] \wedge L = [f,f,g] \wedge C = [f,2,g,1] \\ & \qquad \wedge HL = f \wedge TL = [f,g] \\ & \vee \ldots \end{aligned}$$

At Step 4, the recursive atoms are syntactically inserted, depending on the number of tails of the induction parameter introduced by the instance of $Decompose$ chosen at Step 3. Here there is only one tail (namely TL) of the induction parameter L, so only one recursive atom (namely $compress(TL,TC)$) is introduced for every disjunct that reflects a non-minimal example. The values of TC can then be deductively computed using the properties, respectively by querying the user (analogy-based methods are under investigation as well).
For our problem, the deductive method is sufficient: e.g. the third disjunct becomes:

$$\vee L = [\_|\_]\ \wedge L = [HL|TL] \wedge L = [f,f,g] \wedge C = [f,2,g,1]\ \wedge HL = f \wedge TL = [f,g]$$

The value $[f,g]$ of TL matches the value $[X,Y]$ of the first parameter in the head of the second and the third properties, both times under the substitution $X/f, Y/g$. However, only the body $X \neq Y$ of the third property:

$$X \neq Y \Rightarrow compress([X,Y],[X,1,Y,1])$$

is satisfiable under this substitution, so we obtain the value of TC for the third disjunct by instantiating the second parameter of the head of the third property via the substitution $X/f, Y/g$. Hence:

$$\begin{aligned} & \vee L = [\_|\_]\ \wedge L = [HL|TL] \wedge compress(TL,TC) \wedge L = [f,f,g] \wedge C = [f,2,g,1] \\ & \qquad \wedge HL = f \wedge TL = [f,g] \wedge TC = [f,1,g,1] \end{aligned}$$

Similarly for the other non-minimal disjuncts.

Reduction Phase

At Step 5, a method similar to the ones of Steps 6 and 7 is used to find that $C = []$ is a good instance of $Solve(L,C)$ (which in turn is the current instance of $Solve(X,Y)$).
At Step 6, we instantiate $Process(HX,HY) \wedge Compose(HY,TY,Y)$, which can also be seen as instantiating $ProcessCompose(HX,TY,Y)$ (namely by compiling away the intermediate variable HY). In our problem, we thus have to instantiate $ProcessCompose(HL,TC,C)$ so as to compute the compact list C from its tail TC and the head HL of L. This involves partitioning the non-minimal disjuncts such that in each partition this computation is performed in the same way.
If all the desired instances can be expressed only in terms of equality atoms, then computing the MSGs (most-specific generalizations) of the $< HL,TC,C >$ triples extracted from the current program will readily yield this partition. Otherwise, e.g. if the instance contains recursive calls, the synthesizer

reinvokes itself using these $< HL, TC, C >$ triples as new examples as well as properties derived from the initial properties as new properties.
For our problem, the MSG method is sufficient and partitions the disjuncts into two classes (hence instantiating the schema-variable c of the divide-and-conquer schema above to 2). We have

$$\begin{aligned}
compress(L,C) \Leftrightarrow\ & L = [] \wedge C = [] \\
& \wedge L = [] \wedge C = [] \\
& \vee L = [\_|\_] \wedge L = [HL|TL] \wedge compress(TL,TC) \wedge C = [HL,1|TC] \\
& \qquad \wedge L = [a] \wedge C = [a,1] \wedge HL = a \wedge TL = [] \wedge TC = [] \\
& \vee \ldots \\
& \vee L = [\_|\_] \wedge L = [HL|TL] \wedge compress(TL,TC) \wedge TC = [HL,M|TTC] \\
& \qquad \wedge C = [HL,s(M)|TTC] \\
& \qquad \wedge L = [f,f,g] \wedge C = [f,2,g,1] \wedge HL = f \wedge TL = [f,g] \\
& \qquad \wedge TC = [f,1,g,1] \wedge M = 1 \wedge TTC = [g,1] \\
& \vee \ldots
\end{aligned}$$

where $s(M)$ stands for the successor of integer M.
At Step 7, discriminants deciding on which instance of the $Process_k(HX,HY) \wedge Compose_k(HY,TY,Y)$ is applicable are synthesized. This is achieved by proving that the properties logically follow from the currently synthesized logic program: if the proof fails, then analysis of the failure reveals the instances of the $Discriminate_k$.
For our problem, the proof of the first property yields:

$$discriminate_1(HL,[],[HL,1]) \Leftarrow true$$

while the proof of the second property gives:

$$discriminate_2(HL,[HTL],[HL,2]) \Leftarrow HL = HTL$$

and the proof of the third property yields:

$$discriminate_1(HL,[HTL],[HL,1,HTL,1]) \Leftarrow HL \neq HTL$$

After some regrouping and generalizing, we obtain the following instances:

$$\begin{aligned}
discriminate_1(HL,TL,C) \Leftrightarrow\ & TL = [] \\
& \vee (TL = [HTL|\_] \wedge HL \neq HTL) \\
& discriminate_2(HL,TL,C) \Leftrightarrow TL = [HTL|\_] \wedge HL = HTL
\end{aligned}$$

At Step 8, all equality atoms involving constants introduced from the examples are removed. The results looks as follows.

$$\begin{aligned}
compress(L,C) \Leftrightarrow\ & L = [] \wedge C = [] \\
& \vee L = [\_|\_] \wedge L = [HL|TL] \\
& \qquad \wedge TL = [] \vee (TL = [HTL|\_] \wedge HL \neq HTL) \\
& \qquad \wedge compress(TL,TC) \wedge C = [HL,1|TC] \\
& \vee L = [\_|\_] \wedge L = [HL|TL] \\
& \qquad \wedge TL = [HTL|\_] \wedge HL = HTL \\
& \qquad \wedge compress(TL,TC) \wedge TC = [HL,M|TTC] \wedge C = [HL,s(M)|TTC]
\end{aligned}$$

A first version of the SYNAPSE system has been written in Quintus Prolog as a meta-program based on the ground representation of object-variables (100K, including about 42% of comments). Synthesis is interactive (the user may express preferences and hints) and non-deterministic (a family of alternative programs is synthesized). Not all the features of the synthesis mechanism described in [15] have been implemented yet. Predicate invention is done by self-invocation on a derived specification by examples and properties. A very general divide-and-conquer schema is hardwired into the system.

Other problems within the scope of the SYNAPSE synthesis mechanism are:

- $plateau(N,E,P)$ iff P is a plateau of N elements equal to E, where N is a positive integer, E is a term, and P is a non-empty list, e.g. $plateau(2,a,[a,a])$, but not $plateau(0,f,[])$, nor $plateau(2,a,[a,a,b]$;
- $firstPlateau(L,P,S)$ iff P is the first plateau [1] of L, and S is the corresponding suffix of L, where L is a non-empty list, P is a plateau, and S is a list, e.g. $firstPlateau([a,a,b],[a,a],[b])$, but not $firstPlateau([a,a,b],[a],[a,b])$;
- $delete(E,L,R)$ iff R is L without its first (existing) occurrence of E, where E is a term, L is a non-empty list, and R is a list, e.g. $delete(a,[a,b,a],[b,a])$, but not $delete(a,[a,b,a],[a,b])$;
- $sort(L,S)$ iff S is a non-descendingly ordered permutation of L, where L, S are integer lists.

[1] a *plateau* is a non-empty list of identical elements

These problems range from easy (*plateau*/3: 3 examples, 2 properties) to moderately difficult (*compress*/2 : 8 examples, 3 properties; *firstPlateau*/3: 7 examples, 3 properties; and *delete*/3: 6 examples, 3 properties) to intricate (*sort*/2: 10 examples, 3 properties). For the latter, the Insertion-Sort, Merge-Sort, and Quicksort programs can be found upon backtracking of the synthesis mechanism, provided (for the latter two) that the predicates *split*/2 and *partition*/3 are available in the knowledge-base for the *Decompose* place-holder. This approach reflects a structured provision of knowledge to the synthesis mechanism, and thus solves the efficiency problem in the presence of (too) much background knowledge. The needed instantiations of the *Compose* place-holder, namely *insert*/3, *merge*/3, and *append*/3, respectively, can be invented from scratch by recursive re-invocation of the synthesis mechanism on a derived specification by examples and properties. □

In the following, we discuss how transformation techniques can significantly improve the capabilities of ILP systems.

4 Logic Program Transformations and ILP

The efficiency of logic programs obtained through different ILP techniques depends significantly on the used representation language and on the available background knowledge. The primary goal of ILP is to create a program covering the presented examples. Moreover, in some domains the resulting program represents new knowledge, which should be understandable to its expected human user in order to be verified or accepted by him/her. That is why ILP does not have to seek the most elaborate programs, ever. On the contrary, sometimes simple and well understandable programs should be the primary goal. Unfortunately, simplicity of expression and program efficiency do not go hand in hand often. That is why we believe that program transformation techniques should be included in an ILP environment. There is one more reason for this belief, namely the problem of "good choice" of examples and of the language. The type of program created by an ILP system is highly dependent on available background knowledge. Suppose we want to synthesize a program for reversing lists. If the background knowledge provides a definition of *append*/3, it is unwise to insist that the ILP system has to derive from the examples of the input-output relation of the intended program directly the well known linear version of *reverse*/2 using an auxiliary predicate with the accumulator parameter.

We believe that it is sufficient if an ILP system identifies the correct "naive" reverse computing the result in quadratic time w.r.t. the length of the input. Later the problem of efficiency of the resulting program can be treated as a separate task of further "optional" processing. This last step can be supported by a transformational approach. In such a case the learning process will follow a natural sequence reminding closely the classical didactic approach utilized by human teachers:

1. learning a "simple" or "naive" logic program covering the given examples,
2. transformation of the former result into a more efficient version based on the results of deeper experience or experimentation with the intermediate version.

The invention/definition of new predicates — the so called "eureka" — is one of the crucial problems of the search for a useful program transformation aiming at higher efficiency. Most papers on program transformations are devoted to the search for provably equivalent programs exhibiting some requested properties, e.g. higher efficiency. But it may well be the case that in the area of ILP another approach can be followed:

> Would it not be wise to try to combine a heuristic technique offering a good hint for improvement of the input logic program with the strong debugging facilities of an MIS-like system [27]?

Under this assumption, the important property of the transformed program is no longer full equivalence to the original input program, because a nearly correct version can be tuned to the provided examples by the debugging system. The transformation step should suggest a new version of a program, syntax of which differes from that of the input program (e.g. a new predicate is introduced), while both programs exhibit the same beahviour on most provided examples of the input-output relation.

It has been shown in many papers [e.g. [9], [32]] that practical experience with a naive program can give good hints leading to eureka occurrence. The eureka step is one of the bottle-necks of the classical approach to program transformations as they have been defined by Burstall and Darlington [10]. Their basic operations are definition, instantiation, abstraction, unfold, fold, and laws. Eureka occures in the definition step and it is often impossible to be applied without utilization of laws — reasoning capabilities — which are very difficult to be automated. We advocate a heuristic approach to program transformations. A brief description of a primary version of such a system is given in section 4.2. Our system extends the original set of transformation steps by a new operation called meta-abstraction, because it abstracts on predicates rather on terms.

4.1 Meta-abstraction

Using pure abstraction during the program transformation, we can often loose some important information on data processed in the studied case. In order to make-up for this draw-back, background knowledge (laws) can be utilised for exact description of the relation between the abstracted variable and the original data-structure. Let us call meta-abstraction a transformation operation which

1. introduces a substitution variable for a specific instance in a program clause (this is what pure abstraction does),
2. adds to the original clause full information on the construction of the introduced variable using predicates from background knowledge.

Let us consider a predicate sub/2, refered to as sublist, defined as follows: sub(X,L) holds iff list X can be obtained from the list L by leaving-out some of its elements

```
sub([],Y).
sub([A|X],[A|Y]):-sub(X,Y).
sub(X,[A|Y]):-sub(X,Y).
```

Suppose we have identified that our input examples correspond to the relation, denoted by csub/3, defined by the following clause

```
csub(X,Y,Z):-sub(X,Y),sub(X,Z).
```

The logic program consisting of this clause and the above definition of the predicate sub represents a generate-and-test program which is obviously far from efficient. Let us try to transform it into a more efficient version. During the transformation course we reach a clause

```
csub([A|C1],[A|L1],[B|M1]):-sub(C1,L1),sub([A|C1],M1).
```

Now, meta-abstraction is used to obtain a new version of this clause, which reads as follows

```
csub([A|C1],[A|L1],[B|M1]):-sub(C1,L1),append([A],C1,D),sub(D,M1).
```

The body of this clause will serve as a basis for definition of a new auxiliary predicate dsub/4

$$dsub(B,C,L,M) \Leftrightarrow \exists D(sub(C,L) \wedge append(B,C,D) \wedge sub(D,M))$$

After some more transformational steps, we obtain finaly a more efficient version, denoted by csub1/3, of the original predicate csub utilizing the auxiliary predicate dsub:

```
csub1(C,L,M):-dsub([],C,L,M).
dsub([],[],L,M).
dsub([],[A|C1],[A|L1],[A|M1]):-dsub([],C1,L1,M1).
dsub([],[A|C1],[A|L1],[B|M1]):-dsub([A],C1,L1,M1).
dsub([],C,[A|L1],M):-dsub(([],C,L1,M).
dsub([A],C,L,[A|M1]):-dsub([],C,L,M1).
dsub([A],C,L,[B|M1]):-dsub([A],C,L,M1).
```

This program is not only more efficient then the original csub, but it compares well even with transformed version of csub/3 obtained in [33].

4.2 *STRATEGY*

Actually, we are developing a system STRATEGY for generation of nearly equivalent transformed versions of an input program. This system [34] implements a heuristic search for this purpose. STRATEGY generates the transformation graph of the input logic program using all transformation operations. The definition operation is restricted in order to provide a semi-finite search space. This restriction is motivated by the results on normal forms of logic programs [32]. The transformation graph is searched by an A* algorithm using iterative deepening. The applied heuristic function reflects the case history of the successful applications of the transformation operations with respect to the treated logic program (and the specific predicate it is applied to). The first experiments with the implementation of STRATEGY [34] seem to indicate that it leads rather quickly to a new more efficient "nearly correct" version of the original program for "small problems".

The actual version serves as an input to the debugging facilities of MIS [27], which produces a correct final version using the original program as an oracle.
□

A program construction is an interactive process. Automatic programming should be as well. Some aspects of that statement are discussed bellow. Then, an assumption-based interactive learning is introduced.

5 Interactive Automatic Programming

5.1 Interactive Program Synthesis

Constructing programs is a process with informal intentions at the very beginning and a hopefully correct program at the end. In the case of batch learning, teacher is not always able to collect all the information needed for a program synthesis, namely to build a good example set, in advance. [2] On the other side, a solution offered by interactive learners like MIS [27] or $CLINT$ [11], to ask examples one-by-on, is too bothering for a teacher.

In our case, as a teacher is a human, learning from examples can be seen as a programming by examples. But this kind of programming is nothing but programming. This means that it suffers from the same disease like the classical programming - an informality of teacher's intentions. Because of that, more attention should be paid to a methodology programming by examples, in our case to a problem of driving teacher to choose good examples [24].

Constructing programs is an *iterative* process, where in each step a "better" program is constructed, i.e. a program respecting the teacher's intentions more precisely. [3] We see two ways of solving this drawback: to ask a teacher to formalize his or her intentions, or to support the transfer of knowledge on program being learned. Here, and in the next section, we will address the latter. First, we briefly describe a new approach to interactive model-based learning, an assumption-based leraning.

5.2 Assumption-Based Interactive Learning: hM scenario

In our approach we wish to exploit the advantages of interactive learners and limit their drawbacks, namely:

- to lower the amount of information needed (number of examples, information about the needed background knowledge predicates, bias);
- to limit the number of queries to the teacher during a synthesis.

Ideas on the assumption-based framework underlying our methodology may be found in [7, 22]. The generic hM program [26] consists of three parts:

1. **Generator of assumptions**, which generates extensions of the learning set.
   An assumption can be any formula consistent with that learning set. The generator employs a $preference\ relation$ to generate the most-preferable assumptions first.
2. **Inductive learner**
3. **Truth maintenance system**, which evaluates the acceptability of assumptions. It may ask queries to the teacher.

In the next section, WiM, an assumption-based interactive system, is introduced. Results reached with WiM in learning simple list processing predicates are described and compared with those of $FILP$ [4], CRUSTACEAN [2] and BMWk [23].

6 WiM

WiM is a first implemenation of the assumption-based method. It is implemented in standard Prolog. WiM needs only small example set (2 or 3 positive examples) for learning list processing predicates. Before describing a session with the WiM program, we describe $Markus^+$, our modification of $Markus$ learner, which we use as the instance of the inductive learner in WiM.

6.1 $Markus^+$

$Markus^+$ [26] is an improved version of the $Markus$ learner [19, 20, 21]. $Markus$ improved Shapiro's MIS system [27] by an optimal generation of the refinement graph, by controlling the search space with several parameters, and by use of iterative deepening search, among others. Unlike MIS, $Markus$ processes positive and negative examples in batch.

The improved version, $Markus^+$, allows shifting of bias and the definition of a second-order schema which the learned program has to match. Three parameters are used for shifting bias — the maximal number of free variables in a clause, the maximal number of goals of a clause body, and the maximal head argument depth ($X, [X|Y], [X, Y|Z]$, etc. are of depths 0, 1, 2, etc., respectively). The teacher

[2] Even the assumption on the consistency of the example set, which is common for all ILP learners, can be found too restrictive. Here, we suppose that an example set is consistent.

[3] Intentions are assumed to be fixed. The fact that intentions may change during the period of program construction will not be addressed here.

specifies lower and upper bounds of those parameters. $Markus^+$ starts with the minimal values of these settings. If no acceptable result has been found, one of the settings is increased by 1 so that all variations are being tried. It means that the bias of $Markus^+$ may be weaker than that of $Markus$.

6.2 *WiM*: Main algorithm

A learning session for a predicate P starts from a learning set L, a second-order schema SP of the predicate P, a definition of modes and types of arguments D, and a set of background knowledge predicates BK given by teacher. We will demonstrate it by learning $last(Elem, List)$ predicate ($Elem$ is the last element of a list $List$) using WiM with an input knowledge as follows:

$$L = \{last(a, [a]),\ \ last(b, [c, b])\}$$
$$SP = P :- Q\ \ ;\ \ P :- Q, P.$$
$$D = \{-x, +[x]\}$$
$$BK = \{\}$$

Q may be a conjunction of background knowledge predicates, i.e. recursive programs containing two clauses are acceptable. The set BK of background knowledge predicates is empty so that only `last/2` predicates may appear in the learned program.

WiM processes the learning set L. If it has found an intensional definition of a predicate P such that it matches the second-order schema, then the teacher is asked to confirm/reject that predicate definition. If the teacher doesn't agree with the found solution, it may either ask for an another solution – in that case an assumption is being generated to extend the learning set L – or give a new example. If s/he agree with the found predicate definition, the session is finished.

In our example, an incorrect definition

```
last(X,[Y|Z]) :- last(X,Z).
last(X,[X|Y]).
```

was found for the learning set L. That is why the *generator of assumptions* is called to generate an assumption. As an assumption, a **near miss** to a chosen positive example is generated. A near-missed example is a negative example that differs from a positive example of the intended predicate "as little as possible". A *preference relation* on the set of examples is defined [26] to generate nearmisses of less complex examples first. For list processing predicates, a sum of argument lengths has been found suitable. [4]

The constant set for the two examples above, if extended by a constant `d`, is {`a,b,c,d`}. The example

```
last(a,[a])
```

is chosen for generating near misses. The following syntactic approach is used for computing near misses: extend a set of constants by a new constant. Then take a positive example and modify it by adding/deleting a list element, or by replacing an atom, using the extended constant set. For `last(a,[a])`

```
not last(d,[a]), not last(a,[]), not last(a,[d,a]),
not last(a,[a,d])
```

are generated. For each of assumptions, a *truth maintenance system* calls $Markus^+$ to process the original example set extended by the assumptions. If the new predicate definition differs from the definitions found so far, the teacher is asked to confirm/reject the assumption. For the assumptions `not last(d,[a])` and `not last(a,[])`, the same incorrect solution as above is found. For `not last(a,[d,a])`, WiM has found no solution because of inconsistency in the example set. That's why user is asked only the following question

```
last(a,[a,d]) assumed to be false. OK ? (yes /no /unknown /why)
```

If the teacher agrees (`yes`), then the following information is displayed to the teacher

```
Found predicate

    last(X,[Y|Z])  :- last(X,Z).
    last(X,[X])    :- true.

under assumption that

    last(a,[a,d]) = false
```

and an equivalence query [3] like in MIS [27]) is asked.

```
OK (yes. / new example) ?
```

[4] A length of atoms is equal to 0, a length of list argument is equal to the length of the list.

If the teacher has no example to add to the learning set, the learning session is finished.

If the answer is `no`, i.e. the assumption is rejected, the example is added into the learning set as positive one, and the learning session goes on with this extended learning set. If `unknown` is answered, *WiM* search for another assumption deleting the old one from the learning set. If the answer is `why`, the learned predicate definition using the assumption `last(d,[a,d])` is displayed and the same question is asked again. The example set is left unchanged.

In the following section, experimental results obtained by *WiM* are summarized and compared with results of *FILP* [4], CRUSTACEAN [2] and BMWk [23].

6.3 Discussion of results

For the synthesis of list-processing predicates, the upper bounds of the maximal number of free variables in a clause as well as the maximal head argument depth were set to 2. The maximal number of goals of a clause body was set to 3. We tested *WiM* on the following predicates

- $member(E, L)$ iff the element E appears in the list L;
- $append(L1, L2, L3)$ iff the list $L3$ is equal to the list $L1$ appended by the list $L3$;
- $delete(E, L1, L2)$ iff the list $L2$ is the non-empty list $L1$ without its first (existing) occurrence of E;
- $reverseConcat(L1, L2)$ iff the list $L2$ has the same elements as the list $L1$ but in the reverse order. It uses $concat(L1, E, L2)$ predicate which appends the element E to the list $L1$;
- $reverseAppend(L1, L2)$ is the same as $reverseConcat(L1, L2)$ but using $append(L1, L2, L3)$;
- $last(E, L)$ iff the element E is the last element of the list L;
- $split(L1, L2, L3)$ iff the lists $L2$ and $L3$ contain only odd and even elements, respectively, of the list $L1$.

In the table bellow, a number of the examples needed are summarized:

	WiM	FILP	CRUSTACEAN
member	2+	4	4
append	2+1	4	4
delete	2+1	-	3
reverseConcat	2+	4	-
reverseAppend	3+	-	3
last	2+1	-	3
split	2+1	-	6

Table 2: Learning list processing predicates

For *WiM*, the number before the + sign means a number of positive examples in the initial example set. 1 after the + signs that one assumption was added to the initial example set. We need at worst as many examples as *CRUSTACEAN* and less then or *FILP*. The initial example set need not be so carefully choosen like this one for *FILP*. The number of queries to a teacher is less than for *MIS* [27]. It would be interesting to compare *WiM* with the *BMWk* methodology [23]. *WiM* doesn't need to have examples on the same computational chain. *WiM*, too, needs less examples for simple list processing predicates. However, for more complex tasks it has not been verified.

One can argue that the used bias in *WiM* is too strong. It is true that the upper bound of head argument depth plays the role. However, it is just an upper bound. As $Markus^+$ uses shifting of bias, starting with the simplest head of a clause, a high value of this bound implies only a less efficient computation, (e.g. after `member(X,[X|Z] :- true`, the clause `member(X,[Y,X|Z]) :- true` is generated, which is unacceptable because of the absence of recursion in the found solution). Actualy, a bias of *WiM* is not stronger than that of *CRUSTACEAN*.

Current research is focusing on:

- a synthesis of more complex predicates;
- multiple predicate synthesis where assumptions on different predicates appear;
- abducing more complex assumptions, eg. non-recursive clauses.

We intend to exploit integrity constraints [11] as a generalization of negative examples, and incomplete specifications like properties [16, 15]), as well as algorithm design knowledge (namely the second-order schema to guide synthesis) and domain knowledge.

7 Conclusion

We described three approaches to a logic program synthesis – schema-driven synthesis, program transformation, and assumption-based interactive synthesis – and argued that more attention should be payed to them in an ILP research on automatic programming. The schema-driven synthesis seems to be overcomming the most of existing ILP synthesizers by its capability to learn more complex predicates, even if some of background knowledge predicates are unknown. Existence of powerfull transformation techniques could allow us to focus on a synthesis of simple programs, which can later be transformed into more efficient ones. Different kinds of human-computer interaction can overcome some of program synthesis drawbacks in a natural way. We believe that joining these three approaches will allow us to match the task of automatic logic programming better.

Acknowledgments

We would like to thank Norbert Fuchs for lot of discussion and for his comments on the earlier version of this paper. Thanks of the second author are due to Alípio Jorge for fruitful discussions. Thanks to those two of three referees who realy read the previous version of this contribution.

References

[1] Deville Y. and Lau K.-K.: Logic program synthesis: A survey. Special Volume on Ten Years of Logic Programming, *Journal of Logic Programming*, 1994.

[2] Aha D.W., Lapointe S., Ling C.X., and Matwin S.: Inverting implication with small training sets. In Bergadano F., De Raedt L. (eds): *Proc. of ECML'94*, Catania, pages 31-48. LNCS 784, Springer Verlag, 1994.

[3] Angluin D.: Queries and concept learning. *Machine Learning 2(4):319–342*, April 1988.

[4] Bergadano F. and Gunetti D.: An interactive system to learn functional logic programs. *Proc. of IJCAI'93*, Chambéry, pp. 1044–1045.

[5] Biermann A.W.: Dealing with search. In Biermann A.W., Guiho G., and Kodratoff Y. (eds): *Automatic Program Construction Techniques*, pp. 375–392. Macmillan, 1984.

[6] Biermann A.W.: Automatic programming. In Shapiro S.C. (ed.): *Encyclopedia of Artificial Intelligence*, pp. 59–83. John Wiley, 1992. (Second, extended edition.)

[7] Bondarenko A., Toni F., and Kowalski R.A.: An assumption-based framework for non-monotonic reasoning. In Perreira L.M. and Nerode A. (eds): *Proc. of the 2nd Int'l Workshop on Logic Programming and Non-Monotonic Reasoning*, Lisbon, pp. 171–189. MIT Press, 1993.

[8] Brázdil P. and Jorge A.M.: Learning by refining algorithm sketches. To appear in *Proc. of ECAI'94*, Amsterdam, 1994.

[9] Bruynooghe M. and de Schreye D.: Some thoughts on the role of examples in program transformations and its relevance for EBL. In K.P. Jantke (ed): *Proc. of the AII'89.* LNCS 397:60–77, Springer-Verlag, 1989.

[10] Burstall R.M. and Darlington J.: A transformation system for developing recursive programs. *Journal of the ACM 24:44–67*, 1977.

[11] De Raedt L.: *Interactive Theory Revision: An Inductive Logic Programming Approach.* Academic Press, 1992.

[12] Dershowitz N. and Lee Y.-J.: Logical debugging. *Journal of Symbolic Computation 15(5-6):745–773*, May/June 1993.

[13] Deville Y.: *Logic Programming: Systematic Program Development.* Addison Wesley, 1990.

[14] Drabent W., Nadjm-Tehrani S., and Maluszynski J.: Algorithmic debugging with assertions. In: H. Abramson and M.H. Rogers (eds): *Meta-Programming in Logic Programming: Proceedings of META'88*, pp. 501–521. MIT Press, 1988.

[15] Flener P.: *Logic Algorithm Synthesis from Examples and Properties.* Ph.D. Thesis, Université Catholique de Louvain, Louvain-la-Neuve (Belgium), June 1993. To be published by Kluwer Academic Publishers, 1994.

[16] Flener P. and Deville Y.: Logic program synthesis from incomplete specifications. *Journal of Symbolic Computation, 15(5–6):775–805*, May/June 1993.

[17] Flener P. and Popelínský L.: On the use of inductive reasoning in program synthesis: Prejudice and prospects. To appear in: Fribourg F. and Turini F. (eds): *Proc. of LOPSTR'94 and META'94*, Pisa. LNCS, Springer-Verlag, 1994.

[18] Hagiya M.: Programming by example and proving by example using higher-order unification. In: M.E. Stickel (ed): *Proc. of CADE'90.* LNCS 449:588–602, Springer-Verlag, 1990.

[19] Grobelnik M.: Markus: An optimized Model Inference System. *Proc. of the ECAI'92 Workshop on Logical Approaches to Machine Learning*, Vienna, 1992.

[20] Grobenik M.: Induction of Prolog programs with Markus. In Deville Y.(ed.): *Proceedings of LOPSTR'93*, pp.57-63. Workshops in Computing Series, Springer Verlag, 1994.

[21] Grobelnik M.: Declarative Bias in Markus ILP system. *Working notes of the ECML'94 Workshop on Declarative Bias*, Catania, 1994. (chairperson Rouveirol C.)

[22] Kakas A.C., Kowalski R.A., and Toni F.: Abductive logic programming. *Journal of Logic and Computation 2(6):719–770*, 1993.

[23] Le Blanc G.: BMWk Revisited. In Bergadano F., De Raedt L. (eds): *Proc. of ECML'94*, Catania, pages 183-197. LNCS 784, Springer Verlag, 1994.

[24] Ling X.C.: Inductive learning from good examples. In *Proc. of IJCAI'91*, Sydney, pages 751-756, Morgan Kaufmann, 1991.

[25] Muggleton S. (ed): *Inductive Logic Programming.* Volume APIC-38, Academic Press, 1992.

[26] Popelínský L.: *Interactive assumption-based learning.* TR Dept. of Comp. Sci., Masaryk University, Brno, 1994.

[27] Shapiro Y.: *Algorithmic Program Debugging.* MIT Press, 1983.

[28] Smith D.R.: The synthesis of LISP programs from examples: A survey. In: Biermann A.W., Guiho G., and Kodratoff Y. (eds): *Automatic Program Construction Techniques*, pages 307–324. Macmillan, 1984.

[29] Smith D.R.: Top-down synthesis of divide-and-conquer algorithms. *Artificial Intelligence 27(1):43–96*, 1985.

[30] Smith D.R.: KIDS: A semi-automatic program development system. *IEEE Transactions on Software Engineering 16(9):1024–1043*, September 1990.

[31] Štěpánková O. and Štěpánek P.: Transformations of logic programs. *Journal of Logic Programming 1:489–501*, 1984.

[32] Štěpánková O. and Štěpánek P.: Developing logic programs: Computing through normalizing. *Proc. of Computer Science and Logic LICS'87*, Karlsruhe. LNCS 329:304–321, Springer-Verlag, 1988.

[33] Tamaki H., Sato T.: Unfold/fold Transformations of Logic Programs. In *Proc. of the 2nd Int. Logic Programming Conference ICLP'84*, Uppsala 1984, S.A. Tarnlund (ed.), pp.127-138

[34] Zídek J.: *Development of Logic Programs.* MS Thesis, CTU Prague, 1994.