Inductive Synthesis of Functional Programs

Ute Schmid

Program synthesis from incomplete information is a challenging problem for machine learning algorithms, addressing discovery of generalized, recursive rules from observations. Dividing synthesis in two steps - generation of finite terms and folding into recursion - allows application in different domains such as control-rule learning for planning.

1 Program Synthesis

Automatic program synthesis is an active area of research since the sixties. The application goal is to support human programmers in developing correct and efficient program code and in reasoning about programs - that is, program synthesis is the core of knowledge based software engineering. The second goal of program synthesis research is to gain more insight in the knowledge and strategies underlying the process of code generation. Developing algorithms for automatic program construction is therefore an area of AI research.

There are two main approaches to program synthesis – deduction and induction. Deductive program synthesis addresses the problem of deriving executable programs from high-level specifications. Typically, the employed transformation or inference rules guarantee that the resulting program is correct with respect to the given specification – but of course, there is no guarantee that the specification is valid with respect to the informal idea a programmer has about what the program should do. The challenge for deductive program synthesis is to provide formalized knowledge, that is, a synthesis system can be seen as an expert system incorporating general knowledge about algorithms, data structures, optimization techniques, as well as knowledge about the specific programming domain.

Inductive program synthesis investigates program construction from incomplete information, namely from examples for the desired input/output behavior of the program. Program behavior can be specified on different levels of detail: as pairs of input and output values, as a selection of computational traces, or as an ordered set of generic traces, abstracting from specific input values. For induction from examples it is not possible to give a notion of correctness. The resulting program has to cover all given examples correctly, but the user has to judge whether the generalized program corresponds to his/her intention. The challenge for inductive program synthesis is to provide learning algorithms which can generalize as large a class of programs with as less background knowledge as possible.

In the early days of inductive program synthesis research, there were proposed several approaches to the synthesis of Lisp programs from examples or traces [2]. The most influential approach was proposed by [14], who put inductive synthesis on a firm theoretical foundation. Due to only limited progress, interest decreased in the mid-eighties and research focused on inductive logic programming (ILP) instead [3]. Although ILP proved to be a powerful approach to learning relational concepts, applications to learning of recursive clauses had only moderate success. Probably, one must accept that fully-automated inductive program synthesis can only be applied to small programming problems and does not scale-up to synthesis of complex algorithms. Nevertheless, automatic inductive synthesis is still a problem of interest to basic research – giving us insights in what class of programs can be learned from examples.

The approach to inductive program synthesis presented in my habilitation thesis [8] is based on the recurrence detection method of [14]. Induction of a recursive program is performed in two steps: First, input/output examples are rewritten into a finite program term, and second, the finite term is checked for recurrence. If a recurrence relation is found, the finite program is folded into a recursive function which generalizes over the given examples. The first step of Summers’ approach is knowledge-dependent: in general, there are infinitely many possibilities to represent input/output examples as terms. Because generalizability depends on the form of the finite program, this first step is the bottleneck of program synthesis. Here program synthesis is confronted with the crucial problem of AI and cognitive science – problem solving success is determined by the constructed representation. Summers deals with that problem by restricting his approach to structural list problems (such as reverse) excluding problems where the semantics of the list elements needs to be considered (such as member or sort). Alternatively, the finite program can be generated by the user [13], constructed by AI planning [12], or by genetic programming [11]. The second step of synthesis – folding – is domain-independent: recurrence detection is based on a purely syntactical pattern-matching approach.

2 Recurrence Detection in Finite Terms

Dividing the complex program synthesis problem in two parts allows us to address the sub-problems separately. The knowledge-dependent first part – generating finite programs from examples – can be realized by different approaches, including the rewrite-approach proposed in the seventies. I propose to use state-based planning. While there is a long tradition of combining (deductive) planning and deductive program synthesis [6], up to now there was no interaction between research on state-based planning and research on inductive program synthesis. State-based planning provides a powerful approach to calculate (optimal) transformation sequences from input states to a state fulfilling set of goal relations by providing a powerful domain specification language together with a domain-independent search algorithm for plan construction.

The seminal contribution of Summers was to provide a theoretical foundation for the second step of synthesis – folding of finite programs. He provided a synthesis theorem giving a justi-
fication why generalization of traces based on recurrence detection is legal. He exploits the relation between a given recursive function and its sequences of unfoldings as it is used in fixed point semantics. The theorem represents the converse idea, that is, to find a recurrence relation characterization of a partial function, which is considered as the k-th unfolding of some unknown recursive function. Summers' synthesis theorem and algorithm is restricted to folding a single linear recursive equation with a single list as input parameter. I provided a more powerful framework for folding sets of recursive equations with an arbitrary number of, possibly interdependent, parameters complying to the recursive program scheme (RPS) given in Def. 1.

**Definition 1 (RPS)** Let $\Sigma$ be a signature and $\Phi = \{G_1, \ldots, G_n\}$ a set of function variables with $\Sigma \cap \Phi = \emptyset$ and arity $\alpha(G_i) = m_i > 0$. A recursive program scheme (RPS) $S$ is a pair $(G, t_0)$ with $t_0 \in \mathcal{T}_{\Sigma \cup \Phi}(X)$ and $G$ as a system of $n$ equations:

\[
G_1(x_1, \ldots, x_{m_1}) = t_1, \\
G_2(x_2, \ldots, x_{m_2}) = t_2, \\
\vdots \\
G_n(x_n, \ldots, x_{m_n}) = t_n,
\]

with $t_i \in \mathcal{T}_{\Sigma \cup \Phi}(X), i = 1 \ldots n$.

RPSs which can be folded are restricted such that no nested program calls and no mutual recursion is allowed. The first restriction is semantical, that is, the class of RPSs which can be folded is smaller than the class of calculable functions. The second restriction is only syntactical since each pair of mutually recursive functions can be transformed into semantically equivalent functions which are not mutually recursive.

The formal framework and the folding algorithm are based on the notion of an RPS as a term rewriting system. We showed [8], [5] that the words belonging to the language of an RPS can be constructed inductively, that is, we can give a strong characterization of the relation between an RPS and its unfoldings which can be exploited for synthesis. Stated informally, an RPS recursively explains some finite term $t_{\text{init}}$, if there exists an unfolding $t$ such that $t_{\text{init}}$ is a prefix of $t$ and an unfolding $t'$, derived by at least two applications the rewrite rules derived from the RPS such that $t'$ is a prefix of $t_{\text{init}}$.

Induction of an RPS from some finite term involves the following steps:

1. Identification of recursion points, that is, positions in the term which correspond to the position of recursive calls in an equation.
2. Construction of the program body, that is, obtaining all parts of the term which are constant over the unfoldings of an recursive equation.
3. Identification of substitution terms, that is, obtaining the parameters of a recursive equation, their initial values, and their substitution in the recursive call.

Defining pattern-matching for terms which are elements of an arbitrary term algebra makes the approach independent of a specific programming language. Synthesizing program schemes in contrast to programs allows for a natural combination of induction with analogical reasoning and learning.

### 3 The IPAL Approach

The three components of the IPAL approach – planning, folding of finite terms, and analogical problem solving – can be used in isolation or integrated into a complex system (see Fig. 1). Starting point is a problem specification represented in an extended version of the standard Strips language [8], [9]. I proposed an universal planner – DPlan – for constructing the set of all possible optimal transformations of states of a small finite domain into a desired goal state [12]. The universal plan is represented as DAG. Input in the folder is a finite program term, output is a recursive program scheme (see Def. 1). Finite terms can be obtained in arbitrary ways, for example, by hand-coding.

---

**Figure 1: IPAL: Inductive Program Synthesis, Planning and Learning**
or by recording program traces. I developed an approach of plan transformation (from a DAG to a finite program term) to combine planning with program synthesis.

Generating finite programs by planning is especially suitable for domains, where inputs can be represented as relational objects (sets of literals) and where operations can be described as manipulation of such objects (changing literals). This is true for blocks-world problems and puzzles (such as Tower of Hanoi) and for a variety of list-manipulating problems (such as sorting). Recently, we applied genetic programming to construct XSL transformations which are generalized to transformations with recursive template applications [11].

Input in the analogy module is a finite program term, output is the RPS which is most similar to the term, its re-instantiation or adaptation to cover the current term, and an abstracted scheme, generalizing over the current and the re-used RPS [10].

All components are implemented and we are currently working on a GUI where all components can be accessed uniformly and where interactions between the components can be handled interactively or in batch-mode [1].

## 4 Control-Rule Learning

Control rule learning currently becomes a major interest in planning research [7], [4], [12]: Although a variety of efficient domain-independent planners have been developed in the nineties, for demanding real world applications it is necessary to guide search for (optimal) plans by exploiting knowledge about the structure of the planning domain. Learning control rules allows to gain the efficiency of domain-dependent planning without the need to hand-code such knowledge which is a time consuming and error-prone knowledge engineering task. Our functional approach to program synthesis is very well suited for control rule learning because in contrast to logic programs the control flow is represented explicitly. Furthermore, a lot of ILP systems do not provide an ordering for the induced clauses and literals. Therefore, evaluation of such clauses by a Prolog interpreter is not guaranteed to terminate with the desired result.

Our overall approach to control-rule learning is to first generate a plan for a predefined problem-domain, then rewrite the plan to a term, afterwards using this term as input to the folding algorithm and finally, save the induced RPS as domain specific control rule for the domain (see Fig. 1). Rewriting of a plan into a term is done by identifying the relevant literals – used to provide for a termination condition for the recursion – and introducing datatypes – used for manipulating the input state.

For example, a recursive control rule for solving Rocket transportation problems [16] can be learned fully automatically from a plan for transporting four objects. This and further examples can be found in [12], [8], [15].

## References


