Analysis and Evaluation of Inductive Programming Systems in a Higher-Order Framework*

Martin Hofmann, Emanuel Kitzelmann, and Ute Schmid

University of Bamberg, Germany; *lastname.surname*@uni-bamberg.de

Abstract. In this paper we present a comparison of several inductive programming (IP) systems. IP addresses the problem of learning (recursive) programs from incomplete specifications, such as input/output examples. First, we introduce conditional higher-order term rewriting as a common framework for inductive program synthesis. Then we characterise the ILP system GOLEM and the inductive functional system MAGICHASKELLER within this framework. In consequence, we propose the inductive functional system IGOR II as a powerful and efficient approach to IP. Performance of all systems on a representative set of sample problems is evaluated and shows the strength of IGOR II.

1 Introduction

Inductive programming (IP) is concerned with the synthesis of *declarative* (logic, functional, or functional logic) programs from incomplete specifications, such as input/output (I/O) examples. Depending on the target language, IP systems can be classified as inductive logic programming (ILP), inductive functional programming (IFP) or inductive functional logic programming (IFLP).

Beginnings of IP research [1] addressed inductive synthesis of functional programs from small sets of positive I/O examples only. Later on, some ILP systems had their focus on learning recursive logic *programs* in contrast to learning classifiers (FFOIL [2], GOLEM [3], PROGOL [4], DIALOGS-II [5]). Synthesis of functional logic programs is studied with the system FLIP [6]. Now, induction of functional programs is covered by the analytical approaches IGOR I [7] and IGOR II [8] and by the search-based approach ADATE [9] and MAGICHASKELLER [10]. Analytical approaches work example-driven and are guided by the structure of the given I/O pairs, while search-based approaches enumerate hypothetical programs and evaluate them against the I/O examples.

At the moment, neither a systematic empirical evaluation of IP systems under a common framework nor a general vocabulary for describing and comparing the different approaches in a systematic way exists. Both are necessary for further progress in the field exploiting the strengths and tackling the weaknesses of current approaches.

We present conditional combinatory term rewriting as a uniform framework for describing IP systems and characterise and compare some systems in it. Then, we introduce IGOR II, which realises a synthesis strategy which is more powerful and not less efficient as the older approaches, and evaluate their performance on a set of example problems and show the strength of IGOR II. We conclude with some ideas on future research.

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

2 A Unified Framework for IP

2.1 Conditional Constructor Systems

We sketch term rewriting, conditional constructor systems and an extension to higher-order rewriting as, e.g., described in [11]. Let Σ be a set of function symbols (a signature), then we denote the set of all terms over Σ and a set of variables $\mathcal{X}$ by $\mathcal{T}_\Sigma(\mathcal{X})$ and the (sub)set of ground (variable free) terms by $\mathcal{T}_\Sigma$. We distinguish function symbols that denote datatype *constructors* from those denoting (user-)defined functions. Thus, $\Sigma = \mathcal{C} \cup \mathcal{F}, \mathcal{C} \cap \mathcal{F} = \emptyset$ where $\mathcal{C}$ contains the constructors and $\mathcal{F}$ the defined function symbols. Induced programs are represented in a functional style as sets of recursive rewrite rules over a signature Σ, so called *constructor (term rewriting) systems (CS)*.

The lefthand side (lhs) $l := F(p_1, \ldots, p_n)$ of a rewrite rule $l \to r$ consists of a defined function symbol F and a pattern $p_i \in \mathcal{T}_\mathcal{C}(\mathcal{X})$ which is built up from constructors and variables only. We call terms from $\mathcal{T}_\mathcal{C}(\mathcal{X})$ *constructor terms*. This allows for *pattern matching* known from functional languages such as HASKELL. Consequently, all variables of the righthand side (rhs) must also occur in the lhs, i.e. they must be *bound* (by the lhs). If no rule applies to a term the term is in *normal form*. If we apply a defined function to ground constructor terms $F(i_1, \ldots, i_n)$, we call the i_i *inputs* of F. If such an application normalises to a ground constructor term o we call o *output*.

In a *conditional constructor system (CCS)*, each rewrite rule may be augmented with a *condition* that must be met to apply the rule. Conditional rules are written: $l \to r \Leftarrow v_1 = u_1, \ldots, v_n = u_n$ (cf. Fig. 1(1)). So, a condition is an ordered conjunction of equality constraints $v_i = u_i$ with $v_i, u_i \in \mathcal{T}_\Sigma(\mathcal{X})$. A constraint $v_i = u_i$ holds if, after instantiating the lhs and evaluating all $v_j = u_j$ with $j < i$, (i) v_i and u_i evaluate to the same normal form, or (ii) v_i is a pattern that matches u_i and binds the variables in v_i.

To lift a CCS into the higher-order context and extend it to a (conditional) combinatory rewrite system ((C)CRS) [11] we introduce meta-variables $\mathcal{X}_M = X, Y, Z, \ldots$ which are assumed to be different from any variable in $\mathcal{X}$. Meta-variables occur as $X(t_1, \ldots, t_n)$ and allow for generalisation over functions with arity n. To preserve the properties of a CS, we need to introduce an abstraction operator $[-]-$ to bind variables locally to a context. The term $[A]t$ is called abstraction and the occurences of the variable A in t are bound. For example the recursive rule for the well known function map would look like $map([A]Z(A), cons(B, C)) \to cons(Z(B), map([A]Z(A), C))$ and would match a term like $map([A]square(A), cons(1, nil))$.

2.2 Target Languages in the CCRS Framework

To compare all systems under equal premises, we have to fit the different occurences of declarative languages into the CCRS framework[1]. Considering functional target languages, the underlying concepts are either based on abstract theories, as e. g. equational theory [6], constructor term rewriting systems [8], or concrete functional languages as ML [9] or HASKELL[10]. Applying the CCRS framework to IFP or IFLP systems is straight forward, since they all share the basic principles and functional semantics.

[1] Note the subset relationship between that CS, CCS, and CCRS. So, if the higher-order context is of no matter we use the term CCS, otherwise CCRS.

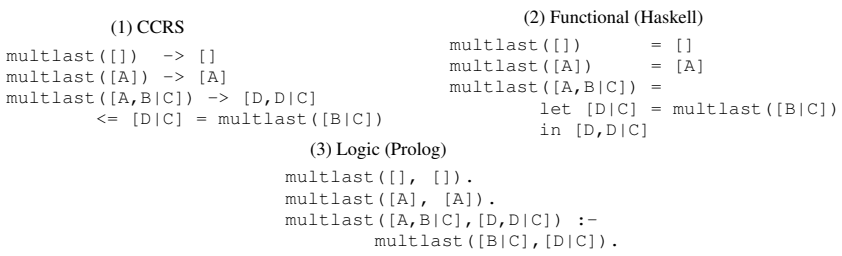

Fig. 1. Equivalent programs of *multlast* (overwriting a list with last element)

In addition to pattern matching and functional operational semantics of CS, CCS can express constructs such as `if`-, `case`-, and `let`-*expressions* in a rewriting context. An `if`-expression is modelled by a condition $v = u$ (case (i) in Sect. 2.1). A `case`-expression is modeled following case (ii), where $v \in \mathcal{T}_{\mathcal{C}}(\mathcal{X})$ and $v \notin \mathcal{X}$. If $v \in \mathcal{X}$, case (ii) models a local variable declaration as in a `let`-expression. Fig. 1 shows a CCRS for a HASKELL program containing a `let`-expression.

In the context of IP, we only consider logic target programs which represent functions, i.e., programs where the output is uniquely determined by the input. Such programs usually are expressed as "functional" predicates such as *multlast* in Fig. 1(3). A *functional predicate* is a relation $r(V_1, \ldots, V_n)$, where for any ground "input" values $i_1, \ldots, i_{n-1}$ of $V_1, \ldots, V_{n-1}$, there is a single "output" value o for V_n such that $\langle i_1, \ldots, i_{n-1}, o\rangle$ belongs to r.The evaluation binds o to V_n using the *input parameters* $V_1, \ldots, V_{n-1}$. If a predicate does not have an output variable it is a "boolean" predicate in the usual sense and evaluates to *true* or *false* if all input parameters are bound. Note that input and output parameters do not have to be variables but may also be constructor terms (containing variables) as known from PROLOG.

In the context of IP, ILP systems require all variable bindings to be directly or indirectly determined by the bindings of the input variables. A Horn clause $h \leftarrow b_1, \ldots, b_n$ is transformed straight forward to a conditional rewrite rule $h' \rightarrow V_h \Leftarrow V_{b_1} = b'_1, \ldots, V_{b_n} = b'_n$, where h' and b'_i are the predicates h and b_i stripped off their output parameters $V_h, V_{b_i} \in \mathcal{T}_{\mathcal{C}}(\mathcal{X})$. For boolean predicates holds $V_h, V_{b_i} \in \{true, false\}$. Transforming Horn clauses containing functional predicates into CCSs is a generalisation of representing Horn clauses as conditional identities as shown in [12].

2.3 IP in the CCRS Framework

Let us now formalise the IP problem in the CCRS setting. Given a CCRS, both, the set of defined function symbols $\mathcal{F}$ and the set of rules R be further partitioned into disjoint subsets $\mathcal{F} = \mathcal{F}_T \cup \mathcal{F}_B \cup \mathcal{F}_I$ and $R = E^+ \cup E^- \cup BK$, respectively. $\mathcal{F}_T$ are the function symbols of the functions to be synthesised, also called *target functions*. $\mathcal{F}_B$ are the symbols of predefined functions that can be used for synthesis. These can either be built in or defined by the user in BK (see below). $\mathcal{F}_I$ is a pool of function variables that can be used for defining invented functions on the fly. E^+ is the set of *positive examples* or *evidence* and E^- the set of *negative examples*, both containing a finite number of I/O pairs as unconditional rewrite rules $F(t_1, \ldots, t_n) \rightarrow r$, where $F \in \mathcal{F}_T$ and $t_1, \ldots, t_n, r \in \mathcal{T}_{\mathcal{C}}(\mathcal{X})$. However, the rules in E^- are interpreted as inequality

constraints. BK is a finite set of rules $F(t_1,\ldots,t_n) \rightarrow r \Leftarrow v_1 = u_1 \wedge \ldots \wedge v_n = u_n$ defining auxiliary concepts that can be used for synthesising the target function, where $F \in \mathcal{F}_B$, $t_i \in \mathcal{T}_C(\mathcal{X} \cup \mathcal{X}_M)$ for $i = 1 \ldots n$, and $r, u_i, v_i \in \mathcal{T}_B(\mathcal{X} \cup \mathcal{X}_M)$.

With such a given CCRS, the IP task can be now described as follows: find a finite set R_T of rules $F(t_1,\ldots,t_n) \rightarrow r \Leftarrow v_1 = u_1 \wedge \ldots \wedge v_n = u_n$ (or program for short) where $F \in \mathcal{F}$, $t_1,\ldots,t_n \in \mathcal{T}_C(\mathcal{X} \cup \mathcal{X}_M)$, and $r, u_i, v_i \in \mathcal{T}_\Sigma(\mathcal{X} \cup \mathcal{X}_M)$, such that it covers all positive examples ($R_T \cup BK \models E^+$, *posterior sufficiency or completeness*) and none of the negative examples ($R_T \cup BK \not\models E^-$, *posterior satisfiability or consistency*). In general, this is done by discriminating between different inputs using patterns on the lhs or conditions modelling `case`-expressions and computing the correct output on the rhs. Constructors, recursive calls, functions from the background knowledge, local variable declarations, and invented functions can be used for this. An invented function is hereby a function which symbol occurs only in $\mathcal{F}_I$, i. e. is neither a target function nor defined in BK and is defined by the synthesis system on the fly.

However, there is usually an infinite number of programs satisfying these conditions, e. g. E^+ itself, and therefore two further restrictions are imposed: A restriction on the terms constructed, the so called restriction bias and a restriction on which terms or rules are chosen, the preference bias.

The *restriction bias* may allow only a specific subset of the terms defined for u_i, v_i, t_i, r in a rule $F(t_1,\ldots,t_n) \rightarrow r \Leftarrow u_1 = v_1 \wedge \ldots \wedge u_n = v_n$. It may restrict nested or mutual recursion, allow for or prohibit abstraction and meta-variables, i. e. higher-order context, or demand the rhs to follow a certain program scheme.

The *preference bias* imposes a partial ordering on terms, lhss, rhss, conditions or whole programs defined by the CCS framework and the restriction bias. A correct program is optimal w. r. t. this ordering and satisfying completeness and consistency.

3 Systems Description in the CCRS Framework

So lets put on the CCRS glasses and have a closer look at the systems. The scope of this paper allows us only to consider GOLEM, MAGICHASKELLER, and IGOR II as representatives of ILP, higher-order search-based and analytic approaches. They were chosen as the most powerfull or most suitable to exemplarily illustrate the strength and weaknesses of their kind. Where appropriate, we will refer to other systems to stress differences or similarities.

GOLEM [3] is, as e. g. FOIL/FFOIL, one of the classic ILP systems. It uses a bottom-up, or example driven approach based on Plotkin's framework of relative least general generalisation (*rlgg*) . This avoids searching a large hypothesis space for consistent hypothesis as, but rather constructs a unique clause covering a subset of the provided examples relative to the given background knowledge.

$\mathcal{C}$ and $\mathcal{F}_B$ are unrestricted, but $\mathcal{F}_T$ is restricted to a singleton set and $\mathcal{F}_I$ is always empty (no function invention). E^+ and E^- are both sets of unconditional rules $F(i_1,\ldots,i_n) \rightarrow o$, where i_i is the i^{th} input and o the output. E^- is very important to prune the search space. $F \in \mathcal{F}_T$, $i_i \in \mathcal{C}$, and $o \in \mathcal{C} \cup \{true, false\}$. For BK, full PROLOG syntax, i. e. in CCRS unrestricted conditional rewrite rules are allowed.

It's restriction bias is quite similar to that of FFOIL, however, predicates can now take constructor terms and not only variables as arguments. Thus, l and v_i are proper functional heads, $r \in \mathcal{T}_{\mathcal{C}}(\mathcal{X})$, and $u_i \in \mathcal{T}_{\mathcal{C}}(\mathcal{X}) \cup \{true, false\}$. It synthesises in first-order, so abstraction and meta-variables are not allowed. The preference bias is defined as the clause covering most of the positive and no negative examples in a lattice over clauses constructed by computing the rlgg of two examples relative to the background knowledge. However, such a search space explodes and makes search nearly intractable.

Therefore, to generate a single clause, GOLEM first randomly picks pairs of positive examples, computes their rlggs and chooses the one with the highest coverage, i.e., with the greatest number of positive examples covered. By randomly choosing additional examples and computing the rlgg of the clause and the new examples, the clause is further generalised. After removing irrelevant literals in a postprocessing step, this is repeated using the clause with the highest coverage until generalisation does not yield a higher coverage. To generate further clauses GOLEM uses the sequential covering approach. It generates one clause that covers some positive and no negative examples, removes the covered examples from the training set and generates the next clause until every positive example is covered by some clause.

MAGICHASKELLER [10] is a comparable new search-based synthesiser which generates HASKELL programs. Exploiting type-constraints, it searches the space of λ-expressions for the smallest program satisfying the user's specification. The expressions are created from user provided functions and data-type constructors via function composition, function application, and λ-abstraction (anonymous functions in HASKELL).

Generally, $\mathcal{C}$ and $\mathcal{F}_B$ are unrestricted, so for BK fully-fledged higher-order functions are allowed. It is noteworthy that this has a direct impact on the synthesiseable functions, since functions in BK immediately define the search space. The system itself is not able to detect recursion, but depends on functions to iterate over or through the defined data types. Therefore to be successful, it needs in addition to the type constructors a paramorphism, i. e. a function that decomposes a given data type, probably applying some function to the primitive part and applying the paramorphism to the rest (i. e. an extended map-function for lists). Only one target-function can be learnt at a time and no function invention is possible. So, $\mathcal{F}_T$ is a singleton set and $\mathcal{F}_I$ is always empty. E^- is empty, too, but E^+ is defined as constraints expressed in a boolean function, so it is possible to define allowed and prohibited outputs of the target function.

It's restriction bias, similar to other search-based approaches (cf. ADATE), is determined by the data types and functions defined in its BK library. So only functions that can be constructed out of these can be synthesised. The system's preference bias can be characterised as a breadth-first search over the length of the candidate programs guided by the type of the target function. Therefore it prefers the smallest program constructable from the provided functions that satisfies the user's constraints.

Forecast As far as one can already say, GOLEM, typically for ILP systems, is hampered by a greedy sequential covering strategy. Consequently, partial rules are never revised and lead to local optima, and dependencies between rules become lost. Nevertheless, it is more flexible in discriminating the inputs on the lhss, because it, contrarily to FFOIL, allows for constructor terms. However, random sampling is too unreliable to balance

out the greedy search and assure for an optimal partition of the inputs, especially when the data structures are more complex or programs with many rules are needed.

MAGICHASKELLER is a promising example of including higher-order features into IP and how functions like map or $filter$ can be applied effectively, when used advisedly, as some kind of program pattern or scheme. Nevertheless, it exhibits the usual pros and cons common to all search-based approaches: The more extensive the BK library, the more powerfull the synthesised programs are, the greater is the search space and the longer are the runs. However, contrarily to GOLEM, it is not mislead by partial solutions and shows again that only a complete search can be satisfactory for IP.

4 IGOR II

In contrast to GOLEM and MAGICHASKELLER, IGOR II is a system specialised to learn *recursive programs*. In order to do this reliably, partitioning of input examples, i.e., the introduction of patterns and predicates, and the synthesis of expressions computing the specified outputs, are strictly separated. Partitioning is done systematically and completely instead of randomly (GOLEM) or by a greedy search (FFOIL). All subsets of a partition are created in parallel, i.e., IGOR II follows a "simultaneous" covering approach. Also the search for expressions is complete. A complete search is tractable even for relative complex programs because construction of hypotheses is data-driven. IGOR II combines analytical program synthesis with search.

IGOR II induces several dependent target functions in one run, no restrictions apply to $\mathcal{F}_T$, $\mathcal{F}_B$ and $\mathcal{C}$. Auxiliary functions are invented if needed, but $\mathcal{F}_I$ is restricted that the domain of each invented function is equal to the domain of the "calling" function, in particular it is not possible to introduce accumulator variables by invention of a auxiliary function. E^- is empty and both E^+ and BK are given as unconditional example equations which may contain variables. In order to achieve confluence it is assured that the induced lhss for one target function do not overlap, i. e., they can be regarded as set and imply a unique partition of the inputs. Rhss of induced rules are restricted in that invented functions cannot be applied at the root. Conditions are restricted to alternative (i) (Sect. 2.1), i. e., simulation of `let`-expressions is not possible.

Fewer case distinctions, most specific patterns, and fewer recursive calls or calls to background functions are preferred. Thus, the initial hypothesis is a single rule per target function. Initial rules are least general generalisations (lggs) [13] of the example equations, i.e., patterns are lggs of the example inputs, rhss are lggs of the outputs w.r.t. the substitutions for the pattern, and conditions are empty. Successor hypotheses have to be computed, if unbound variables occur in rhss. Three ways of getting successor hypotheses are applied: (i) Partitioning of the inputs by replacing one pattern by a set of disjoint more specific patterns or by adding a predicate to the condition. (ii) Replacing the rhs by a (recursive) call of a defined function, where finding the argument of the function call is treated as a new induction problem, i.e., a help function is invented. (iii) Replacing the rhs *subterms* in which unbound variables occur by a call to new subprograms. In cases (ii) and (iii) help functions are invented. This includes the generation of I/O-examples from which they are induced. For case (ii) this is done as follows: Function calls are introduced by matching the currently considered outputs with the outputs

Table 1. System specific runtimes on different problems in seconds

	lasts	*last*	*member*	*odd/even*	*multlast*	*isort*	*reverse*	*weave*	*shiftr*	*mult/add*	*allodds*
I/O size (G/M/I)	$3/2/2_\vee$	$4/3/3_\vee$	$4/3/3_\vee$	$//3_\vee$	$4/3/3_\vee$	$4/3/3_\vee$	$/3/3_\vee$	$4/4/4_\vee$	$4/4/4_\vee$	$//4_\vee$	$4/3/3_\vee$
GOLEM	1.062	$<$ 0.000	0.033	—	$<$ 0.000	0.714	—	$0.049^\perp$	0.298	—	$0.016^\perp$
MAGICH.	7.620	0.040	0.540	—	0.230	—	0.100	31.480	3.620	—	2.100
IGOR II	5.695	0.007	0.152	0.019	0.023	0.105	0.103	0.200	0.127	$\odot$	$\odot$

— not tested $\times$ stack overflow $\odot$ time out $\perp$ wrong $\vee$ variabilised

of any defined function. If all current outputs match, then the matched defined function can be called. The argument of the call must map the currently considered inputs to the corresponding inputs of the matched defined function. For case (iii), the example inputs of the new defined function also equal the currently considered inputs. The outputs are the corresponding subterms of the currently considered outputs. The search ends when the best hypothesis regarding the preference bias has no unbound variables.

5 Empirical Results

As problems we have chosen some of those occurring in the accordant papers and some to bring out the specific strengths and weaknesses. They have the usual semantics on lists: *multlast* was introduced above, *lasts* applies *last* on a list of lists, *isort* is insertion-sort, *allodds* checks for odd numbers, *shiftr* makes a right-shift and *weave* alternates elements from two lists into one. For *odd/even* and *mult/add* both functions need to be learnt at once. The functions in *odd/even* are mutually recursive, *lasts*, *multlast*, *isort*, *reverse*, *mult/add*, *allodds* suggest to use function invention, but only *reverse* is explicitly only solvable with. *lasts*, *allodds* and *odd/even* split up in more than two rules.

Because GOLEM usually performs better with more examples, whereas MAGICHASKELLER and IGOR II do better with less, each system got as much examples as necessary up to certain complexity, but then all examples completely, so no specific cherry-picking was allowed. For synthesising *isort* all systems had a function to insert into a sorted list, and the predicate $<$ as background knowledge. The definition of the background knowledge was extensional (except for MAGICHASKELLER), IGOR II was allowed to use variables and for GOLEM additionally the accordant negative examples were provided. MAGICHASKELLER had paramorphic functions to iterate over a data type in BK. Table 1 shows the runtimes of the different systems on the example problems and the data type size up to which examples were provided.

Due to GOLEM's random sampling, the best result of ten runs was chosen. Despite its randomisation, it exceeds other ILP and IFLP systems due to its capability of introducing `let`-expressions (cf. *multlast*) where IGOR II needs function invention to balance this weak-point. So `let`-constructs can be considered as "poor man's function invention" showing to be quite usefull and promise to help pushing the boundaries of learnable problems even further. On *reverse* and *allodds* MAGICHASKELLER

demonstrates the power of higher order functions. Although it does not invent auxiliary functions, *reverse* was solved using its paramorphism which provides some kind of accumulator. The time increase with *weave* shows the limits of search-based approaches.

6 Conclusions and Further Work

Based on a uniform description of some well-known IP systems and as result of our empirical evaluation of IP systems on a set of representative sample problems, we could show that the analytical approach of IGOR II is highly promising. It can induce a large scope of recursive programs, including mutual recursion and incorporates a straightforward technique for function invention. Background knowledge, in form of example equations, can be included in the inference process in a natural way. As consequence of IGOR II's generalisation principle, induced programs are guaranteed to terminate and to be the least generalisations. Although construction of hypotheses is not restricted by some greedy heuristics, induction is highly time efficient. Furthermore, it needs only a small set of positive I/O examples together with the data type specification of the target function and no further information such as schemes.

The most challenging problem will be to allow function invention for the outmost function without prior definition of the positions of recursive calls and to include the introduction of `let`-expressions and higher-order functions (e. g. map, reduce, filter).

References

1. Biermann, A.W. et al.: Automatic Program Construction Techniques. Free Press, NY (1984)
2. Quinlan, J.R.: Learning first-order definitions of functions. Journal of Artificial Intelligence Research **5** (1996) 139–161
3. Muggleton, S., Feng, C.: Efficient induction of logic programs. In: Proceedings of the 1st Conference on Algorithmic Learning Theory, Ohmsma, Tokyo, Japan (1990) 368–381
4. Muggleton, S.: Inverse entailment and Progol. New Generation Computing, Special issue on Inductive Logic Programming **13**(3-4) (1995) 245–286
5. Flener, P.: Inductive logic program synthesis with Dialogs. In Muggleton, S., ed.: Proceedings of the 6th International Workshop on ILP, Stockholm University (1996) 28–51
6. Hernández-Orallo, J., et al.: Inverse narrowing for the induction of functional logic programs. In Freire-Nistal, et al. , eds.: Joint Conference on Declarative Programming. (1998) 379–392
7. Kitzelmann, E., Schmid, U.: Inductive synthesis of functional programs: An explanation based generalization approach. Journal of Machine Learning Research **7** (2006) 429–454
8. Kitzelmann, E.: Data-driven induction of recursive functions from input/output-examples. In Kitzelmann, E., Schmid, U., eds.: Proceedings of the ECML/PKDD 2007 Workshop on Approaches and Applications of Inductive Programming (AAIP'07). (2007) 15–26
9. Olsson, R.J.: Inductive functional programming using incremental program transformation. Artificial Intelligence **74**(1) (1995) 55–83
10. Katayama, S.: Systematic search for λ-expressions. In: Trends in Functional Programming (2005) 111–126
11. Terese: Term Rewriting Systems. Volume 55 of Cambridge Tracts in Theoretical Computer Science. Cambridge Univ. Press (2003)
12. Baader, F., Nipkow, T.: Term Rewriting and All That. Cambridge Univ. Press, UK (1998)
13. Plotkin, G.: A further note on inductive generalization. In: Machine Intelligence. Vol. 6. Edinburgh Univ. Press (1971)