A Unifying Framework for Analysis and Evaluation of Inductive Programming Systems*

Martin Hofmann and **Emanuel Kitzelmann** and **Ute Schmid**
Faculty Information Systems and Applied Computer Science
University of Bamberg, Germany
{martin.hofmann, emanuel.kitzelmann, ute.schmid}@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 logic and inductive functional program synthesis. Then we characterise the several ILP systems which belong either to the most recently researched or currently to the most powerful IP systems 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.

Introduction

Inductive programming (IP) is concerned with the synthesis of programs or algorithms from incomplete specifications, such as input/output (I/O) examples. Focus is on the synthesis of *declarative*, i.e., logic, functional, or functional logic programs. Research in IP provides better insights in the cognitive skills of human programmers. Furthermore, powerful and efficient IP systems can enhance software systems in a variety of domains—such as automated theorem proving and planning—and offer novel approaches to knowledge based software engineering and model driven software development, as well as end user programming support in the XSL domain (Hofmann 2007). 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 addressed inductive synthesis of functional programs from small sets of positive I/O examples only (Biermann et al. 1984). One of the most influential classical systems was THESYS (Summers 1977) which synthesised linear recursive LISP programs by rewriting I/O pairs into traces and folding of traces based on recurrence detection. Currently, induction of functional programs is covered by the analytical approaches IGOR I (Kitzelmann and Schmid 2006), and IGOR II (Kitzelmann 2007) and by the evolutionary/search-based approaches ADATE (Olsson 1995) and MAGICHASKELLER (Katayama 2005). Analytical approaches work example-driven, so the structure of the given I/O pairs is used to guide the construction of generalised programs. Search-based approaches first construct one or more hypothetical programs, evaluate them against the I/O examples and then work on with the most promising hypotheses.

In the last decade, some inductive logic programming (ILP) systems were presented with focus on learning recursive logic *programs* in contrast to learning classifiers: FFOIL (Quinlan 1996), GOLEM (Muggleton and Feng 1990), PROGOL (Muggleton 1995), and DIALOGS-II (Flener 1996). Synthesis of functional logic programs is covered by the system FLIP (Hernández-Orallo et al. 1998).

IP can be viewed as a special branch of machine learning because programs are constructed by inductive generalisation from examples. Therefore, as for classification learning, each approach can be characterised by its restriction and preference bias (Mitchell 1997). However, IP approaches cannot be evaluated with respect to some covering measure or generalisation error since (recursive) programs must treat *all* I/O examples correctly to be an acceptable hypothesis.

Current IP systems not only differ with respect to the target language and the synthesis strategies but also with respect to the information which has to or can be presented and the scope of programs which can be synthesised. Unfortunately, up to now there is neither a systematic empirical evaluation of IP systems nor a general vocabulary for describing and comparing the different approaches in a systematic way (see (Hofmann, Kitzelmann, and Schmid 2008) for a preliminary evaluation of some systems and (Flener and Yilmaz 1999) for a treatment of ILP systems). We believe that both is necessary for further progress in the field: Only if the relative strengths and weaknesses of the different systems become transparent, more powerful and efficient approaches can be designed by exploiting the strengths of the given approaches and tackling their weaknesses.

We first present conditional combinatory term rewriting as a framework for describing IP systems. Afterwards, several systems are characterised and compared in this framework and their performance is evaluated on a set of representative sample problems and shows the strength of IGOR II. We conclude with ideas on further research.

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

A Unified Framework for IP

Conditional Constructor Systems

We shortly introduce term rewriting and conditional constructor systems as, e.g., described in (Baader and Nipkow 1998; Terese 2003). For a signature, i.e., a set of function symbols Σ and a set of variables $\mathcal{X}$ we denote the set of all terms over Σ and $\mathcal{X}$ 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 respectively. We uniformly represent an induced program in a functional style as a set R of recursive equations (rules) over a signature Σ. The equations are applied as simplification (or rewrite) rules (as known from functional programming) from left to right, i.e., they form a *term rewriting system*. The lefthand side (lhs) l of a rewrite rule $l \to r$ has the form $F(p_1, \ldots, p_n)$, called *function head*, where $F \in \mathcal{F}$ is the name of the function implemented by (amongst others) this rule, i. e., a defined function symbol, and the $p_i \in \mathcal{T}_\mathcal{C}(\mathcal{X})$ are built up from constructors and variables only. We call terms from $\mathcal{T}_\mathcal{C}(\mathcal{X})$ *constructor terms*. The sequence of the p_i is called *pattern*. This format of rules or equations is known as *pattern matching* in functional languages such as HASKELL. In the term rewriting setting, a TRS with this form is called *constructor (term rewriting) system (CS)*. 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*.

Each rewrite rule may be augmented with a *condition* that must be met to apply the *conditional* rule. A term rewriting system or constructor system is called conditional term rewriting system or *conditional constructor system (CCS)* respectively if it contains at least one conditional rule. A condition is an ordered conjunction of equality constraints $v_i = u_i$ with $v_i, u_i \in \mathcal{T}_\Sigma(\mathcal{X})$. Each u_i must be grounded if the lhs of the rule is instantiated and if all equalities $v_j = u_j$ with $j < i$ evaluate to true, then u_i evaluates to some ground normal form. For the v_i must hold (i) either the same as for the u_i or (ii) v_i may contain unbound variables but then it must be a constructor term. In the first case also v_i evaluates to some ground normal form and the equality evaluates to true if both normal forms are equal. In the second case the equality evaluates to true if v_i and the ground normal form of u_i unify. Then the free variables in v_i are bound and may be used in the following conjuncts and the rhs of the rule. We write conditional rules in the form: $l \to r \Leftarrow v_1 = u_1, \ldots, v_n = u_n$. Figure 1(1) shows an example. Rules without a condition are called *unconditional*. 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*. A CCS is *terminating* if all rewriting processes end up in a normal form. In order to implement functions the outputs are required to be unique for each particular input vector. This is the case if the TRS is *confluent*.

To lift a CCS into the higher-order context and extend it to a (conditional) combinatory rewrite system ((C)CRS) (Terese 2003) we introduce meta-variables $\mathcal{X}_M = X, Y, Z, \ldots$ such that $\mathcal{X} = \mathcal{X}_M \cup \mathcal{X}_\mathcal{T}$ with $\mathcal{X}_M \cap \mathcal{X}_\mathcal{T} = \emptyset$ where $\mathcal{X}_\mathcal{T}$ includes all first-order variables over terms. 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 occurrences 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 following term $map([A]square(A), cons(1, nil))$.

Target Languages in the CCRS Framework

To compare all systems under equal premises, the different occurrences of declarative languages are out into the CCRS framework[1]. Considering functional target languages, the underlying concepts are either based on abstract theories (equational theory (Hernández-Orallo et al. 1998), CS (Kitzelmann 2007)), or concrete functional languages (ML (Olsson 1995), HASKELL (Katayama 2005)). Applying the CCRS framework to IFP or IFLP systems is straight forward, since they all share the basic principles and functional semantics. This is in particular the case with IFLP, which provided the theoretical framework for FLIP. However contrarily to IFLP, we additionally want to qualify for expressing the basic constructs of functional languages in the CCRS framework and both apply it to existing systems for a well-founded analysis and evaluation.

In addition to pattern matching and the functional operational semantics of CS, CCS can express constructs as `if`-, `case`-, and `let`-*expressions* in a rewriting context. The *if-then* part of an `if`-expression can be modeled by a condition $v = u$ following case (i) in the previous section. 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(2) shows a CCRS for the HASKELL program containing a `let`-expression.

In the context of IP, only logic target programs where the output is uniquely determined by the input are considered. Such programs usually are expressed as "functional" predicates such as *multlast* in Fig. 1(3). Transforming Horn clauses containing functional predicates into CCSs is a generalisation of representing Horn clauses as conditional identities (Baader and Nipkow 1998).

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

[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.

(1) CCRS

```
mlast([])  -> []
mlast([A]) -> [A]
mlast([A,B|C]) -> [D,D|C]
  <= [D|C] = mlast([B|C])
```

(2) Functional (Haskell)

```
mlast([])      = []
mlast([A])     = [A]
mlast([A,B|C]) =
  let [D|C] = mlast([B|C])
  in [D,D|C]
```

(3) Logic (Prolog)

```
mlast([], []).
mlast([A], [A]).
mlast([A,B|C],[D,D|C]) :-
  mlast([B|C],[D|C]).
```

Figure 1: Equivalent programs of *multlast*

symbols of predefined functions that can be used for synthesis. These can either be built in or defined by the user in the background knowledge BK. $\mathcal{F}_I$ 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 $t_1,\ldots,t_n,r \in \mathcal{T}_\mathcal{C}(\mathcal{X}_T)$. The rules in E^- are interpreted as inequality constraints. BK is a finite set of conditional or unconditional 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 $F \in \mathcal{F}_B$, $t_1,\ldots,t_n \in \mathcal{T}_\mathcal{C}(\mathcal{X})$, and $r,u_i,v_i \in \mathcal{T}_B(\mathcal{X})$. Furthermore, it is requested that for each symbol $f \in \mathcal{F}_T$ ($\mathcal{F}_B$), there is at least one rule in R_T (R_B) with function name f.

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 $F \in \mathcal{F}$, $t_1,\ldots,t_n \in \mathcal{T}_\mathcal{C}(\mathcal{X})$, and $r,u_i,v_i \in \mathcal{T}(\mathcal{X})$, 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. To compute the output constructors, recursive calls, functions from the background knowledge, local variable declarations, and invented functions can be used. 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 the $BK$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* allows only a specific subset of the terms defined for u_i,v_i,l,r in a rule $F(t_1,\ldots,t_n) \rightarrow r \Leftarrow u_1 = v_1 \wedge \ldots \wedge u_n = v_n$, e.g., prohibiting nested or mutual recursion or demanding the rhs to follow a certain 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 synthesised by a specific system is optimal w. r. t. this ordering and satisfying completeness and consistency.

Table 1: Systems' characteristics summary

	$\mathcal{C}$	$\mathcal{F}_T$	$\mathcal{F}_B$	$\mathcal{F}_I$	E^+	E^-	BK	$\mathcal{X}_M$
ADATE	•	$\{\cdot\}$	•	•	•	•	•	$\emptyset$
FLIP	•	•	•	$\emptyset$	○	○,$\emptyset$	•	$\emptyset$
FFOIL	c	•	$\supset$	$\emptyset$	○	○,$\emptyset$	○	$\emptyset$
GOLEM	•	$\{\cdot\}$	•	$\emptyset$	○	○	•	$\emptyset$
IGOR I	•	$\{\cdot\}$	$\emptyset$	•	○	$\emptyset$	$\emptyset$	$\emptyset$
IGOR II	•	•	•	•	○	$\emptyset$	○	$\emptyset$
MAGH.	•	$\{\cdot\}$	•	$\emptyset$	•	•	•	○

• unrestricted / conditional rules ○ restricted / unconditional rules
$\{\cdot\}$ singleton set $\emptyset$ empty set
c constants $\supset$ built in predicates

Systems Description in the CCRS Framework

We will consider only FFOIL, GOLEM, FLIP, MAGIC-HASKELLER IGOR I and IGOR II. The prominent systems FFOIL and GOLEM shall provide a baseline, as representatives of ILP systems, against which the other systems can be compared. The others belong either to the most recent or currently to the most powerfull IP systems and attest the current focus of research on IFLP and IFP.

PROGOL and DIALOGS-II have been excluded, because they make heavily use of background knowledge which goes beyond the notion the other systems have. DIALOGS-II as an interactive system collects much more evidence which is not expressable in I/O examples, because it virtually allows for Horn clauses in E^+ and E^-. Similarly the mode declarations of PROLOG. To allow for learning programs (unlike learning classifiers), mode declarations specify positions of recursive calls and correct data type decompositions would be necessary, which makes the task of IP uninteresting.

Table 1 summarises the classification of the mentioned systems into the CCRS framework and Table 2 the systems' restriction bias and the expressiveness of the conditions in a rule. The following paragraphs sketching the covering and search strategy, the search space and the preference bias.

ADATE as an evolutionary computation system employs search techniques that are inspired by basic biological principles of evolution, like for instance mutation and crossover. Starting from a trivial initial program, offsprings are generated which are tested against the I/O examples and rated using criteria as time and memory usage as preference bias. Only the "fittest" programs are developed further as the search progresses until eventually one program covers all I/O examples and it is aborted by the user. In this way, ADATE searches the space of all programs, in a subset of ML, globally, covering the examples via a generate and test.

FFOIL (Quinlan 1996) is an early representative of a functional top-down learning system. Starting with an empty set of conditions, it continues adding *gainful* literals to a rule until it covers no negative examples anymore. Which conditions to add is determined by a information-based heuristic called *foil gain*. It favours literals with a high information gain, i e., which discriminate notably between positive and

negative evidence when added to a rule. All examples explained by this rule are removed from E^+ and another rule is learnt until E^+ is empty. If no candidate literals are gainfull enough, all so called $determinate$ literals are added. A determinate literal does not deteriorate the foil gain of the current rule, but introduces new variables. So FFOIL traverses the space of Horn clauses heuristically lead by the *foil gain* following a greedy sequential covering strategy.

FLIP (Hernández-Orallo et al. 1998) is a representative of the IFLP approach. It starts from all possible consistent restricted generalisations (CRG) deriveable from the positive example equations. A restricted generalisation (RG) is a generalisation without introducing new variables on the rhs of the equation. A RG is consistent if it does not cover any negative example when interpreted as a rewrite rule.

Informally, narrowing combines resolution from logic programming and term reduction from functional programming. FLIP uses its inverse, similar as inverse resolution, called inverse narrowing to solve the induction problem.

FLIP's core algorithm is based CRG and inverse narrowing which induce a space of hypothesis programs. It is searched heuristically using a combination of minimum description length and coverage of positive examples as preference bias, following a sequential covering strategy.

GOLEM (Muggleton and Feng 1990) uses a bottom-up, or example driven approach based on Plotkin's framework of relative least general generalisation (*rlgg*) (Plotkin 1971). This avoids searching a large hypothesis space for consistent hypothesis as, e.g, FFOIL, but rather constructs a unique clause covering a subset of the provided examples relative to the given 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 example, the clause is further generalised. This generalisation 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. 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 rlggs of two examples.

MAGICHASKELLER (Katayama 2005) 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). 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.

IGOR I is a modern extension of the seminal THESYS system (Summers 1977) adopting its two-step approach. In a first step I/O examples are rewritten to traces which explain each output given the respective input based on a datatype theory. All traces are integrated into one conditional expression computing exactly the output for the inputs as given in the examples as a non-recursive program. In a second step, this initial program is generalised into recursive equations by searching for syntactic regularities in this term.

Synthesis is still restricted to structural problems, where only the structure of the arguments matters, but not their contents, such as in list reversing (and contrary to *member*). Nevertheless, the scope of synthesisable programs is considerably larger. For instance, tree-recursive functions and functions with hidden parameters can be induced. Most notably, programs consisting of a calling function and an arbitrary set of further recursive functions can be induced.

IGOR II is, contrarily to others, specialised to learn *recursive programs*. 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, still remaining tractable even for relative complex programs because construction of hypotheses is data-driven. IGOR II combines analytical program synthesis with search.

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) (Plotkin 1971) 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: (1) Partitioning of the inputs by replacing one pattern by a set of disjoint more specific patterns or by adding a predicate to the condition. (2) 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. (3) Replacing the rhs *subterms* in which unbound variables occur by a call to new subprograms. In cases (2) and (3) auxiliary functions are invented, abducing I/O-examples for them.

Forecast As far as one can generally already say, the "old" systems GOLEM and FFOIL are hampered by their greedy sequential covering strategy. Consequently, partial rules are never revised and lead to local optima, and thus losing dependencies between rules. This is especially the case with FFOIL learning predicates or finding a separate rule for the

Table 2: Overview of systems' restriction bias

	$F(i_1,\ldots,i_n)/lhs$	rhs	v_i/u_i
ADATE	$i_i \in \mathcal{X}_\mathcal{T}$	$\mathcal{T}_\mathcal{C}(\mathcal{X}_\mathcal{T})$	ilc
FLIP	CRG of E^+	inverse narrowing of CRG of E^+	—
FFOIL	$i_i \in \mathcal{X}_\mathcal{T}$	$\mathcal{T}_\mathcal{C}(\mathcal{X}_\mathcal{T}) \cup \{true, false\}$	il
GOLEM	$i_i \in \mathcal{T}_\mathcal{C}(\mathcal{X}_\mathcal{T})$	$\mathcal{T}_\mathcal{C}(\mathcal{X}_\mathcal{T}) \cup \{true, false\}$	ilc
IGOR I	$i_i \in \mathcal{T}_\mathcal{C}(\mathcal{X}_\mathcal{T})$	$\mathcal{T}_\mathcal{C}(\mathcal{X}_\mathcal{T})$	—
IGOR II	$i_i \in \mathcal{T}_\mathcal{C}(\mathcal{X}_\mathcal{T})$	$\mathcal{T}_\mathcal{C}(\mathcal{X}_\mathcal{T})$	i
MAGH.	composition of functions from BK, higher-order via paramorphisms		

i `if` l `let` c `case`

base case, where the foil gain may be misleading. FFOIL is heavily biased towards constructing the next clause to cover the most frequent function value in the remaining tuples.

Where FFOIL has only very general lhss, GOLEM and FLIP try to be more flexible in discriminating the inputs there, but not effective enough. Random sampling is too unreliable for an optimal partition of the inputs, especially for more complex data structures or programs with many rules.

FLIP generates the lhss using the CRG on basis of common subterms on the lhs and rhs of the examples. Necessary function-carrying subterms on both sides may be generalised and the lhss may tend to be overly general. Also, neither overlap of the lhss is prohibited, nor are the rules ordered. Consequently, one input may be matched by several rules resulting in a wrong output. The rhs are constructed via inverse narrowing inducing a huge search space, so with increasing complexity of examples the search becomes more and more intractable when relying solely on heuristics.

MAGICHASKELLER is a promising example of including higher-order features into IP and shows how functions like map or $filter$ can be applied effectively, when used advisedly, as some kind of program pattern or scheme. Nevertheless, MAGICHASKELLER and ADATE 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.

IGOR I and IGOR II will have problems were many examples are required (*mult/add* & *allodds*), but will be in other respects very fast.

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* replaces all elements with the last and *shiftr* makes a right-shift of all elements in a list. Therefore it is necessary to access the last element for further computations. Further functions are *lasts* which applies *last* on a list of lists, *isort* which is insertion-sort, *allodds* checks for odd numbers, 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 and need more than two rules, *lasts*, *multlast*, *isort*, *reverse*, *mult/add*, *allodds* suggest to use function invention, but only *reverse* is explicitly only solvable with. *lasts* and *allodds* also split up in more than two rules if no function invention is applied. To solve *member* pattern matching is required, because equality is not provided. The function *weave* is especially interesting, because it demands either for iterating over more than one argument resulting in more than one base case, or swapping the arguments at each recursive call.

Table 3: Systems' runtimes on different problems in seconds

Problems	ADATE	FFOIL	FLIP	GOLEM	IGOR I	IGOR II	MAGH.
lasts	365.62	$0.7^{\perp}$	×	1.062	0.051	5.695	19.43
last	1.0	0.1	0.020	< 0.001	0.005	0.007	0.01
member	2.11	$0.1^{\perp}$	17.868	0.033	—	0.152	1.07
odd/even	—	$< 0.1^{\perp}$	0.130	—	—	0.019	—
multlast	5.69	< 0.1	$448.900^{\perp}$	< 0.001	0.331	0.023	0.30
isort	83.41	×	×	0.714	—	0.105	0.01
reverse	30.24	—	—	—	0.324	0.103	0.08
weave	27.11	0.2	$134.240^{\perp}$	0.266	$0.001^{\perp}$	0.022	⊙
shiftr	20.14	$< 0.1^{\perp}$	$448.550^{\perp}$	0.298	0.041	0.127	157.32
mult/add	—	$8.1^{\perp}$	×	—	—	⊙	—
allodds	466.86	$0.1^{\perp}$	×	$0.016^{\perp}$	$0.015^{\perp}$	⊙	×

— not tested × stack overflow ⊙ time out ⊥ wrong

Because FFOIL and GOLEM usually perform better with more examples, whereas FLIP, MAGICHASKELLER and IGOR II do better with less, each system got as much examples as necessary up to certain complexity, but then exhaustively, 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. FLIP needed an additional function if to relate the *insert* function with the $<$. For all systems except FLIP and MAGICHASKELLER the definition of the background knowledge was extensional. 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. Note that we did not test a system with a problem which it per se cannot solve due to its restriction bias. This is indicated with '—' instead of a runtime. A timeout after ten minutes is indicated with ⊙. Table 3 shows the runtimes of the different systems on the example problems.

All tests have been conducted under Ubuntu 7.10 on a Intel Dual Core 2.33 GHz with 4GB memory. Following versions have been used: FLIP v0.7, FFOIL 1.0, GOLEM version of August 1992, the latest version of IGOR I, IGOR II version 2.2, ADATE version 0.50 and MAGICHASKELLER 0.8.3-1. The input files can be obtained under `http://www.cogsys.wiai.uni-bamberg.de/effalip/download.html`.

As the empirical results affirm the previous considerations, FFOIL fails with nearly all problems, and *multlast* was only solved with more examples. This can easily be explained with the greedy foil gain and a sequential cover-

ing strategy. Due to GOLEM's random sampling, the best result of ten runs have been chosen.

Long run times and the failures of FLIP testify for the intractable search space induced by the inverse narrowing operator. Wrong programs are due to overlapping lhss and its generalisation strategy of the inputs. Despite its randomisation, GOLEM overtrumps FLIP due to its capability of introducing `let`-expressions (cf. *multlast*). IGOR I and IGOR II need function invention to balance this weak-point.

On *reverse* and *isort* MAGICHASKELLER demonstrates the power of higher-order functions. Although it does not invent auxiliary functions, *reverse* was solved using its paramorphism over lists which provides some kind of accumulator. The paramorphisms are also the reason why MAGICHASKELLER fails with *weave*, since swapping the inputs with each recursive call does not fit in the schema induced by the paramorphism for lists.

These results showed, that the field of IP is not yet fully researched, and there are improvements discovered since the "golden times" of ILP and still to be discovered. Basically, ILP systems need a vast number of I/O examples which is usually impractical for a normal user to provide. Contrarily, IFP systems get along with much less examples but are still much more reliable in their results than ILP systems. Among the IFP it is significant analytic approaches rule out ADATE or MAGICHASKELLER on more complex problems where the search space increases. Also the ability of generically inventing functions is a big advantage.

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. IGOR II can induce a large scope of recursive programs, including mutual recursion using a straight-forward technique for function invention. Background knowledge can be provided 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, IGOR II works with minimal information provided by the user. 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.

Due to the nature of specification by example, IP systems in general, cannot scale up to complex software development problems. However, integrating IP principles in the software engineering process might relieve developers from specifying or coding specific functions. Although IGOR II cannot tackle problems of complex size, it can tackle problems which are intellectually complex and therefore might offer support to inexperienced programmers.

Function invention for the outmost function without prior definition of the positions of recursive calls will be our greatest future challenge. Furthermore, we plan to include the introduction of `let`-expressions and higher-order functions (such as map, reduce, filter).

References

Baader, F., and Nipkow, T. 1998. *Term Rewriting and All That*. United Kingdom: Cambridge University Press.

Biermann, A. W.; Kodratoff, Y.; and Guiho, G. 1984. *Automatic Program Construction Techniques*. NY, Free Press.

Flener, P., and Yilmaz, S. 1999. Inductive synthesis of recursive logic programs: Achievements and prospects. *J. Log. Program.* 41(2-3):141–195.

Flener, P. 1996. Inductive logic program synthesis with Dialogs. In Muggleton, S., ed., *Proc. of the 6th International Workshop on ILP*, 28–51. Stockholm University

Hernández-Orallo, J., and Ramírez-Quintana, M. J. 1998. Inverse narrowing for the induction of functional logic programs. In Freire-Nistal, et al., eds., *Joint Conference on Declarative Programming*, 379–392.

Hofmann, M.; Kitzelmann, E.; and Schmid, U. 2008. Analysis and evaluation of inductive programming systems in a higher-order framework. In *31st German Conference on Artificial Intelligence*, LNAI. Springer-Verlag.

Hofmann, M. 2007. *Automatic Construction of XSL Templates – An Inductive Programming Approach*. VDM Verlag, Saarbrücken.

Katayama, S. 2005. Systematic search for lambda expressions. In *Trends in Functional Programming*, 111–126.

Kitzelmann, E., and Schmid, U. 2006. Inductive synthesis of functional programs: An explanation based generalization approach. *J. of ML Research* 7:429–454.

Kitzelmann, E. 2007. Data-driven induction of recursive functions from input/output-examples. In Kitzelmann, E., and Schmid, U., eds., *Proc. of the ECML/PKDD 2007 Workshop on Approaches and Applications of Inductive Programming*, 15–26.

Mitchell, T. M. 1997. *Machine Learning*. McGraw-Hill Higher Education.

Muggleton, S., and Feng, C. 1990. Efficient induction of logic programs. In *Proc. of the 1st Conference on Algorithmic Learning Theory*, 368–381. Ohmsma, Tokyo, Japan.

Muggleton, S. 1995. Inverse entailment and Progol. *New Generation Computing, Special issue on Inductive Logic Programming* 13(3-4):245–286.

Olsson, R. J. 1995. Inductive functional programming using incremental program transformation. *Artificial Intelligence* 74(1):55–83.

Plotkin, G. 1971. A further note on inductive generalization. In *Machine Intelligence*, vol. 6. Edinb. Univ. Press.

Quinlan, J. R., and Cameron-Jones, R. M. 1993. FOIL: A midterm report. In *Machine Learning: ECML-93, Proc.*, vol. 667, 3–20. Springer-Verlag.

Quinlan, J. R. 1996. Learning first-order definitions of functions. *Journal of AI Research* 5:139–161.

Summers, P. D. 1977. A methodology for LISP program construction from examples. *Journal ACM* 24:162–175.

Terese. 2003. *Term Rewriting Systems*, vol. 55 of *Cambridge Tracts in Theoretical Computer Science*. Cambridge University Press.