Inductive Programming Meets the Real World

Sumit Gulwani José Hernández-Orallo Emanuel Kitzelmann

Stephen H. Muggleton Ute Schmid Benjamin Zorn

Since most end users lack programming skills they often spend considerable time and effort performing tedious and repetitive tasks such as capitalizing a column of names manually. Inductive Programming has a long research tradition and recent developments demonstrate it can liberate users from many tasks of this kind.

Key insights

- Real-world applications emerge with spreadsheet tools, scripting, and intelligent program tutors.
- Learning from few examples is possible because users and systems share the same background knowledge.
- Search is guided by domain-specific languages and the use of higher-order knowledge.

Much of the world's population use computers for everyday tasks, but most fail to benefit from the power of computation due to their inability to program. Most crucially, users often have to perform repetitive actions manually because they are not able to use the macro languages which are available for many application programs. Recently, a first mass-market product was presented in the form of the Flash Fill feature in Microsoft Excel 2013. Flash Fill allows end users to automatically generate string processing programs for spreadsheets from one or more user-provided examples. Flash Fill is able to learn a large variety of quite complex programs from only a few examples because of incorporation of inductive programming methods.

Inductive Programming (IP) is an inter-disciplinary domain of research in computer science, artificial intelligence, and cognitive science that studies the automatic synthesis of computer programs from examples and background knowledge. IP developed from research on inductive program synthesis, now called *inductive functional programming (IFP)*, and from inductive inference techniques using logic, nowadays termed *inductive logic programming (ILP)*. IFP addresses the synthesis of recursive functional programs generalized from regularities detected in (traces of) input/output examples [42, 20] using generate-and-test approaches based on evolutionary [35, 28, 36] or systematic [17, 29] search or data-driven analytical approaches [39, 6, 18, 11, 37, 24]. Its development is complementary to efforts in synthesizing programs from complete specifications using deductive and formal methods [8].

ILP originated from research on induction in a logical framework [40, 31] with influence from artificial intelligence, machine learning and relational databases. It is a mature area with its own theory, implementations, and applications and recently celebrated the 20th anniversary [34] of its inception as an annual series of international conferences.

Over the last decade Inductive Programming has attracted a series of international workshops. Recent surveys [7, 19, 14] reflect the wide variety of implementations and applications in this area.

In the domain of end-user programming, programming by demonstration approaches were proposed which support the learning of small routines from observing the input behavior of users [5, 25, 23]. The above-mentioned Microsoft Excel's sub-system Flash Fill provides an impressive illustration that program synthesis methods developed in IP can be successfully applied to gain more flexibility and power for end-user programming [9, 11]. Further applications are being realized for other desktop applications (for example, PowerShell scripting in the ConvertFrom-String cmdlet [2]) as well as for special-purpose devices such as home robots and smartphones.

In this paper, several of these current applications are presented. We contrast the specific characteristics of IP with those of typical machine learning approaches and we show how IP is related to cognitive models of human inductive learning. We finally discuss recent techniques —such as use of domain-specific languages and meta-level learning— that widen the scope and power of IP and discuss new challenges.

1. REAL-WORLD APPLICATIONS

Originally, IP was applied to synthesizing functional or logic programs for general purpose tasks such as manipulating data structures (e.g., sorting or reversing a list). These investigations showed that small programs could be synthesized from a few input/output examples. The recent IT revolution has created real-world opportunities for such techniques. Most of today's large number of computer users are non-programmers and are limited to being passive consumers of the software that is made available to them. IP can empower such users to more effectively leverage computers for automating their daily repetitive tasks. We discuss below some such opportunities, especially in the areas of End-user Programming and Education.

1.1 End-User Programming

End users of computational devices often need to create *small* (and perhaps one-off) scripts to automate repetitive tasks. These users can easily specify their intent using *examples*, making IP a great fit. For instance, consider the domain of data manipulation. Documents of various types, such as text/log files, spreadsheets, and webpages, offer their

creators great flexibility in storing and organizing hierarchical data by combining presentation and formatting with the underlying data model. However, this makes it extremely hard to extract the underlying data for common tasks such as data processing, querying, altering the presentation view, or transforming data to another storage format.

Existing programmatic solutions to manipulating data (such as Excel macro language, regular expression libraries inside Perl/Python, and JQuery library for Javascript) have three key limitations. First, the solutions are domain-specific and require expertise in different technologies for different document types. Second, they require understanding of the entire underlying document structure including the data fields the end-user is not interested in (some of which may not even be visible in the presentation layer of the document). Third, and most significantly, they require knowledge of programming. As a result, users have to resort to manual copy-paste, which is both time consuming and error prone.

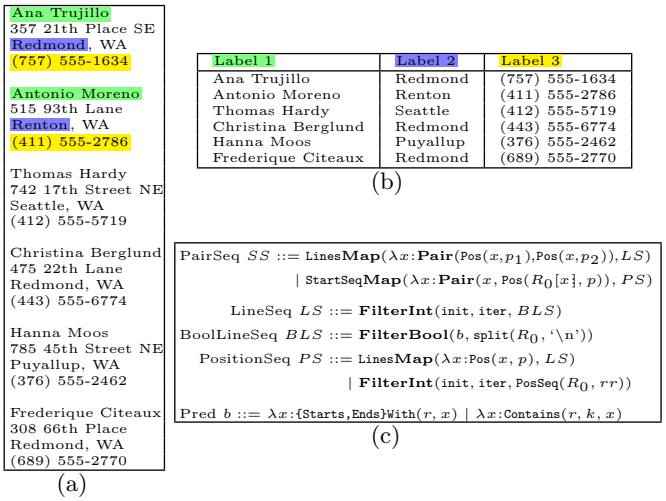

Figure 1: FlashExtract [24]: A framework for extracting data from documents of various kinds such as text files and web pages using examples. Once the user highlights one or two examples of each field in the textfile in (a), FlashExtract extracts more such instances and arranges them in a structured format in the table in (b). This is enabled by synthesis of a program in the domain-specific language (DSL) in (c) that is consistent with the examples in (a) followed by execution of that program on the textfile in (a).

Inductive synthesis can help out with a variety of data manipulation tasks. These include: (a) Extracting data from semi-structured documents including text files, web pages, and spreadsheets [24] (as in Fig. 1). (b) Transformation of atomic data types such as strings [9] (as in Fig. 2) or numbers. Transformation of composite data types such as tables [11] and XML [36]. (c) Formatting data [37]. Combining these technologies in a pipeline of extraction, transformation, and formatting can allow end users to perform sophisticated data manipulation tasks.

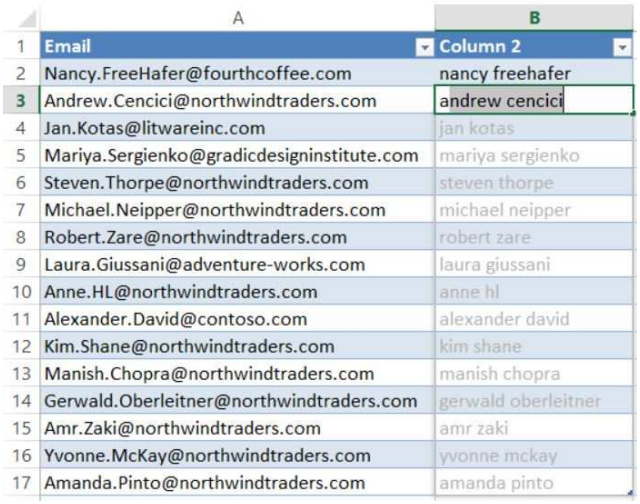

Figure 2: Flash Fill [9]: An Excel 2013 feature that automates repetitive string transformations using examples. Once the user performs one instance of the desired transformation (row 2, col. B) and proceeds to transforming another instance (row 3, col. B), Flash Fill learns a program Concatenate(ToLower(Substring(v,WordToken,1)), " ", ToLower(SubString(v,WordToken,2))), which extracts the first two words in input string v (col. A), converts them to lowercase, and concatenates them separated by a space character, to automate the repetitive task.

Example Problem ⇓	$\frac{\sin A}{1+\cos A} + \frac{1+\cos A}{\sin A} = 2\csc A$
Generalized Problem Template ⇓	$\frac{T_1 A}{1 \pm T_2 A} + \frac{1 \pm T_3 A}{T_4 A} = 2\ T_5 A$ where $T_i \in \{\cos, \sin, \tan, \cot, \sec, \csc\}$
New Similar Problems	$\frac{\cos A}{1-\sin A} + \frac{1-\sin A}{\cos A} = 2\tan A$ $\frac{\cos A}{1+\sin A} + \frac{1+\sin A}{\cos A} = 2\sec A$ $\frac{\cot A}{1+\csc A} + \frac{1+\csc A}{\cot A} = 2\sec A$ $\frac{\tan A}{1+\sec A} + \frac{1+\sec A}{\tan A} = 2\csc A$ $\frac{\sin A}{1-\cos A} + \frac{1-\cos A}{\sin A} = 2\cot A$

Figure 3: Problem generation for algebraic proof problems involving identities over analytic functions. A given problem is generalized into a template and valid instantiations are found by testing on random values for free variables.

1.2 Computer-aided Education

Human learning and communication is often structured around examples —be it a student trying to understand or master a certain concept using examples, or be it a teacher trying to understand a student's misconceptions or provide feedback using example behaviors. Example-based reasoning techniques developed in the inductive synthesis community can help automate several repetitive and structured tasks in education including problem generation, solution generation, and feedback generation [10]. These tasks can

be automated for a wide variety of STEM subject domains including logic, automata theory, programming, arithmetic, algebra, and geometry. For instance, Fig. 3 shows the output of an inductive synthesis technique for generating algebraic proof problems similar to a given example problem.

1.3 Future opportunities

We have described important real world applications of IP. We believe there are many other domains to which IP can and will be applied in the near future. Any domain in which a set of high-level abstractions already exists is a strong candidate for IP. For example, the If This Then That (IFTTT) service (`http://ifttt.com/`), which allows end users to express small rule-based programs using triggers and actions, is an excellent candidate for application of IP. IFTTT programs connect triggers (such as *I was tagged in a photo*) with actions (send an email) over specific channels such as Facebook. In such a domain, IP can be used to learn programs from examples of a user doing the task. For instance, it can learn a program to send a text message every time a smartphone user leaves work for home. Looking further ahead, automatically building robot strategies from user provided examples [31] is a promising new direction for IP.

As the frameworks to build IP-based solutions mature, including meta-synthesis frameworks that simplify the process of building synthesizers (see section 3.2), it will become easier for developers to create new IP-empowered applications. Currently developers must 1) design a domain-specific language that encodes domain knowledge suitable for the task, 2) capture user intent by extracting examples, and 3) search the large space of programs represented by the domain-specific language (DSL) to find those that satisfy the constraints established by the examples, and then rank them to find the one likely intended by the user. In the future, just as compiler generation frameworks such as lex and yacc simplify compiler development, we believe IP frameworks will simplify the synthesis problem.

2. IP VS. MACHINE LEARNING

IP is concerned about making machines learn programs automatically and can hence be considered another machine learning paradigm. So, what is distinctive about inductive programming? Table 1 outlines a series of differences, some of which we discuss below.

We will also use a running example to indicate some of the features about IP. Fig. 4 shows an illustrative application where the goal is to identify repetitive patterns and rules about a user's personal contacts. The IP system learns an easy rule stating that the user's boss must be added to her circles and a more complex one that states that any person who is married to a family member should also be added to her circles. There we can see some of the distinctive features of IP systems, such as the small number of examples, the kind and source of data, the role of background knowledge, the interaction and feedback from the user, the use of a common declarative language, the use of recursion, and the comprehensibility and expressiveness of the learned patterns. Though machine learning and IP are quite complementary, they can also work well together. For instance, if IP generates multiple programs that are consistent with the provided examples, then machine learning can be used to rank such programs [12].

2.1 Small data

As collecting and storing data is becoming cheaper, it is easy to gain the impression that the only interesting datasets nowadays involve *big data*. However, datasets from a single user's interaction with whatever kind of device are usually quite small, such as the amount of data gathered about a person's agenda, as shown at the top of Fig. 4.

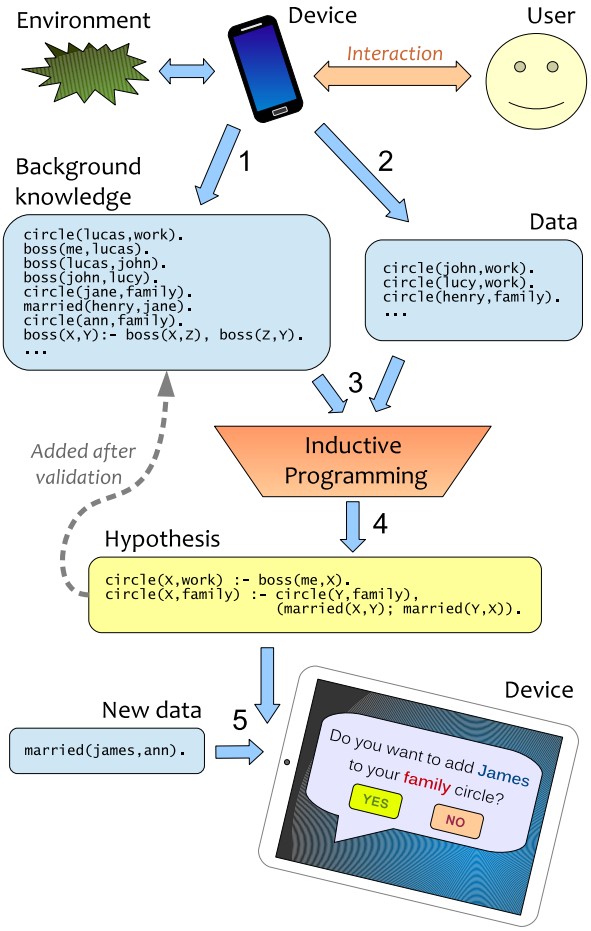

Figure 4: An example of the interaction with an IP system and the key role of the background knowledge. A user interacts with several devices about his/her agenda and contact groups. The system gathers a collection of facts, rules and operators into its background knowledge. From this background knowledge and a few examples, the system is able to infer new rules that —once validated— can be used to enlarge the background knowledge and also to make recommendations or suggest choices for the user.

It is well known that learning from small numbers of examples is more difficult and unreliable than learning from lots of data. The fewer examples we have, the more prone we are to overfitting, especially with expressive languages. IP is particularly useful when the number of examples is small but the hypothesis space is large (Turing-complete).

2.2 Declarative representation

Most (statistical) machine learning techniques are based on probabilities, distances, weights, kernels, matrices, etc. None of these approaches, except for techniques based on (propositional) decision trees and rules, are *declarative*, i.e.,

Table 1: A simplified comparison between Inductive Programming and other machine learning paradigms

	Inductive Programming	**Other Machine Learning Paradigms**
Number of examples	Small.	Large, e.g. *big data*.
Kind of data	Relational, constructor-based datatypes.	Flat tables, sequential data, textual data, etc.
Data source	Human experts, software applications, HCI, etc.	Transactional databases, Internet, sensors (IoT), etc.
Hypothesis language	Declarative: general programming languages or domain-specific languages.	Linear, non-linear, distance-based, kernel-based, rule-based, probabilistic, etc.
Search strategy	Refinement, abstraction operators, brute-force.	Gradient-descent, data partition, covering, instance-based, etc.
Representation learning	Higher-order and predicate/function invention.	Deep learning and feature learning.
Pattern comprehensibility	Common.	Uncommon.
Pattern expressiveness	Usually recursive, even Turing-complete.	Feature-value, not Turing-complete.
Learning bias	Using background knowledge and constraints.	Using prior distributions, parameters and features.
Evaluation	Diverse criteria, including simplicity, comprehensibility and time/space complexity.	Oriented to error (or loss) minimisation.
Validation	Code inspection, divide-and-conquer debugging, background knowledge consistency.	Statistical reasoning (only a few techniques are locally inspectable).

expressed as potentially comprehensible rules. Hence, another distinctive feature of IP is that it uses a rich symbolic representation, as hypotheses are usually *declarative* programs.

The declarative approach permits the use of a single language to represent background knowledge, examples and hypotheses, as shown in Fig. 4. Apart from the accessibility of one single language for the (end-)user, knowledge can be inspected, revised and integrated with other sources of knowledge much more easily. As a result, incremental, cumulative or life-long learning becomes easier [14]. For instance, NELL (Never-Ending Language Learner) [3] uses an ILP algorithm that learns probabilistic Horn clauses.

Nowadays, many languages in IP are *hybrid* such as functional logic programming languages, logic programming with types and higher-order constructs, constraints, probabilities, etc. The logic (ILP) vs functional (IFP) debate has also been surpassed recently by the breakthrough of domain-specific languages (DSL), which are usually better suited for the application at hand, as we will discuss in Section 3.1.

2.3 Refinement and abstraction

Another particular issue about IP is the way the hypothesis space is arranged by properly combining several inference mechanisms such as deduction, abduction and induction. Many early IP operators were inversions of deduction operators, leading to bottom-up and top-down approaches, where generalization and specialization operators, respectively, are used [34]. More generally, *refinement* and *abstraction* operators, including the use of higher-order functions, predicate and function invention, can be defined according to the operational semantics of the language.

This configures many levels between merely extensional facts and more intensional knowledge, leading to a hierarchical structure. Actually, the use of the same representation language for facts, background knowledge and hypotheses, as illustrated in Fig. 4, facilitates this hierarchy.

2.4 Deep knowledge

Because of the abstraction mechanisms and the use of background knowledge, IP considers learning as a knowledge acquisition process. In Fig. 4, for instance, inductive programming has access to some information about contact groups as well as relationships between the contacts (such as family bonds or work hierarchies). Such knowledge is known as *background knowledge* and works as a powerful explicit bias to reduce the search space and to find the right level of generalization.

Knowledge can be considered *deep* if it references lower level definitions, including recursively referencing itself. Representation of such structured and deep knowledge is achieved by programming languages that feature variables, rich operational semantics and, most especially, *recursion*. Recursion is a key issue in inductive programming [42, 31, 39, 20]. Note that both the background knowledge and the new hypothesis in the example of Fig. 4 are recursive.

This is in contrast to other machine learning approaches where background knowledge has only the form of prior distributions, probabilities or features. The difference is also significant with other non-symbolic approaches to *deep learning* [1], a new approach in machine learning where more complex models and features are also built in a hierarchical way, but data, knowledge and bias are represented differently.

2.5 Purpose and evaluation

In other machine learning approaches, hypotheses are measured by different metrics accounting for a degree of error. The purpose of IP is not just to maximize some particular error metric, but to find meaningful programs that are operational, according to the purpose of the IP application. This usually implies that they have to be consistent with most of (if not all) the data but also with the background knowledge and other possible constraints. Also, as hypotheses are declarative (and possibly recursive), the evaluation is more diverse, including criteria such as simplicity, comprehensibility, coherence and time/space complexity.

3. RECENT TECHNIQUES

IP is essentially a search problem, and can benefit from techniques developed in various communities. We present below certain classes of techniques used in recent IP work.

3.1 DSL synthesizers

Domain-specific languages (DSL) have been introduced in the IP scenario under the following methodology:

1. Problem Definition: Identify a vertical domain of tasks and collect common scenarios by studying help forums and conducting user studies.
2. Domain-specific language (DSL): Design a DSL that is expressive enough to capture several real-world tasks in the domain, but also restricted enough to enable efficient learning from examples. Fig. 1(c) describes one such DSL for extracting data from textfiles. (The full version of this DSL along with its semantics is described in [24].) This DSL allows for extracting a sequence of substrings using composition of filter and map operations.
3. Synthesis Algorithm: Most of these algorithms work by systematically reducing the problem to the synthesis of sub-expressions of the original expression (by translating the examples for the expression to the examples for the sub-expressions). These algorithms typically end up computing a set of DSL programs.
4. Ranking: Rank the various programs returned by the synthesizer perhaps using machine learning techniques.

The above methodology has been applied to various domains including the transformation of syntactic strings [9, 30], semantic strings, numbers, and tables [11].

3.2 Meta-synthesis frameworks

Domain-specific synthesizers (as opposed to general purpose synthesizers) offer several advantages related to efficiency (i.e., the ability to synthesize programs quickly) and ranking (i.e., the ability to synthesize intended programs from fewer examples). However, the design and development of a domain-specific synthesizer is a non-trivial process requiring critical domain insights and implementation effort. Furthermore, any changes to the DSL require making non-trivial changes to the synthesizer.

A meta-synthesis framework allows easy development of synthesizers for a related family of DSLs that are built using the same core set of combinators. Building such a framework involves the following steps:

1. Identify a family of vertical task domains which allow a common user interaction model.
2. Design an algebra for DSLs. A DSL is an ordered set of grammar rules (to model ranking).
3. Design a search algorithm for each algebra operator such that it is compositional and inductive.

Meta-synthesis frameworks can allow synthesizer writers to easily develop domain-specific synthesizers, similar to how declarative parsing frameworks allow a compiler writer to easily write a parser. The FlashExtract framework [24] and the Test Driven Synthesis framework [36] allows easy development of synthesizers for extracting and transforming data from documents of various types such as text files, web pages, XML documents, tables and spreadsheets. Fig. 1(c) describes a DSL that is composed of Filter and Map operators, which are supported by the FlashExtract framework. The FlashExtract framework is thus able to automatically construct an efficient synthesis algorithm for this DSL.

3.3 Higher-order functions

Higher-order functions are a possibility to provide a bias when searching for hypotheses. In contrast to DSLs, higher-order functions do not tailor IP to a predefined domain, but instead provide common patterns for processing recursive (linked) data as background knowledge to the IP system. For instance, the `fold` higher-order function (also known as *reduce*) iterates over a list of elements and combines the elements by applying another function which is also given as a parameter to the `fold`. For example, `fold (+) [1, 2, 3, 4]` would combine the numbers in the list with the plus function and return 10. Rather than learning a recursive function, the IP system then only needs to pick the suitable higher-order function and instantiate it appropriately. One of the first systems which made use of higher-order functions in IP was MagicHaskeller [17], which generates Haskell functions from a small set of positive inputs. The generated programs are instantiations of a predefined set of higher-order functions such as `fold`. An extension of the analytical IP system Igor2, also implemented in Haskell, takes a similar approach. In contrast to [17], which finds programs by enumeration, Igor2 analyzes the given data to decide which higher-order function fits. The argument function to instantiate the higher-order function is either picked from background knowledge or, if not existing, is invented as an auxiliary function. The use of this technique not only results in a speed-up of synthesis but also enlarges the scope of synthesizable programs [15]. An example for induction with higher-order functions in given in Fig. 5. Finally, Henderson [13] proposed to use higher order to constrain and guide the search of programs for cumulative learning where functions induced from examples are abstracted and can then be used to induce more complex programs. Unlike a DSL, higher-orderness does not restrict IP to a predefined domain, instead it guides search.

```
reverse x = fold f [] x
f x0 [] = [x0]
f x0 (x1 : xs) = x1 : f x0 xs
```

Figure 5: Given examples to reverse a list for lengths zero to three, Igor2 synthesizes a program using the higher-order function `fold`. The auxiliary function `f`, which appends a single element to the end of a list and parameterizes the `fold`, is not given as background knowledge but is invented.

3.4 Meta-interpretive learning

Meta-Interpretive Learning (MIL) is a recent ILP technique [33, 32] aimed at supporting learning of recursive definitions. A powerful and novel aspect of MIL is that when learning a predicate definition it automatically introduces sub-definitions, allowing decomposition into a hierarchy of reuseable parts. MIL is based on an adapted version of a Prolog meta-interpreter. Normally such a meta-interpreter derives a proof by repeatedly fetching first-order Prolog clauses whose heads unify with a given goal. By contrast, a meta-interpretive learner additionally fetches higher-order meta-rules whose heads unify with the goal, and saves the resulting meta-substitutions to form a program. To illustrate the idea, consider the meta-rule below relating P, Q and R:

Name	Meta-Rule
Chain	$P(x,y) \leftarrow Q(x,z), R(z,y)$
Example	**Background knowledge**
boss(lucas,lucy)	boss(lucas,john). boss(john,lucy).
Meta-substitution	**Chain Hypothesis**
P=Q=R=boss	boss(X,Y) :- boss(X,Z), boss(Z,Y)

In this example, the Chain Hypothesis on the boss predicate from Fig. 4 is learned from the Meta-substitutions into the X, Y and Z meta-variables by proving the Example using the Meta-rule and the Background knowledge.

Given the higher-order substitutions, instantiated program clauses can be reconstructed and re-used in later proofs, allowing a form of IP which supports the automatic construction of a hierarchically defined program.

In [26] the authors applied MIL to a task involving string transformations tasks previously studied by Gulwani [9]. Fig. 6 shows the outcome of applying MIL to learning a set of such tasks, using two approaches, *dependent* and *independent* learning, where in the former new definitions are allowed to call already learned definitions at lower levels. Dependent learning produced more compact programs owing to the re-use of existing sub-definitions. This in turn led to reduced search times since the smaller task definitions required less search to find them.

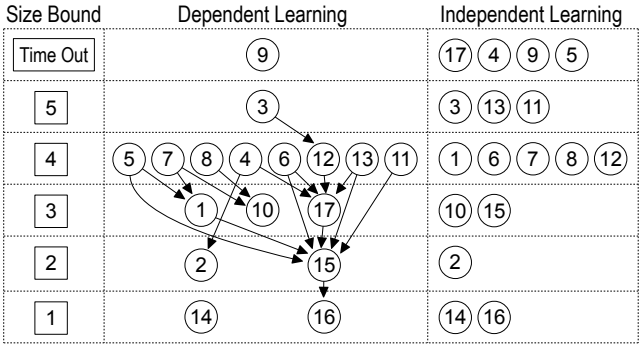

Figure 6: A comparison of the programs generated by Dependent and Independent Learning using MIL. Nodes marked n correspond to programs which solve task n, and nodes are arranged vertically according to their sizes. For Dependent Learning (left), the arrows correspond to the calling relationships of the induced programs. Dependent learning produces re-use of existing sub-definitions which in turn leads to reduced search times.

3.5 New kinds of brute-force search

The general idea here is to systematically explore the entire state space of artifacts and check the correctness of each candidate against the given examples. This approach works relatively well when the specification consists of examples (as opposed to a formal relational specification) since checking the correctness of a candidate solution against examples can be done much faster than validating the correctness against a formal relational specification. However, this is easier said than done because of the huge underlying state space of potential artifacts and often requires innovative nontrivial optimizations, such as goal-directed search, branch and bound, complexity-guided evolutionary approaches, clues based on textual features of examples [29], and offline indexing [17].

3.6 Constraint solving

The general idea here is to reduce the synthesis problem to an equivalent satisfiability problem that is expressed as a standard logical formula. Then, this formula can be solved by a general off-the-shelf tool using the recent advances made in the technology of Satisfiability (SAT) and Satisfiability Modulo Theory (SMT) solvers. This approach has been applied to synthesis from complete formal specifications, but its applicability has been limited to synthesizing restricted forms of programs. On the other hand, if the specification is in the form of examples, then the reduction of the synthesis problem to solving of SAT/SMT constraints can be performed for a larger variety of programs. These examples may be generated inside a counter-example guided inductive synthesis loop [41] (which involves using a validation technology to find new test inputs on which the current version of the synthesized program does not meet the given specification), or using a distinguishing-input based methodology [16] (which involves finding new test inputs that distinguish two semantically distinct synthesized programs, both of which are consistent with the given set of examples).

4. CHALLENGES

There is an ongoing research effort in IP to address increasingly challenging problems in terms of size, effectiveness and robustness.

4.1 Compositionality

The ability of IP to perform adequately for more complex tasks will require breakthroughs in several areas. First, the underlying complexity of the search space for correct solutions limits the overall usability of IP, especially in interactive settings where instant feedback is required. There will undoubtedly be improvements in the performance of such algorithms, including approaches such as version space algebras which provide compact representations of the search space. Ultimately, there will be limits to complexity which no algorithm improvements can address. In such cases, new approaches are needed which allow users to decompose more complex tasks into sufficiently small subtasks and then incrementally compose the solutions provided by IP for each subtask.

4.2 Domain change

Applying IP to new domains efficiently will also require new approaches, including the creation of meta-synthesizers as mentioned above. Because the application of IP techniques in real-world applications is relatively new, there is insufficient experience in exploring the space of applications to clearly identify common patterns that might arise across domains. It is likely that in the short term, domain-specific IP systems will be developed in an ad hoc way, and which over time, as experience with such systems grows, new approaches will systematize and formalize the ad hoc practices, so systems become more general and reusable across different domains.

4.3 Validation

It is important that the artifacts produced by IP give the end user confidence that what they have created is correct and makes sense. For instance, the plethora of automatically named hierarchy of invented sub-tasks generated by approaches such as Meta-Interpretive Learning (Section 3.4)

can lead to confusion if the new names do not bear a clear correspondence to the semantics of the sub-tasks being defined. To address such challenges, we must find new approaches to explain the behavior of the resulting program to the user in intuitive terms and find ways for them to guide the solution if it is incorrect. There is great room for creativity on this problem, such as the use of abstractions which connect the user to the IP result, the ability to highlight those inputs where the tool is less confident and the user should consider inspecting the results, explicitly showing the inferences the synthesized program is applying in domain-specific intuitive ways (e.g., using pictures), and paraphrasing synthesized programs in natural language and letting the user make stylized edits.

4.4 Noise tolerance

Real data is often unclean—some values might be missing and/or incorrect, while some values might occur in different formats (as in representations for dates and numbers). Sometimes even the background knowledge can be incorrect if the user accidentally makes mistakes in providing it.

Addressing the issue of robustness to such noise may be best done in a domain-specific manner. For example, if a table contains mostly correct data with a few outliers, existing techniques to detect and report outliers (or even just missing values) will help the IP process. Fortunately, there is a body of work in the existing ML literature which can be applied to this problem (see, e.g., [4]).

4.5 Making IP more cognitive

Cognitive science and psychology have shown that humans learn from a small number of —usually positive— examples and are relatively intolerant to exceptions [27]. Coherence, simplicity and explanatory power are guiding rules in human inductive inference. The role of background knowledge and the necessary constructs that need to be developed in order to acquire more abstract concepts have also been studied in cognitive science [43]. The progressive acquisition of deep knowledge in humans is especially prominent in language learning but also in learning from problem solving experience. Recently, IP has been used in the context of cognitive modeling, demonstrating that generalized rules can be learned from only a few, positive examples [38]. For instance, Fig. 7 shows the result of learning the *Tower of Hanoi* problem induced by the IP system IGOR2. This result is equivalent to the generalization from three disc to n disc problems (some) humans would infer from the same examples [21]. However, up to now there are no empirical studies which allow for a detailed comparison between high-level human learning of complex routines and the training input and the induced programs of IP systems, to see whether they lead to similar solutions and, when they diverge, to see whether the IP solution is still comprehensible to humans.

Presupposing that the declarative nature of learning in IP systems is sufficiently similar to knowledge level learning in humans, IP systems could be augmented with a Cognitive User Interface [44] with the ultimate goal that machines interact like humans, and evaluate whether in this way they can become more intuitive, trustable, familiar and predictable —including predicting when the system is going to fail. In order to achieve this through IP we need to settle the *interaction model*. For instance, the supervision from the user can be limited to some rewards ("OK" buttons) or penalties ("Cancel" buttons) about what the system is doing, as illustrated in Fig. 4. Alternatively, the user can give a few examples, the IP system makes guesses for other examples and the user corrects them [11, 5]. In this interactive (or query) learning process the user can choose among a set of candidate hypotheses by showing where they differ, using a *distinguishing input* generated by the user —or more effectively— by the IP system itself [16].

5. CONCLUSION

Since the 1970s basic research in IFP and ILP resulted in the development of fundamental algorithms tackling the problem of inducing programs from input/output examples. However, these approaches remained within the context of artificial intelligence research and did not trigger a successful transfer into technologies applicable in a wider context. In 2009 Tessa Lau presented a critical discussion of programming by demonstration systems noting that adoption of such systems is not yet widespread, and proposing that this is mainly due to lack of usability of such systems [22]. In this paper we have presented recent work in IP where we identified several new approaches and techniques which have the potential to overcome some restrictions of previous systems: learning from very few positive examples becomes possible when users and systems share background knowledge that can be represented in a declarative way, which, combined with name inference, is likely to be more easily understandable. Using algorithmic techniques developed in either or both ILP and IFP as well as the use of higher-order functions and meta-interpretative learning resulted in more powerful IP algorithms; the adoption of techniques based on domain-specific languages has allowed the realization of technologies which are ready to use in mass-market products as demonstrated by Flash Fill. Hopefully, the recent achievements will attract more researchers from the different areas in which IP originated—AI, machine learning, functional programming, ILP, software engineering, and cognitive science—to tackle the challenge of bringing IP from the lab into the real world.

6. REFERENCES

[1] Y. Bengio, A. Courville, and P. Vincent. Representation learning: A review and new perspectives. *Pattern Analysis and Machine Intelligence*, 35(8):1798–1828, 2013.

[2] B. Bielawski. Using the convertfrom-string cmdlet to parse structured text. *PowerShell Magazine, http://www.powershellmagazine.com/2014/09/09/using-the-convertfrom-string-cmdlet-to-parse-structured-text/*, September 9, 2004.

[3] A. Carlson, J. Betteridge, B. Kisiel, B. Settles, E.R. Hruschka-Jr, and T.M. Mitchell. Toward an architecture for never-ending language learning. In *AAAI*, 2010.

[4] Varun Chandola, Arindam Banerjee, and Vipin Kumar. Anomaly detection: A survey. *ACM Computing Surveys (CSUR)*, 41(3):15, 2009.

[5] A. Cypher, editor. *Watch What I Do: Programming by Demonstration*. MIT Press, Cambridge, MA, 1993.

[6] C. Ferri-Ramírez, J. Hernández-Orallo, and M.J. Ramírez-Quintana. Incremental learning of functional logic programs. In *FLOPS*, pages 233–247, 2001.

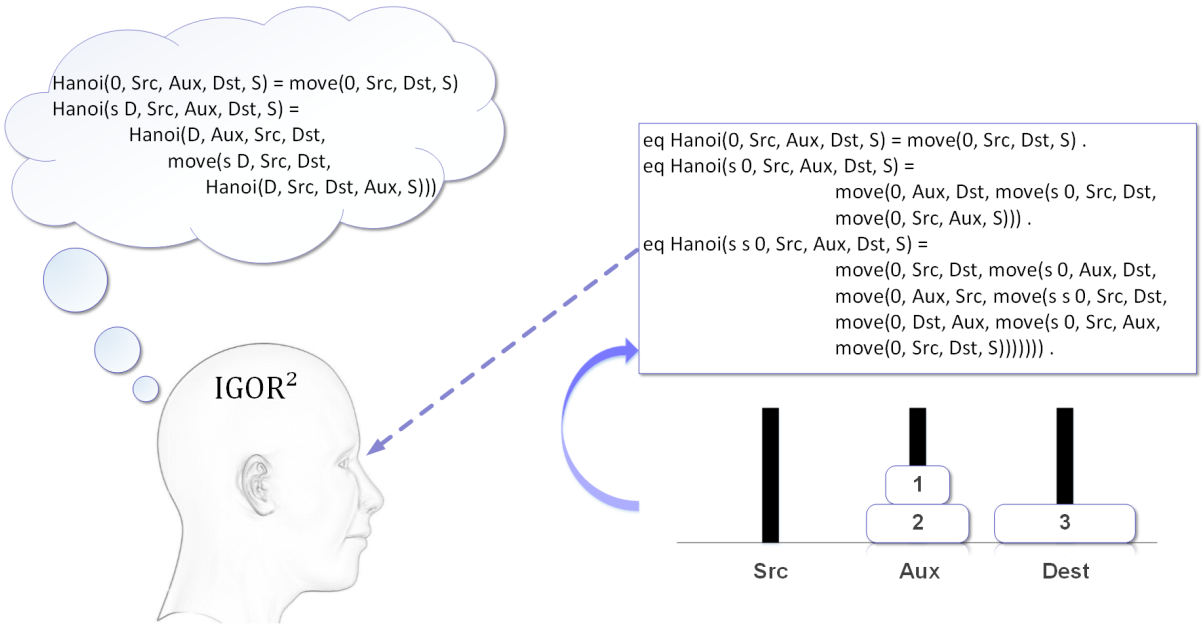

Figure 7: Input of the *Tower of Hanoi* problem for IGOR2 and the induced recursive rule set.

[7] P. Flener and U. Schmid. An introduction to inductive programming. *Artificial Intelligence Review*, 29(1):45–62, 2009.

[8] S. Gulwani. Dimensions in program synthesis. In *PPDP*, 2010.

[9] S. Gulwani. Automating string processing in spreadsheets using input-output examples. In *POPL*, 2011. `http://research.microsoft.com/users/sumitg/flashfill.html`.

[10] S. Gulwani. Example-based learning in computer-aided STEM education. *CACM*, Aug 2014.

[11] S. Gulwani, W. Harris, and R. Singh. Spreadsheet data manipulation using examples. *CACM*, Aug 2012.

[12] Sumit Gulwani, William R Harris, and Rishabh Singh. Spreadsheet data manipulation using examples. *Communications of the ACM*, 55(8):97–105, 2012.

[13] R.J. Henderson and S.H. Muggleton. Automatic invention of functional abstractions. *Latest Advances in Inductive Logic Programming.*, 2012.

[14] J. Hernández-Orallo. Deep knowledge: Inductive programming as an answer, Dagstuhl TR 13502, 2013.

[15] M. Hofmann and E. Kitzelmann. I/O guided detection of list catamorphisms – towards problem specific use of program templates in IP. In *ACM SIGPLAN PEPM*, 2010.

[16] S. Jha, S. Gulwani, S. Seshia, and A. Tiwari. Oracle-guided component-based program synthesis. In *ICSE*, 2010.

[17] S. Katayama. Efficient exhaustive generation of functional programs using Monte-Carlo search with iterative deepening. In *PRICAI*, 2008.

[18] E. Kitzelmann. Analytical inductive functional programming. In *LOPSTR 2008*, volume 5438 of *LNCS*, pages 87–102. Springer, 2009.

[19] E. Kitzelmann. Inductive programming: A survey of program synthesis techniques. In *AAIP*, pages 50–73. Springer, 2010.

[20] E. Kitzelmann and U. Schmid. Inductive synthesis of functional programs: An explanation based generalization approach. *Journal of Machine Learning Research*, 7(Feb):429–454, 2006.

[21] K. Kotovsky, J. R. Hayes, and H. A. Simon. Why are some problems hard? Evidence from Tower of Hanoi. *Cognitive Psychology*, 17(2):248–294, 1985.

[22] T. A. Lau. Why programming-by-demonstration systems fail: Lessons learned for usable AI. *AI Magazine*, 30(4):65–67, 2009.

[23] T. A. Lau, S. A. Wolfman, P. Domingos, and D. S. Weld. Programming by demonstration using version space algebra. *Machine Learning*, 53(1-2):111–156, 2003.

[24] V. Le and S. Gulwani. FlashExtract: A framework for data extraction by examples. In *PLDI*, 2014.

[25] H. Lieberman, editor. *Your Wish is My Command: Programming by Example*. Morgan Kaufmann, 2001.

[26] D. Lin, E. Dechter, K. Ellis, J.B. Tenenbaum, and S.H. Muggleton. Bias reformulation for one-shot function induction. In *ECAI*, 2014.

[27] G.F. Marcus. *The Algebraic Mind. Integrating Connectionism and Cognitive Science*. Bradford, Cambridge, MA, 2001.

[28] F. Martínez-Plumed, C. Ferri, J. Hernández-Orallo, and M.J. Ramírez-Quintana. On the definition of a general learning system with user-defined operators. *arXiv preprint arXiv:1311.4235*, 2013.

[29] A. Menon, O. Tamuz, S. Gulwani, B. Lampson, and A. Kalai. A machine learning framework for programming by example. In *ICML*, 2013.

[30] R. C. Miller and B. A. Myers. Multiple selections in smart text editing. In *IUI*, pages 103–110, 2002.

[31] S.H. Muggleton. Inductive Logic Programming. *New Generation Computing*, 8(4):295–318, 1991.

[32] S.H. Muggleton and D. Lin. Meta-interpretive learning of higher-order dyadic datalog: Predicate invention

revisited. In *IJCAI 2013*, pages 1551–1557, 2013.

[33] S.H. Muggleton, D. Lin, N. Pahlavi, and A. Tamaddoni-Nezhad. Meta-interpretive learning: application to grammatical inference. *Machine Learning*, 94:25–49, 2014.

[34] S.H. Muggleton, L. De Raedt, D. Poole, I. Bratko, P. Flach, and K. Inoue. ILP turns 20: biography and future challenges. *Machine Learning*, 86(1):3–23, 2011.

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

[36] D. Perelman, S. Gulwani, D. Grossman, and P. Provost. Test-driven synthesis. In *PLDI*, 2014.

[37] M. Raza, S. Gulwani, and N. Milic-Frayling. Programming by example using least general generalizations. In *AAAI*, 2014.

[38] U. Schmid and E. Kitzelmann. Inductive rule learning on the knowledge level. *Cognitive Systems Research*, 12(3):237–248, 2011.

[39] U. Schmid and F. Wysotzki. Induction of recursive program schemes. In *ECML*, volume 1398 of *LNAI*, pages 214–225, 1998.

[40] E.Y. Shapiro. An algorithm that infers theories from facts. In *IJCAI*, pages 446–451, 1981.

[41] A. Solar-Lezama. *Program Synthesis by Sketching*. PhD thesis, UC Berkeley, 2008.

[42] P. D. Summers. A methodology for LISP program construction from examples. *Journal ACM*, 24(1):162–175, 1977.

[43] J.B. Tenenbaum, T.L. Griffiths, and C. Kemp. Theory-based Bayesian models of inductive learning and reasoning. *Trends in Cognitive Sciences*, 10(7):309–318, 2006.

[44] S. Young. Cognitive user interfaces. *Signal Processing Magazine, IEEE*, 27(3):128–140, 2010.

Sumit Gulwani (sumitg@microsoft.com) is principal researcher at Microsoft Corporation, Redmond, WA, USA.

José Hernández-Orallo (jorallo@dsic.upv.es) is a reader at Universitat Politècnica de València, Spain. He is supported by EU (FEDER) and Spanish projects PCIN-2013-037, TIN 2013-45732-C4-1-P and GV PROMETEOII2015/013.

Emanuel Kitzelmann (ekitzelmann@gmail.com) is a teacher at Adam-Josef-Cüppers Commercial College, Ratingen, Germany.

Stephen H. Muggleton (s.muggleton@imperial.ac.uk) is professor at the Department of Computing, Imperial College London, UK.

Ute Schmid (ute.schmid@uni-bamberg.de) is professor at University of Bamberg, Germany.

Benjamin Zorn (Ben.Zorn@microsoft.com) is a principal researcher and research co-manager of the Research in Software Engineering (RiSE) group at Microsoft Research in Redmond, WA, USA.