CSCI 699: Introduction to Software Synthesis
Units: 4
Fall 2017 – Monday, Wednesday —10:00am – 11:50am

Location: WPH 603

Instructor: Chao Wang
Office: SAL 334
Office Hours: Tuesday 2:00pm – 3:30pm (tentative)
Contact Info:
Http: http://www-bcf.usc.edu/~wang626/
Email: wang626@usc.edu

Course Description

The goal of this course is to provide an introduction to software synthesis, an emerging research field at the intersection of formal methods, artificial intelligence, and programming systems. It is concerned with automated programming techniques for constructing software from their formal, or informal, specifications as opposed to relying on human to write software code. Specifically, the course will cover classical work on automata based deductive synthesis of reactive systems, as well as constraint solver based techniques for inductive synthesis. The course will also discuss the potential applications of these advanced techniques in hardware, software, and embedded systems.

The course will be graded on the basis of paper reading, class presentation, class participation, as well as a semester-long research project.

Learning Objectives

Students will gain a deep understanding of the foundations, current research, as well as prototype tools of software synthesis. Major learning objectives are:

1. Understand the theoretical foundations of program synthesis, such as propositional logic, temporal logic, constraint solving, and model checking
2. Understand the state-of-the-art techniques for program synthesis, including
   - Reactive synthesis techniques based on automata and two-player games
   - Inductive synthesis techniques based on counter-example guided refinement
   - Proof-based synthesis
   - Syntax-guided synthesis
   - Type-based synthesis,
   - Statistical search and machine learning
3. Gain practical experience with computer-aided programming tools and their applications

Prerequisites

1. General proficiency in discrete mathematics
2. Good programming skills

Required Readings and Supplementary Materials

There are no textbooks. Students are expected to read research papers.

Description and Assessment of Assignments

The grades will be based on reading research papers and writing summaries (20%), class presentation (30%), class participation (10%), and a research project (40%).

The course will be discussion-based. At the start of each lecture, the instructor will give an introduction of the technical background, followed by student-led discussion of 1-3 research papers. Each paper is assigned to a student, who is responsible for giving a presentation and leading the discussion. All students are required to read the assigned papers and submit a short (1-page) summary. Students will also work on an open-ended research project throughout the semester and submit a final report at the end of the semester.

Paper summaries
Students are expected to read and understand the research papers. In addition, they are expected to write and submit the detailed summaries.

Class presentation

Students are expected to present a subset of research papers in class and lead the discussion. The goal is to teach others about the topic, so in the end everyone can have a better understanding of the most recent development in the field.

To get a good grade in paper presentation, you need to excel in the following aspects:

- Clarity in presentation (how well you understand the paper and handle questions, how smooth your talk is, etc.)
- Quality of the slides (your slides must to be informative and thorough, with technical depths, figures, etc.)

Classic participation

Students are expected to participate in the class discussion, e.g., listening to the presentation and asking good questions.

Final project

Final project is a research project that asks students to develop new program synthesis techniques or identify innovative uses of existing program synthesis techniques.

At the end of the project, each student must submit a project report. Your grade on the final project depends on the following aspects:

- Novelty of the project design,
- Thoroughness in the execution, and
- Clarity in the project report.

Grading Breakdown

Assignment	% of Grade
Paper summaries	20%
Class presentation	30%
Class participation	10%
Final project	40%
TOTAL	**100%**

Additional Policies

Late submissions will be accepted up to 24 hours after the announced deadline, with a penalty of 20%. Submissions received more than 24 hours late will receive a grade of 0.

If you feel that an error has been made in grading, please notify the instructor within one week after the material is returned. For exams and final projects, please present a short written appeal to the instructor.

Course Schedule: A Weekly Breakdown

	Topic	Research Paper
Week 1	Introduction to Program Synthesis	Course introduction, survey papers
Week 2	Reactive Synthesis	Temporal logic, automata theory, two-player games
Week 3	Reactive Synthesis	On the synthesis of a reactive module, POPL 1989 Synthesis of reactive(1) designs, VMCAI 2006
Week 4	Inductive Synthesis	SAT solvers, SMT solvers
Week 5	Inductive Synthesis	Syntax-guided synthesis, Dependable software systems engr., 2015
Week 6	Enumerative search algorithms	TRANSIT: Specifying protocols with concolic snippets, PLDI 2013
Week 7	Solver-based search algorithms	Constraints as control, POPL 2012
Week 8	Machine learning based algorithms	Learning programs from noisy data, POPL 2016 Code completion with statistical language models, PLDI 2014
Week 9	Proof-based Synthesis	From program verification to program synthesis, POPL 2010 Complete functional synthesis, PLDI 2010
Week 10	Type-guided Synthesis	Type-and-example-directed program synthesis, PLDI 2015 Synthesizing data structure transformation from examples, PLDI 2015
Week 11	Synthesis with version-space algebra	Automating string processing in spreadsheets using input-output examples, POPL 2010
Week 12	Synthesis for Performance	Fast synthesis of fast collections, PLDI 2016 User-guided device driver synthesis, OSDI 2014
Week 13	Synthesis for Security	Synthesis of masking countermeasures against side-channel attacks, CAV 2014
Week 14	Synthesis for Concurrency	Sketching concurrent data structures, PLDI 2008 Abstraction-guided synthesis of synchronization, POPL 2010
Week 15	Synthesis for Networking	Efficient synthesis for network updates, PLDI 2015 Event-driven network programming, PLDI 2016
FINAL	**(Project report due)** no exam	

Academic Conduct

Plagiarism – presenting someone else's ideas as your own, either verbatim or recast in your own words – is a serious academic offense with serious consequences. Please familiarize yourself with the discussion of plagiarism in *SCampus* in Part B, Section 11, "Behavior Violating University Standards" https://policy.usc.edu/student/scampus/part-b. Other forms of academic dishonesty are equally unacceptable. See additional information in *SCampus* and university policies on scientific misconduct, http://policy.usc.edu/scientific-misconduct.

Discrimination, sexual assault, intimate partner violence, stalking, and harassment are prohibited by the university. You are encouraged to report all incidents to the *Office of Equity and Diversity/Title IX Office* http://equity.usc.edu and/or to the *Department of Public Safety* http://dps.usc.edu. This is important for the health and safety of the whole USC community. Faculty and staff must report any information regarding an incident to the Title IX Coordinator who will provide outreach and information to the affected party. The sexual assault resource center webpage http://sarc.usc.edu fully describes reporting options. Relationship and Sexual Violence Services https://engemannshc.usc.edu/rsvp provides 24/7 confidential support.

Support Systems

A number of USC's schools provide support for students who need help with scholarly writing. Check with your advisor or program staff to find out more. Students whose primary language is not English should check with the *American Language Institute* http://ali.usc.edu, which sponsors courses and workshops specifically for international graduate students. *The Office of Disability Services and Programs* http://dsp.usc.edu provides certification for students with disabilities and helps arrange the relevant accommodations. If an officially declared emergency makes travel to campus infeasible, *USC Emergency Information* http://emergency.usc.edu will provide safety and other updates, including ways in which instruction will be continued by means of Blackboard, teleconferencing, and other technology.

Inductive programming can liberate users from performing tedious and repetitive tasks.

BY SUMIT GULWANI, JOSÉ HERNÁNDEZ-ORALLO,
EMANUEL KITZELMANN, STEPHEN H. MUGGLETON,
UTE SCHMID, AND BENJAMIN ZORN

Inductive Programming Meets the Real World

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 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 interdisciplinary 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[19,41] using generate-and-test approaches based on evolutionary[27,34,35] or systematic[16,28] search or data-driven analytical approaches.[6,11,17,23,38] 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[30,39] 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[33] of its inception as an annual series of international conferences.

Over the last decade IP has attracted a series of international workshops. Recent surveys[7,13,18] reflect the wide variety of implementations and applications in this area.

» **key insights**

■ Supporting end users to automate complex and repetitive tasks using computers is a big challenge for which novel technological breakthroughs are demanded.

■ The integration of inductive programming techniques in software applications can support users by learning domain-specific programs from observing interactions of the user with the system.

■ Inductive programming is being transferred to real-world applications such as spreadsheet tools, scripting, and intelligent program tutors.

■ In contrast to standard machine learning, in inductive programming learning from few examples is possible because users and systems share the same background knowledge.

In the domain of end-user programming, programming by demonstration approaches were proposed that support the learning of small routines from observing the input behavior of users.[5,22,24] Excel's 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 Convert-From-String cmdlet[2]) as well as for special-purpose devices such as home robots and smartphones.

In this article, 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 the use of domain-specific languages and meta-level learning—that widen the scope and power of IP and discuss new challenges.

Real-World Applications

Originally, IP was applied to synthesizing functional or logic programs for general-purpose tasks such as manipulating data structures (for example, 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 here some such opportunities, especially in the areas of end-user programming and education.

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 Web pages, 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 difficult 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 must resort to manually performing repetitive tasks, which is both time consuming and error prone.

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[23] (as shown in Figure 1). (b) Transformation of atomic data types such as strings[9] (as illustrated in Figure 2) or numbers. Transformation of composite data types such as tables[11] and XML.[35] (c) Formatting data.[36] Combining these technologies in a pipeline of extraction, transformation, and formatting can allow end users to perform sophisticated data manipulation tasks.

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

Figure 1. FlashExtract.[23]

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 text file 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 text file in (a).

Ana Trujillo
357 21st Place SE
Redmond, WA
(757) 555-1634

Antonio Moreno
515 93th Lane
Renton, WA
(411) 555- 2786

Thomas Hardy
742 17th Street NE
Seattle, WA
(412) 555-5719

Christina Berglund
475 22th Lane
Redmond, WA
(443) 555-6774

Hanna Moos
785 45th Street NE
Puyallup, WA
(376) 555-2462

Frederique Citeaux
308 66th Place
Redmond, WA
(689) 555-2770

(a)

Label 1	Label 2	Label 3
Ana Trujillo	Redmond	(757) 555-1634
Antonio Moreno	Renton	(411) 555-2786
Thomas Hardy	Seattle	(412) 555-5719
Christina Berglund	Redmond	(443) 555-6774
Hanna Moos	Puyallup	(376) 555-2462
Frederique Citeaux	Redmond	(689) 555-2770

(b)

$$PairSeq\ SS ::= \text{LinesMap}(\lambda x: \text{Pair}(\text{Pos}(x, p_1), \text{Pos}(x, p_2)), LS)$$
$$|\ \text{StartSeqMap}(\lambda x: \text{Pair}(x, \text{Pos}(R_0[x], p)), PS)$$
$$LineSeq\ LS ::= \text{FilterInt}(init, iter, BLS)$$
$$BoolLineSeq\ BLS ::= \text{FilterBool}(b, \text{split}(R_0'/\text{'n'}))$$
$$PositionSeq\ PS ::= \text{LinesMap}(\lambda x: \text{Pos}(x, p), LS)$$
$$|\ \text{FilterInt}(init, iter, \text{PosSeq}(R_0, rr))$$
$$Pred\ b ::= \lambda x: \{\text{Starts},\text{Ends}\}\text{With}(r,x)\ |\ \lambda x: \text{Contains}(r, k, x)$$

(c)

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, Figure 3 shows the output of an inductive synthesis technique for generating algebraic proof problems similar to a given example problem.

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 (such as *send an email message*) 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[30] 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 (as we will discuss later), it will become easier for developers to create new IP-empowered applications.

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 here.

We will also use a running example to indicate some of the features about IP. Figure 4 shows an illustrative ap-

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.

	A	B
	Email	Column 2
1	Email	Column 2
2	Nancy.FreeHafer@fourthcoffee.com	nancy freehafer
3	Andrew.Cencici@northwindtraders.com	andrew cencici
4	Jan.Kotas@litwareinc.com	jan kotas
5	Mariya.Sergienko@gradicdesigninstitute.com	mariya sergienko
6	Steven.Thorpe@northwindtraders.com	steven thorpe
7	Michael.Neipper@northwindtraders.com	michael neipper
8	Robert.Zare@northwindtraders.com	robert zare
9	Laura.Giussani@adventure-works.com	laura giussani
10	Anne.HL@northwindtraders.com	anne hl
11	Alexander.David@contoso.com	alexander david
12	Kim.Shane@northwindtraders.com	kim shane
13	Manish.Chopra@northwindtraders.com	manish chopra
14	Gerwald.Oberleitner@northwindtraders.com	gerwald oberleitner
15	Amr.Zaki@northwindtraders.com	amr zaki
16	Yvonne.McKay@northwindtraders.com	yvonne mckay
17	Amanda.Pinto@northwindtraders.com	amanda pinto

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.

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

Table 1. A simplified comparison between inductive programming and other machine learning paradigms.

	Inductive Programming	Other Machine Learning Paradigms
Number of examples	Small	Large, for example, big data
Kind of data	Relational, constructor-based datatypes	Flat tables, sequential data,
Data Source	Human experts, software applications, HCI, and others	Transactional databases, Internet, sensors (IoT), and others.
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	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).

plication where the goal is to identify repetitive patterns and rules about a user's personal contacts. The IP system learns an easy rule stating the user's boss must be added to her circles and a more complex one that states 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.[11]

Small data. As collecting and storing data is becoming cheaper, it is easy to gain the impression the only interesting datasets today 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 Figure 4.

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 ex-pressive languages. IP is particularly useful when the number of examples is small but the hypothesis space is large (Turing-complete).

Declarative representation. Most (statistical) machine learning techniques are based on probabilities, distances, weights, kernels, matrices, and so on. None of these approaches, except for techniques based on (propositional) decision trees and rules, are *declarative*, that is, expressed as potentially comprehensible rules. Hence, another distinctive feature of IP is 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 Figure 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 lifelong learning becomes easier.[13] For instance, NELL (Never-Ending Language Learner)[3] uses an ILP algorithm that learns probabilistic Horn clauses.

Today, many languages in IP are *hybrid* such as functional logic programming languages, logic programming with types and higher-order constructs, constraints, probabilities, and so on. 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 discuss later.

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.[33] 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 Figure 4, facilitates this hierarchy.

Deep knowledge. Because of the abstraction mechanisms and the use of background knowledge, IP considers learning as a knowledge acquisition process. In Figure 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.[19,30,38,41] Note that both the background knowledge and the new hypothesis in the Figure 4 example 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.

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 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.

Recent Techniques

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

DSL synthesizers. DSLs 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:* 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. Figure 1(c) de-

scribes one such DSL for extracting data from text files. (The full version of this DSL along with its semantics is described in Le and Gulwani.[23]) 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 applying divide-and-conquer deductive techniques that systematically reduce the problem to the synthesis of subexpressions 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.

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.
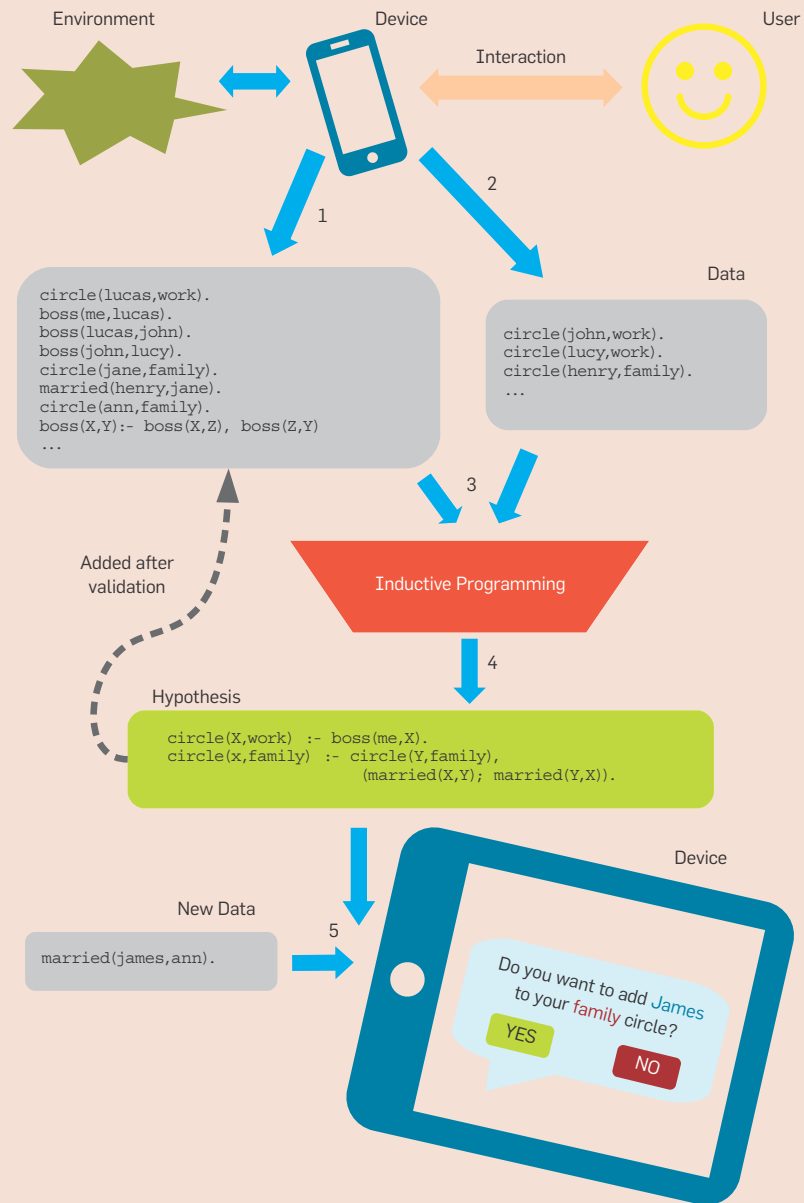
Figure 5. Given examples to reverse a list for lengths zero to three, Igor2 synthesizes a program using the higher-order function fold.

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.

```
reverse x = fold f [] x

f x0 [] = [x0]

f x0 (x1 : xs) = x1 : f x0 xs
```

Table 2. Use of the chain hypothesis on the boss predicate from Figure 4 to prove a new example of boss(lucas, lucy) using the meta-rule and the background knowledge.

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)

This methodology has been applied to various domains including the transformation of syntactic strings,[9,29] semantic strings, numbers, and tables.[11]

Meta-synthesis frameworks. Domain-specific synthesizers (as opposed to general-purpose synthesizers) offer several advantages related to efficiency (such as the ability to synthesize programs quickly) and ranking (such as the ability to synthesize intended programs from fewer examples). However, the design and development of a domain-specific synthesizer is a nontrivial process requiring critical domain insights and implementation effort. Furthermore, any changes to the DSL require making nontrivial 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 that 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 al-low synthesizer writers to easily develop domain-specific synthesizers, similar to how declarative parsing frameworks like lex and yacc allow a compiler writer to easily write a parser. The FlashExtract framework[23] and the Test Driven Synthesis framework[35] allow 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. Figure 1(c) describes a DSL 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.

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 that 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 re-turn 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 that made use of higher-order functions in IP was MAGICHASKELLER,[16] 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 Katayama,[16] 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.[14] An example for induction with higher-order functions in given in Figure 5. Finally, Henderson[12] 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.

Meta-interpretive Learning (MIL) is a recent ILP technique[31,32] 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 reusable 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 this idea, consider the meta-rule in Table 2 relating to *P, Q,* and *R*. In this example, the Chain Hypothesis on the boss predicate from Figure 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 reused in later proofs, allowing a form of IP that supports the automatic construction of a hierarchically defined program.

In Lin et al.[25] the authors applied MIL to a task involving string transformations tasks previously studied by Gulwani.[9] Figure 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 reuse of existing subdefinitions. This in turn led to reduced search times since the smaller task definitions required less search to find them.

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[40] (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[15] (which involves finding new test inputs that distinguish two semantically distinct synthesized programs, both of which are consistent with the given set of examples).

Challenges

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

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 that provide compact representations of the search space. Ultimately, there will be limits to complexity that no algorithm improvements can address. In such cases, new approaches are needed that allow users to decompose more complex tasks into sufficiently small subtasks and then incrementally compose the solutions provided by IP for each subtask.

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,[28] and offline indexing.[16]

Domain change. Applying IP to new domains efficiently will also require new approaches, including the creation of meta-synthesizers as mentioned earlier. 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.

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

Figure 6. A comparison of the programs generated by Dependent and Independent Learning using MIL.

Nodes marked *n* correspond to programs that 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 reuse of existing subdefinitions that in turn leads to reduced search times.
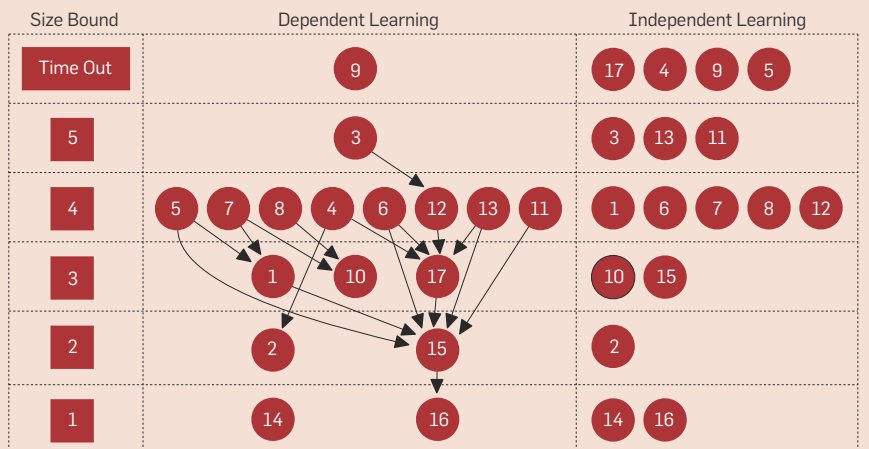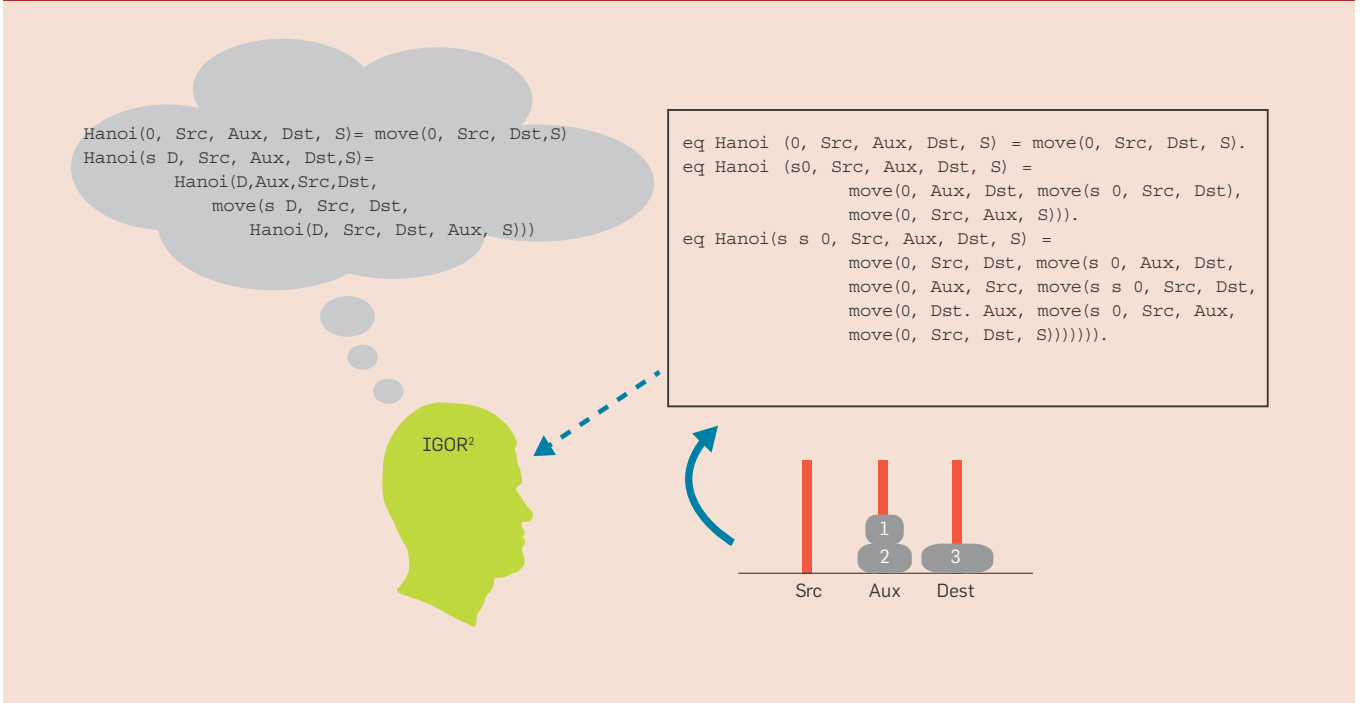
Figure 7. Input of the *Tower of Hanoi* problem for Igor2 and the induced recursive rule set.

instance, the plethora of automatically named hierarchy of invented sub-tasks generated by approaches such as Meta-Interpretive Learning (as discussed previously) 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 that 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 (for example, using pictures), and paraphrasing synthesized programs in natural language and letting the user make stylized edits.

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 that can be applied to this problem (see, for example, Chandola et al.[4]).

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.[26] 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.[42] 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.[37] For instance, Figure 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.[20] However, up until now there are no empirical studies that 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 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[43] 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 Figure 4. Alternatively, the user can give a few examples, the IP system makes guesses for other examples and the user corrects them.[5,11] In this interac-

tive (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.[15]

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 this is mainly due to lack of usability of such systems.[21] In this article, we have presented recent work in IP where we identified several new approaches and techniques that 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 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. ⊂

References
1. Bengio, Y., Courville, A. and Vincent, P. Representation learning: A review and new perspectives. *Pattern Analy. Machine Intell. 35*, 8 (2013), 1798–1828.
2. Bielawski, B. Using the convertfrom-string cmdlet to parse structured text. *PowerShell Magazine*, (Sept. 9, 2004); http:// www. powershellmagazine. com/ 2014/ 09/09/ using-the-convertfrom-string-cmdlet-to-parse-structured-text/
3. Carlson, A., Betteridge, J., Kisiel, B., Settles, B., Hruschka-Jr, E.R. and T.M. Mitchell, T.M. Toward an architecture for never-ending language learning. In *AAAI*, 2010.
4. Chandola, V., Banerjee, A. and V. Kumar, V. Anomaly detection: A survey. *ACM Computing Surveys 41*, 3 (2009), 15.
5. Cypher, A. (Ed). *Watch What I Do: Programming by Demonstration.* MIT Press, Cambridge, MA, 1993.
6. Ferri-Ramírez, C., Hernández-Orallo, J. and Ramírez-Quintana, M.J. Incremental learning of functional logic programs. In *Proceedings of FLOPS*, 2001, 233–247.
7. Flener , P. and Schmid, U. An introduction to inductive programming. *AI Review 29*, 1 (2009), 45–62.
8. Gulwani, S. Dimensions in program synthesis. In *Proceedings of PPDP*, 2010.
9. Gulwani, S. Automating string processing in spreadsheets using input-output examples. In *Proceedings of POPL*, 2011; http://research.microsoft.com/users/sumitg/flashfill.html.
10. Gulwani, S. Example-based learning in computer-aided STEM education. *Commun. ACM 57*, 8 (Aug 2014), 70–80.
11. Gulwani, S., Harris, W. and Singh, R. Spreadsheet data manipulation using examples. *Commun. ACM 55*, 8 (Aug. 2012), 97–105.
12. Henderson, R.J. and Muggleton, S.H. Automatic invention of functional abstractions. *Latest Advances in Inductive Logic Programming*, 2012.
13. Hernández-Orallo, J. Deep knowledge: Inductive programming as an answer, Dagstuhl TR 13502, 2013.
14. Hofmann, M. and Kitzelmann, E. I/O guided detection of list catamorphisms—towards problem specific use of program templates in IP. In *ACM SIGPLAN PEPM*, 2010.
15. Jha, J., Gulwani, S., Seshia, S. and Tiwari, A. Oracle-guided component-based program synthesis. In *Proceedings of the ICSE*, 2010.
16. Katayama, S. Efficient exhaustive generation of functional programs using Monte-Carlo search with iterative deepening. In *Proceedings of PRICAI*, 2008.
17. Kitzelmann, E. Analytical inductive functional programming. *LOPSTR 2008, LNCS 5438.* Springer, 2009, 87–102.
18. Kitzelmann, E. Inductive programming: A survey of program synthesis techniques. In *AAIP*, Springer, 2010, 50–73.
19. Kitzelmann, E. and Schmid, U. Inductive synthesis of functional programs: An explanation based generalization approach. *J. Machine Learning Research 7*, (Feb. 2006), 429–454.
20. Kotovsky, K., Hayes, J.R. and Simon, H.A.. Why are some problems hard? Evidence from Tower of Hanoi. *Cognitive Psychology 17*, 2 (1985), 248–294.
21. Lau, T.A. Why programming-by-demonstration systems fail: Lessons learned for usable AI. *AI Mag. 30*, 4, (2009), 65–67.
22. Lau, T.A., Wolfman, S.A., Domingos, P. and Weld, D.S. Programming by demonstration using version space algebra. *Machine Learning 53*, 1–2 (2003), 111–156.
23. Le, V. and Gulwani, S. FlashExtract: A framework for data extraction by examples. In *Proceedings of PLDI*, 2014.
24. Lieberman, H. (Ed) *Your Wish is My Command: Programming by Example.* Morgan Kaufmann, 2001.
25. Lin, D., Dechter, E., Ellis, K., Tenenbaum, J.B. and Muggleton, S.H. Bias reformulation for one-shot function induction. In *Proceedings of ECAI*, 2014.
26. Marcus, G.F. The Algebraic Mind. *Integrating Connectionism and Cognitive Science.* Bradford, Cambridge, MA, 2001.
27. Martínez-Plumed, C. Ferri, Hernández-Orallo, J. and M.J. Ramírez-Quintana. On the definition of a general learning system with user-defined operators. *arXiv preprint arXiv:1311.4235*, 2013.
28. Menon, A., Tamuz, O., Gulwani, S., Lampson, B. and Kalai, A. A machine learning framework for programming by example. In *Proceedings of the ICML*, 2013.
29. Miller, R.C. and Myers, B.A. Multiple selections in smart text editing. In *Proceedings of IUI*, 2002, 103–110.
30. Muggleton, S.H. Inductive Logic Programming. *New Generation Computing 8*, 4 (1991), 295–318.
31. Muggleton, S.H. and Lin, D. Meta-interpretive learning of higher-order dyadic datalog: Predicate invention revisited. *IJCAI 2013*, 1551–1557.
32. Muggleton, S.H., Lin, D., Pahlavi, N. and Tamaddoni-Nezhad, A. Meta-interpretive learning: application to grammatical inference. *Machine Learning 94* (2014), 25–49.
33. Muggleton, S.H., De Raedt, L., Poole, D., Bratko, I., Flach, P. and Inoue, P. ILP turns 20: Biography and future challenges. *Machine Learning 86*, 1 (2011), 3–23.
34. Olsson, R. Inductive functional programming using incremental program transformation. *Artificial Intelligence 74*, 1 (1995), 55–83.
35. Perelman, D., Gulwani, S., Grossman, D. and Provost, P. Test-driven synthesis. *PLDI*, 2014.
36. Raza, M., Gulwani, S. and Milic-Frayling, N. Programming by example using least general generalizations. *AAAI*, 2014.
37. Schmid, U. and Kitzelmann, E. Inductive rule learning on the knowledge level. *Cognitive Systems Research 12*, 3 (2011), 237–248.
38. Schmid, U. and Wysotzki, F. Induction of recursive program schemes. *ECML 1398 LNAI* (1998), 214–225.
39. Shapiro, E.Y. An algorithm that infers theories from facts. *IJCAI* (1981), 446–451.
40. Solar-Lezama, A. *Program Synthesis by Sketching.* Ph.D thesis, UC Berkeley, 2008.
41. Summers, P.D. A methodology for LISP program construction from examples. *J ACM 24*, 1 (1977), 162–175.
42. Tenenbaum, J.B., Griffiths, T.L. and Kemp, C. Theory-based Bayesian models of inductive learning and reasoning. *Trends in Cognitive Sciences 10*, 7 (2006), 309–318.
43. Young, S. Cognitive user interfaces. *IEEE Signal Processing 27*, 3 (2010), 128–140.

Sumit Gulwani (sumitg@microsoft.com) is principal researcher and research manager of a programming-by-example research and engineering group at Microsoft Corp., Redmond, WA.

José Hernández-Orallo (jorallo@dsic.upv.es) is a reader at the Universitat Politècnica de València. 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 a professor in the Department of Computing at Imperial College London, U.K.

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

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