Report from Dagstuhl Seminar 13502

Approaches and Applications of Inductive Programming

Edited by

Sumit Gulwani[1], Emanuel Kitzelmann[2], and Ute Schmid[3]

1 **Microsoft – Redmond, US,** sumitg@microsoft.com
2 **Universität Duisburg – Essen, DE,** ekitzelmann@gmail.com
3 **Universität Bamberg, DE,** ute.schmid@uni-bamberg.de

Abstract

This report documents the program and the outcomes of Dagstuhl Seminar 13502 "Approaches and Applications of Inductive Programming". After a short introduction to inductive programming research, an overview of the talks and the outcomes of discussion groups is given.

Seminar 8.–11. December, 2013 – www.dagstuhl.de/13502
1998 ACM Subject Classification I.2.2 Automatic Programming, Program Synthesis
Keywords and phrases inductive program synthesis, end-user programming, universal artificial intelligence, constraint programming, probabilistic programming, cognitive modeling
Digital Object Identifier 10.4230/DagRep.3.12.43
Edited in cooperation with Umair Z. Ahmed

1 Executive Summary

Sumit Gulwani
Emanuel Kitzelmann
Ute Schmid

Inductive programming (IP) research addresses the problem of learning programs from incomplete specifications, such as input/output examples, traces, or constraints. In general, program synthesis is a topic of interest to researchers in artificial intelligence as well as in programming research since the 1960s [2]. On the one hand, this research aims at relieving programmers from the tedious task of explicit coding on the other hand it helps to uncover the complex cognitive processes involved in programming as a special domain of complex problem solving. From the beginning, there were two main directions of research – deductive knowledge based approaches and inductive machine learning based approaches. Due to the progress in machine learning, over the last decades the inductive approach currently seems to be the more promising.

Researchers working on the topic of IP are distributed over different communities, especially inductive logic programming (ILP) [12, 6], evolutionary programming [13], functional programming [15, 5, 10], grammar inference [1], and programming languages and verification [7]. Furthermore, domain specific IP techniques are developed for end-user programming [4, 9] and in the context of intelligent tutoring in the domain of programming [8]. In cognitive science, researchers concerned with general principles of human inductive reasoning have constructed computer models for inductive generalization which also have some relation to IP [3, 16].

Approaches and Applications of Inductive Programming, *Dagstuhl Reports*, Vol. 3, Issue 12, pp. 43–66
Editors: Sumit Gulwani, Emanuel Kitzelmann, and Ute Schmid

Dagstuhl Reports
Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

In general, approaches can be classified by (1) the strategy of program construction which can be example-driven or generate-and-test driven; (2) the implicit or explicit restriction bias which can be Horn clauses, functional programs, or domain specific languages possibly with further constraints given as meta-interpreters, templates or program schemes; (3) the possibility to consider background knowledge.

IP research had its first boost in the 1970s in the context of learning Lisp programs from examples. Due to only limited progress, this direction of research decayed and in the 1990s was newly addressed in the context of ILP and evolutionary programming. Again, after first promising results, disappointment set in [14, 11]. However, over the last years, a new revival in IP research can be observed in different communities and promising results, for example in the domain of enduser programming, give rise to new expectations.

Therefore, in the Dagstuhl Seminar AAIP we brought together researchers from these different communities as well as researchers of related fields. The possibility to discuss and evaluate approaches from different perspectives helped to (a) gaining better insights in general mechanisms underlying inductive programming algorithms, (b) identifying commonalities between induction algorithms and empirical knowledge about cognitive characteristics of the induction of complex rules, and (c) open up new areas for applications for inductive programming in enduser programming, support tools for example driven programming, and architectures for cognitive systems.

The presentations covered several aspects of inductive programming and were grouped in the topic sessions

- Inductive Programming Systems and Algorithms (with an introductory talk by Stephen Muggleton),
- Enduser Programming (with an introductory talk by Sumit Gulwani),
- Intelligent Tutoring and Grading,
- Cognitive Aspects of Induction (with an introductory talk by José Hernández-Orallo),
- Combining Inductive Programming with Declarative Programming and with Other Approaches to Program Synthesis (with an introductory talk by Luc de Raedt).

In an initial discussion round three focus topics were identified and further discussed in working groups

- Comparing Inductive Logic and Inductive Functional Programming as well as other Approaches to Program Synthesis,
- Potential New Areas of Applications and Challenges for Inductive Programming,
- Benchmarks and Metrics.

Concluding Remarks and Future Plans

In the final panel discussion the results of the seminar as well as future plans were identified. Participants stated that they learned a lot about different inductive programming techniques and tools to try. The general opinion was that it was very inspiring to have researchers from different backgrounds. To facilitate mutual understanding it was proposed to give introductory lectures, define the vocabulary of the different groups, collect a reading list, and identify common benchmark problems.

To progress in establishing inductive programming as a specific area of research it was proposed to write a Wikipedia page, and to collect introductory literature from the different areas covered in the seminar. Furthermore, plans for joint publications and joint grant proposals were made.

This seminar was highly productive and everybody hoped that there will be a follow-up in the near future.

References

1. D. Angluin and C. H. Smith. Inductive inference: theory and methods. *Computing Surveys*, 15(3):237–269, September 1983.
2. A. W. Biermann, G. Guiho, and Y. Kodratoff, editors. *Automatic Program Construction Techniques*. Macmillan, New York, 1984.
3. P. A. Carpenter, M. A. Just, and P. Shell. What one intelligence test measures: A theoretical account of the processing in the Raven Progressive Matrices test. *Psychological Review*, 97:404–431, 1990.
4. A. Cypher. EAGER: Programming repetitive tasks by example. In *Human Factors in Computing Systems, Proc. of CHI'91*, pages 33–39. ACM Press, 1991.
5. C. Ferri-Ramírez, José Hernández-Orallo, and M. José Ramírez-Quintana. Incremental learning of functional logic programs. In *FLOPS '01: Proceedings of the 5th International Symposium on Functional and Logic Programming*, pages 233–247, London, UK, 2001. Springer-Verlag.
6. P. Flener and S. Yilmaz. Inductive synthesis of recursive logic programs: Achievements and prospects. *Journal of Logic Programming*, 41(2–3):141–195, 1999.
7. Sumit Gulwani. Synthesis from examples: Interaction models and algorithms. In *SYNASC*, pages 8–14, 2012.
8. Sumit Gulwani. Example-based learning in computer-aided stem education. *To appear in Commun. ACM*, 2014.
9. Sumit Gulwani, William R. Harris, and Rishabh Singh. Spreadsheet data manipulation using examples. *Commun. ACM*, 55(8):97–105, 2012.
10. 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.
11. Tessa Lau. Why programming-by-demonstration systems fail: Lessons learned for usable ai. *AI Magazine*, 30(4):65–67, 2009.
12. S. Muggleton and L. De Raedt. Inductive logic programming: Theory and methods. *Journal of Logic Programming, Special Issue on 10 Years of Logic Programming*, 19-20:629–679, 1994.
13. Roland Olsson. Inductive functional programming using incremental program transformation. *Artificial Intelligence*, 74(1):55–83, March 1995.
14. J.R. Quinlan and R.M. Cameron-Jones. FOIL: A midterm report. In P. B. Brazdil, editor, *Proceedings of the ECML'93*, number 667 in LNCS, pages 3–20. Springer, 1993.
15. U. Schmid and F. Wysotzki. Induction of recursive program schemes. In *Proc. 10th European Conference on Machine Learning (ECML-98)*, volume 1398 of *LNAI*, pages 214–225. Springer, 1998.
16. Ute Schmid and Emanuel Kitzelmann. Inductive rule learning on the knowledge level. *Cognitive Systems Research*, 12(3):237–248, 2011.

2 Table of Contents

Executive Summary
Sumit Gulwani, Emanuel Kitzelmann, and Ute Schmid 43

Overview of Talks on: Inductive Programming Systems and Algorithms

Predicate invention and learning of recursive logic programs
Stephen Muggleton 48

Cumulative Learning by Refinement and Automated Abstraction
Robert J. Henderson 48

Example-driven Inductive Functional Programming with IGOR2
Emanuel Kitzelmann 49

gErl: an Inductive Programming System with user-defined operators
Fernando Martínez-Plumed 49

MagicHaskeller on the Web: Inductive Functional Programming System for Casual Programming
Susumu Katayama 50

Project of IDRE I&D oriented to implement arbitrary DSLs by learning and develop applications by demonstrating examples of their execution represented on implemented DSL
Alexey Grigoryev 50

Overview of Talks on: Enduser Programming

Flash Fill: An Excel 2013 feature
Sumit Gulwani 51

Empowering Users with Data
Benjamin Zorn 51

Test-Driven Synthesis
Daniel Perelman 52

Overview of Talks on: Intelligent Tutoring and Grading

Autograder: Automated Feedback Generation for Programming Problems
Rishabh Singh 52

Automated Grading for DFA Constructions
Dileep Kini 53

Automatically Generating Problems and Solutions for Natural Deduction
Umair Zafrulla Ahmed 53

Personalized Mathematical Word Problem Generation
Oleksandr Polozov 54

Overview of Talks on: Cognitive Aspects of Induction

Small but deep. What can we learn from inductive programming?
José Hernández-Orallo 54

A Cognitive Model Approach to Solve IQ-Test Problems
Marco Ragni 55

Applying IGOR to Cognitive Problems
Ute Schmid 55

Learning Analogies
Tarek R. Besold 56

Towards Quantifying Program Complexity and Comprehension
Mike Hansen 56

Overview of Talks on: Combining Inductive Programming with Declarative Programming and with Other Approaches to Program Synthesis

Towards declarative languages for learning
Luc De Raedt 57

Probabilistic programming and automatic programming
Iurii Perov 58

Programming with Millions of Examples
Eran Yahav 58

Type Inhabitation Problem for Code Completion and Repair
Ruzica Piskac 59

Learning a Program's usage of Dynamic Data Structures from Sample Executions
David White 59

SMT-based Videogame Synthesis
Sam Bayless 59

System Demonstrations

Storyboard Programming of Data Structure Manipulations
Rishabh Singh 60

Working Groups

Comparing Inductive Logic and Inductive Functional Programming as well as other Approaches to Program Synthesis
Stephen Muggleton 60

Potential New Areas of Applications and Challenges for Inductive Programming
Ben Zorn 62

Benchmarks and Metrics
José Hernández-Orallo and Marco Ragni 64

Participants 66

3 Overview of Talks on: Inductive Programming Systems and Algorithms

3.1 Predicate invention and learning of recursive logic programs

Stephen Muggleton (Imperial College London, UK)

Joint work of Lin, Dianhuan; Pahlavi, Niels; Tamaddoni-Nezhad, Alireza
URL http://ilp.doc.ic.ac.uk/metagolD/

Inductive Logic Programming (ILP) is the sub-area of Machine Learning concerned with inductive inference of logic programs. Since logic programs can be used to encode arbitrary computer programs, ILP is a highly flexible form of Machine Learning, which has allowed it to be successfully applied in a number of complex areas. However, despite this fact, state-of-the-art ILP systems such as FOIL, Golem and Progol are unable to effectively learn representations such as regular and context-free grammars from example sequences. Such tasks require the automated introduction of new recursively defined non-terminals into the description language. Within ILP this is referred to as predicate invention. Early ILP systems achieved limited forms of predicate invention. However, this approach was abandoned since it was unclear how to bound the complexity of the search involved. A recent review of ILP emphasised an urgent need for renewed attention to Predicate Invention in order to broaden the applications of ILP. Recent work has substantially generalised this idea to support learning of regular and context-free grammars as well as higher-order dyadic datalog programs. The key innovation was the introduction of a Prolog meta-interpreter driven by a set of higher-order Datalog definite clauses. Instantiation of the predicate symbols of these higher-order clauses is achieved using a form of ground abduction. The Meta-interpreter is used to prove the set of examples. If any sub-proof fails then a rule is formed by abduction, allowing the proof to be completed. A set of abduced rules which allows all the positive examples and none of the negative examples to be proved comprises a valid hypothesis. Completeness of the inductive process is ensured by the use of a complete form of ground abduction. The proof space of the meta-interpreter is several orders of magnitude more efficient than the refinement-graph search of the state-of-the-art ILP systems.

3.2 Cumulative Learning by Refinement and Automated Abstraction

Robert J. Henderson (Imperial College London, GB)

Main reference R. J. Henderson, "Cumulative Learning in the Lambda Calculus," PhD thesis, Imperial College London, 2014.

I describe a new approach to 'cumulative learning', in which an inductive inference system automatically acquires knowledge necessary for solving harder problems through experience of solving easier ones. I have implemented this approach in an inductive functional programming system called RUFINSKY, which employs an alternating two-phase policy in order to solve a sequence of related but successively more difficult learning problems. In the first phase, a hypothesis is learned to fit training data using a guided search technique called refinement which has been adapted from inductive logic programming. In the second phase, new elements of background knowledge are abstracted from syntactic patterns in hypotheses

by anti-unification of subterms and a form of inverse beta-reduction. The new background knowledge has the effect of shifting RUFINSKY's inductive bias, allowing it to automatically adapt to a problem domain. Cumulative learning techniques are promising as a route towards practical AIXI-style Artificial General Intelligence (AGI). Taking into account new literature on the potential existential risks to humanity posed by AGI, I discuss whether research into cumulative learning should or should not be considered ethical.

3.3 Example-driven Inductive Functional Programming with IGOR2

Emanuel Kitzelmann (Universität Duisburg – Essen, DE)

Main reference E. Kitzelmann, "A Combined Analytical and Search-based Approach for the Inductive Synthesis of Functional Programs," KI – Künstliche Intelligenz, 25(2):179–182, 2011; see also corresponding PhD Thesis, Faculty of Information Systems and Applied Computer Sciences, University of Bamberg, 2010.
URL http://dx.doi.org/10.1007/s13218-010-0071-x
URL http://opus4.kobv.de/opus4-bamberg/frontdoor/index/index/docId/250

IGOR2, the inductive functional programming (IFP) system presented in this talk, learns recursive functional programs from incomplete specifications such as input-output examples or computation traces. The main contribution of IGOR2 is a combination of techniques from generate-and-test systems with techniques from analytical approaches. The goal is to gain a better trade-off between expressivity of the hypothesis language and efficiency of the synthesis procedure. Therefore, IGOR2 searches in program spaces but computes candidate programs directly from the specification. IGOR2 has been successfully applied in different domains such as list processing and strategy learning.

3.4 gErl: an Inductive Programming System with user-defined operators

Fernando Martínez-Plumed (Polytechnic University of Valencia, ES)

Joint work of Martínez-Plumed, Fernando; Ferri, Cesar; Hernández-Orallo, José; Ramirez-Quintana, Maria-José
Main reference F. Martínez-Plumed, C. Ferri, J. Hernández-Orallo, M.-J. Ramérez-Quintana, "On the definition of a general learning system with user-defined operators," arXiv:1311.4235v1 [cs.LG] , 2013
URL http://arxiv.org/abs/1311.4235v1

GErl was born as an advocacy of a more general framework for machine learning: a general rule-based learning system where operators can be defined and customised according to the problem, data representation and the way the information should be navigated. Since changing operators affect how the search space needs to be explored, heuristics are learnt as a result of a decision process based on reinforcement learning where each action is defined as a choice of operator and rule guided by an optimality criteria (based on coverage and simplicity) which feed a rewarding module. States and actions are abstracted as tuples of features in a Q matrix from which a supervised model is learnt. Erlang is used to represent theories and examples in an understandable way: examples as equations, patterns as rules, models as sets of rules, and, to defining operators: gErl provides some meta-level facilities called meta-operators which allow the user to define well-known generalisation and specialisation operators in Machine Learning.

3.5 MagicHaskeller on the Web: Inductive Functional Programming System for Casual Programming

Susumu Katayama (University of Miyazaki, JP)

MagicHaskeller on the Web is a web-based tool for inductive functional programming in Haskell. Its main focus is on offhandedness, and its users can use it in a similar way to using a web search engine. In this talk, I first demonstrated its usage. Then, I explained its implementation by shared memoization table. I concluded my presentation by discussing possible future work, including application of the algorithm to other languages.

3.6 Project of IDRE I&D oriented to implement arbitrary DSLs by learning and develop applications by demonstrating examples of their execution represented on implemented DSL

Alexey Grigoryev (National Nuclear Research University MEPhI, RU)

Joint work of Grigoryev, Alexey; Sergievsky, George M.
Main reference G M. Sergievsky, "Concept of Inductive Programming Supporting Anthropomorphic Information Technology," Journal of Computer and Systems Sciences International, 50(1):38–50, 2011.
URL http://dx.doi.org/10.1134/S1064230711010163

Inductive synthesis usually consists of two stages: 1) front-end transformation which builds a non-recursive program using the implementation language. This programs executes calculations specified by examples (usually is done by a program chosen by ad hoc developer). 2) Back-end transformation which properly implements inductive synthesis and has is a recursive program as the result of it. The main property of this program is that its unfolding-transformations allow to get a representation containing a non-recursive program (as stated above). In IDRE I&D a front-end transformation is implemented by a program which is synthesized by learning. It brings a possibility to represent instances on arbitrary DSLs specified by user in learning mode. Back-end transformation supports in increment mode, controlling structure predicate synthesis and implementation language based on non-deterministic model of calculations. This brings possibility to develop arbitrary applications by demonstrating examples of their execution represented on implemented DSLs.

4 Overview of Talks on: Enduser Programming

4.1 Flash Fill: An Excel 2013 feature

Sumit Gulwani (Microsoft Research – Redmond, US)

Joint work of Gulwani, Sumit; Singh, Rishabh; Zorn, Ben
Main reference S. Gulwani, W. Harris, R. Singh, "Spreadsheet data manipulation using examples," Comm. of the ACM, 55(8):97–105, 2012.
URL http://dx.doi.org/10.1145/2240236.2240260
URL http://research.microsoft.com/en-us/um/people/sumitg/flashfill.html

Flash Fill is a new feature in Excel 2013 for automating string transformations by examples. In this talk, I will demo Flash Fill and talk about the underlying algorithms. I will also describe some extensions to Flash Fill that enable semantic string transformations, number transformations, and table layout transformations.

References

1 Sumit Gulwani. Automating string processing in spreadsheets using input-output examples. In *POPL*, 2011.
2 Sumit Gulwani, William Harris, and Rishabh Singh. Spreadsheet data manipulation using examples. *Communications of the ACM*, Aug 2012.

4.2 Empowering Users with Data

Benjamin Zorn (Microsoft Research – Redmond, US)

Spreadsheets are valuable because they allow the free-form aggregation of data from arbitrary sources. Users can then manipulate that data using built-in tools, including filtering, aggregation, and visualization. Unfortunately the free- form nature of spreadsheets often prevents relational queries from being applied to the data, limiting its value. We propose a new language, Flare, and a new synthesis algorithm, FlashRelate, that allow users to extract relational data from semi-structured data using examples. Flare programs combine constraints on the contents of cells expressed as traditional regular expressions with spatial constraints that express how cells are related on the grid. Just as regular expressions result in a sequence of matching strings, Flare programs produce a set of tuples containing the contents of cells matching the specified constraints. Our FlashRelate synthesis algorithm can automatically generate Flare programs from a few positive and negative examples of tuples in the desired output table. FlashRelate searches over a space of possible Flare programs and iteratively builds up a solution tree, selecting the next highest ranked constraint (similar to Kruskal's spanning tree algorithm). If the resulting tree does not satisfy the negative examples, the algorithm backtracks. We describe the algorithm and evaluate its effectiveness in extracting structured data from real semi-structured spreadsheets.

4.3 Test-Driven Synthesis

Daniel Perelman (University of Washington – Seattle, US)

Joint work of Perelman, Daniel; Gulwani, Sumit; Grossman, Dan; Provost, Peter

Programming-by-example technologies allow an end-user to create simple programs to automate their workflows merely by providing input/output examples, but existing systems are specific to the domain-specific language (DSL) they were designed for. We present a new programming-by-example language LaSy which is parameterized by an arbitrary DSL that may contain conditionals and loops and therefore able to synthesize programs in any domain. In LaSy, the user provides a sequence of increasingly sophisticated input/output examples along with an expert-written DSL definition. LaSy is powered by our novel test-driven synthesis methodology which performs program synthesis iteratively, consuming a sequence of input/output examples one at a time, and DSL-based synthesis which efficiently synthesizes programs within a given DSL. We present applications of our synthesis methodology to end-user programming with LaSy programs for transformations over strings, XML, and table layouts. We compare our synthesizer on these applications to state-of-the-art DSL-specific synthesizers as well to the general purpose synthesizer Sketch.

5 Overview of Talks on: Intelligent Tutoring and Grading

5.1 Autograder: Automated Feedback Generation for Programming Problems

Rishabh Singh (MIT – Cambridge, US)

Joint work of Singh, Rishabh; Gulwani, Sumit; Solar-Lezama, Armando
Main reference R. Singh, S. Gulwani, A. Solar-Lezama, "Automated Feedback Generation for Introductory Programming Assignments," in SIGPLAN Notices, 48(6):15–26, 2013.
URL http://dx.doi.org/10.1145/2491956.2462195

In this talk, I will present the Autograder tool for providing automated feedback for introductory programming problems. Using a reference implementation and an error model, the tool automatically derives minimal corrections to student's incorrect solutions, providing them with a measure of exactly how incorrect a given solution was, as well as feedback about what they did wrong. Our results on thousands of student attempts from edX 6.00x course show that relatively simple error models can correct a significant fraction of all incorrect submissions. Towards the end of the talk, I will also present some of the recent work and many interesting open research challenges.

5.2 Automated Grading for DFA Constructions

Dileep Kini (University of Illinois – Urbana Champaign, US)

Joint work of Alur, Rajeev; D'Antoni, Loris; Gulwani, Sumit; Kini, Dileep; Viswanathan, Mahesh
Main reference R. Alur, L. D'Antoni, S. Gulwani, D. Kini, M. Viswanathan, "Automated Grading of DFA Constructions," in Proc. of the 23rd Int'l Joint Conf. on Artificial Intelligence (IJCAI'13), pp. 1976–1982, AAAI, 2013; available as pre-print from the author's webpage.
URL http://ijcai.org/papers13/Papers/IJCAI13-292.pdf
URL http://www.cis.upenn.edu/~alur/Ijcai13.pdf

One challenge in making online education more effective is to develop automatic grading software that can provide meaningful feedback. This paper provides a solution to automatic grading of the standard computation-theory problem that asks a student to construct a deterministic finite automaton (DFA) from the given description of its language. We focus on how to assign partial grades for incorrect answers. Each student's answer is compared to the correct DFA using a hybrid of three techniques devised to capture different classes of errors. First, in an attempt to catch syntactic mistakes, we compute the edit distance between the two DFA descriptions. Second, we consider the entropy of the symmetric difference of the languages of the two DFAs, and compute a score that estimates the fraction of the number of strings on which the student answer is wrong. Our third technique is aimed at capturing mistakes in reading of the problem description. For this purpose, we consider a description language MOSEL, which adds syntactic sugar to the classical Monadic Second Order Logic, and allows defining regular languages in a concise and natural way. We provide algorithms, along with optimizations, for transforming MOSEL descriptions into DFAs and vice- versa. These allow us to compute the syntactic edit distance of the incorrect answer from the correct one in terms of their logical representations. We report an experimental study that evaluates hundreds of answers submitted by (real) students by comparing grades/feedback computed by our tool with human graders. Our conclusion is that the tool is able to assign partial grades in a meaningful way, and should be preferred over the human graders for both scalability and consistency.

5.3 Automatically Generating Problems and Solutions for Natural Deduction

Umair Zafrulla Ahmed (Indian Institute of Technology – Kanpur, IN)

Joint work of Ahmed, Umair Zafrulla; Gulwani, Sumit; Karkare, Amey
Main reference U. Z. Ahmed, S. Gulwani, A. Karkare, "Automatically generating problems and solutions for natural deduction," in Proc. of the 23rd Int'l Joint Conf. on Artificial Intelligence (IJCAI'13), pp. 1968–1975, AAAI, 2013.
URL http://ijcai.org/papers13/Papers/IJCAI13-291.pdf

In this talk, I will present our recent work on automatically generating problems and solutions for Natural Deduction proofs. Natural deduction, which is a method for establishing validity of propositional type arguments, helps develop important reasoning skills and is thus a key ingredient in a course on introductory logic. We present two core components, namely solution generation and practice problem generation, for enabling computer-aided education for this important subject domain. The key enabling technology is use of an offline-computed data-structure called Universal Proof Graph (UPG) that encodes all possible

applications of inference rules over all small propositions abstracted using their bitvector-based truth-table representation. This allows an efficient forward search for solution generation. More interestingly, this allows generating fresh practice problems that have given solution characteristics by performing a backward search in UPG. We obtained around 300 natural deduction problems from various textbooks. Our solution generation procedure can solve many more problems than the traditional forward- chaining based procedure, while our problem generation procedure can efficiently generate several variants with desired characteristics.

5.4 Personalized Mathematical Word Problem Generation

Oleksandr Polozov (University of Washington – Seattle, US)

Word problems are an established technique for teaching mathematical mathematical modeling skills in elementary and middle school education. However, the effectiveness of word problems widely varies among students. A large fraction of students finds word problems unconnected to real life, artificial, and uninteresting. Most students find them much more difficult than the corresponding symbolic representations. To account for these opinions, an ideal textbook should consist of a individually crafted progression of unique word problems, that form a personalized plot.

We propose a novel technique for automatic generation of personalized word problems. In our system, word problems are generated procedurally using answer-set programming from general specification. The specification includes tutor requirements (mathematical features, problem class) and student requirements (personalization, plot characters, setting). Our system generates a narrative plot, a mathematical representation, and a natural language description according to the provided specification. It makes use of a rich language of plot elements that can be parametrized by a narrative setting (fantasy world, science fiction, etc.). We are currently investigating the connection of our plot language with FrameNet, the database of semantic knowledge elements.

6 Overview of Talks on: Cognitive Aspects of Induction

6.1 Small but deep. What can we learn from inductive programming?

José Hernández-Orallo (Polytechnic University of Valencia, ES)

Main reference J. Hernández-Orallo, "Deep Knowledge: Inductive Programming as an Answer," Manuscript, 2013.
URL http://users.dsic.upv.es/~flip/papers/deepknowledge2013.pdf

Inductive programming has focussed on problems where data are not necessarily big, but representation and patterns may be deep (including recursion and complex structures). In this context, we will discuss what really makes some problems hard and whether this difficulty is related to what humans consider hard. We will highlight the relevance of background knowledge in this difficulty and how this has influence on a preference of inferring small

hypotheses that are added incrementally. When dealing with the techniques to acquire, maintain, revise and use this knowledge, we argue that symbolic approaches (featuring powerful construction, abstraction and/or higher-order features) have several advantages over non-symbolic approaches, especially when knowledge becomes complex. Also, inductive programming hypotheses (in contrast to many other machine learning paradigms) are usually related to the solutions that humans would find for the same problem, as the constructs that are given as background knowledge are explicit and shared by users and the inductive programming system. This makes inductive programming a very appropriate paradigm for addressing and better understanding many challenging problems humans can solve but machines are still struggling with. Some important issues for the discussion will be the relevance of pattern intelligibility, and the concept of scalability in terms of incrementality, learning to learn, constructive induction, bias, etc.

6.2 A Cognitive Model Approach to Solve IQ-Test Problems

Marco Ragni (Universität Freiburg))

Current computational approaches outperform humans for most reasoning problem classes. Nevertheless, humans do perform better for some specific problem classes, for instance when only imprecise information is available or "insight" is a necessary precondition. Typically, in such domains not all information is given, therefore, functions and operators must often first be identified. By imitating human approaches it is possible to develop artificial intelligence (AI) systems that can deal with problems in such domains (e.g., IQ-test problems). In my talk I will elaborate on two specific domains: number series and IQ-tests. Results and limitations of the approaches are discussed.

6.3 Applying IGOR to Cognitive Problems

Ute Schmid (Universität Bamberg, DE)

Joint work of Schmid, Ute; Kitzelmann, Emanuel
Main reference U. Schmid, E. Kitzelmann, "Inductive rule learning on the knowledge level," Cognitive Systems Research, 12(3–4):237–248, 2011.
URL http://dx.doi.org/10.1016/j.cogsys.2010.12.002

IGOR is an inductive programming system based on an analytical, example-driven technique for generalization. IGOR learns recursive functional programs (in Maude or Haskell) from a small set of positive examples. Main features of IGOR are the possibility to rely on background knowledge, automated invention of sub-functions, guaranteed extensional corrections wrt. the input/output examples and guaranteed termination of the induced programs. Main restrictions of IGOR are that the given input/output examples need to be correct and to cover the first k instances over the input data type. While induction typically is very fast (due to the analytical approach), IGOR can run into memory problems when background knowledge is provided due to its generalization strategy.

Aside from typical applications in the context of automated program construction, we investigated how IGOR can learn generalized rules in cognitive domains. Taking the cognitive perspective, IGOR addresses the acquisition of constructive rules. For example, it can generalize a solution strategy for the Tower of Hanoi from an example solution trace for an Hanoi problem with three discs. Further applications we investigated are from typical planning domains such as blocksworld, learning relations such as the transitivity of "isa", simple natural language grammars, and – most recently – rules to continue number series. While application of IGOR to examples from such cognitive domains is straight-forward, there is no general mechanism to provide the necessary examples in the necessary representation. That is, we need to explore how IGOR can be embedded in a system which generates suitable example experience.

6.4 Learning Analogies

Tarek R. Besold (Universität Osnabrück, DE)

Joint work of Besold, Tarek R.; Jäkel, Frank

Analogical reasoning is a core capacity of human cognition. In an analogy superficially dissimilar domains of knowledge are regarded as similar with respect to their relational structure. Discovering this structure allows us to transfer knowledge from one domain to another.

Over the last decades, a significant body of research in cognitive AI and cognitive science studied how this capacity can be modeled in computational terms. Most systems are focusing on the process of analogical mapping between given domain theories, i.e., on establishing correspondences between elements of the source and the target domain of the analogy, and on the following transfer of knowledge from source to target.

We want to combine the abstract mechanism of analogy-making with learning capacities from IP/ILP. Our aim is to develop a two-level model for cross-domain analogies. Starting out from independent sets of observations from several domains, we want to learn the most likely governing base theory within each domain by means of IP/ILP. In parallel, we are also trying to establish an overall cross-domain general theory encompassing the abstract structure underlying the learned base theories. If a general theory, i.e, an analogy between domains can be established, this theory, in turn, can inform the base theories in each domain.

6.5 Towards Quantifying Program Complexity and Comprehension

Mike Hansen (Indiana University – Bloomington, US)

Joint work of Hansen, Mike; Lumsdaine, Andrew; Goldstone, Robert
Main reference M. Hansen, R. L. Goldstone, A. Lumsdaine, "What Makes Code Hard to Understand?" arXiv:1304.5257v2 [cs.SE], 2013.
URL http://arxiv.org/abs/1304.5257v2

Psychologists have studied the behavioral aspects of programming for at least 40 years. In the last few decades, multiple qualitative cognitive models of program comprehension have been proposed. These models describe important aspects of a programmer's knowledge, and provide a framework for discussing the dimensions along which a program's cognitive complexity

may vary. In this talk, we outline work being done on a quantitative cognitive model that seeks to operationalize the shared aspects of existing qualitative models. Taking inspiration from the ACT-R cognitive architecture, our model will include active vision, behavioral, and declarative/spatial memory components. In the context of inductive programming, we propose using this model for the automated culling of generated programs. Less complex programs – i.e., those that minimize some measure of resource expenditure in the model – may be preferred by human operators.

7 Overview of Talks on: Combining Inductive Programming with Declarative Programming and with Other Approaches to Program Synthesis

7.1 Towards declarative languages for learning

Luc De Raedt (KU Leuven, BE)

One of the long standing goals of artificial intelligence and machine learning is to develop machines that can be programmed automatically. To realize this dream, researchers have investigated programming and modeling languages that support machine learning. These languages provide primitives for specifying the machine learning task of interest, that is, the training instances and constraints on the programs to be learned, and the system should automatically solve the learning task. This is an effective way to realize inductive programming.

In this context, I shall report on three different languages for learning: 1) the probabilistic (logic) programming language ProbLog [1, 2], which extends Prolog with probabilistic facts and which supports parameter estimation, 2) the modeling language MiningZinc [3], which extends the constraint programming language MiniZinc with primitives for data mining, and which allows to declaratively model (and solve) a wide variety of pattern mining problems, and 3) kLog [4], the logical and relational language for kernel-based learning, which allows users to specify logical and relational learning problems at a high level in a declarative way. I shall also discuss how these languages can be used for automatic and inductive programming.

References

1 http://dtai.cs.kuleuven.be/problog/index.html
2 D. Fierens, G. Van den Broeck, J. Renkens, D. Shterionov, B. Gutmann, I. Thon, G. Janssens, and L. De Raedt. Inference and learning in probabilistic logic programs using weighted Boolean formulas. arXiv preprint arXiv:1304.6810 (to appear in Theory and Practice of Logic Programming), 2013. http://arxiv.org/abs/1304.6810
3 T. Guns, A. Dries, G. Tack, S. Nijssen, and L. De Raedt. MiningZinc: A Modeling Language for Constraint-based Mining, In Proceedings of IJCAI, 2013. https://lirias.kuleuven.be/bitstream/123456789/399282/1/miningzinc_ijcai_final.pdf
4 P. Frasconi, F. Costa, L. De Raedt, K. De Grave. kLog: A Language for Logical and Relational Learning with Kernels. arXiv:1205.3981, 2012. http://arxiv.org/abs/1205.3981

7.2 Probabilistic programming and automatic programming

Iurii Perov (University of Oxford, GB)

Joint work of Perov, Iurii; Wood, Frank; Mansinghka, Vikash; Kulkarni, Tejas, Kulkarni; Tenenbaum, Joshua

Probabilistic programming has recently attracted much attention in Machine Learning and Computer Science communities. Probabilistic programming (PP) took from "traditional" programming its convenient way to define generative models as "algorithms", which in the case of PP generally contain much uncertainty. In addition to evaluation (program execution) component, probabilistic programming involves inference, so that models can converge from its prior to its posterior (given observations).

That is, in traditional programming you usually define inputs, define a precise algorithm, and computer gives you outputs. In probabilistic programming you define inputs, specify outputs (i.e. observations) or at least part of them (i.e. a train set), provide a prior model with uncertainty, and computer is supposed to get you into posterior via general- and special-purpose inference techniques (e.g. by providing you with approximate posterior distribution on latent variables). One can imagine on some models that converging from prior to posterior is a reduction of uncertainty given observations. Many Machine Learning problems could be written compactly and easily in probabilistic programming languages.

The idea of combining probabilistic programming and automatic programming (i.e. inference happens on a generative metamodel, which is written as a probabilistic program and which produces a stochastic model – a desired synthesized program) and the draft of road map for this direction will be introduced for discussion.

There is much connection with related fields, including Inductive Programming (especially with Probabilistic Inductive Logic Programming and Inductive Functional Programming), and these relations should be further explored.

7.3 Programming with Millions of Examples

Eran Yahav (Technion – Haifa, IL)

Joint work of Peleg,Hila; Mishne, Alon; Shoham, Sharon; Yang, Hongseok
Main reference H. Peleg, S. Shoham, E. Yahav, H. Yang, "Symbolic Automata for Static Specification Mining," in Proc. of the 20th Int'l Symp. on Static Analysis (SAS'13), LNCS, Vol. 7935, pp. 63–83, Springer, 2013; available as pre-print from the author's webpage.
URL http://dx.doi.org/10.1007/978-3-642-38856-9_6
URL http://www.cs.technion.ac.il/~yahave/papers/sas13-symaut.pdf

We present a framework for data-driven synthesis, aiming to leverage the collective programming knowledge captured in millions of open-source projects. Our framework analyzes code snippets and extracts partial temporal specifications. Technically, partial temporal specifications are represented as symbolic automata where transitions may be labeled by variables, and a variable can be substituted by a letter, a word, or a regular language. Using symbolic automata, we consolidate separate examples to create a database of snippets that can be used for semantic code-search and component synthesis. We have implemented our approach in a tool called PRIME and applied it to analyze and consolidate thousands of snippets per tested API.

7.4 Type Inhabitation Problem for Code Completion and Repair

Ruzica Piskac (Yale University, US)

Joint work of Gvero, Tihomir; Kuncak, Viktor; Kuraj, Ivan; Piskac, Ruzica
Main reference T. Gvero, V. Kuncak, I. Kuraj, R. Piskac, "Complete completion using types and weights," in Proc. of the 34th ACM SIGPLAN Conf. on Programming Language Design and Implementation (PLDI'13), pp. 27–38, ACM, 2013.
URL http://dx.doi.org/10.1145/2491956.2462192

Developing modern software typically involves composing functionality from existing libraries. This task is difficult because libraries may expose many methods to the developer. In this talk I will describe a project called InSynth. InSynth synthesizes and suggests valid expressions of a given type at a given program point, to help developers overcome the problems described in the above scenarios. As the basis of InSynth we use type inhabitation for lambda calculus terms in long normal form. For ranking solutions we introduce a system of weights derived from a corpus of code. I will conclude with an idea how to extend this approach so that it also automatically repairs ill-typed code expressions.

7.5 Learning a Program's usage of Dynamic Data Structures from Sample Executions

David White (Universität Bamberg, DE)

Joint work of White, David; Lüttgen, Gerald
Main reference D. White, G. Lüttgen, "Identifying Dynamic Data Structures by Learning Evolving Patterns in Memory," in Proc. of the 19th Int'l Conf. on Tools and Algorithms for the Construction and Analysis of Systems (TACAS'13), LNCS, Vol. 7795, pp. 354–369, Springer, 2013.
URL http://dx.doi.org/10.1007/978-3-642-36742-7_25

Programs making heavy use of pointers are notoriously difficult to understand and analyze, especially when the programmer is given the freedom allowed by languages such as C. We aim to simplify such analyses by employing machine learning and pattern matching to automatically identify the pointer-based dynamic data structures used by a program. Through observing a sample execution of a program, we are able to discover and label operations responsible for manipulating dynamic data structures. The output of the approach is used for program comprehension and to partially automate contract-based verification.

7.6 SMT-based Videogame Synthesis

Sam Bayless (University of British Columbia – Vancouver, CA)

Joint work of Bayless, Noah; Bayless, Sam

In recent years there has been interest in using formal methods to do constraint-based content generation for videogames, for example using Answer-Set Programming to generate mazes. We introduce an SMT solver for directed graph reachability, and show how it can be used to efficiently generate mazes, puzzles, and dungeons in the style of traditional 2D videogames. We demonstrate an actual working videogame using this technique to generate levels online, and in real-time.

8 System Demonstrations

8.1 Storyboard Programming of Data Structure Manipulations

Rishabh Singh (MIT – Cambridge, US)

Joint work of Singh, Rishabh; Solar-Lezama, Armando
Main reference R. Singh, A. Solar-Lezama, "Synthesizing Data Structure Manipulations from Storyboards," in Proc. of the 19th ACM SIGSOFT Symp. and the 13th Europ. Conf. on Foundations of Software Engineering (ESEC/FSE'11), pp. 289–299, ACM, 2011.
URL http://dx.doi.org/10.1145/2025113.2025153

We present the Storyboard Programming framework, a new synthesis system designed to help programmers write imperative low-level data-structure manipulations. The goal of this system is to bridge the gap between the "boxes-and-arrows" diagrams that programmers often use to think about data-structure manipulation algorithms and the low-level imperative code that implements them. The system takes as input a set of partial input-output examples, as well as a description of the high-level structure of the desired solution. From this information, it is able to synthesize low-level imperative implementations in a matter of minutes.

The framework is based on a new approach for combining constraint-based synthesis and abstract-interpretation-based shape analysis. The approach works by encoding both the synthesis and the abstract interpretation problem as a constraint satisfaction problem whose solution defines the desired low-level implementation. We have used the framework to synthesize several data-structure manipulations involving linked lists and binary search trees, as well as an insertion operation into an And Inverter Graph.

9 Working Groups

9.1 Comparing Inductive Logic and Inductive Functional Programming as well as other Approaches to Program Synthesis

Stephen Muggleton (Imperial College London, UK)

Refinement graphs in ILP and IFP

Muggleton and De Raedt explained how refinement graphs are central to the theory of Inductive Logic Programming (ILP). Logic programs consist of a set of definite clauses. When formulating a hypothetical logic program, clauses can be constructed by successive refinement operations. For instance the empty clause can be refined to a clause with a head by adding an atom with all variables distinct. Body atoms and variable bindings can be added by applying further refinements. Robert Henderson discussed how in his thesis he had adapted refinement theory from ILP to Inductive Functional Programming (IFP). A key idea was successive refinement operations which added functions, function application and lambda variables to the initially unspecified functional program $\perp$.

Predicate invention

Muggleton described how predicate invention was a key idea in early work on Inverse Resolution in ILP as well as more recent work on Meta-Interpretive Learning. Predicate invention involves the introduction of auxilliary definitions to support the induction of the target predicate. For instance, when learning a definition for reversing a list it may be necessary to invent a predicate which appends two lists. The analogous IFP notion of function invention was also shown to be valuable and applicable in both Kitzelman's talk on IGOR2 and Henderson's talk on his recently completed thesis. However, according to Kitzelman it is not possible for IGOR2 to invent partition and append when learning quicksort because of incompleteness in the search. Hernandez-Orallo pointed out the problems that can be produced by generating too many auxilliary predicates, which overwhelm the search.

Abstraction methods in Formal Methods

Gulwani explained how formal methods use logic-based techniques for the specification, development and verification of software. Within Formal Methods abstraction techniques are used to constructed simplified models of a program, such as a finite state machine, in order to allow model checking to be used in verification. There was a discussion about the applicability of such abstraction methods within Inductive Programming. Henderson point out that Abstraction made learning more effective within IFP. It was agreed that the use of MetaRules in Meta-interpretive Learning is related to abstraction, though this needs further investigation to understand the relationship in more detail. Michael Hansen pointed out that multiple levels of abstraction are used by humans when carrying out programming tasks.

Search and Constraint Solving – Sat Solvers

Search is a key element of Inductive Programming which is used to uncover hypotheses which are consistent with the given examples. De Raedt explained declarative languages and modelling can be used to specify an Inductive Learning problem. This approach can be used with constraint programming and Sat solvers to provide an efficient way of carrying out machine learning tasks. In this way the constraints are presented to a solver along with the input data, which generates candidate hypotheses.

Multi-task learning

It was discussed whether it is more effective to solve several inductive programming tasks together. For example, Gulwani's FlashFill system is aimed at inducing a broad class of Inductive transformations, which include reformatting social security numbers, extracting names from email addresses and formulating acronyms such as "IBM" from a company name such as "International Business Machines". Eyal Dechter suggested that it should be possible to build common re-usable libraries which can be used multiply across such tasks.

Probabilistic reasoning versus ranking

Most Inductive Functional Programming approaches, such as Gulwani's FlashFill, rank hypotheses according to a score. Within other areas of Machine Learning, including ILP, it has become common to rank hypotheses according to use Baysian ranking based on posterior probability of the hypotheses given the examples. The relationship between informal ranking schemes and Bayesian ones was discussed. One key difference identified was the ability to use well-founded methods of Bayes' prediction with the probabilistic approaches.

Probabilistic Programs as density estimators

Yura Perov discussed the relationship between Inductive Programming and Probabilistic Programming. In place of a deterministic algorithm, probabilistic programming involves the use of a probabilistic generative model to transform the input to the output. Probabilistic Programs are useful for machine learning since they can be used to directly represent a prior of a structural space. For instance, a prior might be a latent Dirichlet allocation model, and the program can be used to implement a Gibbs sampler for density estimation over a space of solutions. Inductive Programming and Probabilistic Programming have been combined over the last decade within the areas of Statistical Relational Learning and Probabilistic inductive Logic Programming.

Development of benchmarks and data and system repositories

The group discussed the importance of developing benchmarks and repositories for comparing approaches. It was agreed that it would be difficult to find a common data format which would be applicable to all Inductive Programming systems. However, it was pointed out that in practice such datasets are transformed by experimenters into the appropriate form before application of their particular system.

Teaching materials – Joint meetings IP and ILP – Summer school

There was a discussion about the value of arranging joint meetings between the IP and ILP communities. This could be done by way of workshop co-location and/or joint Summer Schools. The advantage of the latter is that it could be used to develop teaching materials for use in undergraduate and graduate courses.

9.2 Potential New Areas of Applications and Challenges for Inductive Programming

Benjamin Zorn (Microsoft Research – Redmond, US)

Joint work of Grigoryev, Alexey; Kitzelmann, Emanuel; White, David; Yahav, Eran; Perelman, Daniel; Bayless, Sam; Zorn, Ben

The focus of the working group was to think about the classes of applications for which inductive programming would provide significant advantages over other machine learning approaches. Identifying these applications would both drive the research agenda of the community in applying the technology to important problems and also serve the basis for commercial applications of the technology. Part of our discussion was to consider why existing machine learning approaches, which have been very successful for certain classes of problems, would not also be sufficient for these inductive programming applications. In our discussion, we found that the following qualities might distinguish inductive programming solutions from other ML approaches: abstraction level of the solution (captured by the operators and data types available in a domain-specific language), performance of the learning process and solution (fewer training examples needed and the ability to apply conventional code generation techniques to the result), and readability.

In thinking generally, we made some specific observations about the class of interesting applications. We felt that many classification tasks alone were not complex enough to allow

inductive programming solutions to outperform existing ML techniques. Furthermore, we felt that IP solutions to common tasks for which many traditional human-written implementations exist, such as scheduling an airline flight, were not good candidates for applications because synthesized solutions are likely to be significantly weaker than existing human-written ones. A better category of task would be one-off solutions to relatively simple problems for which an existing solution already written by a human is unlikely to exist. Examples of such tasks include extracting data, converting data formats, transforming text (as is currently done in Excel Flash Fill). These simple one-off applications have the negative aspect that it is unclear whether IP addresses the problem better than other ML techniques or not.

In thinking about IP and other ML techniques, we discussed the validity of the assumption that there are applications for which IP is a better solution. For example, could the things that are encoded in an IP solution (such as the set of operators and types in a domain-specific language) also be encoded as features in a neural net, etc. Is it possible that neural nets are equally effective as IP when the output of the neural net is a program? We discussed whether research addressing this distinction could be useful and help guide which IP applications provide the best opportunity. For example, does the existence of loops in IP solutions distinguish the class of applications for which IP is a better solution? Is there a middle-ground where some of the programming language artifacts present in IP solutions, such as types, recursion, and functional composition, are encoded into a neural net or other ML classification structure, getting the best of both approaches simultaneously.

In considering possible applications, we discussed several fruitful areas. Applications that require constraint solving, such as decision support, could be a productive area of investigation. Solutions expressed as programs have the property that they are human readable and hence can be checked for correctness and debugged as needed. Problems in this area often have a legitimate need for auditing. Hence applications related to manipulating data in spreadsheets or databases would be amenable to IP-based task synthesis solutions.

Reverse engineering of code from obfuscated sources is another possible area of application. Examples of possible areas where such an approach is needed include reverse engineering device drivers and code de-obfuscation. We all agreed that there are many common simple tasks on mobile devices or in specific applications that could be solved using IP. For example, simplifying the user interface to common mobile phone tasks (like sending a text message or checking the weather) or document handling (such as printing all files linked from a web page) would be valuable and result in relatively simple programs. Manipulating email was a particular area we agreed was both a real problem and a significant opportunity for applying IP.

We also discussed the problem of editing collections of pictures. The model where IP could be applied would be to edit one picture and then apply "similar" edits to the remaining collection of pictures, which might be quite large, saving time. A similar approach to editing presentations could be taken. Robots are another interesting source of opportunity. An approach where a robot is first trained to do a task by example and then uses IP to generalize that experience is compelling and appears to be an excellent fit for IP compared to other ML approaches. The same approach could be applied in a virtual space where an avatar is trained to do a task and then set free to repeat it indefinitely. Gold "farming" is a common task currently carried out in many virtual worlds by humans where automation could be commercially lucrative (although illegal by game rules). We also considered business process mining where many examples of traces of a process exist and a IP solution could extract structure from the trace data.

Another part of our discussion focused on challenges. One goal for identifying applications

is to push the state of the art on specific important problems that need to be solved with research. We discussed several related challenges. One issue that arose was how to reliably determine in many scenarios where one training example starts and ends. Partitioning a stream of observations effectively into training instances is a difficult problem in itself. In thinking about the duality of IP versus other forms of ML, we wondered if there was a class of DSLs where there could be an automatic translation from an IP-based solution to a neural net, etc. Other challenges we discussed include the common concerns around scale: in the size of the hypothesis and the amount of background knowledge required. We also consider the problem of applying datacenter computers at scale to important tasks to be a challenge and not well investigated. This would be motivated more by having a "grand challenge" application or benchmark to drive the research like the recent efforts to push deep learning techniques using large scale application of unsupervised learning. Another important challenge is understanding the user experience aspects of such systems, specifically related to expectations about what user behavior is relevant to learning and how the resulting program artifact relates to it, especially for when the user is a non-programmer.

9.3 Benchmarks and Metrics

José Hernández-Orallo (Polytechnic University of Valencia, ES), Marco Ragni (Universität Freiburg)

Joint work of Martínez-Plumed, Fernando; Schmid, Ute; Siebers, Michael; Solar-Lezama, Armando; Hernández-Orallo, José; Katayama, Susumu

Many different inductive programming systems (including those in inductive logic programming) have been developed in the past 40 years. One of the key features of inductive programming is the use of very general and expressive languages for examples, hypotheses and background knowledge: logic programming, functional programming, higher-order, constraints, etc. Another feature is the wide range of applications: program synthesis, data manipulation, artificial intelligence, robotics, programming by demonstration, etc. As a result, it has been very difficult to compare different inductive programming systems, as they can use different languages and are used in diverse applications. This lack of comparison makes it difficult to properly evaluate the improvement and real breakthrough of new systems, and also makes it difficult to tell when a new system is performing worse (or no better) than other previous systems.

The existence of benchmarks would make it easy to test and develop new systems in inductive programming, as well as also recognising inductive programming as a distinctive discipline, in terms of the kind of problems it can solve, rather than the language representation or its applications. Consequently, the motivation of this working group is to see the possibility of elaborating benchmarks and metrics for inductive programming, arranged in a form of a repository, in order to assess the capabilities and limitations of existing and future inductive programming systems.

Regarding the language, we seemed to agree that a common representation syntax should be used for the benchmark problems in the repository. We would need to define a standard, as the ARFF file format for attribute-value data. In order to use the problems for different representations and systems, we would develop converters for some common languages, such as Haskell and Prolog, to ensure that we cover the inductive functional programming and

inductive logic programming communities.

We were more precise about problem representation, and we identified that we needed to define the name of the function/predicate to be inferred, the datatypes involved and the examples. In the discussion, we clarified that a dataset does not configure a problem. Rather, from a dataset, we can do different sampling of examples and generate problems with more or less examples.

Moving from a repository to a benchmark requires further things. The first question is whether we are going to check solutions extensionally (over a test set) or intensionally (inspecting the code). In order to automate the process, it seems more reasonable to do this by separating between train and test cases and do it extensionally, as in machine learning.

In order to cover several domains, the working group suggested the following areas as a start:

- Traditional IP and ILP problems.
- AI problems, planning, robotics, etc.
- Program synthesis problems.
- Programming-By-Example problems.
- Trigonometry and other educational sources.
- IQ-like problems: number series, geometrical analogy problems (Raven's progressive matrices), Bongard Problems, etc.
- Grammatical inference problems.
- Structured prediction problems.
- Data manipulation and editing problems.

We identified several sources for these problems (such as IP and ILP repositories, but other sources, such as the program synthesis competition).

At the end of the meeting, we proposed a roadmap, starting with the problem representation, then going through a problem repository, before reaching the state of a benchmark repository or even a competition.

Participants

- Umair Zafrulla Ahmed
Indian Institute of Technology – Kanpur, IN
- Sam Bayless
University of British Columbia – Vancouver, CA
- Tarek R. Besold
Universität Osnabrück, DE
- Luc De Raedt
KU Leuven, BE
- Eyal Dechter
MIT – Cambridge, US
- Alexey Grigoryev
National Nuclear Research University MEPhI, RU
- Sumit Gulwani
Microsoft Res. – Redmond, US
- Mike Hansen
Indiana University – Bloomington, US
- Robert J. Henderson
Imperial College London, GB
- José Hernández-Orallo
Polytechnic University of Valencia, ES
- Petra Hofstedt
TU Cottbus, DE
- Frank Jäkel
Universität Osnabrück, DE
- Susumu Katayama
University of Miyazaki, JP
- Dileep Kini
University of Illinois – Urbana Champaign, US
- Emanuel Kitzelmann
Univ. Duisburg – Essen, DE
- Mark Marron
Microsoft Res. – Redmond, US
- Fernando Martínez-Plumed
Polytechnic University of Valencia, ES
- Martin Möhrmann
Universität Osnabrück, DE
- Stephen H. Muggleton
Imperial College London, GB
- Daniel Perelman
University of Washington – Seattle, US
- Iurii Perov
University of Oxford, GB
- Ruzica Piskac
Yale University, US
- Oleksandr Polozov
University of Washington – Seattle, US
- Marco Ragni
Universität Freiburg, DE
- Ute Schmid
Universität Bamberg, DE
- George M. Sergievsky
National Nuclear Research University MEPhI, RU
- Michael Siebers
Universität Bamberg, DE
- Rishabh Singh
MIT – Cambridge, US
- Armando Solar-Lezama
MIT – Cambridge, US
- Janis Voigtländer
Universität Bonn, DE
- David White
Universität Bamberg, DE
- Eran Yahav
Technion – Haifa, IL
- Benjamin Zorn
Microsoft Res. – Redmond, US