Report from Dagstuhl Seminar 25491

Approaches and Applications of Inductive Programming

Ute Schmid*[1], Gust Verbruggen*[2], and Sonja Niemann†[3]

1 **Universität Bamberg, DE.** ute.schmid@uni-bamberg.de
2 **Microsoft – Keerbergen, BE.** gustverbruggen@gmail.com
3 **bidt – München, DE.** sonja.niemann@bidt.digital

Abstract

The Dagstuhl Seminar "Approaches and Applications of Inductive Programming" (AAIP) took place in 2025 for the seventh time. The focus of this seminar series is methods and applications of learning computer programs from incomplete and informal specifications such as input/output examples or natural language prompts. Researchers come from different areas, mostly from machine learning and other branches of artificial intelligence research, cognitive scientists interested in human learning in complex domains, and researchers with a background in formal methods and programming languages. The focus of the AAIP 2025 seminar was the application of large language models to code generation and the evaluation of quality of generated code in comparison with other, classic inductive programming approaches. Furthermore, neuro-symbolic approaches to inductive programming, were explored. Applications of inductive programming in different domains such as biomedical scientific discovery, control code generation in complex industrial settings, and programming education were discussed.

Seminar November 30 – December 5, 2025 – https://www.dagstuhl.de/25491
2012 ACM Subject Classification Computing methodologies → Artificial intelligence; Human-centered computing → Human computer interaction (HCI); Theory of computation → Logic; Computing methodologies → Machine learning; Theory of computation → Semantics and reasoning; Software and its engineering → Software creation and management
Keywords and phrases Diffusion Models, Inductive Programming, LLMs, Program Synthesis, Programming by Examples
Digital Object Identifier 10.4230/DagRep.15.11.134

1 Executive Summary

Ute Schmid (Universität Bamberg, DE)
Gust Verbruggen (Microsoft – Keerbergen, BE)

Inductive programming (IP) research, also called inductive program synthesis, is concerned with approaches to learn computer programs from incomplete and informal specifications, such input/output examples or natural language prompts. IP research incorporates different areas of computer science, especially machine learning, automated reasoning, program verification, programming language theory, and software engineering. Furthermore, IP has strong relations to cognitive science, where IP can help build models of human inductive learning, and to provide intelligent tutoring systems for programming education. IP is also relevant to industry, where it supports tools for end-user programming.

* Editor / Organizer
† Editorial Assistant / Collector

Approaches and Applications of Inductive Programming, *Dagstuhl Reports*, Vol. 15, Issue 11, pp. 134–156
Editors: Ute Schmid, Gust Verbruggen, and Sonja Niemann
Dagstuhl Reports
Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

The Dagstuhl Seminar "Approaches and Applications of Inductive Programming" (AAIP) took place in 2025 for the seventh time. Given the recent developments in generative methods and their applications to code generation, the focus topic of the seminar was the application of large language models to code generation and the evaluation of quality of generated code in comparison with other classic inductive programming approaches. Recent research on large language models as well as diffusion methods for code generation from examples were presented. Suitable benchmarks and evaluation strategies taking into account reliability and quality of generated code were discussed. Furthermore, advances in inductive logic programming, functional program synthesis, and probabilistic programming were presented.

Cognitive science research on IP approaches as models for human concept learning, abstract visual reasoning and problem solving were presented together with empirical results. Furthermore, applications of IP methods for intelligent tutoring in the domain of programming education were discussed. IP methods have been matured over the last decade such that applications in complex real world domains can be tackled. Applications of classic as well as generative IP methods in scientific discovery in biomedicine as well as control code generation in complex industrial settings, were presented.

Core aspects of ongoing IP research were discussed in four working groups: Neuro-symbolic program induction, gradual refinement methods for concept induction and human-like iterative programming, programming education in the age of large language models, inductive programming for complex biological problems, and IP benchmarks and evaluation strategies suitable for comparing performance and quality of programs generated with classic as well as generative IP approaches.

Based on the presentations and in-depth discussions of ongoing research in IP, there was a general agreement that despite the impressive progress of generative methods for code generation, inductive programming is not solved. The main challenges are in combinations of generative approaches with other methods of inductive programming and machine learning to support generation of programs that are reliable and comprehensible. Furthermore, there was a general agreement between participants that the interdisciplinary exchange between core IP researchers with cognitive scientists, and programming language researchers was highly fruitful. Finally, the participation of researchers from industry was perceived as very inspiring, showing advances and challenges for real world applications of inductive programming.

2 Table of Contents

3 Overview of Talks

3.1 Concept Learning as Coarse-To-Fine Probabilistic Program Synthesis

Maddy Bowers (MIT – Cambridge, US)

Joint work of Maddy Bowers, Mauricio Barba da Costa, Xiaoyan Wang, Joshua Tenenbaum, Vikash Mansinghka, Armando Solar-Lezama, Alexander Lew

In this talk I'll discuss a family of new concept induction techniques inspired by two key aspects of human concept learning – our ability to judge how promising a vague, partial hypothesis is, and our ability to gradually refine these vague explanations of observations to precise ones. First, I will argue that coarse hypotheses should be represented as probabilistic programs that combine deterministic logic that explains part of the program structure with probabilistic sampling to encode uncertainty about what else might be going on. Second, I'll show how we can measure the quality of one of these coarse hypotheses precisely by calculating likelihoods of input-output examples, and present Pluck, a new tool we have developed computes these likelihoods really efficiently. Third, I'll show how these likelihood scores can be used to guide synthesis, and finally I'll conclude with some preliminary experimental results.

3.2 Hypothesis Space Processing for Efficient Rule Learning Through Inductive Logic Programming

David M. Cerna (Dynatrace Research – Linz, AT)

Joint work of David M. Cerna, Andrew Cropper
Main reference Andrew Cropper, David M. Cerna, Matti Järvisalo: "Symmetry breaking for inductive logic programming", CoRR, Vol. abs/2508.06263, 2025.
URL https://doi.org//10.48550/ARXIV.2508.06263
Main reference Andrew Cropper, Filipe Gouveia, David M. Cerna: "Honey, I shrunk the hypothesis space (through logical preprocessing)", CoRR, Vol. abs/2506.06739, 2025.
URL https://doi.org//10.48550/ARXIV.2506.06739
Main reference Andrew Cropper, David M. Cerna: "Efficient rule induction by ignoring pointless rules", CoRR, Vol. abs/2502.01232, 2025.
URL https://doi.org//10.48550/ARXIV.2502.01232

Inductive Logic programming (ILP) combines knowledge representation and machine learning [7, 3]. The goal is to search a hypothesis space, consisting of logic programs, to find a hypothesis that generalizes the given training examples and background knowledge. Early ILP approaches such as *foil* iteratively specialize rules till only positive examples are accepted and repeat this process until all positive examples are accepted. This process was refined in systems like *Aleph* by restricting the search space of specializations to hypotheses that θ-subsume the so-called *bottom clause*, i.e, the most specific clause which accepts a given positive example.

Modern ILP often follows the *meta-learning* approach, i.e., perform search through an encoding of the hypothesis space. In particular, *Popper* [4] encodes the generation of logic programs modulo the background knowledge as an ASP program (*generate phase*). As *Popper* tests the hypothesis produced by the generate phase, additional constraints are added guiding the search at subsequent steps. This leads to a significant decrease in the number

of tested hypotheses while maintaining optimality guarantees. These constraints are based on propositional subsumption and are thus computationally feasible to check. Popper's approach to ILP has led to significant advancements in the tasks such systems can handle. For example learning with *negative invention* [1], *Higher-order* [8], *very large programs* [5], and competative performance on ARC [6].

In this abstract, we focus on alternative constraints based on the semantics of the background knowledge and **symmetry breaking**, which can improve performance in all of the above areas. For example, in [2] we introduce the notion of *reducible* literal, i.e., given a rule r containing l, the following holds: $B \models r \leftrightarrow r \setminus \{l\}$ where B is the background knowledge. We use this concept to introduce additional constraints while learning. In some cases, we can constrain the hypothesis space prior to learning, for example, if a predicate in the background knowledge is satisfied by a finite number of instantiations. This limits the number of times it can occur in a rule. For example, consider the following where $r_1 \preceq_\theta r_2$:

$$B = \{edge(a,b),\ edge(b,c),\ edge(c,a)\}$$
$$r_1 = h \leftarrow edge(A,B),\ edge(B,C),\ edge(C,D),\ edge(D,E)$$
$$r_2 = h \leftarrow edge(A,B),\ edge(B,C),\ edge(C,A)$$
$$\theta = \{D \mapsto A, E \mapsto B\}$$

We refer to this property as *recall redundant*, similar properties are definable for literal combinations that make a rule unsat, literal combinations that are always true, and implication relations between literals. A final approach to constraining the hypothesis space is to remove rule variants. This is achieved by ordering the literals by their argument tuples and normalizing with respect to certain properties.

$$r_1 : h(A,B) \leftarrow p(A,E), p(B,\mathbf{C}), p(C,D).$$
$$r_2 : h(A,B) \leftarrow p(A,C), p(B,E), p(C,\mathbf{D}).$$

Observe $r_1\sigma_2 = r_2$ where $\sigma_2 = \{E \mapsto C, C \mapsto E\}\{E \mapsto D\}$. In the above case, we order the variables and ensure that variables are witnessed by smaller variables (with respect to a variable order). This normalization is sound (does not remove all variants), but incomplete. We will cover these approaches in more detail during the talk.

References

1 David M. Cerna and Andrew Cropper. Generalisation through negation and predicate invention. *AAAI*, 38(9):10467–10475, 2024.

2 Andrew Cropper and David M. Cerna. Efficient rule induction by ignoring pointless rules, 2025.

3 Andrew Cropper and Sebastijan Dumancic. Inductive logic programming at 30: A new introduction. *J. Artif. Intell. Res.*, 74:765–850, 2022.

4 Andrew Cropper and Rolf Morel. Learning programs by learning from failures. *Mach. Learn.*, 110(4):801–856, 2021.

5 Céline Hocquette, Andreas Niskanen, Rolf Morel, Matti Järvisalo, and Andrew Cropper. Learning big logical rules by joining small rules. In *IJCAI*, pages 3430–3438, 2024.

6 Céline Hocquette and Andrew Cropper. Relational decomposition for program synthesis, 2025.

7 Stephen Muggleton. Inductive logic programming. *New Generation Computing*, 8(4):295–318, 1991.

8 Stanislaw J. Purgal, David M. Cerna, and Cezary Kaliszyk. Learning higher-order logic programs from failures. In Luc De Raedt, editor, *IJCAI*, pages 2726–2733, 2022.

3.3 Learning cancer programs

Jasmin Fisher (University College London, GB)

Digital twins of cancer tumours are emerging as transformative tools in oncology, enabling more personalised and adaptive treatment strategies. In this talk, I will present a growing library of cancer digital twins spanning multiple cancer types, including breast, lung, skin, and brain. These interpretable, executable models capture molecular mechanisms within tumour cells and their microenvironment, allowing us to uncover and anticipate treatment-resistance mechanisms and design patient-specific strategies to improve outcomes. Currently, these models are manually constructed by experts, a process that is time-consuming and costly. The next challenge is to harness generative AI to learn from vast cancer datasets and automatically build explainable, executable programs ("glass-box" models). This approach would accelerate model development, enhance scalability and align with the goals of inductive programming, making it a timely topic for the Dagstuhl Seminar.

3.4 Computer-Aided Programming and Problem Solving

Frank Jäkel (TU Darmstadt, DE)

Computers are tools for thought. Scientists and engineers use computers to solve problems that they cannot solve without them. However, more often than not learning the relevant tools is cumbersome. An important goal of basic research in cognitive science is to build the conceptual foundations for future AI tools that naturally support human problem solving. In this talk I will present the results of two studies on how people solve two important classes of problems: constraint-satisfaction problems and guesstimation problems. I will also discuss how these results might inform the development of human-AI interaction.

3.5 Inductive Functional Programming: Igor2 (vs ChatGPT) and Generalized I/O-Examples

Emanuel Kitzelmann (Technische Hochschule Brandenburg, DE)

The inductive functional programming system Igor2 induces recursive functions from i/o-examples over user-defined algebraic data types. In this talk, I review its refinement operators using the task of swapping two elements in a list at specified indices, assuming no library functions are available (only list constructors and integers). I demonstrate that even ChatGPT 5.1 still struggles with this problem. Based on an example from the Lists benchmark for inductive programming, I introduce and motivate the problem of generalizing i/o-examples as a preprocessing step for recursive generalization.

3.6 LLM-based feature generation from text for interpretable machine learning

Tomáš Kliegr (University of Economics – Prague, CZ)

Joint work of Vojtech Balek, Lukás Sýkora, Vilém Sklenák, Tomáš Kliegr
Main reference Vojtech Balek, Lukás Sýkora, Vilém Sklenák, Tomáš Kliegr: "LLM-based feature generation from text for interpretable machine learning", Mach. Learn., Vol. 114(11), p. 241, 2025.
URL https://doi.org//10.1007/S10994-025-06867-1

Applying symbolic rule learning to unstructured text is difficult: bag-of-words features produce sparse, brittle rules, while neural embeddings deliver accuracy at the cost of transparency. We bridge this gap in two complementary ways. First, we present SMER, a transparency approach for classifiers built from averaged word embeddings + logistic regression, where logit linearity enables an exact (perfect-fidelity) decomposition of document decisions into word-level contributions. Second, we propose a structural approach that uses LLMs to map documents to low-dimensional, human-interpretable concepts (e.g., rigor, novelty, replicability), which can be directly consumed by symbolic learners such as action rules. On scientometric benchmarks, fusing LLM-derived concepts with standard text features improves predictive performance over either representation alone, while remaining interpretable. We also address semantic bias in LLM prompting – where priors can override logic – via the "Meaningless is better" hypothesis: hashing bias-inducing words into internally referenceable strings improves performance on logical and statistical tasks. We conclude with an ecosystem overview, including the FeatureLab application and a GPU-accelerated action-rules package, and illustrate applications in research-quality prediction and microbial media modeling.

References

1 V. Balek, L. Sýkora, V. Sklenák, and T. Kliegr, "LLM-based feature generation from text for interpretable machine learning," *Machine Learning*, vol. 114, no. 11, p. 241, 2025. doi: 10.1007/s10994-025-06867-1.
2 L. Dvořáčková, M. P. Joachimiak, M. Černý, A. Kubecová, V. Sklenák, and T. Kliegr, "Explaining word embeddings with perfect fidelity: a case study in predicting research impact," *Machine Learning*, vol. 114, no. 12, p. 265, 2025. doi: 10.1007/s10994-025-06870-6.
3 M. Chadimová, E. Jurášek, and T. Kliegr, "Meaningless is better: Hashing bias-inducing words in LLM prompts improves performance in logical reasoning and statistical learning," *Information Processing & Management*, vol. 63, no. 3, p. 104493, 2026. doi: 10.1016/j.ipm.2025.104493.
4 L. Sýkora and T. Kliegr, "action-rules: GPU-accelerated Python package for counterfactual explanations and recommendations," *SoftwareX*, vol. 29, p. 102000, 2025. doi: 10.1016/j.softx.2024.102000.
5 P. Máša, T. Kliegr, and M. P. Joachimiak, "Explainable rule-based prediction of cultivation media for microbes," *Computational and Structural Biotechnology Journal*, vol. 27, pp. 5194–5206, 2025. doi: 10.1016/j.csbj.2025.10.014.

3.7 Generating Industrial Control Logic from Natural Language Requirements

Heiko Koziolek (ABB – Mannheim, DE)

Joint work of Heiko Koziolek, Thilo Braun, Virendra Ashiwal, Sofia Linsbauer, Marthe Ahlgreen Hansen, Karoline Grotterud
Main reference Heiko Koziolek, Thilo Braun, Virendra Ashiwal, Sofia Linsbauer, Marthe Ahlgreen Hansen, Karoline Grotterud: "Spec2Control: Automating PLC/DCS Control-Logic Engineering from Natural Language Requirements with LLMs – A Multi-Plant Evaluation", CoRR, Vol. abs/2510.04519, 2025.
URL https://doi.org//10.48550/ARXIV.2510.04519

Distributed control systems (DCS) manage the automation for many industrial production processes (e.g., power plants, chemical refineries, steel mills). Programming the software for such systems remains a largely manual and tedious process, incurring costs of millions of dollars for extensive facilities. Large language models (LLMs) have been found helpful in generating DCS control logic, resulting in commercial copilot tools. Today, these tools are focused on textual notations, they provide limited automation, and have not been tested on large datasets with realistic test cases. We introduce Spec2Control, a highly automated LLM workflow to generate graphical control logic directly from natural language user requirements. Experiments using an open dataset with 10 control narratives and 65 complex test cases demonstrate that Spec2Control can successfully identify control strategies, can generate 98.6% of correct control strategy connections autonomously, and can save between 94-96% of human labor. Spec2Control is being integrated into commercial ABB engineering tools, but is also available as an open-source variant for independent validation.

3.8 On Solving Visual Analogies

Johannes Langer (Universität Bamberg, DE)

Joint work of Johannes Langer, Ute Schmid
Main reference Johannes Langer, Ute Schmid: "Flexible Segmentation for Rule Learning over Object Representations in Abstract Visual Reasoning", in Proc. of the Twelfth Annual Conference on Advances in Cognitive Systems, 2025.
URL https://openreview.net/forum?id=PSsAsrCkFx

The ability to reason in abstract domains is a key component of human intelligence, and is tested in many intelligence tests such as Raven's Progressive Matrices (RPM). Such tests often require humans to perform analogical reasoning over visually presented problem premises. A more recent and AI-focused example of such a test is the Abstraction and Reasoning Corpus (ARC). To solve such analogies in a human-like manner, we attempted to use Inductive Logic Programming due to its ability to learn relations from little training data. This talk discussed the challenges of that approach and new ideas which arise from it.

3.9 Evaluating Large Language Models in Programming-by-Example

Félix Martí Pérez (Technical University of Valencia, ES)

Traditionally, PBE relied on Inductive Programming (IP) systems based on symbolic logic. However, the rise of Large Language Models (LLMs) offers a powerful alternative. This presentation starts by explaining our past work on evaluating the performance of LLMs against traditional IP solvers (MagicHaskeller) on a curated dataset of both standard algorithmic problems and novel "atypical" tasks. We demonstrate that while LLMs offer superior flexibility, they exhibit significant language bias (favouring Python over Haskell) and struggle with novel logic patterns. We then summarise recent research on evaluating LLMs within the PBE paradigm and conclude by outlining current limitations and promising future directions for building more robust and general PBE benchmarks.

3.10 Beyond Performance: Towards Reliable Large Language Models

Yael Moros Daval (Technical University of Valencia, ES)

Joint work of Lexin Zhou, Wout Schellaert, Fernando Martínez-Plumed, Yael Moros-Daval, Cèsar Ferri, José Hernández-Orallo
Main reference Lexin Zhou, Wout Schellaert, Fernando Martínez-Plumed, Yael Moros-Daval, Cèsar Ferri, José Hernández-Orallo: "Larger and more instructable language models become less reliable", Nat., Vol. 634(8032), pp. 61–68, 2024.
URL https://doi.org//10.1038/S41586-024-07930-Y

Although scaling and alignment improve LLM capability, they also undermine reliability. Models continue to make errors on simple tasks, and difficulty no longer predicts where failures will occur. Refusal behaviour diminishes as models produce fluent but incorrect answers that users often fail to detect. Confidence-estimation and abstention methods reintroduce calibrated uncertainty signals, helping models recognise low-confidence situations and avoid misleading outputs.

3.11 ConDABench: Interactive Evaluation of Data Analysis Assistants

Arjun Radhakrishna (Microsoft – Redmond, US)

Joint work of Avik Dutta, Priyanshu Gupta, Hosein Hasanbeig, Rahul Pratap Singh, Harshit Nigam, Sumit Gulwani, Arjun Radhakrishna, Gustavo Soares, Ashish Tiwari
Main reference Avik Dutta, Priyanshu Gupta, Hosein Hasanbeig, Rahul Pratap Singh, Harshit Nigam, Sumit Gulwani, Arjun Radhakrishna, Gustavo Soares, Ashish Tiwari: "ConDABench: Interactive Evaluation of Language Models for Data Analysis", CoRR, Vol. abs/2510.13835, 2025.
URL https://doi.org//10.48550/ARXIV.2510.13835

Real-world data analysis tasks often come with under-specified goals and unclean data. In the setting of data analysis assistants, user interaction is necessary to understand and disambiguate a user's intent, and hence, is essential for solving these complex tasks. Existing benchmarks for evaluating LLM-driven data analysis assistants on these tasks do not capture

these complexities or provide first-class support for interactivity. In this task, I will discuss ConDABench, a benchmarking framework and a collection of tasks for evaluating data analysis assistants in the interactive setting. In ConDABench, a user-proxy agent plays the role of the user and is responsible for steering the assistant and responding to any questions or clarifications from the assistant. ConDABench provides a set of 1420 real-world data analysis tasks, each of which asks the data analysis assistant to reproduce key results from public data journalism articles, starting from the raw unclean data. ConDABench is an avenue for model builders to measure progress towards truly collaborative models that can complete complex interactive data analysis tasks.

3.12 A Short History of Inductive Programming

Ute Schmid (Universität Bamberg, DE)

In the talk, the history of the research field of inductive programming (IP) and the Dagstuhl Seminar series (since 2013) as well as the preceding workshops on Approaches and Applications of Inductive Programming (AAIP, since 2005) has been presented. Inductive programming research (see Flener and Schmid 2008; 2024) started in the 1970ies with a focus on construction of functional (Lisp) programs from input/output examples. First approaches were data-driven, that is, generalisazion over the structure of the examples. These were complemented with generate-and-test approaches for functional as well as logic program synthesis. Currently, focus is on prompt-based generation from LLMs. Besides algorithmic approaches to inductive programming, in the talk relations to cognitive science research on learning complex productive rules from examples (e.g., Schmid and Kitzelmann, 2011; Rule et al., 2024) as well as programming education (e.g., Phung et al., 2023) have been presented.

References

1 Flener, P., & Schmid, U. (2008). An introduction to inductive programming. Artificial Intelligence Review, 29(1), 45-62.
2 Flener, P., & Schmid, U. (2024). Inductive Programming. In Encyclopedia of Machine Learning and Data Science. Springer, New York, NY.
3 Schmid, U., & Kitzelmann, E. (2011). Inductive rule learning on the knowledge level. Cognitive Systems Research, 12(3-4), 237-248.
4 Rule, J. S., Piantadosi, S. T., Cropper, A., Ellis, K., Nye, M., & Tenenbaum, J. B. (2024). Symbolic metaprogram search improves learning efficiency and explains rule learning in humans. Nature Communications, 15(1), 6847.
5 Phung, T., Pădurean, V. A., Cambronero, J., Gulwani, S., Kohn, T., Majumdar, R., ... & Soares, G. (2023, August). Generative AI for programming education: benchmarking ChatGPT, GPT-4, and human tutors. In Proceedings of the 2023 ACM Conference on International Computing Education Research-Volume 2 (pp. 41-42).

3.13 Prompt Programming

Adish Singla (MPI-SWS – Saarbrücken, DE)

URL https://www.promptprogram.org/

Computing students increasingly rely on generative AI tools for programming assistance, often without formal instruction or guidance. This change highlights the need to teach students how to interact effectively with AI models, particularly through natural-language prompts, to generate and critically evaluate code when solving computational tasks. To address this, we developed a novel platform for prompt programming. This talk will demonstrate the platform, discuss initial results, and highlight plans for future work.

3.14 Execution-guided within-prompt search for programming-by-example

Gust Verbruggen (Microsoft – Keerbergen, BE) and Mukul Singh (Microsoft – Redmond, US)

Joint work of Gust Verbruggen, Sumit Gulwani, Vu Le, Ashish Tiwari, Mukul Singh
Main reference Gust Verbruggen, Ashish Tiwari, Mukul Singh, Vu Le, Sumit Gulwani: "Execution-guided within-prompt search for programming-by-example", in Proc. of the The Thirteenth International Conference on Learning Representations, 2025.
URL https://openreview.net/forum?id=PY56Wur7S0

Large language models (LLMs) can generate code from examples without being limited to a DSL, but they lack search, as sampled programs are independent. We use an LLM as a policy that generates lines of code and then join these lines of code to let the LLM implicitly estimate the value of each of these lines in its next iteration. We further guide the policy and value estimation by executing each line and annotating it with its results on the given examples. This allows us to search for programs within a single (expanding) prompt until a sound program is found, by letting the policy reason in both the syntactic (code) and semantic (execution) space. We evaluate within-prompt search on straight-line Python code generation using five benchmarks across different domains (strings, lists, and arbitrary Python programming problems). We show that the model uses the execution results to guide the search and that within-prompt search performs well at low token budgets. We also analyze how the model behaves as a policy and value, show that it can parallelize the search, and that it can implicitly backtrack over earlier generations.

3.15 Cyclic Bayesian networks: From probabilistic modelling to probabilistic programming

Felix Weitkämper (German UDS – Potsdam, DE)

Joint work of Vera Koponen, Felix Weitkämper

Bayesian networks are a popular target formalism for statistical/probabilistic machine learning as well as knowledge-based model construction. So what distinguishes learning Bayesian networks from inductive probabilistic programming? In particular, what is missing from

Bayesian networks to be counted a full probabilistic programming language? In this talk, I argue that after relational Bayesian networks [1] have addressed a key expressive deficiency, recursion is the main missing ingredient. For fully random Bayesian networks, in which all probabilities are properly between 0 and 1, approaches based on irreducible Markov chains provide a uniquely determined semantics for networks with cycles [2, 3]. I argue that fixed-point methods from traditional logic and discrete mathematics are able to give semantics also to a large class of cyclic networks with deterministic constraints, which suffices for instance to embed probabilistic logic programming.

References

1 Manfred Jaeger. Relational Bayesian networks. Proceedings of UAI 1997, pp. 266–273.
2 Mieczysław Kłopotek. Cyclic Bayesian network: Markov process approach. Studia Informatica: systems and information technology 1(7), 47–55 (2006)
3 Christel Baier, Clemens Dubslaff, Holger Hermanns, Nikolai Käfer. On the Foundations of Cycles in Bayesian Networks. Principles of Systems Design. Lecture Notes in Computer Science, vol 13660, pp 343-363, 2022

4 Working groups

4.1 Inductive programming for complex biological problems

David Cerna (Dynatrace Research – Linz, AT), Jasmin Fisher (University College London, GB), Pierre Flener (Uppsala University, SE), Tomáš Kliegr (University of Economics – Prague, CZ), Heiko Koziolek (ABB – Mannheim, DE), Raoul van Doren (ETH Zürich, CH), and Felix Weitkämper (German UDS – Potsdam, DE)

The motivating problem of the working group is the induction of gene regulatory networks in biology. Those networks are modelled as qualitative networks, a multi-valued generalisation of Boolean networks. Gene regulatory networks are highly relevant for understanding the mechanism of tumor development.

Some of the defining features of this domain, which add intrinsic interest as a target domain for inductive programming approaches, are non-determinism and heavy use of recursion. Furthermore, transparency of both the final output and the methodology used to get there is essential for trusting the discovered networks.

Recent developments in symbolic and sub-symbolic machine learning raise the prospect of inducing such gene regulatory models directly from data. This raises several challenges:

The first challenge is gathering reliable data in a machine-readable format. Despite recent progress in the availability of structured databases, their reliability can be variable. Due to the discrete nature of the systems, single errors can have unforeseen consequences. Hence, tight quality control with human oversight must be integrated into the data curation process.

The second challenge is the induction of the qualitative network from the garnered data. This involves integrating these various forms of examples into a transparent learning framework that provides appropriate guarantees. Existing approaches involve augmenting human-created network structure with a combination logic supported by machine learning.

The third challenge is maintaining the integrity of existing gene regulatory networks in the face of new data. This extends the traditional belief revision problem by the issue of potentially augmenting the vocabulary involved in the network.

4.2 Neuro-Symbolic Program Induction

Quinten Dewulf (KU Leuven, BE), Wang-Zhou Dai (Nanjing University – Suzhou, CN), Cesar Ferri Ramirez (Technical University of Valencia, ES), Johannes Fürnkranz (Johannes Kepler Universität Linz, AT), Johannes Langer (Universität Bamberg, DE), and Zachary Siegel (MIT – Cambridge, US)

4.2.1 Two Types of Neuro-Symbolic Program Induction

In this discussion group we focus on neuro-symbolic program induction. It quickly became apparent that there was an important distinction between two ways to interpret the topic of neuro-symbolic program induction – as *[Neuro-Symbolic] Program Induction* and *[Neuro-Symbolic Program] Induction*. The former addresses neuro-symbolic approaches for primarily inducing *symbolic* programs. Examples for this line of research include discretizing non-symbolic inputs for symbolic program induction [4], or neurally guiding program search as in DreamCoder [3]. The latter describes the problem of inducing programs which themselves are inherently neuro-symbolic in nature.

4.2.2 What are the benefits of using neuro-symbolic methods to synthesize symbolic programs?

Consider two techniques for synthesizing programs: a symbolic-only system that performs enumerative search over a DSL, and a large language model (LLM) from which we sample programs. The symbolic-only system comes with completeness guarantees and can explore programs that an LLM would not otherwise discover if those types of programs are not within the LLM's training distribution. In addition, particularly if the domain is known or constrained, heuristics can be used to guide search over programs to make the system quite powerful. On the other hand, such systems will struggle to produce longer programs since the search space increases exponentially with the depth of the program. Although techniques such as library learning and forming abstractions can help, an unguided search will likely struggle to work in a general setting and lacks the type of "common sense" knowledge that people may use when writing code.

The neural-only system can sample programs effectively that are within the training distribution of LLMs. We see the success of such methods due to the prevalence of using coding assistants, which are already in high use by software developers. These systems, however, struggle to generate code out of distribution, and additionally struggle to manage large code bases. Each piece of code sampled from an LLM is relatively expensive compared to a piece of code generated via search, so the LLM's prior is highly important for generating useful programs.

These benefits naturally raise the question of whether a neuro-symbolic system could obtain the advantages of both methods; namely, combining the "common sense" statistical knowledge of neural networks with the completeness guarantees and other heuristics of symbolic synthesizers. One example is DreamCoder [3], which uses a neural network to guide symbolic search over programs. The neural network can be viewed as a heuristic that accelerates search, allowing the system to discover programs that would not be encountered in a reasonable time frame otherwise. We believe that future promising work might use neural methods such as LLMs to guide symbolic search for programs, potentially in combination with other heuristics to guide program search more effectively.

4.2.3 What is a neuro-symbolic program?

There are already several languages for representing neuro-symbolic programs. Best-known is possibly DeepProbLog [6], which extends probabilistic logic programming with subsymbolic predicates that can, for example, be defined with deep neural networks. Similarly, NeurASP [9] extends Answer Set Programming with neural networks for estimating a probability distribution over atomic facts. Scallop [5] is a language similar to Datalog, which features an underlying differentiable reasoning engine. ULLER [8] is an attempt to unify various approaches and provide a unified framework for learning and reasoning.

However, all these approaches aim at extending conventional programming paradigms (such as Python, Prolog, ASP, etc.) with mechanisms for bringing in neurosymbolic predicates or functions. In all cases, the program itself has a clearly identifiable symbolic "spine" that connects neurosymbolic predicates or functions. The question of what constitutes a neuro-symbolic program may go deeper and could require re-thinking the paradigm of programming. *Vibe programming*, where one obtains a conventional program through an iterative neuro-symbolic refinement process, is one example. A related possibility is that the *process* that defines the final program may itself be considered the program, and the final program text be less central – analogous to how compilers made machine and assembly code largely obsolete for many programmers. In the seminar we saw several neuro-symbolic programming paradigms that foreshadow such developments.

4.2.4 What are the benefits of synthesizing neuro-symbolic programs?

The main benefits of neuro-symbolic programs as a representation come from inheriting advantages of both symbolic and neural representations. Symbolic programs are naturally verifiable and interpretable, since we understand how they work, which makes them more usable in domains requiring performance guarantees. Relatedly, they can be more accurate for specific problems compared to neural methods: if we understand the structure of a problem, a symbolic solution can provide guarantees on finding solutions. Symbolic programs can also generalize well by leveraging representations tailored to the problem; by learning appropriate abstractions, solutions can become reusable.

Despite these benefits, compared to neural techniques, symbolic programs can be harder to scale with access to additional data, since there generally is not an analogue of "taking a gradient step" to update a symbolic program in response to data. With access to large datasets, neural methods can be scaled up to be quite general in the sense of covering a broad distribution of inputs. This scalability is shown clearly in the success of large neural models like LLMs, which achieve impressive performance across a wide range of tasks.

There are often situations where a problem can be well specified with some outer program structure, even if certain internal components may be better represented with neural predicates. For instance, consider controlling the behavior of a self-driving car at a traffic stop. There is a well-defined symbolic behavior (when the light is green, proceed; when the light is red, stop). On the other hand, identifying the color of the traffic light may be easier to specify using a neural predicate. By combining such neural predicates with symbolic decision flow, neuro-symbolic programs can provide additional guarantees and interpretability over fully neural systems, and additional flexibility over symbolic-only models.

4.2.5 Why induce neuro-symbolic programs in a joint fashion?

Inducing neuro-symbolic programs by jointly learning neural weights and symbolic knowledge is, in general, not reducible to simpler symbolic program induction or to learning only neural weights. Many methods in neuro-symbolic AI rely on hand-engineered symbolic knowledge

to provide guarantees, data efficiency, and improved accuracy when training their neural components. This makes such models highly dependent on the quality and completeness of the knowledge provided by human experts. Inducing neuro-symbolic programs from data alleviates this problem by allowing missing or unknown symbolic knowledge to be learned during training.

One might argue that there is no need for joint learning of neural weights and symbolic knowledge in neuro-symbolic AI, since neural perception can be outsourced to generalist vision-language models, reducing the problem to symbolic program induction [1]. However, such methods may fail to account for problem instances containing niche subsymbolic input that is out of distribution for any generalist model, and they neglect the considerable cost associated with using such models.

4.2.6 How to approach induction of neuro-symbolic programs

Jointly learning structure and parameters is difficult because the search spaces interact: structural choices redefine the parameter landscape, and parameter updates can bias the search toward suboptimal structures. Neuro-symbolic systems face this acutely, as the symbolic component demands discrete structural commitments while the neural component relies on continuous optimization. Effective approaches typically alternate or blend these two processes, using differentiable relaxations, guided search, or meta-learning priors to keep the joint space navigable.

A practical approach is to employ abduction-guided search, where candidate neuro-symbolic programs are generated, scored against raw data, and refined using differentiable components to prune the hypothesis space. Concretely, one can combine perceptual models with an abductive logic engine in a loop: propose symbolic rules, test them against observations, and iteratively update the neural perception and the symbolic hypothesis until convergence [2]. Another approach (inspired by meta-learning) uses a symbolic program representation for compositional generalization together with neural program synthesis, enabling efficient guess-and-verify in program space [7].

References

1 Vít Balek, Lukáš Sýkora, Václav Sklenák, and Tomáš Kliegr. *LLM-based feature generation from text for interpretable machine learning*. *Machine Learning*, 114(11):241, 2025. https://doi.org/10.1007/s10994-025-06867-1

2 Wang-Zhou Dai and Stephen H. Muggleton. *Abductive knowledge induction from raw data*. *arXiv preprint* arXiv:2010.03514, 2020.

3 Kevin Ellis, Lucas Wong, Michael Nye, Mathias Sablé-Meyer, Luc Cary, Laura Anaya Pozo, Luke Hewitt, Armando Solar-Lezama, and Joshua B. Tenenbaum. *DreamCoder: Growing generalizable, interpretable knowledge with wake–sleep Bayesian program learning*. *Philosophical Transactions of the Royal Society A: Mathematical, Physical and Engineering Sciences*, 381(2251):20220050, 2023. https://doi.org/10.1098/rsta.2022.0050

4 Johanna Langer and Ute Schmid. *Flexible Segmentation for Rule Learning over Object Representations in Abstract Visual Reasoning*. In *Twelfth Annual Conference on Advances in Cognitive Systems*, 2025. https://openreview.net/forum?id=PSsAsrCkFx

5 Zhuo Li, Junjie Huang, and Mayur Naik. *Scallop: A Language for Neurosymbolic Programming*. *arXiv* arXiv:2304.04812, 2023. https://doi.org/10.48550/arXiv.2304.04812

6 Robin Manhaeve, Sebastijan Dumančić, Angelika Kimmig, Thomas Demeester, and Luc De Raedt. *DeepProbLog: Neural Probabilistic Logic Programming*. *arXiv* arXiv:1805.10872, 2018. https://doi.org/10.48550/arXiv.1805.10872

7 Maxwell Nye, Armando Solar-Lezama, Joshua B. Tenenbaum, and Brenden M. Lake. *Learning compositional rules via neural program synthesis*. In *Advances in Neural Information Processing Systems*, volume 33, pages 10832–10842, 2020.
8 Emile van Krieken, Selim Badreddine, Robin Manhaeve, and Enrico Giunchiglia. *ULLER: A Unified Language for Learning and Reasoning*. *arXiv* arXiv:2405.00532, 2024. https://doi.org/10.48550/arXiv.2405.00532
9 Zhun Yang, Adam Ishay, and Joohyung Lee. *NeurASP: Embracing Neural Networks into Answer Set Programming*. *arXiv* arXiv:2307.07700, 2023. https://doi.org/10.48550/arXiv.2307.07700

4.3 Gradual refinement in coarse-to-fine concept induction and human-like iterative programming

Alex Lew (Yale University – New Haven, US), Maddy Bowers (MIT – Cambridge, US), Gabriel Grand (MIT – Cambridge, US), Emanuel Kitzelmann (Technische Hochschule Brandenburg, DE), David Nagy (Universität Tübingen, DE), Tejaswini Ojha, Mukul Singh (Microsoft – Redmond, US), Tobias Thomas (TU Darmstadt, DE), Dominik Ürüm (TU Darmstadt, DE), and Hanqi Zhou (Universität Tübingen, DE)

A key research question for the field of program induction is how to generate programs *iteratively*, refining them gradually in response to various signals. We investigate this question in two contexts. (1) Modeling the process by which engineers iteratively improve their code, on both short time scales (fixing bugs, refactoring) and longer ones (evolving large-scale codebases over years); and (2) modeling the coarse-to-fine aspects of how humans learn concepts from examples.

4.3.1 Automating human-like iterative coding

Human engineers develop programs in stages: A programmer may develop one function in an iterative fashion over the course of minutes or hours, writing tests, sketching a possible approach with comments, filling in the sketch with code, running tests and inspecting results, fixing bugs, and so on [1]. A key question is how to model or automate these iterative processes.

Model architectures for iterative revision. Today, autoregressive ("causal") transformers are the predominant architecture for LLMs. Such architectures (1) generate tokens left-to-right at inference time, and (2) are trained to maximize the left-to-right generation probability of programs in the training data. One important research direction for modeling iterative coding processes is to explore alternative architectures that may be able to learn to generate text in more human-like ways.

Diffusion models are one such alternative. Current diffusion models for text assume Gaussian noise in latent space, which recovers some but not the whole structure of the iterative process of coding [2, 5]. Thus, a possible extension would be to replace the continuous noise process with an explicit, possibly discrete, noise process in the token or text space. Another one would be to infer the latent generative process of the code, which is in reality never left-to-right, but involves the whole range of the iterative complexity described above. Yet,

the latter approach requires either a dataset including at least some of the intermediate steps of the generative process, or a more sophisticated training algorithm that co-learns a noiser and denoiser (e.g., based on EM or variational inference – see, e.g., the STaR family of algorithms [3, 4], or using amortized sampling of the latent representation over data [6].

A further architectural challenge is that, in human problem solving, there appear to be intermediate memory representations that are less granular than the raw token sequence in an LLM's context window, yet more granular and rapidly updated than what is stored in the weights. For coding agents, this might include persistent information about how to interact with a given user and project such as preferred coding conventions or the appropriate level of detail for clarifying questions from the coding agent. At present, these forms of task context are implemented through a hand-engineered outer loop that incorporates the LLM, injecting this information into the context window via system prompts, personal custom instructions, and explicit context distillation across sessions. An open question is whether some of this intermediate task memory can be incorporated into the architecture and learning dynamics of the model itself, rather than relying on our current understanding and ability to explicitly specify the human coding process.

4.3.2 Modeling human-like coarse-to-fine concept induction

The second question of our discussion group centered on modeling *concept learning*, rather than human software engineering. When solving concept induction problems, people often begin from coarse, vague hypotheses that they incrementally refine into increasingly precise explanations [9]. The talk "Concept learning as coarse-to-fine probabilistic program induction" [7], presented on the first day of the seminar, offered an approach to modeling this kind of reasoning by representing coarse hypotheses as probabilistic programs and evaluating their quality in terms of the likelihood of the observed examples under the programs. However, an open question with the approach is how to decide what set of refinements to consider. In particular, having a small core set of low-level refinements, such as those given implicitly by the productions of the target language grammar, can be suboptimal since we may require applying multiple low-level refinements in sequence before seeing an improvement in hypothesis quality as measured by the likelihood of the data under the program. This suggests introducing new individual refinements that correspond to introducing a whole chunk of program at once.

We discussed how this kind of refinement learning can be framed as a form of library learning in the style of DreamCoder [8]. Library learning could identify common structure across successful or promising coarse-to-fine search traces, enabling the discovery of useful refinement chunks (e.g., a learned `map`). These chunks can then be added to the library of possible refinements for downstream use in synthesis. This could also help bridge a gap in the refinement process, where the intermediate hypothesis the search would otherwise have to proceed through is judged as a low-quality coarse hypothesis even though the destination is high-quality.

We believe that these insights on the coarse-to-fine aspects of human concept learning are applicable not only to modeling concept induction, but also to scaling synthesis in more general automated coding settings.

References

1 J. S. Rule, *The Child as Hacker: Building More Human-Like Models of Learning*, Ph.D. dissertation, Massachusetts Institute of Technology, 2020.
2 M. Singh, J. Cambronero, S. Gulwani, V. Le, C. Negreanu, and G. Verbruggen, "CodeFusion: A pre-trained diffusion model for code generation," in *Proc. Conf. on Empirical Methods in Natural Language Processing (EMNLP)*, Association for Computational Linguistics, 2023, pp. 11697–11708.
3 E. Zelikman, G. R. Harik, Y. Shao, V. Jayasiri, N. Haber, and N. Goodman, "Quiet-STaR: Language models can teach themselves to think before speaking," in *Proc. First Conf. on Language Modeling*, 2024.
4 G. Dong, Y. Chen, X. Li, J. Jin, H. Qian, Y. Zhu, H. Mao, G. Zhou, Z. Dou, and J.-R. Wen, "Tool-Star: Empowering LLM-brained multi-tool reasoner via reinforcement learning," *arXiv preprint*, 2025.
5 M. Singh, G. Verbruggen, V. Le, and S. Gulwani, "Diffusion is a code repair operator and generator," *arXiv preprint*, 2025.
6 S. Venkatraman et al., "Amortizing intractable inference in diffusion models for vision, language, and control," in *Advances in Neural Information Processing Systems*, vol. 37, A. Globerson et al., Eds., 2024, pp. 76080–76114.
7 M. Bowers, A. Lew, W. Qi, J. S. Rule, V. Mansinghka, J. Tenenbaum, and A. Solar-Lezama, "Concept learning as coarse-to-fine probabilistic program induction," in *Proc. Annual Meeting of the Cognitive Science Society (CogSci)*, vol. 46, 2024.
8 K. Ellis, C. Wong, M. Nye, M. Sablé-Meyer, L. Morales, L. Hewitt, and J. B. Tenenbaum, "DreamCoder: Bootstrapping inductive program synthesis with wake-sleep library learning," in *Proc. 42nd ACM SIGPLAN Int. Conf. on Programming Language Design and Implementation (PLDI)*, 2021, pp. 835–850.
9 N. D. Goodman, J. B. Tenenbaum, J. Feldman, and T. L. Griffiths, "A rational analysis of rule-based concept learning," *Cognitive Science*, vol. 32, no. 1, pp. 108–154, 2008.

4.4 Benchmarks and Evaluation

Yael Moros Daval (Technical University of Valencia, ES), Félix Martí Pérez (Technical University of Valencia, ES), Yewen Pu (Nanyang TU – Singapore, SG), Arjun Radhakrishna (Microsoft – Redmond, US), Ute Schmid (Universität Bamberg, DE), and Gust Verbruggen (Microsoft – Keerbergen, BE)

Benchmarking and evaluation of AI systems are currently in a state of rapid transition, largely driven by the generalization capabilities of large language models (LLMs). Classical inductive programming benchmarks, for example lists, strings, and grid-based tasks such as ARC, were designed to probe abstraction, compositionality, and data efficiency. At present, many of these benchmarks are effectively saturated: they are solvable either through scale, by learning strong statistical priors over program spaces, or through inference-time computation that enables iterative reasoning and code refinement, often at substantial cost. As a result, functional correctness on static input–output tasks no longer provides a meaningful signal for progress in symbolic or hybrid AI systems. Recent evaluation efforts have therefore shifted toward more realistic and complex settings, including software engineering benchmarks, interactive program synthesis, and agent-style tasks that embed systems in execution environments.

These benchmarks trade ease of evaluation for realism and complexity, frequently requiring partial automation, human assessment, or LLM-based judges. While such approaches better reflect real-world use, they introduce challenges in reproducibility, standardization, and interpretability of results. Overall, the field is moving from closed-form, ground-truth-driven benchmarks toward in-situ evaluation that models user interaction, iterative refinement, and environment feedback. For inductive and logic-based programming, this shift necessitates new benchmarks that explicitly test collaboration between symbolic methods, learned models, and users, rather than raw problem-solving ability alone.

4.5 Programming Education in the Age of Large Language Models

Hila Peleg (Technion – Haifa, IL), Frank Jäkel (TU Darmstadt, DE), Sven Mayer (TU Dortmund, DE), and Sonja Niemann (bidt – München, DE)

The wide adoption of large language models (LLMs) to assist programmers raises the question: How should programming be taught in the future? It is sometimes claimed that students will not have to learn programming at all. However, there will always be demand for expert programmers. There is currently still disagreement on how many people will have to acquire how much programming expertise. If programming assistants become better still, this will increase access to automation for many people who do not know how to code and will not have to learn it either. Therefore, the number of people who feel that they have to learn programming in order to become proficient computer users will decrease. Until very recently, programming and computational thinking were considered essential skills for the 21st century, and educators tried to integrate them into school curricula. Since LLMs are marketed as being able to replace programmers [5], this consensus is under increasing pressure. However, computational thinking will still be required of users of LLMs, and programming is a good way to learn it. Furthermore, if programming were to be taught less, there would be fewer people who potentially become experts down the road.

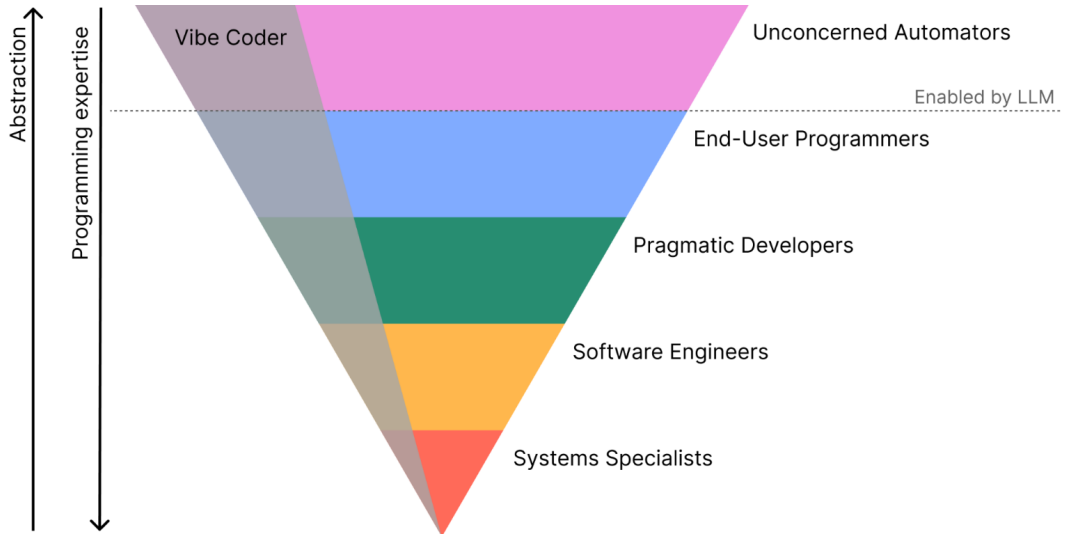

Figure 1 Possible distribution of programmers, abstraction decreases towards the wide base of Unconcerned Automators while programming expertise increases towards the Systems Specialists.

It is useful to think about the people who program today as an inverted pyramid. Most programming is done by users who automate tasks in some application – very often Excel. These end-user programmers are not professional programmers but need some programming skills to get their work done. LLMs currently make this kind of automation accessible to more and more users. These users who don't know how to program will, however, mostly be unconcerned with code. This is in contrast to pragmatic programmers who have more expertise than the end-user programmers. These people have problems that require them to develop custom code, and they often use professional tooling and libraries even though they are not professional software developers. Scientists, engineers, or data analysts fall into this class, but also, e.g., web developers. Their work relies on the work of far fewer software developers who write the libraries and tools that they rely on. Their code, in turn, is built on the work of even fewer system specialists who develop operating systems, compilers, or programming languages. The challenge is to educate each group at the right level while leaving the door open for those on the higher rungs of the pyramid to be able to move to the lower rungs if they need or want to.

To meet this challenge, it is interesting to consider the parallels between programming with an LLM and pair programming, where one person is the "driver," who only types code, and the other the "navigator," who decides what code to write. This is like the acceleration mode in Grounded Copilot [1], removing low-level concerns like syntax. How can we ensure that the roles aren't reversed, i.e., that the human user becomes the "driver" and the LLM the "navigator"? Such a reversal would make it difficult for LLM users to acquire the expertise to transition to the lower rungs of the pyramid. One way to avoid this is to design LLM support in a way that it is more constrained for beginners, thus making sure that they have to think about the code generated by the LLM. Such a system only allows for more and more auto-completion as higher levels of programming expertise are "unlocked". This would be similar to some programming languages specifically created to teach programming at progressively greater degrees of difficulty, like Hedy [2] or Racket [4, 3].

Since expertise is contextual, programmers at all levels might find themselves "vibe coding" at times. Since the definition is cases where the code does not matter, this can be relevant for one-time pieces of code in areas of low expertise, but could also be a very useful tool for things like prototyping an idea, just as a proof-of-concept, generating code that will be thrown out and rewritten with more care and attention later. This makes vibe coding a tool that can be taught, stressing the importance of not keeping any such code around longer than one would keep a prototype, while being aware that this code might find its way into production despite the advice of the tutors.

Programming assistants have the potential to bring the full power of computing to more people, but there is also the danger of skill loss. The design of such tools, therefore, needs to ensure that students can not just solve programming problems faster but also improve their conceptual understanding and programming skills. Otherwise, we might find ourselves in a future without expert programmers at the apex of the inverted pyramid.

References

1 Shraddha Barke, Michael B. James, and Nadia Polikarpova. Grounded Copilot: How programmers interact with code-generating models. *Proceedings of the ACM on Programming Languages*, 7(OOPSLA1), Article 78, 2023.

2 Felienne Hermans. Hedy: A gradual language for programming education. In *Proceedings of the 2020 International Computing Education Research Conference (ICER '20)*, pages 1–12, 2020.

3 Michael Hanus and Jan Rasmus Tikovsky. A parser generator system for level-based programming languages. In *Proceedings of the Software Engineering Workshops 2016 (SE-WS 2016)*, CEUR Workshop Proceedings, volume 1559, pages 3–24, 2016.

4 The Racket Documentation. DrRacket Language Levels. Available at: `https://download.racket-lang.org/docs/5.1.2/html/unstable/DrRacket_Language_Levels.html`, 2011. Version 5.1.2. Accessed: December 5, 2025.

5 Mark Tyson. Jensen Huang says kids shouldn't learn to code – they should leave it up to AI. *Tom's Hardware*, February 25, 2024. Available at: `https://www.tomshardware.com/tech-industry/artificial-intelligence/jensen-huang-advises-against-learning-to-code-leave-it-up-to-ai`.

Participants

- Maddy Bowers
MIT – Cambridge, US
- David Cerna
Dynatrace Research – Linz, AT
- Wang-Zhou Dai
Nanjing University – Suzhou, CN
- Quinten Dewulf
KU Leuven, BE
- Cesar Ferri Ramirez
Technical University of Valencia, ES
- Jasmin Fisher
University College London, GB
- Pierre Flener
Uppsala University, SE
- Johannes Fürnkranz
Johannes Kepler Universität Linz, AT
- Gabriel Grand
MIT – Cambridge, US
- Katerina Hrudková
University of Economics – Prague, CZ
- Frank Jäkel
TU Darmstadt, DE
- Emanuel Kitzelmann
Technische Hochschule Brandenburg, DE
- Tomáš Kliegr
University of Economics – Prague, CZ
- Heiko Koziolek
ABB – Mannheim, DE
- Johannes Langer
Universität Bamberg, DE
- Alex Lew
Yale University – New Haven, US
- Félix Martí Pérez
Technical University of Valencia, ES
- Sven Mayer
TU Dortmund, DE
- Yael Moros Daval
Technical University of Valencia, ES
- David Nagy
Universität Tübingen, DE
- Sonja Niemann
bidt – München, DE
- Stassa Patsantzis
University of Surrey – Guildford, GB
- Hila Peleg
Technion – Haifa, IL
- Yewen Pu
Nanyang TU – Singapore, SG
- Arjun Radhakrishna
Microsoft – Redmond, US
- Ute Schmid
Universität Bamberg, DE
- Zachary Siegel
MIT – Cambridge, US
- Mukul Singh
Microsoft – Redmond, US
- Adish Singla
MPI-SWS – Saarbrücken, DE
- Tobias Thomas
TU Darmstadt, DE
- Dominik Ürüm
TU Darmstadt, DE
- Raoul van Doren
ETH Zürich, CH
- Gust Verbruggen
Microsoft – Keerbergen, BE
- Felix Weitkämper
German UDS – Potsdam, DE
- Hanqi Zhou
Universität Tübingen, DE