Report from Dagstuhl Seminar 23062

Programming Language Processing

Michael Pradel*[1], Baishakhi Ray*[2], Charles Sutton*[3], and Eran Yahav*[4]

1 **Universität Stuttgart, DE.** michael@binaervarianz.de
2 **Columbia University – New York, US.** rayb@cs.columbia.edu
3 **Google – Mountain View, US.** charlessutton@google.com
4 **Technion – Haifa, IL.** yahave@gmail.com

Abstract

This report documents the program and the outcomes of Dagstuhl Seminar 23062 "Programming Language Processing" (PLP). The seminar brought together researchers and practitioners from three communities–software engineering, programming languages, and natural language processing–providing a unique opportunity for cross-fertilization and inter-disciplinary progress. We discussed machine learning models of code, integrating learning-based and traditional program analysis, and learning from natural language information associated with software. The seminar lead to a better understanding of the commonalities and differences between natural and programming languages, and an understanding of the challenges and opportunities in industry adoption of PLP.

Seminar February 5–10, 2023 – https://www.dagstuhl.de/23062
2012 ACM Subject Classification Computing methodologies → Artificial intelligence; Computing methodologies → Machine learning; Software and its engineering
Keywords and phrases ML4PL, ML4SE, Neural Software Analysis
Digital Object Identifier 10.4230/DagRep.13.2.20

1 Executive Summary

Michael Pradel
Baishakhi Ray
Charles Sutton
Eran Yahav

Our 5-day Dagstuhl Seminar on "Programming Language Processing" (PLP) brought together researchers and practitioners from the software engineering, programming languages, and natural language processing communities The seminar focused on activities prepared ahead by the participants, such as talks, demos of tools and challenges, and tutorials, as well as informal discussions anchored around the prepared activities. We provided each participant who wanted to present their work an opportunity for doing so. In addition, we asked specific people to present specific topics, e.g., experts of a particularly relevant subfield to prepare a tutorial or creators of a particularly relevant tool to give a tool demo.

In addition to talks and informal discussions, there were several break-out sessions during which participants discussed specific topics in smaller groups and eventually reported back to the other participants. In particular, we had break-out sessions on the following topics:

* Editor / Organizer

- How (if at all) do AI programming assistants change programming?
- Interpreting neural models of code.
- Do we still need per-task models, or do large language models solve it all?
- What software engineering tasks are not yet explored (sufficiently) by neural models?
- How should and will computer science education change in response to ML-based coding tools?
- What kinds of guarantees can we expect, and do we want, from ML-based developer tools? What human factors in interacting with ML systems are relevant?
- How can learned models use existing tools, e.g., compilers and interpreters, to improve their predictions?

As a result of the seminar, several participants plan to launch various follow-up activities, such as joint publications and transferring promising ideas from academia to industry.

2 Table of Contents

3 Overview of Talks

3.1 Mining Idioms

Rui Abreu (Meta Platforms – Bellevue, US)

Joint work of Aishwarya Sivaraman, Rui Abreu, Andrew Scott, Tobi Akomolede, Satish Chandra
Main reference Aishwarya Sivaraman, Rui Abreu, Andrew Scott, Tobi Akomolede, Satish Chandra: "Mining Idioms in the Wild", in Proc. of the 44th IEEE/ACM International Conference on Software Engineering: Software Engineering in Practice, ICSE (SEIP) 2022, Pittsburgh, PA, USA, May 22-24, 2022, pp. 187–196, IEEE, 2022.
URL https://doi.org//10.1109/ICSE-SEIP55303.2022.9794062

Existing code repositories contain numerous instances of code patterns that are idiomatic ways of accomplishing a particular programming task. Sometimes, the programming language in use supports specific operators or APIs that can express the same idiomatic imperative code much more succinctly. However, those code patterns linger in repositories because the developers may be unaware of the new APIs or have not gotten around to them. Detection of idiomatic code can also point to the need for new APIs.

We share our experiences in mine idiomatic patterns from the Hack repo at Facebook. We found that existing techniques either cannot identify meaningful patterns from syntax trees or require test-suite-based dynamic analysis to incorporate semantic properties to mine useful patterns. The key insight of the approach proposed in this talk – Jezero – is those semantic idioms from a large codebase can be learned from canonicalized dataflow trees. We propose a scalable, lightweight static analysis-based approach to construct a tree that is well suited to mine semantic idioms using nonparametric Bayesian methods.

Our experiments with Jezero on Hack code show a clear advantage of adding canonicalized dataflow information to ASTs: Jezero was significantly more effective than a baseline that did not have the dataflow augmentation in being able to effectively find refactoring opportunities from unannotated legacy code.

3.2 Crafting Code Suggestions Using Large Language Models

Edward E. Aftandilian (GitHub – San Francisco, US) and Albert Ziegler (GitHub – San Francisco, US)

Large Language Models are great at putting out code, but if you want the right code, you have to ask the right question. Concretely, you have to turn your application task (e.g., I want a test for this function) into a completion task (i.e., a string whose most likely completion will give you such a test). Building it is the object of the emerging discipline of promptcrafting.

We state some general principles for promptcrafting, distinguishing between communicating the task (write a test now) and the background information for this task (the function, its dependencies, the test style in the project, etc.). We discuss typical constraints to promptcrafting (context window size, latency, generality), and desiderata of the produced prompt, in particular the "Little Red Rule" of normality.

Then we go into detail of how we construct prompts in GitHub Copilot by breaking up the different types of background information into discrete "wishes" organized in a wish

list that can be quickly optimized over. We describe the strategies we use to construct our "wishes", and how these prompt crafting strategies improve the performance of the underlying OpenAI Codex model for our application task of generating code editing suggestions.

3.3 Counterfactual Explanations for Models of Code

Jürgen Cito (TU Wien, AT)

Joint work of Jürgen Cito, Isil Dillig, Vijayaraghavan Murali, Satish Chandra
Main reference Jürgen Cito, Isil Dillig, Vijayaraghavan Murali, Satish Chandra: "Counterfactual Explanations for Models of Code", in Proc. of the 44th IEEE/ACM International Conference on Software Engineering: Software Engineering in Practice, ICSE (SEIP) 2022, Pittsburgh, PA, USA, May 22-24, 2022, pp. 125–134, IEEE, 2022.
URL https://doi.org//10.1109/ICSE-SEIP55303.2022.9794112

Machine learning (ML) models play an increasingly prevalent role in many software engineering tasks. However, because most models are now powered by opaque deep neural networks, it can be difficult for developers to understand why the model came to a certain conclusion and how to act upon the model's prediction. Motivated by this problem, this paper explores counterfactual explanations for models of source code. Such counterfactual explanations constitute minimal changes to the source code under which the model "changes its mind". We integrate counterfactual explanation generation into models of source code in a real-world setting at Meta. We describe considerations that impact both the ability to find realistic and plausible counterfactual explanations, as well as the usefulness of such explanations to the developers that use the model. In a series of experiments, we investigate the efficacy of our approach on three different models, each based on a BERT-like architecture operating over source code, including models to detect performance regressions, test plan quality, and taint propagation.

3.4 Synthesizing Correctness Properties

Elizabeth Dinella (University of Pennsylvania – Philadelphia, US)

Joint work of Elizabeth Dinell, Gabriel Ryan, Aaditya Naik, Todd Mytkowicz, Shuvendu Lahiri, Mayur Naik
Main reference Elizabeth Dinella, Gabriel Ryan, Todd Mytkowicz, Shuvendu K. Lahiri: "TOGA: A Neural Method for Test Oracle Generation", in Proc. of the 44th IEEE/ACM 44th International Conference on Software Engineering, ICSE 2022, Pittsburgh, PA, USA, May 25-27, 2022, pp. 2130–2141, ACM, 2022.
URL https://doi.org//10.1145/3510003.3510141

Automatically checking for software correctness is a well studied and important problem in software engineering. Despite many successful works, there are still significant barriers to full automation. Namely, manually expressing a precise notion of correctness for which the tools can check for. A program analysis tool cannot check for a property which solely exists in the developer's mind. In this talk I will present works to automatically synthesize such properties. I explore synthesis of both pre and post conditions and evaluate in the automated testing domain. I will present learning techniques for synthesis as well as datasets for future research in this area.

3.5 DiCoder: Code Completion via Dialog

Aryaz Eghbali (Universität Stuttgart, DE)

Joint work of Aryaz Eghbali, Michael Pradel

Large Language Models (LLMs) pretrained on code are assisting developers by suggesting code completions. Since the quality of the prediction can greatly be influenced by the prompt, and because of prompt size limitations, providing appropriate context for these models has been challenging. Current methods rely on heuristic approaches to select context from various locations in code, which fail in cases where there exist relevant information present in the same project, but in non-trival locations. Although some predictions are not using the proper API or are using them in a wrong way, the output of LLMs are in many cases close enough to the desired prediction that can lead to correct completions if provided with hints about the APIs. We propose a novel approach, called DiCoder, for using the output of the model to guide the process of gathering relevant context from the same project. DiCoder iteratively prompts the model with the most relevant context that it has gathered until the most recent iteration, and then uses tokens in the completion to retrieve further relevant context.

3.6 Automated Repair for Static Warnings: PLMs (Codex) vs Template-based Metaprogramming

Khashayar Etemadi Someoliayi (KTH Royal Institute of Technology – Stockholm, SE)

Main reference Khashayar Etemadi Someoliayi, Nicolas Yves Maurice Harrand, Simon Larsén, Haris Adzemovic, Henry Luong Phu, Ashutosh Verma, Fernanda Madeiral, Douglas Wikstrom, Martin Monperrus: "Sorald: Automatic Patch Suggestions for SonarQube Static Analysis Violations", IEEE Transactions on Dependable and Secure Computing, pp. 1–1, 2022.
URL https://doi.org//10.1109/TDSC.2022.3167316

Static analyzers create warnings regarding parts of code that can potentially cause bugs. As addressing the warnings can be very time-consuming, developers sometimes ignore these warnings. This causes significant quality degradation to the program in the long term. Automated program repair (APR) tools are proposed to deal with static warnings and modify the program to preserve their semantics and make them warning free. Existing APR techniques modify programs using fixed metaprogramming templates that address certain types of static warnings. In this work, we assess if using PLMs can replace traditional APR tools for static warnings. We create a prototype PLM-based APR tool that fixes SonarQube static warnings. Our prototype tool fixes warnings with 7 different types in a dataset of five real-world projects, while the state-of-the-art template-based tool fixes warnings of 4 types on the same dataset. This proves the flexibility and usefulness PLM-based APR tools in this area.

3.7 Controlling Large Language Models to Generate Secure and Vulnerable Code

Jingxuan He (ETH Zürich, CH)

Joint work of Jingxuan He, Martin T. Vechev
Main reference Jingxuan He, Martin T. Vechev: "Controlling Large Language Models to Generate Secure and Vulnerable Code", CoRR, Vol. abs/2302.05319, 2023.
URL https://doi.org//10.48550/arXiv.2302.05319

Large language models (LLMs) are increasingly pretrained on large codebases and used for code generation tasks. A fundamental limitation of LLMs is that they can generate secure and vulnerable code, but the users cannot control this. In this work, we formulate a new problem called controlled code generation, which allows users to input a boolean property into LLMs to control if the output code is secure or vulnerable. We propose svGen, which learns soft prompts for solving controlled code generation. Our extensive evaluation on a wide range of vulnerabilities shows the effectiveness of svGen.

3.8 Explaining Code Intelligence to Bridge the Gap Between Testing and Debugging

Reyhaneh Jabbarvand (University of Illinois – Urbana-Champaign, US)

Testing and Debugging are an inseparable part of the software engineering and development pipeline. While performed separately, they are connected in such a way that debugging relies on the result of test execution to localize and ultimately fix the bugs. Without accurate, automated, and explainable oracles, detecting and localizing the bugs could be cumbersome. We proposed SEER, an automated technique that leverages deep learning and attention analysis to produce accurate and explainable oracles. To build the ground truth, SEER jointly embeds unit tests and code into a unified vector space, in such a way that the neural representation of tests are similar to that of code they pass on them, but dissimilar to the code they fail on them. The classifier built on top of this vector representation serves as the oracle to generate "fail" labels, when test inputs detect a bug in code or "pass" labels, otherwise. The extensive experiments on applying SEER to more than 5K unit tests from a diverse set of open-source Java projects show that the produced oracle is (1) effective in predicting the fail or pass labels, achieving an overall accuracy, precision, recall, and F1 measure of 93%, 86%, 94%, and 90%, (2) generalizable, predicting the labels for the unit test of projects that were not in training or validation set with negligible performance drop, and (3) efficient, detecting the existence of bugs in only 6.5 milliseconds on average.

3.9 LExecutor: Learning-Guided Execution

Michael Pradel (Universität Stuttgart, DE)

Joint work of Michael Pradel, Beatriz Souza
Main reference Beatriz Souza, Michael Pradel: "LExecutor: Learning-Guided Execution", CoRR, Vol. abs/2302.02343, 2023.
URL https://doi.org//10.48550/arXiv.2302.02343

Executing code is essential for various program analysis tasks, e.g., to detect bugs that manifest through exceptions or to obtain execution traces for further dynamic analysis. However, executing an arbitrary piece of code is often difficult in practice, e.g., because of missing variable definitions, missing user inputs, and missing third-party dependencies. This talk presents LExecutor, a learning-guided approach for executing arbitrary code snippets in an underconstrained way. The key idea is to let a neural model predict missing values that otherwise would cause the program to get stuck, and to inject these values into the execution. For example, LExecutor injects likely values for otherwise undefined variables and likely return values of calls to otherwise missing functions. We evaluate the approach on Python code from popular open-source projects and on code snippets extracted from Stack Overflow. The neural model predicts realistic values with an accuracy between 80.1% and 94.2%, allowing LExecutor to closely mimic real executions. As a result, the approach successfully executes significantly more code than any available technique, such as simply executing the code as-is. For example, executing the open-source code snippets as-is covers only 4.1% of all lines, because the code crashes early on, whereas LExecutor achieves a coverage of 50.1%.

3.10 How to incorporate semantics in LLM pretraining

Baishakhi Ray (Columbia University – New York, US)

Understanding code semantics is significant for code modeling tasks such as software vulnerability detection, code clone detection, security analysis and code generation. Here we talk about ways how model can learn code semantics while pertaining.

- We present a novel self-supervised model focusing on identifying (dis)similar functionalities of source code. Different from existing works, our approach does not require a huge amount of randomly collected datasets. Rather, we design structure-guided code transformation algorithms to generate synthetic code clones and inject real-world security bugs, augmenting the collected datasets in a targeted way. We propose to pre-train the Transformer model with such automatically generated program contrasts to better identify similar code in the wild and differentiate vulnerable programs from benign ones. To better capture the structural features of source code, we propose a new cloze objective to encode the local tree-based context (e.g., parents or sibling nodes). We pre-train our model with a much smaller dataset, the size of which is only 5% of the state-of-the-art models' training datasets, to illustrate the effectiveness of our data augmentation and the pre-training approach. The evaluation shows that, even with much less data, DISCO can still outperform the state-of-the-art models in vulnerability and code clone detection tasks.

- For a generative setting, we leverage such syntax guided transformation in a de-noising encoder-decoder setting. We inject semantically similar transformation as "noise" and the decoder learns to denoise and retrieve the original code. Learning to generate equivalent, but more natural code, at scale, over large corpora of open-source code, without explicit manual supervision, helps the model learn to both ingest and generate code. We fine-tune our model in three generative Software Engineering tasks: code generation, code translation, and code refinement with limited human-curated labeled data and achieve state-of-the-art performance rivaling CodeT5. We show that our pre-trained model is especially competitive at zero-shot and few-shot learning, and better at learning code properties (e.g., syntax, data flow).
- Incorporating Dynamic Code Feature: We discuss how building machine learning (ML) models toward learning program execution semantics so they can remain robust against transformations in program syntax and generalize to various program analysis tasks and security applications. The corresponding research tools, such as Trex, StateFormer, and NeuDep, have outperformed commercial tools and prior arts by up to 117x in speed and by 35% in precision and have helped identify security vulnerabilities in real-world firmware that run on billions of devices.

3.11 On the role of data in Neural Bug Detection and Repair

Cedric Richter (Universität Oldenburg, DE)

Joint work of Cedric Richter, Heike Wehrheim
Main reference Cedric Richter, Heike Wehrheim: "Can we learn from developer mistakes? Learning to localize and repair real bugs from real bug fixes", CoRR, Vol. abs/2207.00301, 2022.
URL https://doi.org//10.48550/arXiv.2207.00301

Real bug fixes seem to be the perfect source for training neural bug detection and repair models. Yet, existing real bug fix datasets are often too small to effectively train data-hungry neural approaches. For this reason, existing approaches often rely on artificial bugs that can be easily produced via a mutation operator at large scales. However, neural bug detection and repair approaches trained purely on mutants usually underperform when confronted with real bugs.

To address this shortcoming, we introduce a novel dataset of 31k real bug fixes for training. We utilize the dataset to evaluate the impact of artificially generated mutants and real bug fixes on the training of neural bug detection and repair approaches. We find that training on real bug fixes can significantly improve the ability of our model to detect and repair real bugs, while training on mutants is still necessary to achieve high performing models.

3.12 Two Benchmarks for ML4Code: GLUE Code and RunBugRun

Romain Robbes (CNRS – Bordeaux, FR & University of Bordeaux, FR)

Joint work of Romain Robbes, Anjan Karmakar, Julian Prenner, Miltiadis Allamanis
Main reference Anjan Karmakar, Miltiadis Allamanis, Romain Robbes: "JEMMA: An Extensible Java Dataset for ML4Code Applications", CoRR, Vol. abs/2212.09132, 2022.
URL https://doi.org//10.48550/arXiv.2212.09132

We present two benchmarks and datasets that are designed to help the community progress on goals that we think are important.

- GLUE Code (Global and Local Understanding Evaluation of Code) is geared towards the development of models that use a global context beyond a code snippet. Indeed, one of our studies shows that 60% of method calls are project-specific, and 40% come from a distant context. GLUE Code is based on the JEMMA dataset of source code projects, a dataset of 50,000 projects (derived from 50K-C) that include significant post-processing to add source code representations, call graphs, and static analysis tool data. GLUE Code includes tasks that require a model to go beyond the current code snippet and include larger context (file, package, callers/callees). GLUE Code users can use JEMMA to assemble the context they need to solve the GLUE Code tasks. In this way, GLUE Code and JEMMA allow users to experiment with a variety of source code contexts. Find more details on JEMMA at: `https://arxiv.org/abs/2212.09132`.
- RunBugRun is a large-scale, executable, and multi-lingual dataset to incentivize Automated Program Repair models to leverage runtime information in their design. RunBugRun is derived from CodeNet; it includes 450,000 (carefully curated) executable bug/fix pairs that can be validated via running tests. Generated patches can be compiled and executed. RunBugRun includes bug/fix pairs in C, C++, Python, Java, JavaScript, Go, Ruby, and PHP. The bug/fix pairs are also labeled with respect to the kind of changes they include. Initial results on two baselines show both that there is room for future improvement, and the potential of transfer learning from common to uncommon languages. Find more details on RunBugRun at: `https://github.com/giganticode/run_bug_run`.

3.13 Combining compiler IRs with machine learning

Baptiste Rozière (Meta AI – Paris, FR)

Joint work of Baptiste Rozière, Marc Szafraniec, Gabriel Synnaeve, Ruba Mutasim, David Pichardie, Patrick Labatut, Hugh Leather, François Charton
Main reference Marc Szafraniec, Baptiste Rozière, Hugh Leather, François Charton, Patrick Labatut, Gabriel Synnaeve: "Code Translation with Compiler Representations", CoRR, Vol. abs/2207.03578, 2022.
URL https://doi.org//10.48550/arXiv.2207.03578

We leverage low-level compiler intermediate representations (IR) to improve code translation. Traditional transpilers rely on syntactic information and handcrafted rules, which limits their applicability and produces unnatural-looking code. Applying neural machine translation (NMT) approaches to code has successfully broadened the set of programs on which one can get a natural-looking translation. However, they treat the code as sequences of text tokens, and still do not differentiate well enough between similar pieces of code which have different semantics in different languages. The consequence is low quality translation, reducing the

practicality of NMT, and stressing the need for an approach significantly increasing its accuracy. Here we propose to augment code translation with IRs, specifically LLVM IR, with results on the C++, Java, Rust, and Go languages. Our method improves upon the state of the art for unsupervised code translation, increasing the number of correct translations by 11% on average, and up to 79% for the Java → Rust pair with greedy decoding. With beam search, it increases the number of correct translations by 5.5% in average.

Additionally, we train models with high performance on the problem of IR decompilation, generating programming source code from IR, and study using IRs as intermediary pivot for translation. We also show that IR decompilation can be used to simplify source code, or for code repair. As the retrieved source code can be compiled to IR again and compared to the input IR, we can sometimes guarantee the correctness of the output assuming that the compiler is correct.

3.14 Large language models and program invariants

Charles Sutton (Google – Mountain View, US)

Joint work of Kexin Pei, David Bieber, Kensen Shi, Charles Sutton, Pengcheng Yin

Identifying invariants is an important program analysis task with applications towards program understanding, vulnerability analysis, and formal verification. Existing tools for identifying invariants rely on dynamic analysis, requiring traces collected from multiple executions in order to produce reliable invariants. We study the application of large language models to invariant prediction, finding that models trained on source code and fine-tuned for invariant generation can perform invariant prediction as static rather than dynamic analysis. Using a scratchpad approach where invariants are predicted sequentially through a program gives the best performance, finding invariants statically of quality comparable to those obtained by a dynamic analysis tool with access to five program traces.

3.15 Customized Models or Generic Code Language Models?

Lin Tan (Purdue University – West Lafayette, US)

Joint work of Nan Jiang, Thibaud Lutellier, Yiling Lou, Dan Goldwasser, Xiangyu Zhang, and Kevin Liu
Main reference Nan Jiang, Kevin Liu, Thibaud Lutellier, Lin Tan: "Impact of Code Language Models on Automated Program Repair", CoRR, Vol. abs/2302.05020, 2023.
URL https://doi.org//10.48550/arXiv.2302.05020
Main reference Nan Jiang, Thibaud Lutellier, Yiling Lou, Lin Tan, Dan Goldwasser, Xiangyu Zhang: "KNOD: Domain Knowledge Distilled Tree Decoder for Automated Program Repair", CoRR, Vol. abs/2302.01857, 2023.
URL https://doi.org//10.48550/arXiv.2302.01857

This talk presents a novel customized neural-network model KNOD for fixing software bugs automatically. KNOD contains (1) a novel three-stage tree decoder, which directly generates Abstract Syntax Trees of patched code according to the inherent tree structure, and (2) a novel domain-rule distillation, which leverages syntactic and semantic rules and teacher-student distributions to explicitly inject the domain knowledge into the decoding procedure during both the training and inference phases. KNOD outperforms existing program-repair techniques on three widely-used benchmarks.

It then discusses the pros and cons of building customized models such as KNOD for a task versus using and fine-tuning generic code language models for the same task. These discussions are based on two recent ICSE 2023 papers "KNOD: Domain Knowledge Distilled Tree Decoder for Automated Program Repair" and "Impact of Code Language Models on Automated Program Repair". Some surprising results include that the best code language model as is, fixes 72% more bugs than the state-of-the-art deep-learning-based program repair techniques.

3.16 An Empirical Study of Deep Learning Models for Vulnerability Detection

Wei Le (Iowa State University – Ames, US)

Joint work of Wei Le, Benjamin Steenhoek, Mahbubur Rahman, Richard Jiles
Main reference Benjamin Steenhoek, Md Mahbubur Rahman, Richard Jiles, Wei Le: "An Empirical Study of Deep Learning Models for Vulnerability Detection", CoRR, Vol. abs/2212.08109, 2022.
URL https://doi.org//10.48550/arXiv.2212.08109

Deep learning (DL) models of code have recently reported great progress for vulnerability detection. In some cases, DL-based models have outperformed static analysis tools. Although many great models have been proposed, we do not yet have a good understanding of these models. This limits the further advancement of model robustness, debugging, and deployment for the vulnerability detection. In this paper, we surveyed and reproduced 9 state-of-the-art (SOTA) deep learning models on 2 widely used vulnerability detection datasets: Devign and MSR. We investigated 6 research questions in three areas, namely model capabilities, training data, and model interpretation. We experimentally demonstrated the variability between different runs of a model and the low agreement among different models' outputs. We investigated models trained for specific types of vulnerabilities compared to a model that is trained on all the vulnerabilities at once. We explored the types of programs DL may consider "hard" to handle. We investigated the relations of training data sizes and training data composition with model performance. Finally, we studied model interpretations and analyzed important features that the models used to make predictions. We believe that our findings can help better understand model results, provide guidance on preparing training data, and improve the robustness of the models. All of our datasets, code, and results are available at `https://doi.org/10.6084/m9.figshare.20791240`.

Participants

- Rui Abreu
Meta Platforms – Bellevue, US
- Edward E. Aftandilian
GitHub – San Francisco, US
- Jürgen Cito
TU Wien, AT
- Premkumar T. Devanbu
University of California – Davis, US
- Elizabeth Dinella
University of Pennsylvania – Philadelphia, US
- Aryaz Eghbali
Universität Stuttgart, DE
- Khashayar Etemadi Someoliayi
KTH Royal Institute of Technology – Stockholm, SE
- Yoav Goldberg
Bar-Ilan University – Ramat Gan, IL
- Lars Grunske
HU Berlin, DE
- Jingxuan He
ETH Zürich, CH
- Maliheh Izadi
TU Delft, NL
- Reyhaneh Jabbarvand
University of Illinois – Urbana-Champaign, US
- Wei Le
Iowa State University – Ames, US
- Martin Monperrus
KTH Royal Institute of Technology – Stockholm, SE
- Alex Polozov
Google – Mountain View, US
- Michael Pradel
Universität Stuttgart, DE
- Baishakhi Ray
Columbia University – New York, US
- Cedric Richter
Universität Oldenburg, DE
- Romain Robbes
CNRS – Bordeaux, FR & University of Bordeaux, FR
- Baptiste Rozière
Meta AI – Paris, FR
- Jan Arne Sparka
HU Berlin, DE
- Charles Sutton
Google – Mountain View, US
- Lin Tan
Purdue University – West Lafayette, US
- Eran Yahav
Technion – Haifa, IL
- Albert Ziegler
GitHub – San Francisco, US