Equality Saturation Guided by Large Language Models

WENTAO PENG*, Peking University, China
RUYI JI*, Peking University, China
YINGFEI XIONG†, Peking University, China

One critical issue with LLMs is their inability to guarantee correctness. Although this problem can be addressed by applying LLMs to formal rewrite systems, the capability of LLMs is still far from adequate to generate sound rewrite chains. To bridge this gap, this paper proposes *LLM-guided equality saturation*, dubbed as LGUESS, by incorporating e-graphs as an intermediate layer between LLMs and rewrite systems. LGUESS queries LLMs for only high-level rewrite checkpoints and uses e-graphs to supply low-level rewrite chains between these checkpoints. In this procedure, the key technical challenge lies in effectively extracting a suitable checkpoint from a saturated e-graph, and LGUESS addresses this by learning a probabilistic model from the LLM. The model predicts probable checkpoints while remaining simple enough for effective extraction.

We have implemented a prototype of LGUESS and evaluated it on the problem of factorizing multi-variable polynomials. The results demonstrate a significant advantage of LGUESS compared to both straightforward equality saturation and the approach that queries the LLM directly for the rewrite chain.

1 INTRODUCTION

Recently, LLMs have revealed remarkable effectiveness in program optimization. They have been applied to optimize programs in different scenarios, such as compile-time optimization [Cummins et al. 2025; Lange et al. 2025], competitive programming [Chen et al. 2024; Gao et al. 2024], and software development [Garg et al. 2022, 2023], achieving impressive results that sometimes overwhelm traditional techniques, especially on complex tasks that require large-scale modifications.

Despite these achievements, correctness remains a serious issue of LLM-based optimizers. LLMs fundamentally cannot ensure the correctness of their output, making it dangerous to adopt their optimizations. For example, Lange et al. [2025] report that their LLM-based optimizer can produce incorrect results even against a testing script – the optimizer discovers a memory exploit in the testing script and sometimes utilizes the exploit to escape the correctness test.

Rewrite systems. One promising method for ensuring correctness is to *certify the optimizations from LLMs through formal rewrite systems*. Rather than directly generating the optimized program, we can apply LLMs to a pre-defined rewrite system and produce a step-by-step rewrite chain for the optimization. In this way, correctness is guaranteed by the rewrite system, and the result can be formally verified by confirming that each step soundly applies a rewrite rule.

However, the current capability of LLMs is still far from adequate to generate rewrite chains.

- On the one hand, modern rewrite systems are too complex for LLMs to fully grasp. For example, SZALINSKI [Nandi et al. 2020], a tool for optimizing CAD programs, implements 65 rules across ~1,000 LOC and incorporates custom solvers for arithmetic reasoning and list partitioning – such a large system cannot even be fully described in a prompt.

---

*Equal contribution
†Corresponding author

Authors' addresses: Wentao Peng, Key Lab of High Confidence Software Technologies, Ministry of Education Department of Computer Science and Technology, School of Computer Science, Peking University, Beijing, China, 2200012940@stu. pku.edu.cn; Ruyi Ji, Key Lab of High Confidence Software Technologies, Ministry of Education Department of Computer Science and Technology, School of Computer Science, Peking University, Beijing, China, jiruyi910387714@pku.edu.cn; Yingfei Xiong, Key Lab of High Confidence Software Technologies, Ministry of Education Department of Computer Science and Technology, School of Computer Science, Peking University, Beijing, China, xiongyf@pku.edu.cn.
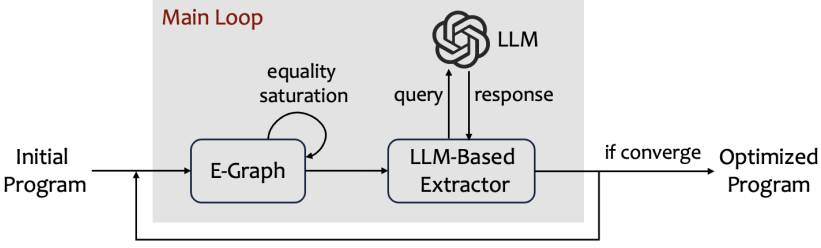
Fig. 1. The workflow of LGuess.

- On the other hand, most rewrite systems operate at a low level, making the rewrite chains too intricate for LLMs to maintain. For example, Szalinski often applies hundreds of rules to optimize CAD programs, resulting in complex rewrite chains comprising millions of AST nodes – this scale far exceeds the capabilities of current LLMs.

Our approach. To bridge the gap, this paper proposes *LLM-guided equality saturation*, dubbed as LGuess, by incorporating e-graphs as an intermediate layer between LLMs and rewrite systems.

LGuess is motivated by the recent work on *guided equality saturation* [Koehler et al. 2024], which demonstrates that human users can be aware of important checkpoints in the rewrite chain even without knowing details on the rewrite system. Based on this observation, LGuess applies LLMs to assume the role of human users in suggesting useful checkpoints, as LLMs have exhibited human-like capabilities; and then utilizes e-graphs to supply the rewrite chains between the checkpoints, mostly low-level rewrites that cannot be identified without knowing the rewrite system.

Fig. 1 illustrates the workflow of LGuess, which runs in multiple rewrite phases. In each phase, LGuess creates a new e-graph for the current program and applies equality saturation until reaching a resource limit. Then, LGuess queries LLMs to identify a checkpoint program from the e-graph and takes the checkpoint as the input of the next phase. This cycle repeats until convergence, i.e., when the checkpoint program remains unchanged after one phase.

Extraction from e-graphs. A key technical problem here is how to effectively extract a suitable checkpoint from the saturated e-graph. After equality saturation, an e-graph is typically huge in its size and the number of represented programs. Hence, it is impractical to describe the entire e-graph in a prompt or to query the LLM individually for each candidate program.

LGuess solves this problem by learning a probabilistic model from the LLM, which predicts probable checkpoints and remains simple enough to enable efficient extraction. LGuess extracts from the e-graph by iteratively refining a default model. In each round, it samples a new program according to the current model and queries the LLM to compare the new program and the previous result. Given the response, LGuess refines the model by enhancing the better program and weakening the worse one, and updates the result if the new program is preferred. After repeating this process for a pre-defined number of rounds, LGuess takes the final result as the checkpoint extracted from the e-graph.

Evaluation. We implement a prototype of LGuess and evaluate it on the problem of factorizing multi-variable polynomials. This problem is challenging for traditional program rewrite techniques because it requires numerous applications of the associative and commutative laws, which will typically cause a serious combinatorial explosion in the search space.

The results demonstrate the effectiveness of LGuess – it has a significant advantage compared to both straightforward equality saturation and querying the LLM directly for the rewrite chain.

(Sqr)	$a^2 \Leftrightarrow a \cdot a$
(Char-2)	$a + a \Rightarrow 0$
(Add-0)	$a + 0 \Rightarrow a$
(Add-C)	$a + b \Leftrightarrow b + a$
(Times-C)	$a \cdot b \Leftrightarrow b \cdot a$
(Add-A)	$a + (b + c) \Leftrightarrow (a + b) + c$
(Distr)	$(a + b) \cdot c \Rightarrow a \cdot c + b \cdot c$

Fig. 2. Rewrite rules in a ring with char. 2.

$$(x + y)^2$$
$$\Rightarrow^* \quad \{\text{Sqr, Distr, Times-C} \times 2\}$$
$$(x + y) \cdot x + (x + y) \cdot y$$
$$\Rightarrow^* \quad \{\text{Distr} \times 2, \text{Add-A} \times 2, \text{Times-C}\}$$
$$(x \cdot x + (x \cdot y + x \cdot y)) + y \cdot y$$
$$\Rightarrow^* \quad \{\text{Sqr} \times 2, \text{Char-2, Add-0}\}$$
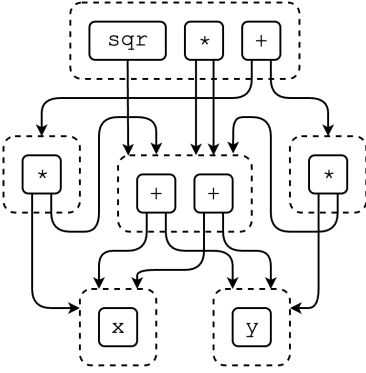$$x^2 + y^2$$

Fig. 3. The outline of the simplification.

Fig. 4. The saturated e-graph in the first phase, where *sqr* denotes the square operator $(\cdot)^2$, and $*$ denotes the multiplication operator.

Table 1. Our bigram models, where each row denotes an operator and each column denotes a context.

(a) The default model.

	$\bot$	sqr_1	$*_1$	$*_2$	$+_1$	$+_2$
sqr						
$*$		all probabilities are				
$+$		initialized to 0.20				
x						
y						

(b) The model updated after querying GPT-4o.

	$\bot$	sqr_1	$*_1$	$*_2$	$+_1$	$+_2$
sqr	0.09	0.22	0.17	0.17	0.13	0.13
$*$	0.36	0.22	0.17	0.17	0.13	0.13
$+$	0.18	0.11	0.33	0.33	0.13	0.13
x	0.18	0.22	0.17	0.17	0.53	0.07
y	0.18	0.22	0.17	0.17	0.07	0.53

2 OVERVIEW

We illustrate LGuess using the following task, which is often dubbed as the *freshman's dream*.

> Simplify the expression $(x + y)^2$ on a commutative ring with characteristic 2.

The expected result is $x^2 + y^2$. Fig. 2 lists the rewrite rules available in this task, and Fig. 3 outlines a rewrite chain for the simplification, which applies 13 rewrites in total.

GPT-4o can identify the expected result given the above task description, but it fails to construct a sound rewrite chain using the rewrite rules. On the other hand, this task is not trivial for equality saturation – it takes EGG [Willsey et al. 2021] 7 iterations to find the target expression.

Below, we show how LGuess solves this task under an extreme restriction on equality saturation, where we assume EGG can only run up to 2 iterations when rewriting each program.

Walkflow. LGuess runs by phases. In the first phase, LGuess starts with creating an e-graph for the initial expression $(x + y)^2$ and saturates it within the resource limit.

Fig. 4 shows the saturated e-graph, which contains 10 expressions: some make progress toward the simplification, such as $x \cdot (x + y) + y \cdot (x + y)$, while others do not, such as $(y + x)^2$. Among them, LGuess aims to extract the expression p^* that makes the most progress. It will record the rewrite chain of p^* into the result, and focus on simplifying p^* in the next phase.

One straightforward approach to extracting p^* is to scan over all programs in the e-graph, query the LLM to compare each program with the previous result, and updates the result if the LLM prefers the former. This approach, however, is impractical because a saturated e-graph typically contains an exponential number of programs.

LGuess improves this approach by incorporating a probabilistic model to predict suitable checkpoints, and querying the LLM only with probable programs, thus reducing the search space. The model is learned on the fly from the responses of the LLM.

Bigram model. LGuess utilizes a bigram model, as shown in Tab. 1. The model assigns a probability for each operator op under each context c, denoted as $\gamma(op \mid c)$; then the probability of a program is computed as the product of the probabilities of its operators. Except for the special context $\perp$ that denotes the topmost operator, each other context comprises two parts: the parent operator and the index of the current operator. For example, $\gamma(x \mid +_1)$ denotes the probability for variable x to appear as the first child of $+$. The index in the context enables our model to distinguish symmetric programs. It can assign different probabilities to $x + y$ and $y + x$, as shown below, which must otherwise be the same without the index.

$$\Pr[x + y] \coloneqq \gamma(+ \mid \perp) \cdot \gamma(x \mid +_1) \cdot \gamma(y \mid +_2) \qquad \Pr[y + x] \coloneqq \gamma(+ \mid \perp) \cdot \gamma(x \mid +_2) \cdot \gamma(y \mid +_1)$$

Another advantage of this model lies in its simplicity, which allows efficient extraction from e-graphs. Within polynomial time, we can both sample a random program and extract the most probable program from the e-graph according to the model.

Extraction guided by the model. To extract from the e-graph, LGuess starts with a default model where all operators are assigned the same probability (shown in Tab. 1a) and samples a random program as the initial result. Then, LGuess refines the model and updates the result by iteratively querying the LLM. In each round, it will (1) sample a new candidate program from the e-graph according to the model, (2) query the LLM to compare the previous result and the new candidate, and (3) update the model and the result according to the response.

Suppose the initial result is $(y + x)^2$, and program $(x + y) \cdot (x + y)$ is sampled in the first iteration. LGuess will query the LLM to decide which program makes more progress, as shown in Fig. 5. Given the response that the new program makes more progress, LGuess will take this program as the new result and refine the model by enhancing the probability of the new program while weakening the probability of the previous program. In more detail, LGuess will collect the model parameters that contribute to the probabilities of the two programs, as listed below.

$$\Pr[(y + x)^2] \coloneqq \gamma(sqr \mid \perp) \cdot \gamma(+ \mid sqr_1) \cdot \gamma(y \mid +_1) \cdot \gamma(x \mid +_2)$$
$$\Pr[(x + y) \cdot (x + y)] \coloneqq \gamma(* \mid \perp) \cdot \gamma(+ \mid *_1) \cdot \gamma(+ \mid *_2) \cdot \gamma(x \mid +_1)^2 \cdot \gamma(y \mid +_2)^2$$

LGuess will multiply each parameter by α for each time it contributes to the better program, and divide it by α for each time it contributes to the worse one, where $\alpha > 1$ is a pre-defined constant. For example, when $\alpha = 2$, $\gamma(x \mid +_1)$ will be multiplied by 4, and $\gamma(y \mid +_1)$ will be divided by 2.

Tab. 1b shows the refined model, where the probability sum is normalized to 1 under each context, and we mark the enhanced parameters as red and weakened parameters as blue. LGuess will use this new model to sample the next candidate program – this process will repeat for a pre-defined number of rounds, and the final result will be taken as the checkpoint.

Compared to the direct approach that queries each program one by one, the key advantage of LGuess is its ability to generalize across programs with similar structures. For example, in the refined model, the probability of $x + y$ is nearly 100 times higher than $y + x$. Such generalization makes LGuess focus on programs involving only $x + y$, thus immediately cutting off the program space by 70% because only 3 out of the 10 programs in the e-graph satisfy this restriction.

User: I am simplifying the s-expression (sqr (+ x y)) on a commutative ring with characteristic 2. Please compare the following two intermediate results, and decide which makes more progress toward simplification.

    (a) (sqr (+ y x))
    (b) (* (+ x y) (+ y x))

Conclude your output with **the answer is (a)/(b)**.
Assistant: **the answer is (b)**.

Fig. 5. A query to GPT-4o and the response.

Table 2. The checkpoints found by LGuess.

Phase	Checkpoint
Init	$(x + y)^2$
1	$(x + y) \cdot (x + y)$
2	$(x \cdot x + y \cdot x) + (x \cdot y + y \cdot y)$
3	$(x \cdot x + 0) + y \cdot y$
4/5	$x^2 + y^2$

Of course, this advantage hinges on the assumption that programs with significant progress share unique syntactic features. We make this assumption because Koehler et al. [2024] shows that human users can identify useful checkpoints via some fixed sketches, which are purely syntactic.

All checkpoints. In the first phase, LGuess will extract $(x + y) \cdot (x + y)$ as the checkpoint. Notably, GPT-4o does not prefer the program $x \cdot (x + y) + y \cdot (x + y)$ in the e-graph, although it is one step closer to the target expression. This is because GPT-4o cannot recognize low-level rewrites – it will directly connect $(x + y) \cdot (x + y)$ to $(x \cdot x + x \cdot y) + (y \cdot x + y \cdot y)$, skipping over five intermediate rewrites. Fortunately, this issue will not affect the effectiveness of LGuess, because $(x + y) \cdot (x + y)$ still makes positive progress toward the target and therefore serves as a valid checkpoint.

Then, LGuess moves to the next rewrite phase, restarts equality saturation from the checkpoint, and extracts the next one from the e-graph. This process repeats until convergence, where LGuess identifies 4 checkpoints (Tab. 2) and finally returns $x^2 + y^2$, the expected result.

3 EVALUATION

We have implemented a prototype of LGuess and evaluated it on a constructed dataset.

Dataset. We consider the problem of factorizing multi-variable polynomials, such as rewriting $x \cdot x + x \cdot y + x \cdot z + y \cdot z$ into $(x + y) \cdot (x + z)$, which can be regarded as an optimization problem for improving the efficiency of evaluating polynomials. This problem is challenging because it requires applying the associative and commutative laws many times, which typically leads to a combinatorial explosion in the search space.

We implement a random generator to construct tasks with various difficulties. Our generator takes two parameters n_d and n_v, denoting the degree of the polynomial and the number of different variables. It first samples n_d random (non-empty) subsets of variables, then multiplies the sum of each subset, unfolds the product, and at last applies the associative and commutative laws to randomly reorganize the expression. The following demonstrates a sample generation.

$$\langle n_d = 2, n_v = 3 \rangle \xrightarrow{\text{sample}} \{y\}, \{x, y, z\} \xrightarrow{\text{multiply}} y \cdot (x + y + z) \xrightarrow{\text{unfold}} y \cdot x + y \cdot y + y \cdot z$$

$$\xrightarrow{\text{randomly reorganize}} z \cdot y + (x \cdot y + y \cdot y)$$

The goal of each task is to rewrite the last expression back to the multiplication form.

We construct our dataset by running the generator 20 times for each combination of n_d and n_v within the range $[2, 5]$, thus creating tasks with a diverse range of difficulties from easy to hard.

Baselines. We compare LGuess with two baseline solvers.

- DirectES denotes direct equality saturation. It creates an e-graph for the initial expression and saturates the e-graph via egg until a multiplication form is reached.

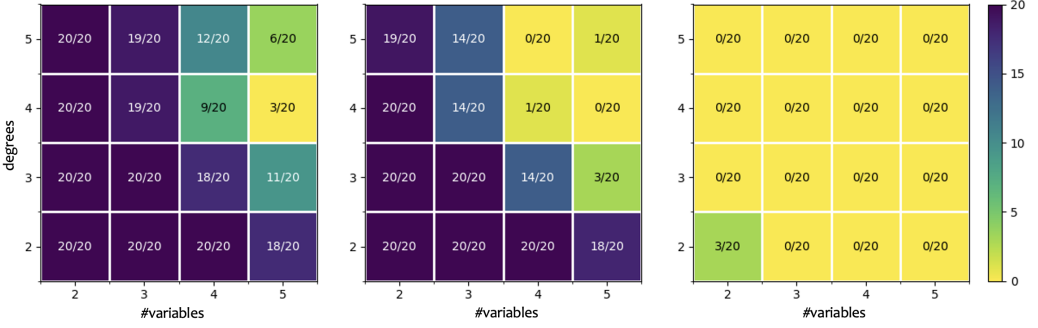(a) LGUESS solves 255/320 tasks.   (b) DIRECTES solves 204/320 tasks.   (c) DIRECTLLM solves 3/320 tasks.

Fig. 6. The results of our evaluation. In these figures, each row denotes a different degree of polynomials, and each column denotes a different number of variables – the factorization task becomes more and more challenging from lower left to upper right. In each cell, we report the number of solved tasks in the corresponding subset, where a deeper color denotes a better performance.

- DIRECTLLM directly generates the rewrite chain by LLMs. It queries the LLM with the task description and the rewrite system, then verifies the rewrite chain in the response.

Configuration. Our experiments are conducted on Intel Xeon Gold 6230 2.1GHz 20-Core Processor, with a timeout of 150 seconds and a memory limit of 24GB per task.

In each phase of LGUESS, we set a timeout of 5 seconds for equality saturation, limit the number of rounds for refining the bigram model to 10, and set the constant α to 1.5. Besides, we use GPT-4o as the backend LLM for both LGUESS and DIRECTLLM.

Evaluation results. We run LGUESS and the two baseline solvers on all tasks in our dataset. The results are summarized in Fig. 6. They demonstrate that LGUESS can solve most task in the dataset (255 out of 320) and significantly outperforms both baseline solvers.

- Compared with DIRECTES, LGUESS has a clear advantage on challenging tasks – it solves 30 tasks when the degree and the number of variables are both no smaller than 4, while DIRECTES can only solve 2. This is because LGUESS performs equality saturation in phases, thus alleviating the combinatorial explosion problem in DIRECTES.
- DIRECTLLM fails to solve almost all tasks because GPT-4o can hardly maintain the low-level rewrite chain – it frequently makes mistakes such as skipping steps and applying unavailable rules. In contrast, LGUESS addresses this issue by incorporating e-graphs to handle all low-level rewrites.

4 CONCLUSION AND FUTURE WORK

In this paper, we study the correctness issue of LLMs and propose a novel approach LGUESS, which bridges the gap between LLMs and formal rewrite systems via e-graphs. We evaluate LGUESS on a constructed dataset for polynomial factorization and preliminarily verify its effectiveness.

In the next step, we will instantiate LGUESS in more practical domains, such as optimizing CAD programs [Yaghmazadeh et al. 2016] and array programs [Hagedorn et al. 2020], to see whether LGUESS can effectively handle real-world problems. Besides, we will also investigate the reasons why LGUESS fails on some tasks in our dataset and work to improve it accordingly.

REFERENCES

Zimin Chen, Sen Fang, and Martin Monperrus. 2024. Supersonic: Learning to Generate Source Code Optimizations in C/C++. *IEEE Trans. Software Eng.* 50, 11 (2024), 2849–2864. https://doi.org/10.1109/TSE.2024.3423769

Chris Cummins, Volker Seeker, Dejan Grubisic, Baptiste Rozière, Jonas Gehring, Gabriel Synnaeve, and Hugh Leather. 2025. LLM Compiler: Foundation Language Models for Compiler Optimization. In *Proceedings of the 34th ACM SIGPLAN International Conference on Compiler Construction, CC 2025, Las Vegas, NV, USA, March 1-2, 2025*, Daniel Kluss, Sara Achour, and Jens Palsberg (Eds.). ACM, 141–153. https://doi.org/10.1145/3708493.3712691

Shuzheng Gao, Cuiyun Gao, Wenchao Gu, and Michael R. Lyu. 2024. Search-Based LLMs for Code Optimization. *CoRR* abs/2408.12159 (2024). https://doi.org/10.48550/ARXIV.2408.12159 arXiv:2408.12159

Spandan Garg, Roshanak Zilouchian Moghaddam, Colin B. Clement, Neel Sundaresan, and Chen Wu. 2022. DeepDev-PERF: a deep learning-based approach for improving software performance. In *Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering, ESEC/FSE 2022, Singapore, Singapore, November 14-18, 2022*, Abhik Roychoudhury, Cristian Cadar, and Miryung Kim (Eds.). ACM, 948–958. https://doi.org/10.1145/3540250.3549096

Spandan Garg, Roshanak Zilouchian Moghaddam, and Neel Sundaresan. 2023. RAPGen: An Approach for Fixing Code Inefficiencies in Zero-Shot. *CoRR* abs/2306.17077 (2023). https://doi.org/10.48550/ARXIV.2306.17077 arXiv:2306.17077

Bastian Hagedorn, Johannes Lenfers, Thomas Koehler, Xueying Qin, Sergei Gorlatch, and Michel Steuwer. 2020. Achieving high-performance the functional way: a functional pearl on expressing high-performance optimizations as rewrite strategies. *Proc. ACM Program. Lang.* 4, ICFP (2020), 92:1–92:29. https://doi.org/10.1145/3408974

Thomas Koehler, Andrés Goens, Siddharth Bhat, Tobias Grosser, Phil Trinder, and Michel Steuwer. 2024. Guided Equality Saturation. *Proc. ACM Program. Lang.* 8, POPL (2024), 1727–1758. https://doi.org/10.1145/3632900

Robert Tjarko Lange, Aaditya Prasad, Qi Sun, Maxence Faldor, Yujin Tang, and David Ha. 2025. The AI CUDA Engineer: Agentic CUDA Kernel Discovery, Optimization and Composition. (2025).

Chandrakana Nandi, Max Willsey, Adam Anderson, James R. Wilcox, Eva Darulova, Dan Grossman, and Zachary Tatlock. 2020. Synthesizing structured CAD models with equality saturation and inverse transformations. In *Proceedings of the 41st ACM SIGPLAN International Conference on Programming Language Design and Implementation, PLDI 2020, London, UK, June 15-20, 2020*, Alastair F. Donaldson and Emina Torlak (Eds.). ACM, 31–44. https://doi.org/10.1145/3385412.3386012

Max Willsey, Chandrakana Nandi, Yisu Remy Wang, Oliver Flatt, Zachary Tatlock, and Pavel Panchekha. 2021. egg: Fast and extensible equality saturation. *Proc. ACM Program. Lang.* 5, POPL (2021), 1–29. https://doi.org/10.1145/3434304

Navid Yaghmazadeh, Christian Klinger, Isil Dillig, and Swarat Chaudhuri. 2016. Synthesizing transformations on hierarchically structured data. In *Proceedings of the 37th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2016, Santa Barbara, CA, USA, June 13-17, 2016*. 508–521. https://doi.org/10.1145/2908080.2908088