Combining E-Graphs with Abstract Interpretation

Samuel Coward
Electrical and Electronic Engineering
Imperial College London
Email: s.coward21@imperial.ac.uk

George A. Constantinides
Electrical and Electronic Engineering
Imperial College London
Email: g.constantinides@imperial.ac.uk

Theo Drane
Numerical Hardware Group
Intel Corporation
Email: theo.drane@intel.com

***Abstract*—E-graphs are a data structure that compactly represents equivalent expressions. They are constructed via the repeated application of rewrite rules. Often in practical applications, conditional rewrite rules are crucial, but their application requires the detection – at the time the e-graph is being built – that a condition is valid in the domain of application. Detecting condition validity amounts to proving a property of the program. Abstract interpretation is a general method to learn such properties, traditionally used in static analysis tools. We demonstrate that abstract interpretation and e-graph analysis naturally reinforce each other through a tight integration because (i) the e-graph clustering of equivalent expressions induces natural precision refinement of abstractions and (ii) precise abstractions allow the application of deeper rewrite rules (and hence potentially even greater precision). We develop the theory behind this intuition and present an exemplar interval arithmetic implementation, which we apply to the FPBench suite of benchmarks.**

***Index Terms*—abstract interpretation, interval arithmetic, static analysis, e-graph**

I. Introduction

Equivalence graphs, commonly called e-graphs, provide a compact representation of equivalence classes (e-classes) of expressions, where the notion of equivalence is with respect to some concrete semantics [1]. The recent `egg` tool introduced *e-class analysis*, a technique to enable program analysis over an e-graph, attaching analysis data to each e-class [2]. This paper formalises some of the concepts required to produce e-class analyses enabling e-graph growth via conditional rewrites. We show that partitioning expressions into e-classes gives rise to a natural lattice-theoretic interpretation for abstract interpretation (AI), resulting in the generation of precise abstractions. Figure 1 provides an example, in which interval analysis of two equivalent expressions can be combined to produce tighter enclosing intervals than either one alone.

We develop the general theoretical underpinnings of AI on e-graphs, exploiting rewrites to produce tight abstractions using a lattice-theoretic formalism. We also provide a sound interpretation of cycles naturally arising in e-graphs, corresponding to extracting abstract fixpoint equations, and show that a known interval algorithm, the Krawczyk method, results as a special case.

This paper extends and generalises a talk abstract presented at the EGRAPHS workshop [3] in the following ways:

- formalization of AI with e-graphs in a lattice-theoretic setting

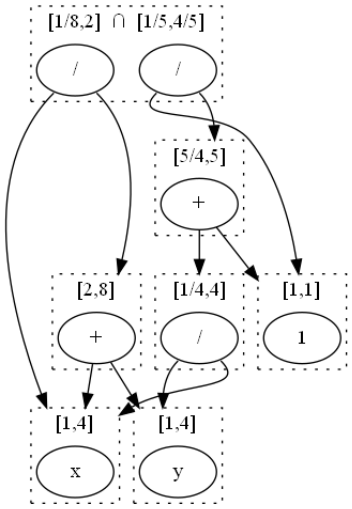

Fig. 1. E-graph containing equivalent real arithmetic expressions $\frac{x}{x+y}$ and $\frac{1}{1+(y/x)}$ in the root e-class (at the top). Intervals are associated with each e-class. Input constraints are propagated upwards via an e-class analysis.

- relating fixpoints to e-graph cycles to automatically discover iterative abstract refinement methods,
- an interval arithmetic implementation with associated expression bounding results.

A short background overview is provided in §II. In §III we present the theoretical application of AI to e-graphs and then demonstrate its viability using an interval arithmetic implementation in §IV. Lastly, in §V we present results generated by this implementation.

II. Background

The e-graph data structure is commonly found in theorem provers and solvers [4], [5]. It represents multiple expressions as a graph (e.g. Figure 1), where the nodes represent functions, grouped together into collections of e-classes. Edges connect nodes to e-classes, as a given sub-expression may be implemented using any of the nodes found in the child e-class. E-graphs are often combined with an optimisation technique called equality saturation [2], [6], [7], which deploys equivalence preserving transformations to monotonically grow the e-graph and discover alternative equivalent expressions. A recent resurgence of e-graph research [2] has seen the

technique applied to floating point numerical stability analysis [8], mapping programs onto hardware accelerators [9] and optimizing linear algebra [10]. Some works make heavy use of conditional rewrites of the form $\phi \Rightarrow \ell \to r$, for example $x \neq 0 \Rightarrow x/x \to 1$, which require the e-graph construction algorithm to determine whether the rewrite is applicable in each case [2], [11].

AI has primarily been applied to static program analysis [12], where computer programs are automatically analyzed without actually being executed. It uses the theory of abstraction to consider over-approximations of the program behaviour using alternative interpretations [13], [14]. Existing tools have incorporated term rewriting to refine their abstractions [15]. Previous work on abstract congruence closure [16], [17] has exploited e-graphs to combine different abstract domains [18], but we are not aware of existing work exploiting the tight interaction between AI and e-graph construction.

III. Theory

A. Abstraction

From a theoretical viewpoint, AI [14] is concerned with relationships between lattices, defined via Galois connections.

Definition 1 (Lattice)**.** A lattice is a partially ordered set (poset) $\langle L, \leq \rangle$, such that $\forall a, b \in L$ the least upper bound (join) $a \sqcup b$ and the greatest lower bound (meet) $a \sqcap b$ both exist.

Definition 2 (Galois connection)**.** Given a poset $\langle \mathcal{C}, \sqsubseteq \rangle$, corresponding to the concrete domain, and a poset $\langle \mathcal{A}, \preccurlyeq \rangle$, corresponding to the abstract domain, a function pair $\alpha \in \mathcal{C} \to \mathcal{A}$, $\gamma \in \mathcal{A} \to \mathcal{C}$, defines a Galois connection iff

$$\forall P \in \mathcal{C}.\ \forall \overline{P} \in \mathcal{A}.\ \alpha(P) \preccurlyeq \overline{P} \Leftrightarrow P \sqsubseteq \gamma(\overline{P}),$$

$$\text{written } \langle \mathcal{C}, \sqsubseteq \rangle \underset{\gamma}{\overset{\alpha}{\rightleftarrows}} \langle \mathcal{A}, \preccurlyeq \rangle.$$

The pair (α, γ) define the abstraction and concretization respectively, allowing us to over-approximate (i.e. abstract) concrete properties in $\mathcal{C}$ with abstract properties in $\mathcal{A}$.

Definition 3 (Sound abstraction [14])**.** $\overline{P} \in \mathcal{A}$ is a sound abstraction of a concrete property $P \in \mathcal{C}$ iff $P \sqsubseteq \gamma(\overline{P})$.

Consider expressions evaluated over a domain $\mathcal{D}$. By imposing a canonical ordering on the variable set, we work within a defined subset $I \subseteq \mathcal{D}^n$, which encodes a precondition on the set of (input) variable values. Now consider a (concrete) semantics of expressions $[\![\cdot]\!]. \in \text{Expr} \to I \to \mathcal{D}$, where Expr denotes the set of expressions, so $[\![e]\!]_\rho$ denotes the interpretation of expression e under execution environment (assignment of variables to values) $\rho \in I$. Let $[\![e]\!] = \{[\![e]\!]_\rho \,|\, \rho \in I\}$. The e-graph data structure encodes equivalence under concrete semantics, which we shall now define precisely.

Definition 4 (Congruence)**.** Two expressions e_a and e_b are congruent, $e_a \cong e_b$, iff $[\![e_a]\!]_\rho = [\![e_b]\!]_\rho$ for all $\rho \in I$.

Lemma 1. If $e_a \cong e_b$ and $\overline{P}$ is a sound abstraction of $[\![e_a]\!]$, then $\overline{P}$ is a sound abstraction of $[\![e_b]\!]$.

```
workqueue = egraph.classes().leaves()
while !workqueue.is_empty():
  s = workqueue.dequeue()
  for n in s.nodes():
    skip_node = false
    for child_s in n.children():
      if child_s.uninitialized:
        workqueue.enqueue(s)
        skip_node = true
    if skip_node:
        continue
    elif s.uninitialized:
        s.data = make(n)
        s.uninitialized = false
        workqueue.enqueue(s.parents())
    elif !(s.data<=meet(s.data,make(n))):
        s.data = meet(s.data,make(n))
        workqueue.enqueue(s.parents())
```

Fig. 2. Pseudocode for abstract property propagation in an e-graph.

Proof. by definition of congruence. □

This lemma implies that a sound abstraction of one expression in an e-class is a sound abstraction of all expressions in the e-class. Precision refinement relies on the following.

Lemma 2. For any two sound abstractions $\overline{P}_a$ and $\overline{P}_b$ of P, the meet $\overline{P}_a \sqcap \overline{P}_b$ is also a sound abstraction of P.

Proof. $P \sqsubseteq \gamma(\overline{P}_a)$ (sound abstraction) $\Rightarrow \alpha(P) \preccurlyeq \overline{P}_a$ (Galois connection) and similarly $P \sqsubseteq \gamma(\overline{P}_b) \Rightarrow \alpha(P) \preccurlyeq \overline{P}_b$. Therefore $\alpha(P) \preccurlyeq \overline{P}_a \sqcap \overline{P}_b$ (meet definition) and hence $P \sqsubseteq \gamma(\overline{P}_a \sqcap \overline{P}_b)$ (Galois connection). □

B. Application to E-graphs

Consider an e-graph. Let S denote the set of e-classes, and $\mathcal{N}_s$ the set of nodes in the equivalence class $s \in S$. With each e-class associate an abstraction $A \in \mathcal{A}$ and write $\mathcal{A}[\![s]\!] = A$. Interpreting a k-arity node n of function f with children classes $s_1, ..., s_k$, using the abstracted function $\bar{f} = \alpha \circ f \circ \gamma$:

$$\mathcal{A}[\![n]\!] = \bar{f}(\mathcal{A}[\![s_1]\!], ..., \mathcal{A}[\![s_k]\!]). \tag{1}$$

0-arity nodes represent either constants with exact abstractions in $\mathcal{A}$ or variables with user specified abstract constraints.

For acyclic e-graphs, we propagate the known abstractions upwards using Eqn. 1, taking the greatest lower bound (meet) across all nodes in the e-class.

$$\mathcal{A}[\![s]\!] = \bigsqcap_{n \in \mathcal{N}_s} \mathcal{A}[\![n]\!] \tag{2}$$

The propagation algorithm is described in Figure 2, where

$$\texttt{make}(n) = \mathcal{A}[\![n]\!] \text{ and } \texttt{meet}(A_1, A_2) = A_1 \sqcap A_2.$$

These functions are analogous to those described for an e-class analysis in [2], but replace their `join` function with a `meet`.

In the abstract domain the notion of equivalence is different, $n_a, n_b \in \mathcal{N}_s \not\Rightarrow \mathcal{A}[\![n_a]\!] = \mathcal{A}[\![n_b]\!]$, which results in tighter abstractions since the meet corresponds to the most precise

abstraction in $\mathcal{A}$. In the algorithm in Figure 2, by initializing the `workqueue` with only the modified e-classes after application of a rewrite, the abstract properties of the e-graph can be evaluated on the fly. On-the-fly evaluation facilitates conditional rewrite application as more precise properties are discovered during construction and is used in §IV.

A positive feedback loop is created by combining AI and e-graphs. A larger space of equivalent expressions is explored as more rewrites can be proven to be valid at exploration time. In turn, expression abstractions are further refined by discovering more equivalent expressions, allowing even more valid rewrites, and the cycle continues. Additionally, several equivalent expressions may contribute to the tight final abstraction. An example is shown in §IV.

C. Cyclic E-graphs and Fixpoints

Cyclic e-graphs arise when an expression is equivalent to a sub-expression of itself with respect to concrete semantics, for example $e \times 1 \cong e$. Let $e \cong e'$ where e appears as a subterm in e'. Treating other subterms as absorbed into the function, let $f : \mathcal{D} \to \mathcal{D}$ be the interpretation of e' as a (concrete) function of $[\![e]\!]_\rho$, so that – in particular – $f([\![e]\!]_\rho) = [\![e]\!]_\rho$ due to the congruence. Abstracting f via a sound abstraction $\bar{f}$, yields the corresponding abstract fixpoint equation $a = a \sqcap \bar{f}(a)$ where the meet operation arises from Equation 2.

Now consider the function $\tilde{f}(a) = a \sqcap \bar{f}(a)$. The decreasing sequence defined by $a_{n+1} = \tilde{f}(a_n)$ corresponds to applying the abstract property propagation around a cycle in the e-graph, given an initial sound abstraction a_0 of $[\![e]\!]$.

Lemma 3. $\alpha([\![e]\!])$ is a fixpoint of $\tilde{f}$.

Proof. $\alpha([\![e]\!]) = \alpha(f([\![e]\!]))$ (congruence) $\preccurlyeq \bar{f}(\alpha([\![e]\!]))$ (sound abstraction). Hence $\tilde{f}(\alpha([\![e]\!])) = \alpha([\![e]\!]) \sqcap \bar{f}(\alpha([\![e]\!])) = \alpha([\![e]\!])$ (meet definition). □

Lemma 4. a_n is a sound abstraction of $[\![e]\!]$ for all $n \in \mathbb{N}$.

Proof. By induction, $[\![e]\!] \sqsubseteq \gamma(a_0)$ and assume $[\![e]\!] \sqsubseteq \gamma(a_n)$. $[\![e]\!] = f([\![e]\!]) \sqsubseteq \gamma(\bar{f}(a_n))$ (sound abstraction of f). Hence $a_{n+1} = a_n \sqcap \bar{f}(a_n)$ is a sound abstraction of $[\![e]\!]$ (Lemma 2) for all n. □

Collecting these results, for some fixpoint a^* we have

$$\alpha([\![e]\!]) \preccurlyeq a^* \preccurlyeq \ldots \preccurlyeq a_1 \preccurlyeq a_0.$$

Thus computing abstractions around the loop refines the abstraction and is guaranteed to terminate if the lattice $\langle \mathcal{A}, \preccurlyeq \rangle$ satisfies the descending chain condition, as any finite abstract domain will [19]. The algorithm in Figure 2 will correctly apply abstract property propagation around loops, terminating if the sequence a_n converges in a finite number of steps.

IV. Implementation

To demonstrate the theory described above, we implement interval arithmetic (IA [20]) for real valued expressions using the extensible `egg` library, as an e-class analysis [2]. We consider a concrete domain corresponding to sets of extended real numbers, i.e. $\mathcal{C} = \mathcal{P}(\mathbb{R} \cup \{-\infty, +\infty\})$ where $\mathcal{P}$ denotes the power set. We associate each expression with a `binary64` [21] valued interval (a finite abstract domain),

$$\mathcal{A} = \{[a, b] \mid a \leq b, a, b \in \texttt{binary64}\} \cup \{\emptyset\}.$$

In this setting the abstraction and concretization functions are as follows (infima and suprema always exist in this setting):

$$\alpha(X) = [\text{round\_down}(\inf X), \text{round\_up}(\sup X)] \quad (3)$$

$$\gamma([a, b]) = [a, b] \quad (4)$$

$$\alpha(\emptyset) = \emptyset, \gamma(\emptyset) = \emptyset \quad (5)$$

To ensure correctness, we use 'outwardly rounded IA' which conservatively rounds upper bounds towards $+\infty$ (round_up) and lower bounds towards $-\infty$ (round_down) [20], [22]. For particular operators, e.g. $\sqrt{x}$, we are unable to control the rounding mode, so conservatively add or subtract one unit in the last place for upper and lower bounds respectively. Provided `NaN`s do not appear in the input expression evaluation they are not generated by the e-graph exploration.

The abstraction of a function f is defined as above $\bar{f} = \alpha \circ f \circ \gamma$. We will denote $\mathcal{A}[\![s]\!] \in \mathcal{A}$, for class s. Under this interpretation Eqn. 2 uses the intersection operation, the meet operation of the lattice of intervals.

$$\mathcal{A}[\![s]\!] = \bigcap_{n \in \mathcal{N}_s} [\![n]\!] \quad (6)$$

This relationship generates monotonically narrowing interval abstractions. 0-arity nodes represent either constants associated with degenerate intervals or variables taking user defined interval constraints.

The classical problem of interval arithmetic is the so-called 'dependency problem', arising because the domain does not capture correlations between multiple occurrences of a single variables. Consider $x \in [0, 1]$, under classical IA:

$$\mathcal{A}[\![x - x]\!] = [0, 1] - [0, 1] = [0 - 1, 1 - 0] = [-1, 1]. \quad (7)$$

Within the e-graph framework we discover, via term rewriting, $x - x \cong 0$ and by Eqn. 6 the expression is now correctly abstracted by the (much tighter) degenerate interval $[0, 0]$.

For this work we use a set of 39 rewrites, defining equivalences of real valued expressions. The basic arithmetic rewrites are commutativity, associativity, distributivity, cancellation and idempotent operation reduction across addition, subtraction, multiplication and division. Conversion rewrites describe the

TABLE I
Additional IA optimization rewrites.

Class	Rewrite	Condition
Common Terms	$ab \pm ac \to a(b \pm c)$	True
Binomial	$1/(1-a) \to 1 + a/(1-a)$	$0 \notin [\![1 - a]\!]$
Frac plus Int	$b/c \pm a \to (b \pm ac)/c$	$0 \notin [\![c]\!]$
Division [20]	$a/b \to 1/(b/a)$ $a/b \to 1 + (a-b)/b$	$0 \notin [\![a]\!] \cup [\![(b/a)]\!]$ $0 \notin [\![b]\!]$
Factorise	$a^2 - 1 \to (a-1)(a+1)$	True
Elementary	$\ln(e^a) \to a$	True

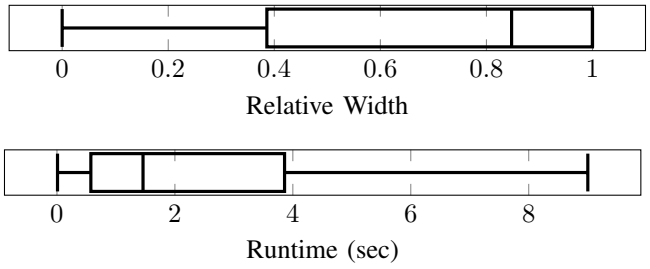

Fig. 3. Relative interval width (optimized width/naive width) and runtime boxplots to demonstrate the distribution of results on the FPBench suite.

natural equivalence between the power function and multiplication/division. Table I contains the remaining rewrites.

Conditional rewrites such as the first "Division" rewrite are only valid on a subset of the input domain. By attaching additional information through an AI we can prove validity of such rules. Eqn. 8 illustrates how conditional rewrites can be applied, as IA can confirm that $0 \notin [\![x + y]\!]$, in order to remove multiple occurrences of variables, improving the expression bound produced by IA.

$$\frac{x+y}{x+y+1} \rightarrow ... \rightarrow \frac{1}{1+\frac{1}{x+y}} \tag{8}$$

To show how multiple expressions in an e-class can independently contribute to a tight abstraction, consider the following equivalent expressions for variables $x \in [0, 1]$ and $y \in [1, 2]$.

$$1 - \frac{2y}{x+y} \qquad \in \left[-3, \frac{1}{3}\right] \tag{9}$$

$$\cong \frac{x-y}{x+y} \qquad \in [-2, 0] \tag{10}$$

$$\cong \frac{2x}{x+y} - 1 \qquad \in [-1, 1]. \tag{11}$$

All three reside in the same e-class within an e-graph with associated interval $[-3, \frac{1}{3}] \cap [-2, 0] \cap [-1, 1] = [-1, 0]$. Thus, given the first expression 9, we are capable of generating a tight interval enclosure using two distinct equivalent expressions for the upper (10) and lower (11) bounds.

V. Results

Based on the theory described above we describe a real valued expression bounding tool in Rust using the `egg` library. All test cases were run on an Intel i7-10610U CPU.

A. Benchmarks

We evaluate the implementation using 40 benchmarks from the FPTaylor [23] supported subset of the FPBench benchmark suite [24]. We allow four iterations of e-graph rewriting for these experiments. Figure 3 summarises the distribution of the results, showing a modest average interval reduction over naive IA, but a substantial improvement in particular cases. There is little correlation between the runtime and bound improvement.

Across this benchmark set the inclusion of IA and domain specific rewrites increased the number of e-graph nodes by 4% on average but by up to 84% in some cases. This demonstrates the additional rewrites that have been applied as a result of combining e-graphs and AI. The overhead of incorporating IA into the e-graph increased runtimes by less than 1% on average.

B. Iterative Method Discovery

The Krawczyk method [20] is a known algorithm to generate increasingly precise element-wise interval enclosures of solutions of linear systems of equations $Ax = b$, where A is an n-by-n matrix and b is an n-dimensional vector. Letting X^0 be an initial interval enclosure of the solutions, the Krawcyzk method uses an update formula of the form

$$X^{k+1} = \big(Yb + (I - YA)X^k\big) \cap X^k, \text{ where } Y = \text{mid}(A)^{-1}.$$

mid(A) is the element-wise interval midpoint of the matrix. This sequence, via interval extension and intersection, corresponds to a sequence of tightening bounds on the solution x, which converges provided the matrix norm $||I - YA|| < 1$.

We consider a specific instance of this problem,

$$\begin{pmatrix} 1 & y \\ y & 1 \end{pmatrix} \begin{pmatrix} x_1 \\ x_2 \end{pmatrix} = \begin{pmatrix} b_1 \\ b_2 \end{pmatrix}, \text{ where } y \in \left[-\frac{1}{2}, \frac{1}{2}\right]. \tag{12}$$

$$X_1^{k+1} = \big(b_1 - yX_2^k\big) \cap X_1^k, \; X_2^{k+1} = \big(b_2 - yX_1^k\big) \cap X_2^k. \tag{13}$$

A naive solution in the concrete domain, $x = A^{-1}b$, yields,

$$x_1 = \frac{1}{1-y^2}(b_1 - b_2 y), \; x_2 = \frac{1}{1-y^2}(b_2 - b_1 y). \tag{14}$$

Initialising the e-graph with these expressions, the solution for x_1 can be automatically rewritten such that Eqn. 13 arises in the abstract domain. The "Binomial" rewrite from Table I introduces a loop into the e-graph, which we combine with distributivity rules and "Common Terms" from Table I,

$$x_1 = b_1 - y\left(b_2 + (b_2 y^2 - b_1 y)\frac{1}{1-y^2}\right). \tag{15}$$

Applying, "Frac plus Int", and cancelling

$$x_1 = b_1 - y(b_2 - b_1 y)\frac{1}{1-y^2} = b_1 - yx_2. \tag{16}$$

When a cycle is introduced, the IA update procedure will continue to iteratively evaluate the loop, taking the intersection with the previous iteration as described in §III-C.

VI. Conclusion

We present a combination of abstract interpretation and e-graphs, demonstrating the natural interpretation of e-class partitions as meet operators in a lattice, resulting in precise abstractions. Of key importance is the positive feedback loop between e-graph exploration and abstraction refinement, as the precision then allows the application of conditional rewrite rules, which can be applied in many domains and may further improve abstraction precision. An exemplar interval arithmetic implementation has demonstrated the value of this idea, including automated discovery of a known algorithm for iterative refinement. Future work will explore additional abstract domains in this setting which can capture variable relations, using the resulting tool for program optimization.

REFERENCES

[1] C. G. Nelson, "Techniques for program verification," Ph.D. dissertation, Stanford University, 1980.

[2] M. Willsey, C. Nandi, Y. R. Wang, O. Flatt, Z. Tatlock, and P. Panchekha, "Egg: Fast and extensible equality saturation," *Proceedings of the ACM on Programming Languages*, vol. 5, no. POPL, 2021.

[3] S. Coward, G. A. Constantinides, and T. Drane, "Abstract Interpretation on E-Graphs," in *EGRAPHS Workshop - co-located with PLDI*, 3 2022.

[4] L. De Moura and N. Bjørner, "Z3: An efficient SMT Solver," in *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, vol. 4963 LNCS, 2008.

[5] D. Detlefs, G. Nelson, and J. B. Saxe, "Simplify: A theorem prover for program checking," *Journal of the ACM*, vol. 52, no. 3, 2005.

[6] R. Tate, M. Stepp, Z. Tatlock, and S. Lerner, "Equality saturation: A new approach to optimization," in *ACM SIGPLAN Notices*, vol. 44, no. 1, 2009.

[7] R. Joshi, G. Nelson, and K. Randall, "Denali: A goal-directed superoptimizer," in *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, 2002.

[8] P. Panchekha, A. Sanchez-Stern, J. R. Wilcox, and Z. Tatlock, "Automatically improving accuracy for floating point expressions," *ACM SIGPLAN Notices*, vol. 50, no. 6, pp. 1–11, 2015.

[9] G. H. Smith, A. Liu, S. Lyubomirsky, S. Davidson, J. McMahan, M. Taylor, L. Ceze, and Z. Tatlock, "Pure tensor program rewriting via access patterns (representation pearl)," in *Proceedings of the 5th ACM SIGPLAN International Symposium on Machine Programming*, 2021.

[10] Y. R. Wang, S. Hutchison, J. Leang, B. Howe, and D. Suciu, "SPORES: Sum-product optimization via relational equality saturation for large scale linear algebra," *Proceedings of the VLDB Endowment*, vol. 13, no. 11, 2020.

[11] S. Coward, G. A. Constantinides, and T. Drane, "Automatic Datapath Optimization using E-Graphs," 4 2022. [Online]. Available: https://arxiv.org/abs/2204.11478

[12] P. Cousot and R. Cousot, "Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints," in *Conference Record of the Annual ACM Symposium on Principles of Programming Languages*, vol. Part F130756, 1977.

[13] P. Cousot, "Abstract Interpretation Based Formal Methods and Future Challenges," in *Informatics: 10 Years Back, 10 Years Ahead*, 2001.

[14] ——, *Principles of Abstract Interpretation*, 1st ed. MIT Press, 9 2021.

[15] M. Daumas and G. Melquiond, "Certification of bounds on expressions involving rounded operators," *ACM Transactions on Mathematical Software*, vol. 37, no. 1, 2010.

[16] L. Bachmair, A. Tiwari, and L. Vigneron, "Abstract congruence closure," *Journal of Automated Reasoning*, vol. 31, no. 2, 2003.

[17] L. Bachmair and A. Tiwari, "Abstract congruence closure and specializations," in *Lecture Notes in Artificial Intelligence (Subseries of Lecture Notes in Computer Science)*, vol. 1831, 2000.

[18] B. Y. E. Chang and K. R. M. Leino, "Abstract interpretation with alien expressions and heap structures," in *Lecture Notes in Computer Science*, vol. 3385, 2005.

[19] G. Birkhoff, "Von Neumann and Lattice Theory," *Bulletin of the American Mathematical Society*, vol. 64, no. 3, 1958.

[20] R. E. Moore, R. B. Kearfott, and M. J. Cloud, *Introduction to Interval Analysis*. SIAM, 2009.

[21] "IEEE Standard for Floating-Point Arithmetic," *IEEE Std 754-2008*, pp. 1–70, 2008.

[22] U. Kulisch, *Computer Arithmetic in Theory and Practice*, 1981.

[23] A. Solovyev, M. S. Baranowski, I. Briggs, C. Jacobsen, Z. Rakamarić, and G. Gopalakrishnan, "Rigorous estimation of floating-point round-off errors with symbolic Taylor expansions," *ACM Transactions on Programming Languages and Systems*, vol. 41, no. 1, 2018.

[24] N. Damouche, M. Martel, P. Panchekha, C. Qiu, A. Sanchez-Stern, and Z. Tatlock, "Toward a standard benchmark format and suite for floating-point analysis," in *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, vol. 10152 LNCS, 2017.