Predicate E-Graphs with Symbolic Conditional Rewriting

Anders Ågren Thuné
Uppsala University, Sweden

Johannes Borgström
Uppsala University, Sweden

Lars-Henrik Eriksson
Uppsala University, Sweden

John Högberg
Ericsson, Stockholm, Sweden

Tjark Weber
Uppsala University, Sweden

Tobias Wrigstad
Uppsala University, Sweden

1 Introduction

Implementing an optimising compiler presents a number of challenges in terms of maintainability, correctness, and efficiency. The optimising compiler must not only implement the code transformation of each optimisation pass, but also the analysis deciding when to apply it. A given pass may require certain circumstances, or *preconditions*, to hold in order to be sound, and for high reliability the applicability of each optimisation may need to be checked separately from the code that implements it.

Our initial concern has been to find a data structure that can accurately and efficiently store multiple optimised variants of a program term (such as a function body). The state-of-the-art data structure for this task is the e-graph [3, 4]. However, e-graphs cannot directly represent that terms are equal under different preconditions.

Coloured e-graphs [2] are an efficient representation of several related congruence relations (each represented by an e-graph). In coloured e-graphs, each e-graph is associated with a colour: an identifier that is opaque to the operations of the data structure. Concretely, a coloured e-graph is implemented as a rooted tree of e-graphs, where a node (i.e., colour) in the tree may consider congruent two terms that the node's parent considers to be different (yielding a semi-lattice of congruences). This invariant makes coloured e-graphs interesting for conditional rewriting, where knowing more facts makes more rewrite rules applicable.

We take this idea a step further with *predicate e-graphs* where each colour is associated with a set of truth-valued assumptions (conditions) under which the congruences in the associated e-graph hold. The tree structure of coloured e-graphs is generalised to a DAG where the edges represent an implication relation on assumptions. The assumptions are expressed in the same language as the terms, and indeed are stored in the e-graphs themselves. This allows assumptions to be manipulated in the same way as any other term. Rewriting may thus make the assumptions of two colours equal, in which case the two nodes are merged to a single e-graph.

This extended abstract presents a work-in-progress design for predicate e-graphs. Our goal is to explore representational and operational foundations rather than to provide a complete algorithm, soundness proof, or evaluation. A formalisation and a prototype implementation are in progress.

2 Background

In this section, we briefly introduce the formal definition of e-graphs that we will use in the rest of the paper, and recall the notion of coloured e-graphs [2].

As in [2], we assume a set Σ containing the *function symbols* in the language and their arities. The sets of closed terms $\mathcal{T}$ and of patterns $\mathcal{T}_V$ over the variables V are defined

inductively as expected. We also assume an infinite set $\mathcal{E}$ of opaque *e-class identifiers*, and let the *nodes* over a set $X \subseteq \mathcal{E}$ be the set $\mathcal{N}_X \coloneqq \{f(x_1, \ldots, x_n) \mid f^{(n)} \in \Sigma,\ x_i \in X\}$.

2.1 E-graphs

In this paper, we use a formal definition of an e-graph as consisting of (i) the set of valid e-class IDs in the graph, (ii) the hash-cons structure, and (iii) the union-find structure.

▶ **Definition 1** (E-graph). *An* e-graph *is given by a tuple* (E, H, find)*, where*

$$\begin{array}{lll} \textit{valid e-class IDs} & E & \subseteq \mathcal{E}, \\ \textit{hash-cons} & H & : \mathcal{N}_E \rightharpoonup E, \\ \textit{union-find} & \mathrm{find} & : E \to E. \end{array}$$

For a given e-graph (E, H, find), we derive the e-class map $M : E \to \mathcal{P}(\mathcal{N}_E)$ by

$$M(e) \coloneqq \{\, f(e_1, \ldots, e_n) \mid H(f(\mathrm{find}(e_1), \ldots, \mathrm{find}(e_n))) = \mathrm{find}(e) \,\},$$

although an implementation would represent it explicitly. As usual, an e-graph induces notions of term equivalence $\equiv$ and term representation $[\![\cdot]\!] \colon E \to \mathcal{P}(\mathcal{T})$. The well-formedness conditions are that find is idempotent and that the hash-cons only returns canonical e-class IDs; we may also require that every e-class represents some term.

We model the primitive operations for (i) introducing new e-nodes, (ii) merging e-classes, and (iii) performing pattern matching over the graph structure.

▶ **Definition 2** (E-graph operations). *E-graphs provide operations with the following signatures, where g is an e-graph, and $\mathfrak{E}$ is the set of all e-graphs:*

$$\begin{array}{lll} \textit{e-node insertion} & \mathrm{insert}_g & : N_g \to \mathfrak{E}, \\ \textit{e-class union} & \mathrm{union}_g & : E_g \to E_g \to \mathfrak{E}, \\ \textit{pattern matching} & \mathrm{match}_g & : \mathcal{T}_V \to \mathcal{P}(E_g \times (V \rightharpoonup E_g)). \end{array}$$

Here, insert_g, for a given e-graph g, takes an e-node, and returns the e-graph obtained by inserting the node into g. The operation union_g takes two e-class IDs, and returns a new e-graph in which the two e-classes have been merged. The operation match_g takes a pattern, and returns a set of pairs (e, θ), where $e \in E_g$ is the e-class ID of a class that represents a term matching the pattern, and the substitution $\theta : V \rightharpoonup E_g$ indicates which e-class matched each of the variables in the pattern.

2.2 Coloured E-graphs

A coloured e-graph [2] is an efficient representation of a tree-structured family of e-graphs, where each node (or colour) in the tree intuitively represents a condition under which the equivalences in that e-graph are valid. Formally, we assume an infinite set $\mathcal{C}$ of colours, with a distinguished colour $\bullet$. We can then view a coloured e-graph as a set $C \subseteq \mathcal{C}$ of colours equipped with a parent relation $P : C \setminus \{\bullet\} \to C$ and a family of e-graphs $(E_c, H_c, \mathrm{find}_c)_{c \in C}$.

Here P is required to form a tree with $\bullet$ being its root. Additionally, if $P(c_1) = c_2$, so that c_1 is the child of c_2, then there is a well-formedness condition that the term equivalence associated with c_1 is coarser (i.e., identifies more terms) than that of c_2. This enables efficient incremental representation of coloured e-graphs. Coloured e-graphs support the operations of Definition 2 at any colour, guaranteed to preserve the well-formedness condition.

3 Predicate E-Graphs

Our main contribution is the definition of predicate e-graphs to represent the possible conditionally optimised versions of a program term. Predicate e-graphs are similar to coloured e-graphs, in that they consist of a family of standard e-graphs indexed by colours that form a directed graph. Predicate e-graphs extend coloured e-graphs in several directions:

1. Colours are associated with sets of Boolean terms, and the directed graph structure on colours is compatible with the implication structure on these sets.
   Each colour c is associated with a set of assumptions $\mathrm{facts}(c)$, given by the terms congruent to $\texttt{true}$, under which the equivalences in the associated e-graph are valid. In other words, asking whether terms t and t' are equivalent in the c-coloured e-graph is the same as asking whether t and t' are equivalent *given the set of facts* in $\mathrm{facts}(c)$.
2. Colours form a rooted directed graph instead of a tree, meaning that one colour can have multiple direct "parents."
   We write $c_1 \triangleright c_2$ if there is a path from c_1 to c_2. Every colour has a path to the root $\bullet$.
3. The directed graph structure can grow dynamically.
   A path $c_1 \triangleright c_2$ in the graph intuitively represents that the set of assumed facts associated with c_1 is known to subsume that of c_2. Implication between fact sets needs to be checked dynamically because of rewrites that make terms congruent. If the fact set of one colour is proven to imply that of another (e.g., as attested to by an off-the-shelf SMT solver such as Z3 [1]), we introduce a $\triangleright$ edge in the colour graph. The search for such implications is an expensive operation, but it can be performed periodically and with tunable effort.
   Similarly to coloured e-graphs, colours further away from the root equate more terms. Because of this well-formedness condition, the addition of an edge $c_1 \triangleright c_2$ requires that the nodes and equivalences in the e-graph of c_2 be added to that of c_1, by taking the union of node sets and fusing overlapping e-classes.
4. Colours can be identified, requiring the corresponding e-graphs to be merged.
   Because of the well-formedness condition, all colours in a $\triangleright$ cycle refer to the same term equivalence. Hence, whenever we dynamically introduce a $\triangleright$ cycle, we associate all colours in that cycle with the same e-graph, corresponding to the merger of their original e-graphs as above. The association from colours to e-graphs may go via a union-find structure, ensuring efficient merging of equivalence classes of colours when cycles are introduced. Equivalence classes of colours modulo $\triangleright$ cycles form a rooted DAG.

As an example, consider a predicate e-graph with four distinct colours $\bullet, l, r, o$ where $l \triangleright \bullet$, $r \triangleright \bullet$, $o \triangleright \bullet$, $\mathrm{facts}(l) = \{a \geq 1, \texttt{true}\}$, $\mathrm{facts}(r) = \{a \leq 1, \texttt{true}\}$ and $\mathrm{facts}(o) = \{a = 1, \texttt{true}\}$. As a first step in the example, we introduce a new colour m with $m \triangleright l$ and $m \triangleright r$. Then well-formedness ensures that $\mathrm{facts}(m) = \mathrm{facts}(l) \cup \mathrm{facts}(r)$.

Since predicate e-graphs associate colours with syntactic conditions, they can perform reasoning internally, as rewrite rules on conditions. For instance, in the example above, we may have rewrite rules $\texttt{true} \to \texttt{true} \wedge \texttt{true}$ and $x \geq y \wedge x \leq y \to x = y$. As a second step, we apply these rules to the m-coloured e-graph above, yielding $\mathrm{facts}(m) \supseteq \{a = 1, a \geq 1, a \leq 1, \texttt{true}\}$. From $\mathrm{facts}(m) \supseteq \mathrm{facts}(o)$ we can immediately deduce and introduce an edge $m \triangleright o$.

In the final step, we may deduce that $a = 1 \Rightarrow a \geq 1 \wedge a \leq 1$, and hence we introduce the corresponding edge $o \triangleright m$. This introduces a cycle in the $\triangleright$ graph, so we must ensure that both colours refer to the merger of their e-graphs, here equal to the e-graph of m.

We envision predicate e-graphs to be especially useful for conditional rewriting, where a conditional rewrite rule $\phi \vdash p \to q$ may either be applied or not based on a syntactic membership check $\phi \in \mathrm{facts}(c)$, or may be applied symbolically by dynamically introducing

a new colour associated with the condition ϕ.

3.1 Formal description of predicate e-graphs

▶ **Definition 3** (Predicate e-graph). *A* predicate e-graph *is given by*

colours	C	$\subseteq \mathcal{C}$	*with* $\bullet \in C$,
colour paths	$\triangleright$	$\subseteq C \times C$,	
component e-graphs	$(E_c, H_c, \mathrm{find}_c)$	$\in \mathfrak{E}$	*for all* $c \in C$,
truth e-class	$\top$	$\in E_\bullet$.	

In a predicate e-graph, we also have node sets $N_c \coloneqq \mathcal{N}_{E_c}$ and e-class maps $M_c \colon E_c \to \mathcal{P}(N_c)$ as in Section 2.1, parametrised by colours $c \in C$. Similarly, a predicate e-graph induces parametrised notions of term equivalence $\equiv_c$ and representation $[\![\cdot]\!]_c$.

We are especially interested in the set of terms equivalent to `true` in a given colour, i.e., the facts associated to that colour.

▶ **Definition 4** (Fact sets). *For a given predicate e-graph, we define the set of* facts *associated with a colour* $c \in C$ *to be the terms equivalent to* `true` *in the e-graph associated to* c*:* $\mathrm{facts}(c) \coloneqq \{t \in \mathcal{T} \mid t \equiv_c \texttt{true}\}$.

To relate e-graphs of different colours, we introduce a subsumption relation as follows.

▶ **Definition 5** (Colour subsumption). *If* c_1 *and* c_2 *are colours, we say that* c_1 subsumes c_2 *and write* $c_1 \blacktriangleright c_2$ *if* $E_{c_1} \supseteq E_{c_2} \wedge \forall e \in E_{c_2}.\ M_{c_1}(e) \supseteq M_{c_2}(e)$.

Intuitively, if c_1 subsumes c_2, then c_1 equates at least as many e-nodes as c_2.

▶ **Definition 6** (Well-formed predicate e-graph). *We say that a predicate e-graph is* well-formed *if the following properties hold.*

1. *The truth e-class contains* `true` *in the root colour:* $\texttt{true} \in M_\bullet(\top)$.
2. *The relation* $\triangleright$ *is a preorder with* $\bullet$ *as minimal element, i.e., for all colours* $a, b, c \in C$*:*

   $$c \triangleright \bullet \quad \textit{and} \quad c \triangleright c \quad \textit{and} \quad a \triangleright b \wedge b \triangleright c \Rightarrow a \triangleright c.$$

3. *If there is a path from* c_1 *to* c_2*, then* c_1 *subsumes* c_2*.*

   $$c_1 \triangleright c_2 \quad \textit{implies} \quad c_1 \blacktriangleright c_2.$$

4. *Whenever an e-class ID exists in multiple component e-graphs, they must share an ancestor that also contains that ID:*

   $$e \in E_{c_1} \cap E_{c_2} \quad \textit{implies} \quad \exists c.\ c_1 \triangleright c \wedge c_2 \triangleright c \wedge e \in E_c.$$

We let $\mathfrak{P}$ *denote the set of well-formed predicate e-graphs.*

Items 1 and 2 formally state properties discussed above: they ensure that `true` belongs to the truth class $\top$, that $\triangleright$ is well-behaved, and that $\bullet$ is the root colour. Item 3 is the core semantic invariant, which ensures that edges in the colour graph correspond to inclusions on the associated e-graphs. (Whenever we detect inclusions between fact sets we also add edges to the colour graph, but since inclusion detection is hard in general, not all inclusions are guaranteed to be detected.) Intuitively, this property says that if we have an edge from c_1 to c_2, then c_1 represents a coarser term equivalence than c_2. Finally, Item 4 is a technical requirement to ensure that no spurious ID collisions occur when merging component e-graphs.

Note that from Item 3 it follows that whenever there is a path from c_1 to c_2 in the DAG, then $\text{facts}(c_1) \supseteq \text{facts}(c_2)$, i.e., the set of facts associated with c_1 subsumes that of c_2. Moreover, from Items 1 and 2 we get $\text{facts}(c) = [\![\top]\!]_c$.

If $c_1 \triangleright c_2$ and $c_2 \triangleright c_1$, then Item 3 ensures that c_1 subsumes c_2 and c_2 subsumes c_1. As they then denote the same equivalence relation, an implementation could make them alias each other (perhaps via a union-find structure), referring to the very same e-graph.

3.2 Basic operations on predicate e-graphs

Predicate e-graphs inherit operations from their component e-graphs. Letting $g \in \mathfrak{P}$ be a predicate e-graph, we have a set of colours C_g and a DAG structure $\triangleright_g$. For each colour $c \in C_g$, we have operations:

$$E_{g,c} \subseteq \mathcal{E}, \qquad H_{g,c} : N_{g,c} \rightharpoonup E_{g,c}, \qquad \text{find}_{g,c} : E_{g,c} \to E_{g,c}, \qquad M_{g,c} : E_{g,c} \to \mathcal{P}(N_{g,c})$$

where $N_{g,c}$ denotes $\mathcal{N}_{E_{g,c}}$. We also get a truth class $\top_g \in E_{g,\bullet}$, and a colour-wise pattern matching operation:

$$\text{match}_{g,c} : \mathcal{T}_V \to \mathcal{P}(E_{g,c} \times (V \rightharpoonup E_{g,c})).$$

▶ **Definition 7** (Predicate e-graph operations). *Predicate e-graphs provide operations with the following signatures, where $g \in \mathfrak{P}$ and $c \in C_g$:*

$$\begin{aligned}
&\text{insert}_{g,c} && : N_{g,c} \to \mathfrak{P},\\
&\text{union}_{g,c} && : E_{g,c} \to E_{g,c} \to \mathfrak{P},\\
&\text{addEdge}_g && : C_g \to C_g \to \mathfrak{P},\\
&\text{addColour}_g && : \mathcal{P}(C_g) \to \mathfrak{P} \times \mathcal{C},\\
&\text{colMatch}_{g,c} && : \mathcal{T}_V \to \mathcal{P}(C_g \times \mathcal{E} \times (V \rightharpoonup \mathcal{E})),\\
&\text{colMatchIn}_{g,c} && : E_{g,c} \to \mathcal{T}_V \to \mathcal{P}(C_g \times (V \rightharpoonup \mathcal{E})).
\end{aligned}$$

The operations $\text{insert}_{g,c}$ and $\text{union}_{g,c}$ are colour-annotated variants of the corresponding operations for plain e-graphs, and the return type dictates that these operations must restore the well-formedness properties of Definition 6. The operation addEdge_g adds an edge between two existing colours in the graph. The operation addColour_g takes a set of colours and creates a fresh colour with edges to all the given colours, populating the associated e-graph as necessary, and then returns a pair containing the updated predicate e-graph and the newly created colour. The facts of the new colour will be given by the union of the fact sets of the associated colours, closed under the union of their congruences.

We define two variants of pattern matching: $\text{colMatch}_{g,c}$ takes a pattern, and searches for a match in any of the e-graphs associated to the colours in the sub-DAG rooted at c. The result is a set of matches of the form (c', e, θ), where $c' \triangleright_g c$ is the colour where the match was found, $e \in E_{g,c'}$ is the root e-class of the match, and θ is a substitution sending pattern variables to elements of $E_{g,c'}$. The operation $\text{colMatchIn}_{g,c}$ works the same, but only searches for matching nodes in the given e-class. A typical use for $\text{colMatchIn}_{g,c}$ is to look up the colours where a given fact (Boolean term) is true, as in $\text{colMatchIn}_{g,\bullet}(\top_g, b)$, with $b \in \mathcal{T}$ a concrete term (which is also valid as a pattern).

3.3 Symbolic matching and equality saturation

From the operations above, we can construct a more general pattern matching operation $\text{symbMatch}_{g,c} : \mathcal{T}_V \to \mathcal{P}(\mathcal{P}(C_g) \times C_g \times \mathcal{E} \times (V \rightharpoonup C_g \times \mathcal{E}))$. This operation takes a pattern

and tries to find matches *across different colours* (possibly on diverging $\triangleright$ paths). Its result type is similar to that of colMatch, but instead of tuples (c', e, θ) we get tuples (X, c', e, σ), where $X \subseteq C_g$ is the set of colours in which the different parts of this match are valid, $c' \triangleright_g c$ gives the colour where the matching node was found, and $e \in E_{g,c'}$ is the canonical ID of the matching e-class. Finally, the substitution σ maps the pattern variables to *colour-annotated* canonical e-class IDs (pairs (c_i, e_i) with $e_i \in E_{g,c_i}$).

As an example, consider the pattern $p = f(a, b)$ without any variables, and suppose we have a predicate e-graph g where the term $f(t, u)$ is represented in the root e-graph, the colour c_1 stipulates that a is equivalent to t, and the colour c_2 stipulates that b is equivalent to u. Then, performing $\text{symbMatch}_{g,\bullet}$ with p should yield a match $(\{\bullet, c_1, c_2\}, \bullet, e_f, \emptyset)$, where e_f is the e-class ID of $f(t, u)$ in the root e-graph, indicating that this e-class has a term matching the pattern when the conditions of c_1 and c_2 both hold.

There may be no such colour in the original predicate e-graph, but based on the results of the symbolic match we can let $(g', c') = \text{addColour}_g(\{\bullet, c_1, c_2\})$. Then the newly created colour c' is guaranteed to have a match for p at e-class e_f in g'. For any pattern p, we can implement $\text{symbMatch}_{g,c}(p)$ by recursion on p, using the match operations above.

A predicate e-graph enables a symbolic approach to equality saturation using conditional rewrite rules. Suppose we have a conditional rewrite rule $\phi_1, \ldots, \phi_n \vdash p \to q$, where $\phi_i \in \mathcal{T}_V$ are Boolean term patterns representing preconditions which must hold for the rule to be valid. We can use $\text{colMatch}_{g,\bullet}$ (or $\text{symbMatch}_{g,\bullet}$) to find e-nodes matching p throughout the graph. For each match (c, e, θ), we check if c has a child colour c' which validates the conditions ϕ_i (i.e., such that $\phi_i\theta \in \text{facts}(c')$ for all i). If so, we apply the rewrite to that child e-graph; otherwise, we create a fresh child colour of c where we add the conditions to the facts set and apply the rewrite. In this way, we can apply the rule symbolically without having to verify that the preconditions hold up front. With a database of such rules, we can repeat this until saturation; note that since we perform a syntactic facts-membership check, we avoid unnecessary creation of new colours. Rewrite iterations may also be interleaved with SMT solver reasoning, which can merge colours as saturation proceeds.

4 Conclusion

We propose predicate e-graphs as a work-in-progress design for representing conditional equivalences and symbolic rewriting by way of structuring families of e-graphs under a dynamically checked implication preorder. In the preorder a constituent e-graph may have several direct ancestors; e-graphs are merged and identified whenever a loop occurs.

Our focus has been on articulating the core ideas, invariants, and operations of the model. Work is in progress on a formalisation, and on a prototype implementation, to clarify semantic properties and explore the practical behaviour. Two major aspects under study are the properties of symbolic rewriting and the tradeoffs concerning detection of implications.

References

1 Leonardo de Moura and Nikolaj Bjørner. Z3: an efficient SMT solver. In *Tools and Algorithms for the Construction and Analysis of Systems 2008*, pages 337–340. Springer, Berlin, Heidelberg, March 2008. doi:10.1007/978-3-540-78800-3_24.

2 Eytan Singher and Itzhaky Shachar. Easter egg: Equality reasoning based on e-graphs with multiple assumptions. In Nina Narodytska and Philipp Rümmer, editors, *Proceedings of the 24th Conference on Formal Methods in Computer-Aided Design – FMCAD 2024*, pages 70–83. TU Wien Academic Press, 2024. doi:10.34727/2024/isbn.978-3-85448-065-5_13.

3 Ross Tate, Michael Stepp, Zachary Tatlock, and Sorin Lerner. Equality saturation: a new approach to optimization. In Zhong Shao and Benjamin C. Pierce, editors, *Proceedings of the 36th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2009, Savannah, GA, USA, January 21-23, 2009*, pages 264–276. ACM, 2009. doi:10.1145/1480881.1480915.

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