Transforming Optimization Problems into Disciplined Convex Programming Form

Ramon Fernández Mir[1]✉[0000−0001−7242−5532],
Paul B. Jackson[1]✉[0000−0003−3863−8336], Siddharth Bhat[2][0009−0007−6410−3681],
Andrés Goens[3][0000−0002−0409−1363], and Tobias Grosser[2][0000−0003−3874−6003]

[1] University of Edinburgh, Edinburgh, UK
{ramon.fernandezmir,paul.jackson}@ed.ac.uk
[2] University of Cambridge, Cambridge, UK
sb2743@cam.ac.uk, tobias.grosser@cst.cam.ac.uk
[3] Universiteit van Amsterdam, Amsterdam, The Netherlands
a.goens@uva.nl

Abstract. Disciplined convex programming (DCP) is a popular framework for systematically reducing convex optimization problems to the low-level conic form commonly used by convex solvers. An arbitrary convex problem may not immediately be in a DCP-compliant form, and several manual and error-prone steps are often needed to transform it into an equivalent form that is accepted by DCP frameworks. We automate this process in CvxLean, a convex optimization modeling framework embedded in the Lean theorem prover. While the steps can be described using rewrite rules, there are not clear heuristics for orienting and applying them. Instead, we carry out an efficient breadth-first search for a suitable sequence of steps by making use of the `egg` e-graph-based term rewriting system. When `egg` finds a suitable sequence, we automatically prove it correct in Lean. This procedure is the first generic, proof-producing approach to transform a wide range of optimization problems into DCP-compliant forms. Moreover, it is an important step towards a fully-verified and user-friendly convex programming environment.

1 Introduction

Many problems in engineering, finance, and industry can be phrased as convex optimization problems. Convex optimization generalizes linear programming. A convex optimization problem [5] consists of a convex set (the *feasible set*) defined by constraints on some domain and a convex function (the *objective function*) mapping from the domain into some ordered set, usually the reals. The goal is to find an element of the feasible set that minimizes the objective function's value. The popularity of convex optimization is in great part due to the development of interior-point methods [35], which are able to solve convex problems efficiently.

Disciplined convex programming. A challenge is that interior-point solvers accept a very restricted input format. The original problems, however, can usually be formulated in many equivalent ways and might potentially involve a wide

range of functions and notation. Translating problems into solver-compatible forms such as *conic form* requires specialist knowledge and can be a tedious and error-prone process. A popular approach for automating this translation is *disciplined convex programming* (DCP) [16]. With DCP, input problems have to be phrased in terms of particular functions drawn from an extensible library. The DCP scheme is then able to deduce that input problems are convex from certain properties of these functions (see Sec. 2.1) and to reduce such convex problems to conic form automatically.

Verified problem transformations. There are a number of software tools based on DCP and similar approaches. One example is `CVXPY` [10]. However, these tools generally offer no guarantees that the transforms applied are always mathematically sound and have been programmed correctly. An exception is CvxLean[4] [4], a convex optimization modeling framework written in Lean 4 [22]. The work reported here contributes to CvxLean. With this framework, a user states optimization problems using definitions from Lean's library mathlib [6]. If the problem is in a form acceptable to the DCP procedure (Dcp-*compliant form* or Dcp *form* for short), CvxLean can automatically reduce the problem to conic form and pass it to a solver. Right now, it works with `MOSEK` [3], and it could easily be adapted to support other popular solvers such as `ECOS` [11] or `SDPA` [36]. If the problem is not in DCP form, CvxLean can also help; tactics can be used to manually guide a verified *preDCP* transformation of the problem into DCP form.

Contribution of this paper. The main contribution is an extensible framework for automating preDCP transformations. Doing these manually is difficult since the kinds of transformation steps that might be used are quite varied, the target problem is not always obvious, and even if some target problem is guessed at, it might be tedious to figure out the rewrites to get there. Further, it is challenging to organize these steps and orient equalities as rewrite rules in some simple automated strategy. Instead, we automate the search for a useful sequence of steps by making use of e-graph rewriting as implemented in `egg` [34]. E-graphs efficiently represent sets of optimization problems reachable in an iterative deepening search, and e-graph rewriting easily handles bi-directional rewrite rules. An indication of the flexibility of our approach is that it works for problems that previously required specialized transformations [2,1] (see Sec. 4).

Our running example. Consider the following problem over a single real variable:

$$\begin{aligned} \text{minimize} \quad & x \\ \text{subject to} \quad & 0.001 \leq x \\ & \frac{1}{\sqrt{x}} \leq \exp(x) \end{aligned}$$

[4] `https://github.com/verified-optimization/CvxLean/`

This problem is not in DCP form, as DCP requires that the right-hand side of the second inequality is a concave function, whereas $\exp(x)$ is convex. However, the equivalent problem:

$$\begin{array}{ll} \text{minimize} & x \\ \text{subject to} & 0.001 \leq x \\ & \exp(-x) \leq \sqrt{x} \end{array}$$

is DCP, as $\exp(-x)$ is convex and $\sqrt{x}$ is concave, which allows DCP to recognize the constraint $\exp(-x) \leq \sqrt{x}$ as defining a convex set. Automatically transforming the first problem into the second one is challenging, as the second is not obviously simpler in some sense than the first. One possible sequence of rewrite steps to transform the second constraint is:

$$\begin{aligned} \frac{1}{\sqrt{x}} \leq \exp(x) \quad &\rightsquigarrow \quad 1 \leq \exp(x)\sqrt{x} \quad \rightsquigarrow \quad 1 \leq \sqrt{x}\exp(x) \\ &\rightsquigarrow \quad \frac{1}{\exp(x)} \leq \sqrt{x} \quad \rightsquigarrow \quad \exp(-x) \leq \sqrt{x} \end{aligned}$$

Notice that the third rewrite involves a rewrite rule $a \leq bc \rightsquigarrow a/c \leq b$ (conditional on $c > 0$) which is the reverse of the rule used in the first rewrite.

For automating rewrites like these, sometimes it is possible to define a cost metric which decreases as one applies transformation steps towards some desired end form, and which then can guide the selection and orientation of rewrites. Here, there is no such obvious metric. Moreover, many of the rewrites that will be needed for other problems could still apply, for example, $a \leq b \rightsquigarrow a - b \leq 0$. This situation, when there is an absence of a simple strategy for applying rewrite rules, is exactly when an e-graph rewriting approach is useful.

Outline. In Sec. 2, we give an overview of disciplined convex programming and e-graphs. Sec. 3 explains how we use e-graph rewriting in egg to discover rewrite sequences to DCP-compliant form, and then check these sequences in CvxLean. In Sec. 4, we present some experimental results and discuss benchmarks derived from non-convex optimization problems. In Sec. 5, we discuss some related work; and we conclude with some future work in Sec. 6.

2 Background

In this section, we discuss the relevant background, explaining both the basics of convex optimization and DCP, as well as e-graphs and e-graph rewriting.

2.1 Disciplined convex programming

A *function* $f : \mathbb{R}^n \to \mathbb{R}$ is *convex* if, for any two points $x, y \in \mathbb{R}^n$ and any $t \in [0, 1]$, we have $f((1-t)x + ty) \leq (1-t)f(x) + tf(y)$. We say f is *concave* if

$-f$ is convex, and f is *affine* if it is both convex and concave. An affine function can always be expressed as the sum of a linear function and a constant, i.e., it can be put in form $f(x) = a \cdot x + b$ for some $a \in \mathbb{R}^n$ and $b \in \mathbb{R}$. A *set* $C \subseteq \mathbb{R}^n$ is *convex* if, for any two points $x, y \in C$, the line segment joining x and y also lies in C: $\forall t \in [0,1], (1-t)x + ty \in C$. A set $C \subseteq \mathbb{R}^n$ is *affine* if, for any two distinct points $x, y \in C$, the line through x and y also lies in C: $\forall t \in \mathbb{R}, (1-t)x + ty \in C$. Given an optimization variable $x \in \mathbb{R}^n$, convex functions $f_i : \mathbb{R}^n \to \mathbb{R}$ for $i = 0, \ldots, k$ and affine functions $h_i : \mathbb{R}^n \to \mathbb{R}$ for $i = 1, \ldots, l$, a convex optimization problem in *standard form* is defined as follows:

$$\begin{array}{ll} \text{minimize} & f_0(x) \\ \text{subject to} & f_i(x) \leq 0, \ i = 1, \ldots, k \\ & h_j(x) = 0, \ j = 1, \ldots, l. \end{array}$$

A point $x \in \mathbb{R}^n$ is *feasible* if it satisfies all the constraints. The set of feasible points is always a convex set. A feasible point x is also *optimal* if, for every other feasible point y, we have $f_0(x) \leq f_0(y)$.

Convex optimization solvers typically work with problems in low-level *conic form*. However, most problems that arise in applications are not in conic form. DCP [16] is a framework to address the reduction to conic form. DCP requires that the $f_i(x)$ and $h_j(x)$ defining problems be built from a collection of functions known as *atoms*. The framework has two key components. Firstly, it has an *atom library* holding information on each atom, such as its curvature (whether it is convex, concave, or affine); how it is non-decreasing, non-increasing, or neither in each of its arguments; and, when relevant, its graph implementation. The *graph implementation* of an atom specifies how to replace each occurrence of the atom with an equivalent optimization problem in conic form. Secondly, the framework has a *convexity ruleset* that determines the curvature of combinations of atoms, variables, and constants. For example, given an expression $e := f(a_1, \ldots, a_n)$ where f is a convex atom, e is convex if, for each $i \in \{1, \ldots, n\}$, one of the following is true:

- f is nondecreasing in its i^{th} component & a_i is convex.
- f is nonincreasing in its i^{th} component & a_i is concave.
- a_i is affine.

A problem is in DCP form if it is constructed using atoms from some given atom library and can be easily recognised to be convex by using the convexity rules. For example, with our current library, the constraint $\exp(x)\exp(y) \leq z$ is not DCP-compliant, as $\exp(x)\exp(y)$ is considered to be the product of applications of the $\exp(\_)$ atom to x and y, and the product of two convex functions is not necessarily convex. However, the equivalent $\exp(x+y) \leq z$ is DCP-compliant, as $\exp(x + y)$ is seen as convex by the convexity rules. Conceivably, one could add a new atom for $\exp(x)\exp(y)$, but adding new atoms is significant work, and expressions such as $\exp(x)\exp(y)$ are much more easily handled by transforming them to equivalent expressions that the DCP rules can work on.

Problems in DCP form can always be automatically reduced to conic form, principally by replacing atom occurrences by their graph implementations.

2.2 Equivalence of problems

All the problem transformations considered so far in CvxLean and in our work reported here are equivalence-preserving. We use a constructive notion of equivalence that requires the explicit specification of maps between the untransformed and transformed problems. We can then compose maps from a chain of equivalences to map solutions from solvers back into initial problem domains.

Definition 1 (Equivalence). *Let $P := (f_A, cs_A)$ be a minimization problem defined over a domain A, and with objective function $f : A \to \mathbb{R}$ and constraints $cs : A \to \{\bot, \top\}$, where $cs(x) = \top$ if the point x satisfies all the constraints cs, and $cs(x) = \bot$ otherwise. Similarly, let $Q := (g_B, ds_B)$. We say that P and Q are equivalent if there exist maps $\varphi : A \to B$ and $\psi : B \to A$ such that:*

1. *$\forall x \in A$. x is optimal in $P \Rightarrow \varphi(x)$ is optimal in Q.*
2. *$\forall y \in B$. y is optimal in $Q \Rightarrow \psi(y)$ is optimal in P.*

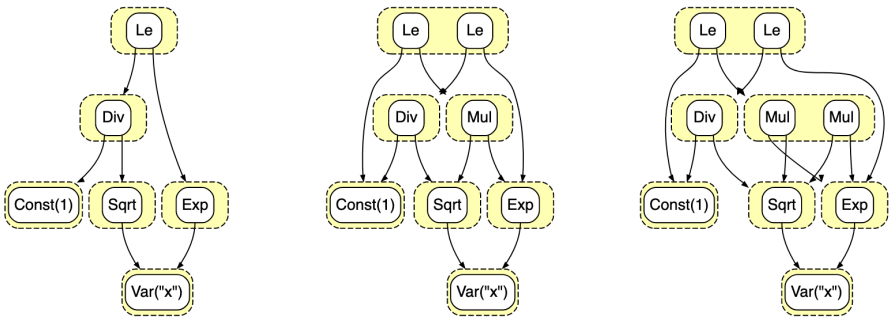
2.3 E-graph-based rewriting systems

Given some language of terms, an *e-graph* is a data structure that compactly represents some set of terms drawn from that language and a congruence relation on that set. E-graphs are used in the standard congruence closure algorithm [23]. In SMT solvers (e.g., [9,8]), they represent congruence relations on ground terms and support instantiation of quantified assumptions. They have also been repurposed to drive compiler optimizations and other applications [29,34].

E-graphs implement equivalence classes of terms that are closed under congruence by making use of a Union-Find data structure [28]. Each equivalence class or *e-class* contains a set of *e-nodes*, with each e-node associated with some term constructor and a list of child e-classes, one for each child of the constructor. If we select some e-node from an e-class c and then recursively select e-nodes from each of its child classes, we can build from the corresponding term constructors one of the terms represented by c.

An e-graph can be built for an initial single term and then iteratively extended by applying rewrite rules to e-nodes. Each application of a rule generates new e-nodes that are then added to the graph, provided there are no e-nodes already there representing the same term sets. E-graph modifications can be instrumented so that, given two terms shown equivalent by some e-graph, a sequence of rewrite rules that justifies their equivalence can be extracted.

In Fig. 1, we show the first steps of building an e-graph for the running example. For simplicity, we focus on the e-graph of the second constraint; in reality, we build one e-graph for the whole problem. Boxes with dashed outlines represent e-classes, and boxes with solid outlines represent e-nodes. From a logical point of view, we always treat the variables we store in e-graphs as Skolem constants, so all terms represented by an e-graph are ground, as required by formal treatments of e-graphs. To start with, in Fig. 1(a), we have an e-graph for the initial constraint. Then, in Fig. 1(b) and Fig. 1(c), the e-graph is augmented with new nodes and classes that represent the right-hand sides of the applied rewrite rules



(a) Initial e-graph representing $1/\sqrt{x} \leq \exp(x)$.

(b) After rewriting with $1/\sqrt{x} \leq \exp(x)$ $\rightsquigarrow 1 \leq \exp(x)\sqrt{x}$.

(c) After rewriting with $1 \leq \exp(x)\sqrt{x}$ $\rightsquigarrow 1 \leq \sqrt{x}\exp(x)$.

Fig. 1: E-graph building steps for the running example.

and capture how these right-hand sides are equivalent to the left-hand sides. One e-graph invariant is that any particular term or set of terms is represented by at most one e-node or e-class. This is already evident in the first graph, where there is only one e-class and e-node for the variable x.

The top e-class in the e-graph in Fig. 1(c) represents just the terms that explicitly appear at the start or that appear after we sequentially apply the two indicated rules, namely $1/\sqrt{x} \leq \exp(x)$, $1 \leq \exp(x)\sqrt{x}$ and $1 \leq \sqrt{x}\exp(x)$. But in general the e-graph captures many further equivalent terms. For example, if we have an e-graph for $f(a, b)$ and we rewrite a to a', we add $f(a', b)$ to the e-graph, and then if we rewrite b to b', the resulting e-graph represents not only $f(a', b')$ but also $f(a, b')$. Further, in the *e-matching* [7] process used to instantiate rewrite rules, the rules' variables get instantiated with e-classes, and the right-hand side from a single rewrite rule instantiation, in general, represents a whole set of new terms, not just one.

The process of iteratively applying a set of rewrites to an initial e-graph for a single term is sometimes referred to as *equality saturation*, as rewriting can continue until no more rules apply. We mostly avoid talking about saturation, as we terminate rewriting as soon as a DCP-compliant version of the initial problem has been found, well before saturation might be achieved.

The `egg` e-graph rewriting system generalises mechanisms provided in other e-graph implementations for annotating e-classes with further data. With an *e-class analysis* [34], each class is annotated with an element from a partial order. The order can be thought of as an information order, with higher elements being, in some sense, more informative. This order must support a join (least upper bound) operation on pairs of data values. When two e-classes are merged because a new equality is added to the e-graph, the join operation is used to compute the data for the merged e-class from that of the two e-classes being merged. It is also necessary to describe a *make* operation for creating the data annotation for

a new singleton e-class when it is added to the e-graph. See Sec. 3.4 for examples of our use of e-class analyses.

3 Automating transforming problems into DCP form

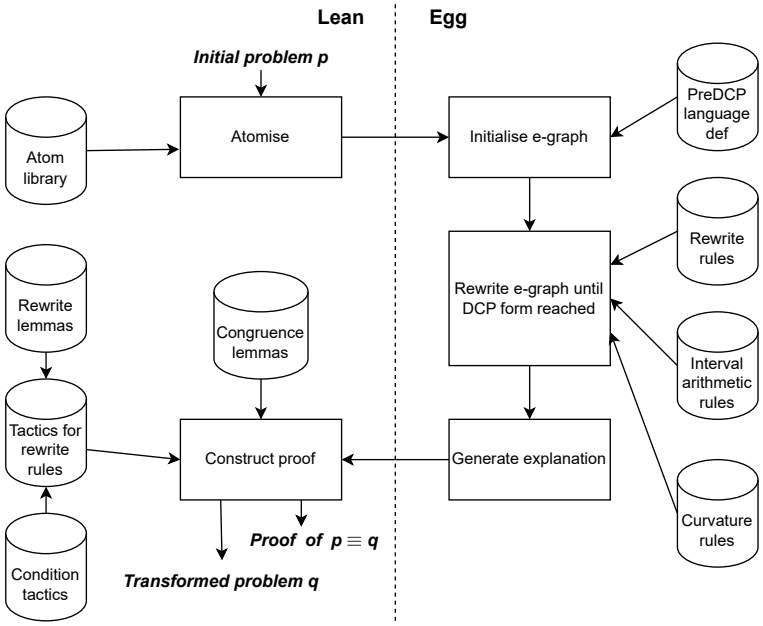


Fig. 2: Architecture of the preDCP tactic.

We automate the transformation using what we call the *preDCP* tactic. In Fig. 2, we show the overall architecture of the tactic and the flow of the input problem through the tactic's stages. As shown in the diagram, the stages are divided between Lean and `egg`. The `egg` system is written as a Rust library, and we customise it using wrapper code also written in Rust. Our `egg` wrapper runs as a subprocess of the Lean process, and Lean communicates with this subprocess via standard input and output. In the following subsections, we explain each of the stages in detail.

3.1 Problem transformation tactics

Top-level tactics for transforming optimization problems in CvxLean are *generative* in that, when applied to a problem p, they generate both some transformed problem p' and a proof of the equivalence $p \equiv p'$. In that respect, they resemble the rewrite conversions originally introduced by Paulson for Cambridge LCF [26].

These top-level tactics work on goals of form $\vdash p \equiv ?q$, where p is a problem to be transformed and $?q$ is a meta-variable, implicitly existentially quantified, which eventually gets instantiated with the final transformed problem after a sequence of transformation steps. When a top-level tactic is applied to a goal of form $\vdash p \equiv ?q$, the tactic generates a subgoal $\vdash p' \equiv ?q$ and a partial proof that, when applied to a later-derived proof of the subgoal, generates a proof of the goal. This partial proof uses transitivity to combine the tactic's proof of $\vdash p \equiv p'$ with the subgoal's proof. Because the form of the subgoal $\vdash p' \equiv ?q$ is the same as that of the goal $\vdash p \equiv ?q$, a sequence of transformations can be realised by simply applying a sequence of these top-level tactics.

One way in CvxLean of running these top-level tactics is by making use of the `equivalence` command:

```
equivalence eqv/q : p := by t1 ; t2 ; ... ; tn
```

Here, `p` is some Lean expression for the initial problem, and `eqv` and `q` are identifiers. When Lean processes this command, the top-level problem transformation tactics `t1` ... `tn` are run in sequence, a reflexivity rule causes $?q$ in the final subgoal $\vdash p_n \equiv ?q$ to be instantiated to p_n and closes this subgoal, the identifier `q` gets bound to the instantiated $?q$, i.e., to p_n, and the identifier `eqv` gets bound to a proof of $\vdash$ `p` $\equiv$ `q`, i.e., `eqv` names this theorem.

The implementation of our problem transformation tactics is similar to that of Lean's conversion-mode tactics [5], though the current Lean distribution does not provide commands such as this `equivalence` command which allow the generative capabilities to be accessed by the user.

3.2 Rephrasing using atoms

Our current preDCP tactic works with a subset of atoms currently supported by CvxLean. This subset allows us to handle problems defined by the grammar:

`prob`	::=	(`expr`, {`constr`}*)	
`constr`	::=	`expr` $=$ `expr`	
	$\mid$	`expr` $\leq$ `expr`	
`expr`	::=	c	a numerical constant, in $\mathbb{F} \setminus \{-\infty, \infty, \text{NaN}\}$
	$\mid$	$\text{var}(s)$	a variable, where s is a string
	$\mid$	u(`expr`)	$u \in \{-(\cdot),\ (\cdot)^{-1},\ \lvert\cdot\rvert,\ \sqrt{\cdot},\ \log,\ \exp\}$
	$\mid$	b(`expr`, `expr`)	$b \in \{+,\ -,\ \times,\ /,\ \hat{},\ \min,\ \max\}$
	$\mid$	cu(`expr`)	$cu \in \{\text{xexp},\ \text{entr}\}$
	$\mid$	cb(`expr`, `expr`)	$cb \in \{\text{qol},\ \text{geo},\ \text{lse},\ \text{norm2}\}$

We distinguish simple unary atoms (u) and binary atoms (b) from composed unary atoms (cu) and composed binary atoms (cb), as the latter can be expressed

[5] `https://leanprover.github.io/theorem_proving_in_lean4/conv.html`

in terms of simple atoms. The curvature analysis used for DCP problem recognition can make use of curvature properties of these composed atoms, whereas the curvature analysis fails on the expansion of these atoms. The atom $\mathrm{lse}(x, y)$ (log-sum-exp) corresponds to $\log(\exp(x) + \exp(y))$, $\mathrm{xexp}(x)$ to $x \exp(x)$, $\mathrm{entr}(x)$ to $-x \log(x)$, $\mathrm{qol}(x, y)$ (quadratic over linear) to x^2/y, $\mathrm{geo}(x, y)$ (geometric mean) to $\sqrt{xy}$, and $\mathrm{norm2}(x, y)$ to $\sqrt{x^2 + y^2}$.

The first step taken by the preDCP tactic is to convert a problem expressed using definitions from Lean's library into a form in this grammar. A similar problem conversion is used by the front-end of the CvxLean tactic that implements the automatic transformation of DCP-compliant problems into conic form (See Sec. 2.1). In that case, the conversion makes choices of atoms to ensure DCP curvature checks are satisfied. With the preDCP tactic, we bypass these checks, and the conversion is driven just by syntactic structure; the whole point of the preDCP tactic is that it takes problems not satisfying these checks and transforms them into equivalent versions that do satisfy the checks.

This conversion distinguishes *simple constraints*, constraints on single domain variables of form $x \leq c$, $x < c$, $c \leq x$, $c < x$, $x = c$ where c is a constant, from the rest of the problem constraints. Simple constraints are used later in the e-graph rewriting process to check the conditions of conditional rewrite rules.

3.3 Initialising e-graphs

The egg system requires declarations of all term constructors, so we provide it with declarations of the constructors in the preDCP language defined above. By drawing on these declarations, our egg wrapper can create an initial e-graph from a problem sent over from Lean.

3.4 Rewriting until DCP form reached

We have egg repeatedly apply rewrite rules until either we find we have reached a problem in DCP form or we reach some limit. Rewrites are applied by egg in iterations. In each iteration egg first looks for all matches of rewrite rules to the current problem e-graph, and then adds the results of the rewrites to the e-graph. After this, at the end of each iteration, we check whether the e-graph contains a problem in DCP form. The limit we currently use is on the size of the e-graph. We set it much higher than is needed to solve all our benchmark problems (see Sec. 4). If we reach this limit, the preDCP tactic fails.

As far as we know, this use of e-graph rewriting to *discover* an expression of a particular form that is equivalent to an initial expression, rather than to find an equivalent expression that *optimizes* some metric, is novel. See Sec. 5 for more on this point.

Rewrite rules. We have to configure egg with the set of rules we want it to try. egg is a generic tool which relies on the user defining a term language and rewrite rule set appropriate to the domain they are interested in. At the time of

writing, we have 68 rules (51 of which are bidirectional). There are essentially three types of rules:

1. Equality rewrites that operate on real-valued terms (terms generated by the `expr` non-terminal in our grammar).
2. If-and-only-if rewrites that operate on propositions (terms generated by the `constr` non-terminal in our grammar).
3. Problem equivalence rewrites that operate on whole problems generated by the root non-terminal `prob` in our grammar. All such rewrites that we currently consider only change the objective function and do not touch the constraints.

We show some rules that are relevant to our running example. Two rules in the first class are:

$$\forall x, y \in \mathbb{R}.\ xy \rightsquigarrow yx, \qquad \forall x \in \mathbb{R}.\ \frac{1}{\exp(x)} \leftrightsquigarrow \exp(-x).$$

In the above and following rewrite rules, we use $\rightsquigarrow$ or $\leftrightsquigarrow$ rather than the appropriate equivalence relation in order to explicitly show the intended rewrite direction or directions. A rule in the second class is:

$$\forall a, b, c \in \mathbb{R}.\ c > 0 \Rightarrow \left(\frac{a}{c} \leq b \leftrightsquigarrow a \leq bc\right)$$

This is an example of a conditional rewrite rule; it only should be applied when the condition $c > 0$ can be inferred.

There are no third-class rules that apply to our example, but they are particularly interesting as they are specific to optimization problems. We have identified two such rules, which involve, respectively, applying a logarithm to the objective function and squaring the objective function:

$$\forall f, cs.\ (\forall x.\ cs(x) \Rightarrow\ f(x) > 0) \Rightarrow (f, cs) \rightsquigarrow (\lambda x.\log(f(x)), cs)$$

$$\forall f, cs.\ (\forall x.\ cs(x) \Rightarrow\ f(x) \geq 0) \Rightarrow (f, cs) \rightsquigarrow \left(\lambda x.(f(x))^2, cs\right)$$

These rules are instances of a more-general rule that applies any strictly-monotone function to the objective function.

E-class analysis for rule conditions. Our rewrite rules currently involve checking the conditions $v > 0$, $v \geq 0$, $v \neq 0$, $v \leq 0$, $v < 0$, $v \leq w$ and $v \in \mathbb{N}$ for variables v, w that occur in the rewrite rule match patterns. To check these conditions we use an e-graph analysis that computes for each e-class and node in the e-graph an interval that contains the possible values of the e-class or node. The lower and upper bound of each interval can each be open or closed, and we allow $\pm\infty$ bounds [19]. Intervals for problem domain variables are inferred from simple constraints (See Sec. 3.2). Rules are provided for computing the interval of each e-graph node for a real-valued *expr* constructor from the intervals for its arguments. The interval for an e-class is computed as the intersection of the intervals of the nodes in the class.

These interval calculations have been sufficient for discharging the side conditions we have come across so far in examples.

E-class analysis for detecting DCP form An optimization problem is in Dcp form when we can check using curvature rules that the objective function is a convex function and each constraint defines a convex set. To detect this, we use an e-class analysis that computes a curvature value for each e-class and node in the e-graph. See Fig. 3 for the partial orders we use for these values.

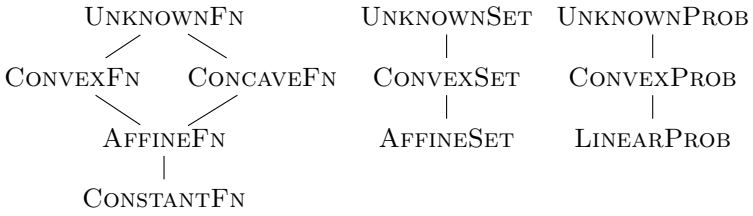


Fig. 3: Hasse diagrams for orderings of curvatures of real-valued (*expr*) terms, proposition-valued (*constr*) constraints, and whole problem (*prob*) terms. Semantically, each label represents a set, and each ordering corresponds to set inclusion on the representations.

Problem domain variables are given AffineFn curvature and constants ConstantFn. For each constructor in the preDcp grammar, we have a curvature rule which is used to compute the curvature of nodes labelled with the constructor from the curvature of the child classes. These rules embody composition rules such as that mentioned at the end of Sec. 2.1. For example, a node for $\exp(x)$ is labelled with ConvexFn if the class for x is labelled with ConvexFn or better [6], and a node for $x \leq y$ is labelled with ConvexSet if the class for x is labelled with ConvexFn or better, and the class for y is labelled with ConvcaveFn or better.

For computing the curvature of a class from that of its nodes, the meet (greatest-lower-bound) operation is used, which corresponds to intersecting the sets that each curvature element represents. A problem is considered to be in Dcp form if the e-class in the graph for the problem is labelled with ConvexProb or better.

Once we know that the e-graph contains a problem term in Dcp form, we need to extract a specific such term. To help us do this, we store along with each curvature label for a node or class a particular term with that curvature, and we augment the curvature computation rules to calculate such terms.

A subtlety occurs when merging two e-classes of an e-graph, one labelled ConvexFn and one labelled ConcaveFn. Using the meet on the order results in the label AffineFn. While we then know that all the terms contained in the two e-classes must mathematically be affine, we do not have at hand any term that can be checked as being affine by the Dcp curvature rules. We need such a term because our eventual goal is to derive an e-graph that contains an

[6] Here, "X or better" means "X or any element below X in the relevant order".

optimization problem that can be recognised to be DCP-compliant by the DCP rules. Our current pragmatic solution is for merge to *not* use the mathematical meet, but rather to simply pick the curvature and corresponding term of one of the classes. This is sound, the curvature labelling of the resulting e-class is mathematically correct, but it is not ideal. We reason that this is acceptable because we expect that simplifications from future applied rewrite rules will eventually result in the further merging of a class labelled with AFFINEFN, in which case we regain the desired invariant that the curvature labelling of a class is the same, no matter in what order the curvatures of the nodes in the class are pairwise combined. We do see examples with our benchmarks (see Sec. 4) where this pragmatic solution is needed and, so far, it has always worked.

3.5 Generating equivalence explanations

Once we have a transformed version of the initial problem in DCP form, we ask `egg` to generate an explanation for why the initial and transformed problems are equivalent [24,12]. This explanation takes the form of a sequence of steps, each a single application of one of the rewrite rules. For each step, information is provided on the rewrite rule name, whether, for bidirectional rules, the rule was applied in a forward or reverse direction, the position in the intermediate problem generated by the previous steps at which the rule was applied, and the instantiated left and right-hand sides of the rule.

3.6 Replaying explanations as proofs in Lean

Using Lean 4's macro system [31], the initial problem of our running example can be expressed in CvxLean in a readable format and bound to a variable `p` with the following text:

```
def p : Minimization ℝ ℝ :=
    optimization (x : ℝ)
        minimize (x)
        subject to
            h1 : 1 / 1000 ≤ x
            h2 : 1 / (sqrt x) ≤ exp x
```

The type `Minimization ℝ ℝ` of problem `p` indicates that the problem has a real-valued domain and the objective function has a real-valued range. After the `optimization` keyword, the `(x : ℝ)` indicates that the problem domain has a single component that is referred to using the variable `x`.

We then can run the preDCP tactic with:

```
equivalence eqv/q : p := by
  pre_dcp

#print q
-- def q : Minimization ℝ ℝ :=
--   optimization (x : ℝ)
```

```
--     minimize x
--     subject to
--       h1 : 1 / 1000 ≤ x
--       h2 : exp (-x) ≤ sqrt x
```

The commented code shows the output of the `#print` command.

Each step of the explanation received from `egg` is justified in Lean using a suitable call of a top-level problem transformation tactic. Internally, this tactic first reduces showing the equivalence of the problems before and after the step to showing the equivalence of just one of the problem components (either the objective function or one of the constraints). We can always do this, as each rewrite step identified using `egg` involves the application of some rewrite rule to a single component. We do not currently have rewrite rules that work with multiple components, such as rewriting one constraint using an equality from another constraint or adding or removing constraints. For example, the first step of the explanation generated by `egg` for transforming our running example problem involves transforming the constraint `h2`. The Lean congruence lemma[7]

```
def rewrite_constraint_2_last (hrw : ∀ x, c1 x → (c2 x ↔ c2' x)) :
    ⟨f, fun x => c1 x ∧ c2 x⟩ ≡ ⟨f, fun x => c1 x ∧ c2' x⟩
```

reduces the before-and-after-step problem equivalence to

```
x: ℝ
h1: 1 / 1000 ≤ x
⊢ 1 / sqrt x ≤ exp x ↔ 1 ≤ sqrt x * exp x
```

The assumption of constraint `h1` here is essential for later discharging a condition of the lemma used to prove this goal. At this point, CvxLean needs to know what rewrite in the Lean library corresponds to the `egg` rewrite rule `div_le_iff`:

$$\forall a, b, c \in \mathbb{R}.\ c > 0 \Rightarrow \left(\frac{a}{c} \leq b \rightsquigarrow a \leq bc\right)$$

Using a custom Lean command, we set up associations between these rewrite rule names on the `egg` side and corresponding lemmas and tactics on the Lean side. A slightly simplified version of the command that associates the `div_le_iff` rule to a tactic is:

```
register_rule_to_tactic "div_le_iff" := apply div_le_iff (by positivity!)
```

Here, the Lean lemma has the same name `div_le_iff`, but this is not always the case. The `positivity!` is a Lean tactic that should be used to discharge the rewrite rule condition. It is a custom extension of Lean's `positivity` tactic which solves arithmetic goals and can support, to some extent, non-linear expressions and transcendental functions. Ideally, an interval arithmetic tactic should be used here, mimicking the logic of the interval arithmetic e-class analysis we use.

[7] We use a `def` rather than `lemma` declaration here for constructive type-theoretic reasons; it enables us to extract from equivalence proofs functions for mapping solutions to initial problem domains.

For now, with the strictly weaker `positivity!` and related arithmetic tactics, we have been able to handle the conditions of all the examples we have considered.

Currently, there is no static checking of this association between `egg` rules and Lean rewrite rules; if there is a mistake, it likely would not manifest itself until it causes the replay in Lean of a `egg`-generated proof to fail. Some checking could be done, or, better, further development could simplify matters by requiring rewrite rules to only be specified in Lean, and having the `egg` versions derived from these Lean versions.

The example `div_le_iff` rule above is applied at the top level of a constraint component of a problem. In general, the needed rule is applied at some interior point of the component, and an extra congruence lemma is applied first to set up a subgoal expressing the required equality at this interior point.

4 Experiments and discussion

We currently have a set of 145 problems that we use for testing and evaluating our preDCP tactic: 114 are unit tests, 4 are derived from exercises for a convex optimization course at Stanford, 10 come from an online DCP quiz, and 17 derive from Geometric Programming (GP) and Quasi-Convex Programming (QCP) problems. GP and QCP are classes of optimization problems that can be transformed into convex optimization problems. Just as there is Disciplined Convex Programming, so there is Disciplined GP [2] and Disciplined QCP [1]. The `CVXPY` tool [10] includes specially designed sets of rules to support both DGP and DQCP. In many cases, our preDCP tactic is able to transform problems in these further classes into DCP form once a CvxLean change-of-variables tactic is first called, without otherwise any special further sets of rules. More specifically, we can solve all GPs after a manual change of variables and many DQCP-compliant problems that are mathematically convex.

We cannot handle general quasiconvex problems as these require extra machinery for solving a series of DCP problems. Regarding GPs, one could hypothetically add a change-of-variables rule to the rule set used by preDCP tactic, so this step, too, would be automated. We have not explored this yet. The challenge is that changing variables is a creative step that hugely increases branching in the search for a DCP-compliant problem and would easily make the search intractable. Intelligent heuristics would need to be designed to carefully select and control the variable changes. Apart from these strong limitations, there are also potential practical limitations with expressions such as $(\sqrt{x_1} + \cdots + \sqrt{x_n})^2$ with $x_i > 0$ (which can be rewritten into a DCP concave expression), where AC rules become an issue for large n. We speculate that these are rare in practice.

In assembling the set of problems that were not unit tests, we were striving to find examples that were representative of problems that come up in practice. These problems heavily guided our selection of rewrite rules and motivated the set of atoms that we currently support.

In Tab. 1, we show some statistics generated by `egg` and CvxLean when run on the most challenging of our benchmark problems. We also include our running

Benchmark	Run-time		Term size		#nodes	#steps	#iters	#rewrites
	egg	total	before	after				
gp4	1439 ms	3170 ms	37	41	22059	31	10	26338
gp5	1340 ms	3265 ms	41	45	24452	38	10	24995
gp8	1083 ms	8193 ms	93	72	19327	79	9	19755
gp9	5799 ms	15463 ms	97	104	41012	123	19	36674
agp3	1129 ms	2819 ms	40	46	19860	29	10	23033
qcp4	344 ms	1711 ms	22	19	10847	17	5	9426
stan3	101 ms	1033 ms	26	31	2572	20	4	2319
stan4	291 ms	1789 ms	34	48	7754	38	5	6458
quiz9	113 ms	614 ms	13	9	1868	12	4	2216
example	16 ms	249 ms	13	15	185	3	2	132

Table 1: Results on selected problems.

Run-time egg – run-time of egg when running the preDcp tactic; *Run-time total* – total run-time of the preDcp tactic; *Term size before / after* – size of the problem term before and after running the preDcp tactic; *#nodes* – size of the final e-graph from which the transformed problem is extracted; *#steps* – number of rewrite steps in the explanation produced by egg; *#iters* – number of iterations of the core egg algorithms needed; *#rewrites* – number of rewrite rule instances used by egg to grow the e-graph.

example on the last line. The experiments were performed on a 2021 Macbook Pro with an Apple M1 Pro and 16GB RAM.

As we can see, the time spent on the Lean side (total - egg) is consistently higher. On average, for all 145 problems, the time spent for each tactic run is around 4.5× that spent in egg; the performance bottleneck is on the Lean side, primarily in proof reconstruction.

Given the number of steps required on these selected problems, the benefits of the automation provided by egg are clear. It would be very tedious for a CvxLean user to write tactic scripts for the problem transformations that apply relevant rewrite rules one-by-one, let alone figure out which rules are needed to reach problems in Dcp-compliant form.

Observe that the size of the resulting problem term (the size of its AST) does not always decrease, which is further evidence that this transformation is not a simplification.

The total number of steps is much higher than the number of iterations because egg explores rewrites on independent parts of the initial problem in parallel. With the basic egg rewriting algorithm, the number of iterations corresponds to the length of the maximum sub-sequence of rewrite rule steps where each rule's application depends on the result of some earlier rule. In practice, further iterations can be needed because egg curtails the growth of the e-graph by periodically disabling for a few iterations rules that get high use.

E-graph rewriting makes tractable the search of the space of terms equivalent to some initial term by an arbitrary set of rewrite rules, a search that, naively, would be utterly intractable.

5 Related work

There are convex optimization modeling frameworks in a number of languages. Even though they require the input problem to comply with the rules of DCP (except for special cases such as DGP or DQCP), they are the main point of reference for our work. The main tool discussed in this paper is `CVXPY` [10], written in Python. It has a large atom library, relevant examples for many applications, and is maintained by an active developer community. Other popular implementations include `CVX` [15] in MATLAB, `CVXR` [13] in R, and `Convex.jl` [30] in Julia. The main difference between these frameworks and CvxLean is that, in CvxLean, problem definitions are mathematically precise, which makes verifying the procedure presented here possible. An advantage that applies in particular for expert users of these tools is that one can confidently extend CvxLean (e.g., add a new transformation step) since any error will result in failure to prove equivalence.

This work focuses on the correctness of the transformations, but there is another related line of research that is worth highlighting that involves verifying the numerical certificates output by the solver. `ValidSDP` [21] is a tool written in Coq where polynomial positivity queries are rephrased as sum-of-squares problems and solved using semidefinite programming. This requires formal reasoning about floating-point errors. There is also a comparable tactic based on sum-of-squares in HOL Light [17]. A different approach is taken by `VSDP` [18], a MATLAB package where rigorous computations of optimality bounds and solution enclosures are performed using interval arithmetic.

The other angle taken by our project is using e-graphs to solve a problem that we have managed to rephrase as a rewriting problem. E-graphs were originally designed for automated theorem proving [23] and are currently used in a crucial way by SMT solvers [7]. In this setting, it is often necessary for an equivalence between two terms established by the e-graph to be justified with a sequence of rewrite rule applications [24,12], usually called an *explanation*. In our work, we use explanations to reconstruct Lean proofs of equivalence between optimization problems expressed in our restricted language. E-graphs have also been used for generic congruence closure algorithms in interactive theorem provers, where the challenge is making them work with the complex underlying languages like Lean's dependent type theory [27] or cubical Agda's homotopy type theory [14].

All of these works use e-graphs for verification through congruence closure, witnessing the equivalence (or congruence) of terms, but not considering any analysis or cost minimization on the e-graphs. In contrast, in our work here, we use both e-class analysis and extraction [34] to find representatives of the equivalence class that are in DCP form. In this sense, our work is more closely related to other uses of equality saturation that come from program optimization [29], where equivalence is necessary, but a particular property of the equivalence class is sought. Frameworks like Herbie [25], Diospyros [32], Tensat [37] or SPORES [33] all use e-graphs and equality saturation to optimize some performance metric of programs (or hardware). In all these cases, the transformation to a minimum cost term – say, identifying a program with optimum performance – is always acceptable. However, to the best of our knowledge, our use case is the first where

inspection of the minimum cost term in the e-graph helps one decide whether or not a transformation is even acceptable.

6 Conclusion and future work

We have introduced a system that is able to transform optimization problems into equivalent Dcp-compliant forms in a manner where all transformation steps are checked in a theorem prover. Using `egg`, we are able to simply list all the rewrite rules that are potentially useful without worrying about their individual effect, thanks to the non-destructive nature of e-graphs. An alternative approach would be to use a tool like `aesop` [20], which we experimented with, fine-tuning the priority of every rewrite rule depending on how likely it is to result in a Dcp-compliant form. In practical terms, however, it is hard to know a priori which rules will be useful, and we have shown that considering all equivalent problems at once works efficiently for several interesting examples.

Next steps include extending the preDcp language to include matrix and vector atoms, which are widely used in `CVXPY`, and binding operators such as summations. Currently, if the preDcp tactic fails, the user gets no feedback. We are considering how instead to show the user why the tactic's best effort still fails to be in Dcp form. Customization for new constructors and new rewrite rules could be simplified by requiring this customization just on the Lean side, and then have Lean take care of suitably configuring the `egg` side. Using `egglog` [38], which unifies Datalog with equality saturation, instead of `egg` could offer enhanced opportunities for checking conditions.

A major project would be porting and extending the `ValidSDP` work in Coq (see Sec. 5) on formally verifying solutions provided by solvers, taking account of floating-point errors. If one also formally checked bounds on the floating-point errors introduced by mapping solver solutions back to initial problem domains, one then would have formal guarantees about the solutions returned by `CvxLean`. Alternatively, one could use the `VSDP` solver (again see Sec. 5), accept its reduced performance compared to state-of-the-art solvers, trust the solution intervals it computes, and use interval arithmetic for mapping back to initial domains. This would be much less work than porting and extending `ValidSDP`, though the degree of assurance in the correctness of solutions would be less. While both these options could be desirable, we argue that there is much value in our focus on ensuring the correctness of problem transformations, even without numerical verification of solutions.

We are keen to have members of the convex optimization community give `CvxLean` a try and are improving robustness and documentation to make this feasible in the coming months. To date, all our most challenging benchmarks are derived from academic courses on optimization. With contact with community members, we hope to see `CvxLean` and our preDcp tactic exercised on real-world industrial examples.

References

1. Agrawal, A., Boyd, S.: Disciplined quasiconvex programming. Optimization Letters **14**, 1643–1657 (2020). https://doi.org/10.1007/s11590-020-01561-8
2. Agrawal, A., Diamond, S., Boyd, S.: Disciplined geometric programming. Optimization Letters **13**, 961–976 (2019). https://doi.org/10.1007/s11590-019-01422-z
3. Andersen, E.D., Andersen, K.D.: The MOSEK interior point optimizer for linear programming: an implementation of the homogeneous algorithm. In: High performance optimization, pp. 197–232. Springer (2000). https://doi.org/10.1007/978-1-4757-3216-0_8
4. Bentkamp, A., Fernández Mir, R., Avigad, J.: Verified reductions for optimization. In: International Conference on Tools and Algorithms for the Construction and Analysis of Systems. pp. 74–92. Springer (2023). https://doi.org/10.1007/978-3-031-30820-8_8
5. Boyd, S.P., Vandenberghe, L.: Convex optimization. Cambridge University Press (2004). https://doi.org/10.1017/CBO9780511804441
6. mathlib Community, T.: The Lean mathematical library. p. 367–381. CPP 2020, Association for Computing Machinery, New York, NY, USA (2020). https://doi.org/10.1145/3372885.3373824
7. De Moura, L., Bjørner, N.: Efficient e-matching for SMT solvers. In: Automated Deduction–CADE-21: 21st International Conference on Automated Deduction Bremen, Germany, July 17-20, 2007 Proceedings 21. pp. 183–198. Springer (2007). https://doi.org/10.1007/978-3-540-73595-3_13
8. De Moura, L., Bjørner, N.: Z3: An efficient SMT solver. In: International conference on Tools and Algorithms for the Construction and Analysis of Systems. pp. 337–340. Springer (2008). https://doi.org/10.1007/978-3-540-78800-3_24
9. Detlefs, D., Nelson, G., Saxe, J.B.: Simplify: a theorem prover for program checking. Journal of the ACM (JACM) **52**(3), 365–473 (2005). https://doi.org/10.1145/1066100.1066102
10. Diamond, S., Boyd, S.: CVXPY: A Python-embedded modeling language for convex optimization. Journal of Machine Learning Research **17**(83), 1–5 (2016)
11. Domahidi, A., Chu, E., Boyd, S.: ECOS: An SOCP solver for embedded systems. In: 2013 European control conference (ECC). pp. 3071–3076. IEEE (2013). https://doi.org/10.23919/ECC.2013.6669541
12. Flatt, O., Coward, S., Willsey, M., Tatlock, Z., Panchekha, P.: Small proofs from congruence closure. In: 2022 Formal Methods in Computer-Aided Design (FMCAD). pp. 75–83. IEEE (2022). https://doi.org/10.34727/2022/isbn.978-3-85448-053-2_13
13. Fu, A., Narasimhan, B., Boyd, S.: CVXR: An R package for disciplined convex optimization. Journal of Statistical Software **94**, 1–34 (2020). https://doi.org/10.18637/jss.v094.i14
14. Gjørup, E.H., Spitters, B.: Congruence closure in cubical type theory. In: Workshop on Homotopy Type Theory/Univalent Foundations. `https://www.cs.au.dk/~spitters/Emil.pdf` (2020)
15. Grant, M., Boyd, S.: CVX: MATLAB software for disciplined convex programming, version 2.1. `http://cvxr.com/cvx` (Mar 2014)
16. Grant, M., Boyd, S., Ye, Y.: Disciplined convex programming. Global optimization: From theory to implementation pp. 155–210 (2006). https://doi.org/10.1007/0-387-30528-9_7

17. Harrison, J.: Verifying nonlinear real formulas via sums of squares. In: Schneider, K., Brandt, J. (eds.) Theorem Proving in Higher Order Logics, 20th International Conference, TPHOLs 2007, Kaiserslautern, Germany, September 10-13, 2007, Proceedings. Lecture Notes in Computer Science, vol. 4732, pp. 102–118. Springer (2007). https://doi.org/10.1007/978-3-540-74591-4_9
18. Härter, V., Jansson, C., Lange, M.: VSDP: A MATLAB toolbox for verified semidefinite-quadratic-linear programming. Optimization Online (2012), `https://optimization-online.org/?p=12271`
19. Hickey, T.J., Ju, Q., van Emden, M.H.: Interval arithmetic: From principles to implementation. J. ACM **48**(5), 1038–1068 (2001). https://doi.org/10.1145/502102.502106
20. Limperg, J., From, A.H.: Aesop: White-box best-first proof search for Lean. In: Proceedings of the 12th ACM SIGPLAN International Conference on Certified Programs and Proofs. pp. 253–266 (2023). https://doi.org/10.1145/3573105.3575671
21. Martin-Dorel, É., Roux, P.: A reflexive tactic for polynomial positivity using numerical solvers and floating-point computations. In: Bertot, Y., Vafeiadis, V. (eds.) Certified Programs and Proofs (CPP) 2017. pp. 90–99. ACM (2017). https://doi.org/10.1145/3018610.3018622
22. de Moura, L., Ullrich, S.: The Lean 4 theorem prover and programming language. In: Automated Deduction–CADE 28: 28th International Conference on Automated Deduction, Virtual Event, July 12–15, 2021, Proceedings 28. pp. 625–635. Springer (2021). https://doi.org/10.1007/978-3-030-79876-5_37
23. Nelson, G., Oppen, D.C.: Fast decision procedures based on congruence closure. Journal of the ACM (JACM) **27**(2), 356–364 (1980). https://doi.org/10.1145/322186.322198
24. Nieuwenhuis, R., Oliveras, A.: Proof-producing congruence closure. In: International Conference on Rewriting Techniques and Applications. pp. 453–468. Springer (2005). https://doi.org/10.1007/978-3-540-32033-3_33
25. Panchekha, P., Sanchez-Stern, A., Wilcox, J.R., Tatlock, Z.: Automatically improving accuracy for floating point expressions. ACM SIGPLAN Notices **50**(6), 1–11 (2015). https://doi.org/10.1145/2813885.2737959
26. Paulson, L.C.: A higher-order implementation of rewriting. Sci. Comput. Program. **3**(2), 119–149 (1983). https://doi.org/10.1016/0167-6423(83)90008-4
27. Selsam, D., de Moura, L.: Congruence closure in intensional type theory. In: Automated Reasoning: 8th International Joint Conference, IJCAR 2016, Coimbra, Portugal, June 27–July 2, 2016, Proceedings 8. pp. 99–115. Springer (2016). https://doi.org/10.1007/978-3-319-40229-1_8
28. Tarjan, R.E.: Efficiency of a good but not linear set union algorithm. Journal of the ACM (JACM) **22**(2), 215–225 (1975). https://doi.org/10.1145/321879.321884
29. Tate, R., Stepp, M., Tatlock, Z., Lerner, S.: Equality saturation: a new approach to optimization. In: Proceedings of the 36th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages. pp. 264–276 (2009). https://doi.org/10.1145/1480881.1480915
30. Udell, M., Mohan, K., Zeng, D., Hong, J., Diamond, S., Boyd, S.: Convex optimization in Julia. In: 2014 First Workshop for High Performance Technical Computing in Dynamic Languages. pp. 18–28. IEEE (2014). https://doi.org/10.1109/HPTCDL.2014.5
31. Ullrich, S., de Moura, L.: Beyond notations: Hygienic macro expansion for theorem proving languages. In: Peltier, N., Sofronie-Stokkermans, V. (eds.) Automated Reasoning. pp. 167–182. Springer International Publishing, Cham (2020). https://doi.org/10.1007/978-3-030-51054-1_10

32. VanHattum, A., Nigam, R., Lee, V.T., Bornholt, J., Sampson, A.: Vectorization for digital signal processors via equality saturation. In: Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems. pp. 874–886 (2021). https://doi.org/10.1145/3445814.3446707
33. Wang, Y.R., Hutchison, S., Suciu, D., Howe, B., Leang, J.: SPORES: sum-product optimization via relational equality saturation for large scale linear algebra. Proc. VLDB Endow. **13**(11), 1919–1932 (2020). https://doi.org/10.14778/3407790.3407799, `http://www.vldb.org/pvldb/vol13/p1919-wang.pdf`
34. Willsey, M., Nandi, C., Wang, Y.R., Flatt, O., Tatlock, Z., Panchekha, P.: egg: Fast and extensible equality saturation. Proceedings of the ACM on Programming Languages **5**(POPL), 1–29 (2021). https://doi.org/10.1145/3434304
35. Wright, M.: The interior-point revolution in optimization: history, recent developments, and lasting consequences. Bulletin of the American mathematical society **42**(1), 39–56 (2005). https://doi.org/10.1090/S0273-0979-04-01040-7
36. Yamashita, M., Fujisawa, K., Kojima, M.: Implementation and evaluation of SDPA 6.0 (semidefinite programming algorithm 6.0). Optimization Methods and Software **18**(4), 491–505 (2003). https://doi.org/10.1080/1055678031000118482
37. Yang, Y., Phothilimthana, P., Wang, Y., Willsey, M., Roy, S., Pienaar, J.: Equality saturation for tensor graph superoptimization. Proceedings of Machine Learning and Systems **3**, 255–268 (2021), `https://proceedings.mlsys.org/paper_files/paper/2021/file/cc427d934a7f6c0663e5923f49eba531-Paper.pdf`
38. Zhang, Y., Wang, Y.R., Flatt, O., Cao, D., Zucker, P., Rosenthal, E., Tatlock, Z., Willsey, M.: Better together: Unifying Datalog and equality saturation. Proceedings of the ACM on Programming Languages **7**(PLDI), 468–492 (2023). https://doi.org/10.1145/3591239