Verified reductions for optimization

Anonymous authors

Abstract. Numerical and symbolic methods for optimization are used extensively in engineering, industry, and finance, and various methods are used to reduce problems of interest to ones that are amenable to solution by such software. We develop a framework for designing and applying such reductions, using the Lean programming language and interactive proof assistant. Formal verification makes the process more reliable, and the availability of an interactive framework and ambient mathematical library provides a robust environment for constructing the reductions and reasoning about them.

Keywords: convex optimization · formal verification · interactive theorem proving · disciplined convex programming

1 Introduction

Optimization problems and constraint satisfaction problems are ubiquitous in engineering, industry, and finance. These include the problem of finding an element of $\mathbb{R}^n$ satisfying a finite set of constraints or determining that the constraints are unsatisfiable; the problem of bounding the value of an objective function over the domain defined by such a set of constraints; and the problem of finding a value of the domain that maximizes (or minimizes) the value of an objective function. Linear programming, revolutionized by Dantzig's introduction of the simplex algorithm in 1947, deals with the case in which the constraints and objective function are linear. The development of interior point methods in the 1980s allows for the efficient solution of problems defined by convex constraints and objective functions, which gives rise to the field of convex programming [10, 34, 41]. Today there are numerous back-end solvers for convex optimization problems, including MOSEK [4], SeDuMi [39], and Gurobi [23]. They employ a variety of methods, each with its own particular strengths and weaknesses. (See [1, Section 1.2] for an overview.)

Using such software requires interpreting the problem one wants to solve in terms of one or more associated optimization problems. Often, this is straightforward; proving the safety of an engineered system might require showing that a certain quantity remains within specified bounds, and an industrial problem might require determining optimal or near-optimal allocation of certain resources. Other applications are less immediate. For example, proving an interesting mathematical theorem may require a lemma that bounds some quantity of interest (e.g. [6]). Once one has formulated the relevant optimization problems, one has to transform them into ones that the available software can solve, and one has to ensure that the conditions under which the software is designed

to work correctly have been met. Mathematical knowledge and domain-specific expertise are often needed to transform a problem to match an efficient convex programming paradigm. A number of modeling packages then provide front ends that apply further transformations so that the resulting problem conforms to a back-end solver's input specification [15, 17, 20, 26, 40]. The transformed problem is then sent to the back-end solver and the solver produces a response, which then has to be reinterpreted in terms of the original problem.

Our goal here is to develop ways of using formal methods to make the passage from an initial mathematical problem to the use of a back-end solver more efficient and reliable. Expressing a mathematical problem in a computational proof assistant provides clarity by endowing claims with a precise semantics, and having a formal library at hand enables users to draw on a body of mathematical facts and reasoning procedures. These make it possible to verify mathematical claims with respect to the primitives and rules of a formal axiomatic foundation, providing strong guarantees as to their correctness. Complete formalization places a high burden on practitioners and often imposes a standard that is higher than users want or need, but verification is not an all-or-nothing affair: users who make use of mathematical and computational results should have the freedom to decide which results they are willing to trust and which ones ought to be formally verified.

With respect to the use of optimization software, the soundness of the software itself is one possible concern. Checking the correctness of a solution to a satisfaction problem is easy in principle: one simply plugs the result into the constraints and checks that they hold. Verifying the correctness of a bounding problem or optimization problem is often almost as easy, in principle, since the results are often underwritten by the existence of suitable *certificates* that are output by the optimization tools. In practice, these tasks are made more difficult by the fact that floating point calculation can introduce numerical errors that bear on the correctness of the solution.

Here, instead, we focus on the task of manipulating a problem and reducing it to a form that a back-end solver can handle. Performing such transformations in a proof assistant offers strong guarantees that the results are correct and have the intended meaning, and it enables users to perform the transformations interactively or partially, and thus introspect and explore the results of individual transformation steps. Moreover, in constructing and reasoning about the transformations, users can take advantage of an ambient mathematical library, including a database of functions and their properties.

In Section 3, we describe the process that CVXPY and other systems use to transform optimization problems expressed in the *disciplined convex program* (DCP) framework to conic form problems that can be sent to solvers like MOSEK [4]. In Section 4, we explain how our implementation in the Lean programming language and proof assistant [30, 31] augments that algorithm so that it at the same time produces a formal proof that the resulting reduction is correct. DCP relies on a library of basic *atoms* that serve as building blocks for reductions, and in Section 5, we explain how our implementation makes it possible to add

new atoms in a verified way. In Section 6, we provide an example of the way that one can further leverage the power of an interactive theorem prover to justify the reduction of a problem that lies outside the DCP framework to one that lies within, using the mathematical library to verify its correctness. In Section 7, we describe our interface between Lean and an external solver, which transforms an exact symbolic representation of a problem into a floating point approximation. Related work is described in Section 8 and conclusions are presented in Section 9.

Our implementation of these methods, CvxLean, is still work in progress. The current prototype is spread between two versions of the Lean; we provide more information in Section 9. Our Lean 4 implementation and an associated Lean 3 library are included as supplementary materials, and will be made public if the paper is accepted.

2 Optimization problems and reductions

The general structure of a minimization problem is expressed in Lean 4 as follows:

```
structure Minimization (D R : Type) :=
(objFun : D → R)
(constraints : D → Prop)
```

Here the data type `D` is the *domain* of the problem and `R` is the data type in which the objective function takes its values. The field `constraints` is a predicate on `D`, which, in Lean, is represented as a function from `D` to propositions: for every value `a` of the domain `D`, the proposition `constraints a`, which says that the constraints hold of `a`, is either true or false. The domain `D` is often $\mathbb{R}^n$ or a space of matrices, but it can also be something more exotic, like a space of functions. The data type `R` is typically the real numbers, but in full generality it can be any type that supports an ordering. A maximization problem is represented as a minimization problem for the negation of the objective function.

A *feasible point* for the minimization problem `p` is an element `point` of `D` satisfying `p.constraints`. Lean's foundational framework allows us to package the data `point` together with the property that it is feasible as follows:

```
structure FeasPoint {D R : Type} [Preorder R] (p : Minimization D R) :=
(point : D)
(feasibility : p.constraints point)
```

A *solution* to the minimization problem `p` is a feasible point, denoted `point`, such that for every feasible point `y` the value of the objective function at `point` is smaller than or equal to the value at `y`.

```
structure Minimization.Solution {D R : Type} [Preorder R]
  (p : Minimization D R) :=
(point : D)
(feasibility : p.constraints point)
(optimality : ∀ y : p.FeasPoint, p.objFun point ≤ p.objFun y.point)
```

Feasibility and bounding problems can also be expressed in these terms. If the objective function is constant (for example, the constant zero function), a solution to the optimization problem is simply a feasible point. And given a domain, an objective function, and constraints, a value `b` is a strict lower bound on the value of the objective function over the domain if and only if the feasibility problem obtained by adding the inequality `objFun x ≤ b` to the constraints has no solution.

We have taken advantage of Lean 4's extensible syntax to implement convenient syntax for defining optimization problems. For example, the following specifies the problem of maximizing $\sqrt{x - y}$ subject to the constraints $y = 2x - 3$ and $x^2 \leq 2$:

```
optimization (x y : ℝ)
  maximize sqrt (x - y)
  subject to
    c1 : y = 2*x - 3
    c2 : x^2 ≤ 2
    c3 : 0 ≤ x - y
```

The third condition, `c3`, is required to ensure that the objective function makes sense and is concave on the domain determined by the constraints. In some frameworks, like CVXPY, this constraint is seen as implicit in the use of the expression `sqrt (x - y)`, but in CvxLean we currently make it explicit.

Problems can also depend on parameters and background conditions. For example, the following defines a family of optimization problems that is parameterized by real numbers a and b, where b is assumed to be greater than zero:

```
def prob (a b : ℝ) (h1 : 0 < b) :=
  optimization (x y : ℝ)
    maximize b * sqrt (x - y)
    subject to
      c1 : y = a*x - 3
      c2 : x^2 ≤ 2
      c3 : 0 ≤ x - y
```

In Section 6, we will consider the covariance estimation for Gaussian variables, which can be expressed as follows:

```
optimization (R : Matrix (Fin n) (Fin n) ℝ)
  maximize (∏ i, gaussianPdf R (y i))
  subject to
    c_pos_def : R.posDef
```

Here `Matrix (Fin n) (Fin n) ℝ` is Lean's representation of the data type of $n \times n$ matrices over the reals, `gaussianPdf` is the Gaussian probability density function defined in Section 6, and the constraint `R.posDef` specifies that `R` ranges over positive definite matrices.

If `p` and `q` are problems, a *reduction* from `p` to `q` is a function mapping any solution to `q` to a solution to `p`. The existence of such a reduction means that to solve `p` it suffices to solve `q`. If `p` is a feasibility problem, it means that the

feasibility of `q` implies the feasibility of `p`, and, conversely, that the infeasibility of `p` implies the infeasibility of `q`. With this framework in place, we can now easily describe what we are after: we are looking for a system that helps a user reduce a problem `p` to a problem `q` that can be solved by an external solver. (For a bounding problem `q`, the goal is instead to find a reduction to `q` from an infeasible problem `p`.) At the same time, we wish to verify the correctness of the reduction, either automatically or with user interaction. This will ensure that the results from the external solver really address the problem that the user is interested in solving.

This notion of a reduction is quite general, and is not restricted to any particular kind of constraint or objective function. In the sections that follow, we explain how the notion can be applied to convex programming.

3 Reduction to conic form

Disciplined Convex Programming (DCP) is a framework for writing constraints and objective functions in such a way that they can automatically be transformed into problems that can be handled by particular back-end solvers, especially ones that are designed to handle convex objective functions and constraints. To start with, the framework guarantees that expressions satisfy the relevant curvature constraints, using the following general facts [1, 21]:

- An expression $f(\mathsf{expr}_1, \ldots, \mathsf{expr}_n)$ is affine if f is an affine function and for each i, expr_i is affine.
- An expression $f(\mathsf{expr}_1, \ldots, \mathsf{expr}_n)$ is convex if f is convex and for each i, one of the following conditions holds:
  - f is increasing in its ith argument and expr_i is convex.
  - f is decreasing in its ith argument and expr_i is concave.
  - expr_i is affine.
- The previous statement holds with "convex" and "concave" switched.

An affine expression is both convex and concave. Some functions f come with side conditions on the range of arguments for which such curvature properties are valid; e.g. $f(x) = \sqrt{x}$ is concave and increasing on the domain $\{x \in \mathbb{R} \mid x \geq 0\}$.

A minimization problem is amenable to the DCP reduction if, following the rules above, its objective function is convex and the expressions occurring in its constraints are concave or convex, depending on the type of constraint. For example, maximizing $\sqrt{x - y}$ requires minimizing $-\sqrt{x - y}$, and the DCP rules tell us that the latter is a convex function of x and y on the domain where $x - y \geq 0$, because $x - y$ is affine, $\sqrt{\cdot}$ is convex and increasing in its argument, and $-(\cdot)$ is affine and decreasing in its argument.

CvxLean registers the properties of atomic functions $f(\bar{a})$ in a library of *atoms*. Each such function f is registered with a formal representation $\mathsf{expr}_f(\bar{a})$, using expressions like `x * log x` or `log (det A)` that can refer to arbitrary functions defined in Lean's library. The atom also registers the relevant properties of f. The curvature of f, curv_f, has one of the values convex, concave, or affine, and

the monotonicity of the function in each of its arguments is tagged as increasing, decreasing, or neither. CvxLean also allows the value auxiliary, which indicates an expression that serves as a fixed parameter in the sense that it is independent of the variables in the optimization problem, although it may depend on other parameters to the problem. Atoms can also come with *background conditions* $\mathsf{bconds}_f(\bar{a})$, which are independent of the domain variables, and *variable conditions* $\mathsf{vconds}_f(\bar{a})$, which constrain the domain on which the properties hold. Notably, the atoms also include *proofs* of properties that are needed to justify the DCP reduction.

By storing additional information with each atom, a DCP framework can use the compositional representation of expressions to represent a problem in a form appropriate to a back-end solver. For example, solvers like MOSEK expect problems to be posed in a certain *conic form* [4]. To that end, CVXPY stores a *graph implementation* for each atomic function f, which is a representation of f as the solution to a conic optimization problem. In greater detail, the graph implementation of an atomic function f is an optimization problem in conic form, given by a list of variables $\bar{v}$, an objective function $\mathsf{obj}_f(\bar{a}, \bar{v})$, and a list of constraints $\mathsf{constr}_f(\bar{a}, \bar{v})$, such that the optimal value of the objective under the constraints is equal to $f(\bar{a})$ for all $\bar{a}$ in the domain of validity. For example, for any $x \geq 0$, the concave function $\sqrt{x}$ can be characterized as the maximum value of the objective function $\mathsf{obj}(x, t) = t$ satisfying the constraint $\mathsf{constr}(x, t)$ given by $t^2 \leq x$. Once again, a notable feature of CvxLean is that that the atom comes equipped with a formal proof of this fact.

Given a well-formed DCP minimization problem, CvxLean must construct a reduced problem in conic form, as well as construct the reduction and prove that it is correct. In this section we describe the construction of the reduced problem, and in the next section we describe the construction of the verified reduction. A more formal description of both of these is given in the appendix.

Let e be a well-formed DCP expression. CvxLean associates to each such expression a tree T_e whose leaves are expressions that are affine with respect to the variables of the optimization problem. For example, the tree in Figure 1 is associated with the expression `-sqrt (x - y)`. Denoting the variables of the optimization problem by $\bar{y}$, we can recursively assign a subexpression $\mathsf{oexpr}_n(\bar{y})$ of e to each node of the tree, such that e is the expression assigned to the root. In the example above, the subexpressions are `x`, `y`, `x - y`, `sqrt (x - y)`, and `-sqrt (x - y)`. To each internal node, we assign a curvature, convex, concave, or affine, subject to the rules of DCP. An expression that is affine can be viewed as either convex or concave. Equalities and inequalities are also atoms; for example, $e_1 \leq e_2$ describes a convex set if and only if e_1 is convex and e_2 is concave. A formalization of the DCP rules allows us to recursively construct formal proofs of these curvature claims, modulo the conditions and assumptions of the problem. We elaborate on this process in the next section.

Now consider a well-formed DCP minimization problem with objective function e_o and constraints $e_1, \ldots, e_n$. We call these expressions the *components* of

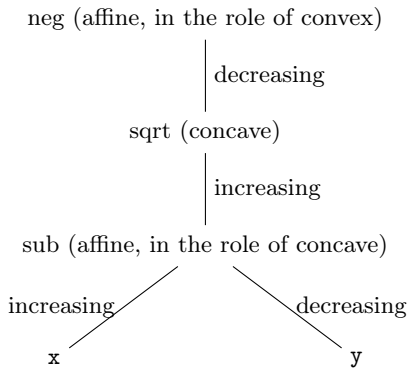

Fig. 1. An expression tree for `-sqrt (x - y)`

the problem. Recall the following example from the previous section, recast as a minimization problem:

```
optimization (x y : ℝ)
  minimize -sqrt (x - y)
  subject to
    c1 : y = 2*x - 3
    c2 : x^2 ≤ 2
    c3 : 0 ≤ x - y
```

Here the components are `-sqrt (x - y)`, `y = 2*x - 3`, `x^2 ≤ 2`, and `0 ≤ x - y`.

First, we assign to each component e an atom tree T_e as described above. If $\bar{y}$ are the variables of the original problem, the variables of the reduced problem are $\bar{y} \cup \bar{z}$, where $\bar{z}$ is a collection of variables consisting of a fresh set of variables for the graph implementation at each internal node of each tree, for those atoms whose graph implementations introduce new variables. To each node n of each atom tree, we assign an expression $\mathsf{rexpr}_n(\bar{y}, \bar{z})$ in the language of the reduced problem representing the expression $\mathsf{oexpr}_n(\bar{y})$ in the original problem. At the leaves, $\mathsf{rexpr}_n(\bar{y}, \bar{z})$ is the same as $\mathsf{oexpr}_n(\bar{y})$. At internal nodes we use the objective function of the corresponding atom's graph implementation, applied to the interpretation of the arguments. The objective of the reduced problem is the expression assigned to the root of T_{e_o}.

As far as the constraints of the reduced problem, recall that each internal node of the original problem corresponds to an atom, which has a graph implementation. The graph implementation, in turn, is given by a list of variables $\bar{v}$, an objective function $\mathsf{obj}_f(\bar{a}, \bar{v})$, and a list of constraints $\mathsf{constr}_f(\bar{a}, \bar{v})$. These constraints, applied to the expressions representing the arguments, are part of the reduced problem. Moreover, the constraints of the original problem, expressed in terms of the reduced problem, become constraints of the reduced problem as well, with one exception. Recall that atoms can impose conditions $\mathsf{vconds}_f(\bar{a})$, which are assumed to be among the constraints of the original problem and to be *implied* by the graph implementation. For example, the condition `0 ≤ x` is

required to characterize $\sqrt{x}$ as the maximum value of a value t satisfying `t^2` $\leq$ `x`, but, conversely, the existence of a `t` satisfying `t^2` $\leq$ `x` implies `0` $\leq$ `x`. So a constraint `0` $\leq$ `x` that is present in the original problem to justify the use of `sqrt x` can be dropped from the reduced problem.

In the example above, there is a tree corresponding to each of the components `-sqrt (x - y)`, `x^2` $\leq$ `2`, `0` $\leq$ `x - y`, and `y = 2*x - 3`. As n ranges over the nodes of these trees, $\mathsf{oexpr}_n(x, y)$ ranges over all the subexpressions of these components, namely, `x`, `y`, `x - y`, `sqrt (x - y)`, `-sqrt (x - y)`, `x^2`, `2`, `x^2` $\leq$ `2`, and so on. The only atoms whose graph implementations introduce extra variables are the square root and the square. Thus, CvxLean introduces the variable `t.0`, corresponding to the expression `sqrt (x - y)`, and the variable `t.1`, corresponding to the expression `x^2`. The values of $\mathsf{rexpr}_n(x, y, t_0, t_1)$ corresponding to some of the expressions above are as follows:

$\mathsf{oexpr}_n(x, y)$	`x - y`	`sqrt (x - y)`	`-sqrt (x - y)`	`x^2`
$\mathsf{rexpr}_n(x, y, t_0, t_1)$	`x - y`	`t.0`	`-t.0`	`t.1`

The constraints `c1` and `c2` of the original problem translate to cone constraints `c1'` and `c2'` on the new variables, the constraint `c3` is implied by the graph representation of `x^2`, and the graph representations of `sqrt (x - y)` and `x^2` become new cone constraints `c4'` and `c5'`. Thus the reduced problem is as follows:

```
optimization (x y t.0 t.1 : ℝ)
  maximize t.0
  subject to
    c1' : zeroCone (2 * x - 3 - y)          -- 2*x - 3 - y = 0
    c2' : posOrthCone (2 - t.1)             -- 2 - t.1 ≥ 0
    c4' : rotatedSoCone 0.5 (x - y) ![t.0]  -- x - y ≥ t.0^2
    c5' : rotatedSoCone t.1 0.5 ![x]        -- t.1 ≥ x^2
```

Here, the meaning of the cone constraints is annotated in the comments. For a description of the relevant conic forms, see the MOSEK modeling cookbook [5].

4 Verifying the reduction

The reduction described in the previous section is essentially the same as the one carried out by CVXPY. The novelty of CvxLean is that it provides a formal justification that the reduction is correct. The goal of this section is to explain how we manage to construct a formal proof of that claim. In fact, given a problem P with an objective function f, CvxLean constructs a new problem Q with an objective g, together with the following additional pieces of data:

- a function φ from the domain of P to the domain of Q such that for any feasible point x of P, $\varphi(x)$ is a feasible point of Q with $g(\varphi(x)) \leq f(x)$, and
- a function ψ from the domain of Q to the domain of P such that for any feasible point y of Q, $\psi(y)$ is a feasible point of P with $f(\psi(y)) \leq g(y)$.

These two conditions guarantee that if y is a solution to Q then $\psi(y)$ is a solution to P, because for any x in the domain of P we have

$$f(\psi(y)) \leq g(y) \leq g(\varphi(x)) \leq f(x).$$

This shows that ψ is a reduction of P to Q, and the argument with P and Q swapped shows that φ is a reduction of Q to P. Moreover, the first condition implies that the value of $g(y)$ for any solution y to Q is less than or equal to the value $f(x)$ for any feasible point x of P, and the second condition implies the version with P and Q swapped. So, the conditions above imply that P has a solution if and only if Q has a solution, and when they do, the minimum values of the objective functions coincide. Below, we will refer to the data (φ, ψ) as a *strong equivalence* between the two problems.

To construct and verify such a strong equivalence between the original problem and the result of applying the transformation described in Section 3, we need to store additional information with each atom. Specifically, for each atomic function $f(\bar{a})$, that atom must provide solutions $\mathsf{sol}_f(\bar{a})$ to the graph implementation variables $\bar{v}$, as well as formal proofs of the following facts:

- The function $f(\bar{a})$ satisfies the graph implementation: for each $\bar{a}$ satisfying the conditions $\mathsf{vconds}_f(\bar{a})$, we have:
  - *solution feasibility:* $\mathsf{sol}_f(\bar{a})$ satisfies the constraints $\mathsf{constr}_f(\bar{a}, \mathsf{sol}_f(\bar{a}))$
  - *solution correctness:* we have $\mathsf{obj}_f(\bar{a}, \mathsf{sol}_f(\bar{a})) = \mathsf{expr}_f(\bar{a})$, where $\mathsf{expr}_f(\bar{a})$ is the expression representing f.
- The function $f(\bar{a})$ is the *optimal* solution to the graph implementation, in the following sense. Write $\bar{a}' \triangle \bar{a}$ to express the assumptions that $a'_i \geq a_i$ for increasing arguments to f, $a'_i \leq a_i$ for decreasing arguments, and a'_i and a_i are syntactically identical for other arguments. If f is convex and $\bar{a}' \triangle \bar{a}$, we require $\mathsf{obj}_f(\bar{a}, \bar{v}) \geq \mathsf{expr}_f(\bar{a}')$ for any $\bar{v}$ such that $\mathsf{constr}_f(\bar{a}, \bar{v})$ holds. If f is concave, we require instead $\mathsf{obj}_f(\bar{a}, \bar{v}) \leq \mathsf{expr}_f(\bar{a}')$ under the same conditions. For affine atoms, we require both.

Finally, as noted in the previous section, the graph implementation generally implies the conditions needed for the reduction. In that case, under the assumptions on $\bar{a}$ and $\bar{a}'$ in the second case above, we also require a proof of $\mathsf{vconds}_f(\bar{a}')$. We refer to this as *condition elimination*.

For a concrete example, consider the atom for the concave function $\sqrt{a}$. In that case, $\mathsf{vconds}(a)$ is the requirement $a \geq 0$, and $\mathsf{expr}(a)$, the Lean representation of the function, is given by Lean's `sqrt` function. The graph implementation adds a new variable v. The only constraint $\mathsf{constr}(a, v)$ is $v^2 \leq a$, and the objective function is $\mathsf{obj}(a, v) = v$. The solution function $\mathsf{sol}(a)$ returns $\sqrt{a}$ when a is nonnegative and an arbitrary value otherwise. The atom for $\sqrt{\cdot}$ stores Lean proofs of all of the following:

- solution feasibility: `∀ a, 0 ≤ a → (sqrt a)^2 ≤ a`
- solution correctness: `∀ a, 0 ≤ a → sqrt a = sqrt a`
- optimality: `∀ v a a', a ≤ a' → v^2 ≤ a → v ≤ sqrt a'`

– condition elimination: ∀ `v a a', a ≤ a' → v^2 ≤ a → 0 ≤ a'`.

More precisely, the atom stores the representation of the graph of the square root function as a cone constraint, and the properties above are expressed in those terms. These properties entail that `sqrt` is concave, but we do not need to prove concavity explicitly.

Let the variables $\bar{y}$ range over the domain of the original problem, P, and let the variables $\bar{y}, \bar{z}$ be the augmented list of variables in the reduced problem, Q. We wish to construct a strong equivalence between P and Q. To that end, we need to define a forward map φ and a reverse map ψ. The definition of the ψ is easy: we simply project each tuple $\bar{y}, \bar{z}$ to $\bar{y}$. The definition of the forward map, φ, is more involved, since we have to map each tuple $\bar{y}$ of values to an expanded tuple $\bar{y}, \bar{z}$. The values of $\bar{y}$ remain unchanged, so the challenge is to define, for each new variable z, an expression $\varphi_z(\bar{y})$ to interpret it.

Recall that for each subexpression $\mathsf{oexpr}_n(\bar{y})$ in the original problem, corresponding to a node n of some component's tree, there is a an expression $\mathsf{rexpr}_n(\bar{y}, \bar{w})$ involving new variables from the reduced problem. Suppose a node n corresponds to an expression $f(u_1, \ldots, u_n)$ in the original problem, and the graph implementation of f introduces new variables $\bar{v}$. For each v_j, we need to devise an interpretation $\varphi_{v_j}(\bar{y})$ of v_j. To start with, sol_f provides a solution to v_j in terms of the arguments $u_1, \ldots, u_n$. For each of these arguments, rexpr provides a representation in terms of the variables $\bar{y}$ and other new variables. Composing these, we get a representation $v_j = e(\bar{y}, w_1, \ldots, w_\ell)$ in terms of the variables $\bar{y}$ of the original problem and new variables $w_1, \ldots, w_\ell$. Recursively, we find interpretations $\varphi_{w_k}(\bar{y})$ of each w_k, and then define $\varphi_{v_j}(\bar{y})$ to be

$$e(\bar{y}, \varphi_{w_1}(\bar{y}), \ldots, \varphi_{w_\ell}(\bar{y})).$$

In words, we read off the interpretation of each new variable of the reduced problem from the intended solution to the graph equation, which may, in turn, require the interpretation of other new variables that were previously introduced.

To show that (φ, ψ) is a strong equivalence, we must show that for any feasible point $\bar{y}$ of the original problem, $\varphi(\bar{y})$ is a feasible point of the reduced problem. This follows from the solution correctness requirement above. We also need to show that if $f(\bar{y})$ is the objective function of the original problem and $g(\bar{y}, \bar{z})$ is the objective function of the reduced problem, $g(\varphi(\bar{y})) \leq f(\bar{y})$. In fact, the solution correctness requirement enables us to prove the stronger property $g(\varphi(\bar{y})) = f(\bar{y})$. Finally, we need to show that for any feasible point $\bar{y}, \bar{z}$ of the reduced problem, the tuple $\bar{y}$ is a feasible point of the original problem and $f(\bar{y}) \leq g(\bar{y}, \bar{z})$. To do that, we recursively use the optimality requirement to show $\mathsf{rexpr}_n(\bar{y}, \bar{z}) \geq \mathsf{oexpr}_n(\bar{y})$ whenever the node n marks a convex expression or an affine expression in the role of a convex expression, and $\mathsf{rexpr}_n(\bar{y}, \bar{z}) \leq \mathsf{oexpr}_n(\bar{y})$ whenever the node n marks a concave expression or an affine expression in the role of a concave expression.

A detailed proof that the maps φ and ψ constructed above form a strong equivalence can be found in the appendix, but it is helpful to work through the

example from Section 3 to get a sense of what the proof means. For this example, the forward map is $\varphi(x, y) = (x, y, \sqrt{x - y}, x^2)$ and the reverse map is $\psi(x, y, t_0, t_1) = (x, y)$. Assuming that (x, y) is a solution to the original problem, the fact that $\varphi(x, y)$ satisfies `c1'` follows from `c1`, the fact that it satisfies `c2'` follows from `c2`, the fact that it satisfies `c4'` and `c5'` follows from the fact that $\sqrt{x - y}$ and x^2 are correct solutions to the graph constraints. In this direction, $g(\varphi(x, y)) = -\sqrt{x - y} = f(x, y)$. In the other direction, assuming that (x, y, t_0, t_1) is a solution to the reduced problem, the fact that (x, y) satisfies the `c1` follows from `c1'`, that fact that it satisfies `c2` follows from `c2'` and `c5'`, and the fact that is satisfies `c3` follows from `c4'`. Here we have $f(\psi(x, y, t_0, t_1)) = -\sqrt{x - y}$ and $g(x, y, t_0, t_1) = -t_0$, and the fact that the former is less than or equal to the latter follows from `c4'`.

5 Adding atoms

One important advantage to using an interactive theorem prover as a basis for solving optimization problems is that it is possible to extend the atom library in a verified way. In a system like CVXPY, one declares a new atom with its graph implementation on the basis of one's background knowledge or a pen-and-paper proof that the graph implementation is correct and that the function described has the relevant properties over the specified domain. In CvxLean, we have implemented syntax with which any user can declare a new atom in Lean and provide formal proofs of these facts. The declaration can be made in any Lean file, and it becomes available in any file that imports that one as a dependency. Lean has a build system and package manager that handles dependencies on external repositories, allowing a community of users to share such mathematical and computational content.

For example, in CvxLean, the declaration of the atom for the logarithm looks as follows:

```
declare_atom log [concave] (x : ℝ)+ : log x :=
  conditions (cond : 0 < x)
  implementationVars (t : ℝ)
  implementationObjective t
  implementationConstraints (c_exp : expCone t 1 x)
  solution (t := log x)
  solutionEqualsAtom by ...
  feasibility (c_exp : by ...)
  optimality by ...
  conditionElimination (cond : by ...)
```

The ellipses indicate places that are filled by formal proofs. Proof assistants like Lean allow users to write such proofs interactively in an environment that displays proof obligations, the local context, and error messages, all while the user types. For example, placing the cursor at the beginning of the optimality block displays the following goal:

```
x t : Real
```

```
c_exp : expCone t 1 x
⊢ ∀ (y : Real), x ≤ y → t ≤ log y
```

In words, given real values x and t and the relevant constraint in terms of the exponential cone, we need to prove that for every $y \geq x$, we have $t \leq \log(y)$.

For the example we present in the next section, we had to implement the *log-determinant* atom [10, Example 9.5], whose arguments consist of a natural number n and a matrix $A \in \mathbb{R}^n \times \mathbb{R}^n$. This function is represented in Lean by the atom expression $\texttt{expr}_{\textit{log-det}} = \texttt{log (det A)}$, where the parameter n is implicit in the type of `A`. The curvature is specified to be concave, the monotonicity in n is auxiliary because we do not support the occurrence of optimization variables in this argument, and the monotonicity in A is neither because the value of $\log(\det A)$ is neither guaranteed to increase nor guaranteed to decrease as A increases. (The relevant order here on matrices is elementwise comparison.) The atom has the atom condition that A is positive definite because only under this condition our reduction of the atom will work correctly. Following CVXPY, we used the following graph implementation:

$$
\begin{array}{ll}
\text{maximize} & \displaystyle\sum_i t_i \\
\text{over} & t \in \mathbb{R}^n,\ Y \in \mathbb{R}^n \times \mathbb{R}^n \\
\text{subject to} & (t, 1, y) \in \text{expcone} \\
& \begin{pmatrix} D & Z \\ Z^T & A \end{pmatrix} \text{ positive semidefinite}
\end{array}
$$

where y is the diagonal of Y; Z is obtained from Y by setting all entries below the diagonal to 0; and D is obtained from Y by setting all entries off the diagonal to 0. Here, saying that the tuple $(t, 1, y)$ is in the exponential cone means that $e^{y_i} \geq t_i$ for each i. Our implementation in CvxLean required proving that this graph implementation is correct. To do so, we formalized an argument in the MOSEK documentation.[1] This, in turn, required proving properties of the Schur complement, triangular matrices, Gram-Schmidt orthogonalization, and LDL factorization. Moreover, the argument uses the subadditivity of the determinant function, for which we followed an argument by Andreas Thom on MathOverflow.[2] Our formalization, putting all of these building blocks together, appears in the file `log_det.lean` in the `optlib` repository.

6 User-defined reductions

An even more important advantage of using an interactive proof assistant as a framework for convex optimization is that, with enough work, users can carry out *any* reduction that can be expressed and justified in precise mathematical terms.

[1] https://docs.mosek.com/modeling-cookbook/sdo.html#log-determinant

[2] https://mathoverflow.net/questions/65424/determinant-of-sum-of-positive-definite-matrices/65430#65430

As a simple example, DCP cannot handle an expression of the form $exp(x)exp(y)$ in a problem, requiring us instead to write it as $exp(x + y)$. But in CvxLean, we have the freedom to express the problem in the first form if we prefer to and then verify that the trivial reduction is justified:

```
reduction red/prob :
  optimization (x y : ℝ)
    maximize x + y
    subject to
      h : (exp x) * (exp y) ≤ 10 := by
  conv_constr => rw [←Real.exp_add]
```

Here the expression `rw [←Real.exp_add]` supplies the short formal proof that $exp(x + y)$ can be replaced by $exp(x) \cdot exp(y)$.

Of course, this functionality becomes more important as the reductions become more involved. As a more substantial example, we have implemented a reduction needed to solve the the covariance estimation problem for Gaussian variables [10, pp. 355]. In this problem, we are given N samples $y_1, \dots y_N \in \mathbb{R}^n$ drawn from a Gaussian distribution with zero mean and unknown covariance matrix R. We assume that the Gaussian distribution is nondegenerate, so R is positive definite and the distribution has density function

$$p_R(y) = (2\pi)^{-n/2} \det(R)^{-1/2} \exp(-y^T R^{-1} y/2).$$

We want to estimate the covariance matrix R using maximum likelihood estimation, i.e., we want to find the covariance matrix that maximizes the likelihood of observing $y_1, \dots y_N$. The maximum likelihood estimate for R is the solution to the following problem:

$$\text{maximize} \quad \prod_{k=1}^{N} p_R(y_k) \quad \text{over} \quad R \quad \text{subject to} \quad R \text{ positive definite.}$$

As stated, this problem has a simple analytic solution, namely, the sample covariance of $y_1, \dots, y_n$, but the problem becomes more interesting when one adds additional constraints, for example, upper and lower matrix bounds on R, or constraints on the condition number of R (see [10]). We can easily reduce the problem to maximizing the logarithm of the objective function above, but that is not a concave function of R. It is, however, a concave function of $S = R^{-1}$, and common constraints on R translate to convex constraints on S. We can therefore reduce the problem above to the following:

$$\text{maximize} \quad \log(\det(S)) - \sum_{k=1}^{N} y_k^T S y_k \quad \text{over} \quad S \quad \text{subject to} \quad S \text{ positive definite,}$$

possibly with additional constraints on S. We express the sum using the sample covariance $Y = \frac{1}{N} \sum_{k=1}^{N} y_k y_k^T$ and the trace operator:

$$\begin{aligned} &\text{maximize} \quad \log(\det(S)) - N \cdot \operatorname{tr}(Y S^T) \quad \text{over} \quad S \\ &\text{subject to} \quad S \text{ positive definite} \end{aligned}$$

The problem can then be solved using disciplined convex programming. The constraint that S is positive definite is eliminated while applying the graph implementation of $\log(\det(S))$.

We formalize the relevant facts in the file `covariance-estimation.lean` in `optlib`, and we use them to justify the reduction. We include an example with an additional sparsity constraint on R in `CvxLean/Examples`.

7 Connecting Lean to a Conic Optimization Solver

Once a problem has been reduced to conic form, it can be sent to an external back-end solver. At this point, we must pass from the realm of precise symbolic representations and formal mathematical objects to the realm of numeric computation with floating point representations. We traverse our symbolic expressions, replacing functions on the reals from Lean's mathematical library with corresponding numeric functions on floats, for example associating the floating point exponential function `Float.exp` to the real exponential function `Real.exp`. Our implementation makes it easy to declare such associations with the following syntax: `addRealToFloat : Real.exp := Float.exp`.

This is one area where more verification is possible. We could use verified libraries for floating point arithmetic [2, 9, 19, 42], we could use dual certificates to verify the results of the external solver, and we could carry out formal sensitivity analysis to manage and bound errors. For now, however, we have set these concerns aside. Our current implementation is only designed to verify correctness up to the point where the problem is sent to the back-end solver, and to facilitate the last step, albeit in an unverified way.

We have implemented a `solve` command in CvxLean which takes a an optimization problem `prob` in DCP form and carries out the following steps:

1. It applies the `dcp` procedure to obtain a reduced problem, `prob.reduced`, and a reduction `red : Solution prob.reduced -> Solution prob`.
2. It carries out the translation to floats, traversing each expression and applying the registered translations.
3. It extracts the numerical data from the problem. At this point, we have scalars, arrays and matrices associated to every type of constraint.
4. It writes the problem to an external file in the conic benchmark format.[3]
5. It calls MOSEK and receives a status code in return, together with a solution, if MOSEK succeeds in finding one. The problem status is added to the environment and if it is infeasible or ill-posed, we stop.
6. Otherwise, the `solve` command interprets the solution so that it matches the shape of the variables of `prob.reduced`. It also expresses these values as Lean reals, resulting in an approximate solution `p` to `prob.reduced`. It declares a corresponding `Solution` to `prob.reduced`, using a placeholder for the proofs of feasibility and optimality (since we simply trust the solver here).

[3] https://docs.mosek.com/latest/rmosek/cbf-format.html

7. It then uses the reduction from `prob` to `prod.reduced`, again reinterpreted in terms of floats, to compute an approximate solution to `prob`.

Finally, the results are added to the Lean environment. In the following example, the command `solve so1` results in the creation of new Lean objects `so1.reduced`, `so1.status`, `so1.value`, and `so1.solution`. The first of these represents the conic-form problem that is sent to the back-end solver, while the remaining three comprise the resulting solution.

```
noncomputable def so1 :=
  optimization (x y : ℝ)
    maximize sqrt (x - y)
    subject to
      c1 : y = 2 * x - 3
      c2 : x ^ 2 ≤ 2
      c3 : 0 ≤ x - y

solve so1
#print so1.reduced  -- shows the reduced problem
#eval so1.status   -- "PRIMAL_AND_DUAL_FEASIBLE"
#eval so1.value    -- 2.101003
#eval so1.solution -- (-1.414214, -5.828427)
```

8 Related work

Our work builds on decades of research on convex optimization, which is too broad to survey here [10, 34, 37, 41]. Our work builds most directly on the CVX family and disciplined convex programming [15, 17, 20, 21, 40]. Other popular packages include Yalmip [26].

Formal methods have been used to solve bounding problems [18, 36], constraint satisfaction problems [16], and optimization problems [25]. This literature is also too broad to survey here, but [14] surveys some of the methods that are used in connection with the verification of cyber-physical systems. Proof assistants in particular have been used to verify bounds in various ways. Some approaches use certificates from numerical packages; Harrison [24] uses certificates from semidefinite programming in HOL Light, and Magron et al. [27] and Martin-Dorel and Roux [28] use similar certificates in Coq. Solovyev and Hales use a combination of symbolic and numeric methods in HOL Light [38]. Other approaches have focused on verifying symbolic and numeric algorithms instead. For example, Muñoz, Narkawicz, and Dutle [32] verify a decision procedure for univariate real arithmetic in PVS and Cordwell, Tan, and Platzer [13] verify another one in Isabelle. Narkawicz and Muñoz [33] have devised a verified numeric algorithm to find bounds and global optima. Cohen et al. [11, 12] have developed a framework for verifying optimization algorithms using the ANSI/ISO C Specification Language (ACSL) [7].

Although the notion of a convex set has been formalized in a number of theorem provers, we do not know of any full development of convex analysis.

The Isabelle [35] HOL-Analysis library[4] includes properties of convex sets and functions, including Carathéodory's theorem on convex hulls, Radon's theorem, and Helly's theorem, as well as properties of convex sets and functions on normed spaces and Euclidean spaces. A theory of lower semicontinuous functions by Grechuk [22] in the Archive of Formal Proofs [8] includes properties of convex functions. Lean's mathlib includes a number of fundamental results,[5] including a formalization of the Riesz extension theorem by Kudryashov and Dupuis and a formalization of Jensen's inequality by Kudryashov. Allamigeon and Katz have formalized a theory of convex polyhedra in Coq with an eye towards applications to linear optimization [3]. We do not know of any project that has formalized the notion of a reduction between optimization problems.

9 Conclusions

We have argued that formal methods can bring additional reliability and interactive computational support to the practice of convex optimization. The success of our prototype shows that it is possible to carry out and verify reductions using a synergistic combination of automation and user interaction.

The implementation of CvxLean is currently spread between two versions of Lean [30, 31]. Lean 3 has a formal library, mathlib [29], which comprises close to a million lines of code and covers substantial portions of algebra, linear algebra, topology, measure theory, and analysis. Lean 4 is a performant programming language as well as a proof assistant, but its language is not backward compatible with that of Lean 3. All of the substantial programming tasks described here have been carried out in Lean 4, but we rely on a binary translation of the Lean 3 library and some additional results proved there. This arrangement is not ideal, but a source-level port of the Lean 3 library is already underway, and we expect to move the development entirely to Lean 4 in the near future.

There is still a lot to do. We have implemented and verified all the atoms needed for the examples presented in this paper, but these are still only a fraction of the atoms that are found in CVXPY. The DCP transformation currently leaves any side conditions that it cannot prove for the user to fill in, and special-purpose *tactics*, i.e. small-scale automation, could help dispel proof obligations like monotonicity. Textbooks often provide standard methods and tricks for carrying out reductions (e.g. [10, Section 4.1.3]), and these should also be supported by tactics in CvxLean. Our project, as well as Lean's library, would benefit from more formal definitions and theorems in convex analysis and optimization. We need to implement more efficient means of extracting numeric values for the backend solver, and it would be nice to verify more of the numeric computations and claims. Finally, and most importantly, we need to work out more examples like the ones presented here to ensure that the system is robust and flexible enough to join the ranks of conventional optimization systems like CVXPY.

[4] https://isabelle.in.tum.de/dist/library/HOL/HOL-Analysis/

[5] https://github.com/leanprover-community/mathlib/tree/master/src/analysis/convex

References

1. Agrawal, A., Verschueren, R., Diamond, S., Boyd, S.: A rewriting system for convex optimization problems. J. Control and Decision **5**(1), 42–60 (2018)
2. Akbarpour, B., Abdel-Hamid, A.T., Tahar, S., Harrison, J.: Verifying a synthesized implementation of IEEE-754 floating-point exponential function using HOL. Comput. J. **53**(4), 465–488 (2010). https://doi.org/10.1093/comjnl/bxp023
3. Allamigeon, X., Katz, R.D.: A formalization of convex polyhedra based on the simplex method. J. Autom. Reason. **63**(2), 323–345 (2019)
4. ApS, M.: Introducing the MOSEK Optimization Suite (2022), https://docs.mosek.com/latest/intro
5. ApS, M.: MOSEK Modeling Cookbook (2022), https://docs.mosek.com/modeling-cookbook
6. Bachoc, C., Vallentin, F.: New upper bounds for kissing numbers from semidefinite programming. J. Amer. Math. Soc. **21**(3), 909–924 (2008). https://doi.org/10.1090/S0894-0347-07-00589-9
7. Baudin, P., Cuoq, P., Filliâtre, J.C., Marché, C., Monate, B., Moy, Y., Prevosto, V.: Acsl: Ansi/iso c specification language (2020), https://frama-c.com/html/acsl.html, version 1.17
8. Blanchette, J.C., Haslbeck, M.W., Matichuk, D., Nipkow, T.: Mining the archive of formal proofs. In: Kerber, M., Carette, J., Kaliszyk, C., Rabe, F., Sorge, V. (eds.) Intelligent Computer Mathematics (CICM 2015). LNCS, vol. 9150, pp. 3–17. Springer (2015)
9. Boldo, S., Filliâtre, J.: Formal verification of floating-point programs. In: 18th IEEE Symposium on Computer Arithmetic (ARITH-18 2007), 25-27 June 2007, Montpellier, France. pp. 187–194. IEEE Computer Society (2007). https://doi.org/10.1109/ARITH.2007.20
10. Boyd, S.P., Vandenberghe, L.: Convex Optimization. Cambridge University Press (2014), https://web.stanford.edu/%7Eboyd/cvxbook/
11. Cohen, R., Davy, G., Feron, E., Garoche, P.L.: Formal verification for embedded implementation of convex optimization algorithms. IFAC-PapersOnLine **50**(1), 5867–5874 (2017), 20th IFAC World Congress
12. Cohen, R., Feron, E., Garoche, P.: Verification and validation of convex optimization algorithms for model predictive control. Journal of Aerospace Information Systems **17**(5), 257–270 (3 2020)
13. Cordwell, K., Tan, Y.K., Platzer, A.: A verified decision procedure for univariate real arithmetic with the BKR algorithm. In: Cohen, L., Kaliszyk, C. (eds.) Interactive Theorem Proving (ITP 2021). LIPIcs, vol. 193, pp. 14:1–14:20. Schloss Dagstuhl - Leibniz-Zentrum für Informatik (2021)
14. Deshmukh, J.V., Sankaranarayanan, S.: Formal techniques for verification and testing of cyber-physical systems. In: Al Faruque, M.A., Canedo, A. (eds.) Design Automation of Cyber-Physical Systems. pp. 69–105. Springer, Cham (2019)
15. Diamond, S., Boyd, S.: CVXPY: A Python-embedded modeling language for convex optimization. J. Machine Learning Research **17**(83), 1–5 (2016)
16. Fränzle, M., Herde, C., Teige, T., Ratschan, S., Schubert, T.: Efficient solving of large non-linear arithmetic constraint systems with complex Boolean structure. J. Satisf. Boolean Model. Comput. **1**(3-4), 209–236 (2007). https://doi.org/10.3233/sat190012
17. Fu, A., Narasimhan, B., Boyd, S.: CVXR: An R package for disciplined convex optimization. Journal of Statistical Software **94**(14), 1–34 (2020)

18. Gao, S., Avigad, J., Clarke, E.M.: δ-complete decision procedures for satisfiability over the reals. In: Gramlich, B., Miller, D., Sattler, U. (eds.) Automated Reasoning (IJCAR 2012). LNCS, vol. 7364, pp. 286–300. Springer (2012)
19. Goodloe, A., Muñoz, C.A., Kirchner, F., Correnson, L.: Verification of numerical programs: From real numbers to floating point numbers. In: Brat, G., Rungta, N., Venet, A. (eds.) NASA Formal Methods, 5th International Symposium, NFM 2013, Moffett Field, CA, USA, May 14-16, 2013. Proceedings. Lecture Notes in Computer Science, vol. 7871, pp. 441–446. Springer (2013). https://doi.org/10.1007/978-3-642-38088-4_31
20. Grant, M., Boyd, S.: CVX: Matlab software for disciplined convex programming, version 2.1. http://cvxr.com/cvx (Mar 2014)
21. Grant, M., Boyd, S., Ye, Y.: Disciplined convex programming. In: Global optimization, pp. 155–210. Springer (2006)
22. Grechuk, B.: Lower semicontinuous functions. Archive of Formal Proofs (Jan 2011), https://isa-afp.org/entries/Lower_Semicontinuous.html, Formal proof development
23. Gurobi Optimization, LLC: Gurobi Optimizer Reference Manual (2022), https://www.gurobi.com
24. Harrison, J.: Verifying nonlinear real formulas via sums of squares. In: Schneider, K., Brandt, J. (eds.) Theorem Proving in Higher Order Logics (TPHOLs 2007). LNCS, vol. 4732, pp. 102–118. Springer (2007)
25. Kong, S., Solar-Lezama, A., Gao, S.: Delta-decision procedures for exists-forall problems over the reals. In: Chockler, H., Weissenbacher, G. (eds.) Computer Aided Verification (CAV 2018, Part II). LNCS, vol. 10982, pp. 219–235. Springer (2018)
26. Löfberg, J.: Yalmip : A toolbox for modeling and optimization in matlab. In: Computer Aided Control System Design (CACSD 2004). pp. 284–289 (2004)
27. Magron, V., Allamigeon, X., Gaubert, S., Werner, B.: Formal proofs for nonlinear optimization. J. Formaliz. Reason. **8**(1), 1–24 (2015). https://doi.org/10.6092/issn.1972-5787/4319
28. 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
29. Mathlib Community: The lean mathematical library. In: Blanchette, J., Hritcu, C. (eds.) Certified Programs and Proofs (CPP 2020). pp. 367–381. ACM (2020)
30. de Moura, L., Ullrich, S.: The lean 4 theorem prover and programming language. In: Platzer, A., Sutcliffe, G. (eds.) Automated Deduction (CADE) 2021. pp. 625–635. Springer (2021). https://doi.org/10.1007/978-3-030-79876-5_37
31. de Moura, L.M., Kong, S., Avigad, J., van Doorn, F., von Raumer, J.: The Lean theorem prover (system description). In: Felty, A.P., Middeldorp, A. (eds.) Automated Deduction (CADE-25). LNCS, vol. 9195, pp. 378–388. Springer (2015)
32. Muñoz, C.A., Narkawicz, A.J., Dutle, A.: A decision procedure for univariate polynomial systems based on root counting and interval subdivision. J. Formaliz. Reason. **11**(1), 19–41 (2018). https://doi.org/10.6092/issn.1972-5787/8212
33. Narkawicz, A., Muñoz, C.A.: A formally verified generic branching algorithm for global optimization. In: Cohen, E., Rybalchenko, A. (eds.) Verified Software: Theories, Tools, Experiments (VSTTE 2013). LNCS, vol. 8164, pp. 326–343. Springer (2013)
34. Nesterov, Y.: Lectures on convex optimization. Springer, Cham (2018). https://doi.org/10.1007/978-3-319-91578-4, second edition

35. Nipkow, T., Paulson, L.C., Wenzel, M.: Isabelle/HOL - A Proof Assistant for Higher-Order Logic, LNCS, vol. 2283. Springer (2002)
36. Ratschan, S., She, Z.: Safety verification of hybrid systems by constraint propagation-based abstraction refinement. ACM Trans. Embed. Comput. Syst. **6**(1), 8 (2007). https://doi.org/10.1145/1210268.1210276
37. Rockafellar, R.T.: Convex analysis. Princeton University Press, Princeton, N.J. (1970)
38. Solovyev, A., Hales, T.C.: Formal verification of nonlinear inequalities with Taylor interval approximations. In: Brat, G., Rungta, N., Venet, A. (eds.) NASA Formal Methods (NFM 2013). LNCS, vol. 7871, pp. 383–397. Springer (2013)
39. Sturm, J.F.: Using sedumi 1.02, a matlab toolbox for optimization over symmetric cones. Optimization methods and software **11**(1-4), 625–653 (1999)
40. Udell, M., Mohan, K., Zeng, D., Hong, J., Diamond, S., Boyd, S.: Convex optimization in Julia. SC14 Workshop on High Performance Technical Computing in Dynamic Languages (2014)
41. Vishnoi, N.: Algorithms for Convex Optimization. Cambridge University Press (2021)
42. Yu, L.: A formal model of IEEE floating point arithmetic. Arch. Formal Proofs **2013** (2013), https://www.isa-afp.org/entries/IEEE_Floating_Point.shtml

Appendix

In this appendix we provide a more formal description of the proof-producing DCP reduction algorithm described in Sections 3 and 4.

Recall that we use the term *component* to mean either the objective function or one of the constraints of a problem. For each component, we construct a tree where each internal node corresponds to an atom. We assign an identifier n to each node of the tree in such a way that the identifiers of the children of node n are of the form $n.i$, and we denote the atom at node n by $A(n)$. Each node has as many branches as the atom has arguments. The leaves of the tree are expressions that are affine w.r.t. the optimization variables.

Each node n represents an expression $\mathsf{oexpr}_n(\bar{y})$ in the variables $\bar{y}$ of the DCP problem, defined as follows. For each leaf node, let $\mathsf{oexpr}_n(\bar{y})$ be the affine expression associated with that node. For each inner node, let $\mathsf{oexpr}_n(\bar{y}) = \mathsf{expr}_{A(n)}(i \mapsto \mathsf{oexpr}_{n.i}(\bar{y}))$ where $\mathsf{expr}_{A(n)}$ is the formal representation of the function associated with the atom. Here and elsewhere, we write $i \mapsto f(i)$ to denote the tuple whose ith entry is $f(i)$. The tree for each component is constructed so that the component is represented by the root, which we denote ε.

With each atom A, we store the monotonicity $\mathsf{mono}_{A,i}$ of each argument i. They determine which atoms can be used in the arguments of another atom:

- If $\mathsf{mono}_{A(n),i} = \mathsf{increasing}$, then $\mathsf{role}_{n.i} = \mathsf{role}_n$.
- If $\mathsf{mono}_{A(n),i} = \mathsf{decreasing}$, then $\mathsf{role}_{n.i} = -\mathsf{role}_n$.
- If $\mathsf{mono}_{A(n),i} = \mathsf{neither}$, then $\mathsf{role}_{n.i} = \mathsf{affine}$.
- If $\mathsf{mono}_{A(n),i} = \mathsf{auxiliary}$, then node $n.i$ is a constant leaf.

Here $-\mathsf{convex} = \mathsf{concave}$, $-\mathsf{concave} = \mathsf{convex}$, and $-\mathsf{affine} = \mathsf{affine}$. If $\mathsf{curv}_{A(n)} = \mathsf{affine}$, we can choose role_n arbitrarily to make the tree fulfill the requirements; otherwise, we must set $\mathsf{role}_n = \mathsf{curv}_{A(n)}$. Moreover, for the objective function, $\mathsf{role}_\varepsilon = \mathsf{convex}$, and for constraints $\mathsf{role}_\varepsilon = \mathsf{concave}$. For constraints, the label "concave" semantically means that the constraint describes a convex set; we will explain the reason for this unfortunate naming below.

Some atoms' graph implementations are only valid under certain conditions, or they fulfill monotonicity or curvature properties only under certain conditions. For such atoms, CvxLean introduces *background conditions* $\mathsf{bconds}_A(\bar{a})$ and *variable conditions* $\mathsf{vconds}_A(\bar{a})$. If a tree contains atoms with such conditions or assumptions, it is only valid if for all nodes n, the statements $\mathsf{bconds}_{A(n)}(i \mapsto \mathsf{oexpr}_{n.i}(\bar{y}))$ and $\mathsf{vconds}_{A(n)}(i \mapsto \mathsf{oexpr}_{n.i}(\bar{y}))$ are fulfilled for all $\bar{y}$. Background conditions and variable conditions differ in where CvxLean will search for them. Variable conditions must be present as constraints of the given optimization problem; background conditions must be present in the local context of the optimization problem. Since the local context cannot refer to the optimization variables, background conditions can only refer to parameters of the atom that are constant over the domain.

Crucially, the construction of the tree for any constraint is allowed to fail as long as some other component requires this constraint as a condition. For example, the atom $\log(x)$ requires the condition $0 < x$, and the DCP framework

cannot handle strict inequalities. The graph implementation constraint `expCone t 1 x` (representing $\exp(t) \leq x$), will then be used to justify the constraint $0 < x$, which disappears during the reduction.

Let $\bar{y}$ the variables of the original problem. Let $\bar{z}$ be the collection of variables consisting of one fresh copy for each node's graph implementation variables. Then the optimization variables of the reduced problem are $\bar{y} \cup \bar{z}$. Let $\mathsf{vars}_n(\bar{z})$ the projection to only those variables in $\bar{z}$ that correspond to node n.

To determine the reduced problem, we iterate through each atom tree as follows, determining a *reduced expression* $\mathsf{rexpr}_n(\bar{y}, \bar{z})$ at each node. For leaves n, let $\mathsf{rexpr}_n(\bar{y}, \bar{z}) = \mathsf{oexpr}_n(\bar{y})$. For inner nodes n, let $\mathsf{rexpr}_n(\bar{y}, \bar{z}) = \mathsf{obj}_{A(n)}(i \mapsto \mathsf{rexpr}_{n.i}(\bar{y}, \bar{z}), \mathsf{vars}_n(\bar{z}))$.

The reduced problem's objective function is $\mathsf{rexpr}_\varepsilon(\bar{y}, \bar{z})$ where ε is the root of the objective function's atom tree. For the root ε of each constraint's atom tree, the expression $\mathsf{rexpr}_\varepsilon(\bar{y}, \bar{z})$ becomes a constraint of the reduced problem, unless the constraint is used as a condition of some atom. Moreover, for each node n occurring in any component's atom tree, the expression $\mathsf{constr}_{A(n)}(i \mapsto \mathsf{rexpr}_{n.i}(\bar{y}, \bar{z}), \mathsf{vars}_n(\bar{z}))$ becomes a constraint of the reduced problem.

To establish the equivalence of the original problem and the reduced problem, we need to store the following information with each atom. We write $\approx$ to denote that two expressions are propositionally equal, and we write $=$ to denote that two expressions are definitonally equal.

- Forward properties:
  - Solution: We must provide instantiations $\mathsf{sol}_{A(n)}(\bar{a})$ of the graph implementation variables $\bar{v}$ such that the following two properties hold. These instantiations may contain the atom arguments $\bar{a}$ as parameters.
  - Solution correctness:
    If $\mathsf{vconds}_{A(n)}(\bar{a})$, then $\mathsf{obj}_{A(n)}(\bar{a}, \mathsf{sol}_{A(n)}(\bar{a})) \approx \mathsf{expr}_{A(n)}(\bar{a})$.
  - Solution feasibility: If $\mathsf{vconds}_{A(n)}(\bar{a})$, then $\mathsf{constr}_{A(n)}(\bar{a}, \mathsf{sol}_{A(n)}(\bar{a}))$.
- Backward properties:
  - Optimality: Assume that $\mathsf{constr}_{A(n)}(\bar{a}, \bar{v})$ for some argument values $\bar{a}$ and some variable values $\bar{v}$. Let $\bar{a}'$ be a second tuple of argument values. For convex atoms $A(n)$, we need to show: if $\bar{a} \triangle \bar{a}'$, then $\mathsf{obj}_{A(n)}(\bar{a}, \bar{v}) \geq \mathsf{expr}_{A(n)}(\bar{a}')$; for concave atoms $A(n)$, we need to to show: if $\bar{a}' \triangle \bar{a}$, then $\mathsf{obj}_{A(n)}(\bar{a}, \bar{v}) \leq \mathsf{expr}_{A(n)}(\bar{a}')$; where $\triangle$ denotes $\geq$ for increasing arguments, $\leq$ for decreasing arguments, and $=$ for neither or auxiliary arguments. For affine atoms, we must show both the convex case and the concave case.
    Intuitively, this property combines two different properties of the atom, namely on the one hand that the objective function of the graph implementation is bounded by the atom's function and on the other hand that the monotonicity properties actually hold.
  - Condition elimination: Some variable conditions cannot be translated into conic form, such as the condition of the logarithm atom. For such conditions, we must prove under the assumptions on $\bar{a}$ and $\bar{a}'$ above that $\mathsf{vconds}_{A(n)}(\bar{a}')$ holds.

To establish these properties also for predicate atoms (i.e., atoms returning a value of type `Prop`), we use the Lean's default order `false` $\leq$ `true`. With this order, registering predicate atoms as "concave" amounts to showing that the predicate describes a convex set. This unfortunate naming could be avoided by using the order `false` $\geq$ `true` instead, but we would like to avoid introducing an additional order on `Prop`.

Constructing a strong equivalence requires us to define a forward map φ and a backward map ψ. The definition of the backward map ψ is simple: Since we only add new variable to construct Q from P, we can simply project the domain of Q onto the domain of P.

For the definition of the forward map φ, we use the identity function on the old variables that Q inherits from P, and for the new variables, we define φ such that $\varphi_{\mathsf{vars}_n(\bar{z})}(\bar{y}) = \mathsf{sol}_{A(n)}(i \mapsto \mathsf{rexpr}_{n.i}(\bar{y}, \varphi_{\bar{z}}(\bar{y})))$.

To show that (φ, ψ) is a strong equivalence, we must show that for any feasible point x of P, $g(\varphi(x)) \leq f(x)$ and $d(\varphi(x))$. For the objective function and for the constraints originating from P, we use the *solution correctness* properties of the involved atoms. We can even show the stronger property that for any feasible point x of P, $g(\varphi(x)) = f(x)$ and $d_i(\varphi(x)) \Leftrightarrow c_i(x)$ where d_i is a constraint of Q that originates from a constraint c_i of P.

To show this, we proceed as follows: Let $\bar{y}$ a feasible point in the domain of the original problem P. We recursively show that

$$\mathsf{rexpr}_n(\bar{y}, \varphi_{\bar{z}}(\bar{y})) \approx \mathsf{oexpr}_n(\bar{y})$$

For leaf nodes, this is true by definition of rexpr. For inner nodes, we observe that

$$\begin{aligned}
\mathsf{rexpr}_n(\bar{y}, \varphi_{\bar{z}}(\bar{y})) &\approx \mathsf{obj}_{A(n)}(i \mapsto \mathsf{rexpr}_{n.i}(\bar{y}, \varphi_{\bar{z}}(\bar{y})), \mathsf{vars}_n(\varphi_{\bar{z}}(\bar{y}))) \\
&\quad \text{by definition of rexpr} \\
&\approx \mathsf{obj}_{A(n)}(i \mapsto \mathsf{rexpr}_{n.i}(\bar{y}, \varphi_{\bar{z}}(\bar{y})), \mathsf{sol}_{A(n)}(i \mapsto \mathsf{rexpr}_{n.i}(\bar{y}, \varphi_{\bar{z}}(\bar{y})))) \\
&\quad \text{by definition of } \varphi \\
&\approx \mathsf{obj}_{A(n)}(i \mapsto \mathsf{oexpr}_{n.i}(\bar{y}), \mathsf{sol}_{A(n)}(i \mapsto \mathsf{oexpr}_{n.i}(\bar{y}))) \\
&\quad \text{by the inductive hypothesis} \\
&\approx \mathsf{expr}_{A(n)}(i \mapsto \mathsf{oexpr}_{n.i}(\bar{y})) \\
&\quad \text{by solution correctness} \\
&\approx \mathsf{oexpr}_n(\bar{y}) \\
&\quad \text{by definition of oexpr}
\end{aligned}$$

Above, for solution correctness, we use that $\bar{y}$ a feasible point of P and hence all atoms' conditions $\mathsf{vconds}_{A(n)}(i \mapsto \mathsf{oexpr}_{n.i}(\bar{y}))$ are fulfilled. Thus, for the objective function and for the constraints originating from P, $g(\varphi(x)) = f(x)$ and $d_i(\varphi(x)) \Leftrightarrow c_i(x)$.

For the constraints introduced due to constraints of graph implementations of atoms, we use the *solution feasibility* property of those atoms. As above,

we assume that $\bar{y}$ is a feasible point of P and hence all atoms' conditions $\mathsf{vconds}_{A(n)}(i \mapsto \mathsf{oexpr}_{n.i}(\bar{y}))$ are fulfilled. By solution feasibility, $\mathsf{constr}_{A(n)}(i \mapsto \mathsf{oexpr}_{n.i}(\bar{y}), \mathsf{sol}_{A(n)}(i \mapsto \mathsf{oexpr}_{n.i}(\bar{y})))$. By the equation derived above, it follows that $\mathsf{constr}_{A(n)}(i \mapsto \mathsf{rexpr}_{n.i}(\bar{y}, \varphi_{\bar{z}}(\bar{y})), \mathsf{sol}_{A(n)}(i \mapsto \mathsf{rexpr}_{n.i}(\bar{y}, \varphi_{\bar{z}}(\bar{y}))))$, which by definition of φ is equivalent to $\mathsf{constr}_{A(n)}(i \mapsto \mathsf{rexpr}_{n.i}(\bar{y}, \varphi_{\bar{z}}(\bar{y})), \varphi_{\mathsf{vars}_n(\bar{z})}(\bar{y}))$. Thus also the constraints introduced due to constraints of graph implementations of atoms are fulfilled.

The second property we need to show is that for any feasible point y of Q, $f(\psi(y)) \leq g(y)$ and $c(\psi(y))$. For the objective function and for those constraints of P that were not omitted in Q, we use the *optimality* property as follows. We recursively show that

$$\mathsf{rexpr}_n(\bar{y}, \bar{z}) \sqsupseteq \mathsf{oexpr}_n(\bar{y})$$

where $\sqsupseteq$ denotes $\geq$ if n is convex (or affine in the role of convex), $\leq$ if n is concave (or affine in the role of concave), and $=$ if n is affine or a leaf. At the leaves, the property is obvious by the definition of rexpr. Affine nodes in the role of affine can always be constructed to be leaves. At other inner nodes, we apply the *optimality* property. The conditions of *optimality* are fulfilled for $\bar{a} = i \mapsto \mathsf{rexpr}_{n.i}(\bar{y}, \bar{z})$ and $\bar{a}' = i \mapsto \mathsf{oexpr}_{n.i}(\bar{y})$ by the inductive hypothesis, provided that the tree is valid. Thus, choosing $\bar{v} = \mathsf{vars}_n(\bar{z})$, we have

$$\mathsf{obj}_{A(n)}(i \mapsto \mathsf{rexpr}_{n.i}(\bar{y}, \bar{z}), \mathsf{vars}_n(\bar{z})) \sqsupseteq \mathsf{expr}_{A(n)}(i \mapsto \mathsf{oexpr}_{n.i}(\bar{y}))$$

By definition of rexpr and oexpr, this is equivalent to $\mathsf{rexpr}_n(\bar{y}, \bar{z}) \sqsupseteq \mathsf{oexpr}_n(\bar{y})$.

For the constraints of P that have been omitted in Q, we use the *condition elimination* property. To be able to invoke it, we use the property that we have shown above. We obtain that the variable conditions $\mathsf{vconds}_{A(n)}(i \mapsto \mathsf{oexpr}_{n.i}(\bar{y}))$ hold, which are exactly the constraints of P that have been omitted in Q.