THE UNIVERSITY *of* EDINBURGH

Verified Transformations for Convex Programming

Ramon Fernández Mir

Doctor of Philosophy
Laboratory for Foundations of Computer Science
School of Informatics
University of Edinburgh
2024

Abstract

This thesis is concerned with developing and enhancing several components of `CvxLean`, a convex optimization modeling framework based on the mechanical proof assistant Lean 4.

Convex programming studies the class of optimization problems that are either convex (convex objective function and feasible set) or equivalent to a convex problem. In practice, solvers expect the input to be in conic form, a low-level representation of the problem. Transforming a problem to an equivalent problem in conic form is not straightforward. It often requires several steps and uses properties of the functions involved in the problem in nontrivial ways. This is exactly the process that we formally verify in this work.

The main transformation we verified is the "DCP transformation" step. This required formalizing and extending disciplined convex programming (DCP), a popular technique to transform problems into conic form. It works by iteratively replacing applications of well-understood functions with new variables and conic constraints. In our development, each replacement is augmented with proof obligations that are then combined to produce an overall proof of equivalence between the starting problem and its conic form counterpart.

DCP requires problems to follow a strict set of rules. Often, users need to adjust how they write problems to follow these rules. Instead, we propose an automated and verified "pre-DCP transformation" step using an e-graph-based rewriting system. This procedure explores the space of equivalent problems and finds a sequence of rewrites to transform initial user problems into DCP-compliant forms.

The applicability of the methods that we describe throughout the thesis to real-world problems is backed by a number of examples and case studies.

Lay Summary

Suppose we have a rectangular piece of cardboard and want to make an open-top box by cutting a square from each corner and folding the flaps. How much should we cut from each corner to maximize the volume of the box? This is the type of problem that mathematical optimization software can solve.

Problems like the one above (minimizing or maximizing some quantity subject to some constraints) appear everywhere. In our work, we are particularly interested in applications in engineering, where obtaining the correct result is often safety-critical. The formulas that we want to minimize or maximize, as well as the constraints, might be complicated mathematical expressions, making these problems very hard to solve in general. Luckily, there are some classes of problems that can be solved efficiently. One such class is convex optimization.

In the 1980s, there were major breakthroughs in convex optimization algorithms, and since then, there has been much interest in developing tools around them. One issue is that these tools have become large pieces of software. They modify the starting problem in several ways before actually solving it, and it is not straightforward to ensure all these transformation steps are correct. This mismatch between different representations of the same problem is worrisome. How can we be sure that we solved the problem that we had in mind? In this thesis, we develop a system, `CvxLean`, that endows these transformation steps with a level of mathematical rigor that other tools lack.

To do so, we use a theorem prover, which can be thought of as a programming language that allows us to write not only regular code but also mathematical proofs. In `CvxLean`, every transformation step must be accompanied by a proof, which serves as evidence of its correctness. We show how this verified approach can be used for many problems of interest, making our work a contribution towards developing practical and reliable mathematical optimization software.

Acknowledgements

The last four years have been life-changing for me. To everyone who I met along the way and who helped me get here: **thank you**.

First and foremost, and not only because it is customary, I would like to thank my supervisor, Paul Jackson. Paul's mentorship, guidance, and encouragement are, in one way or another, behind every word in this thesis. His curious and enthusiastic approach to research has made me feel continuously excited about my work. Every time we encountered a problem, even if it was a small or technical problem, Paul always wanted to get to the bottom of it and fully understand it, which I found very inspiring. According to my notes, we have met at least 150 times to discuss ideas and many other times simply to have lunch or tea. I learned so much about so many things in these meetings. Thanks to these conversations, I now have a wider perspective on our fascinating field, which makes me appreciate it much more. I am incredibly thankful for all of that and I feel very lucky to have had Paul by my side on every step of this journey.

I thank Jacques Fleuriot, my second supervisor, for his advice in the yearly reviews and for having me as a member of the AIML Lab. Being part of Jacques' lab has definitely been one of the highlights of my time here. It has allowed me to meet and befriend many other researchers, each with their own unique approach to AI. It is truly a remarkable group of people, and I have always felt at home (and not only because quite a few of them speak Spanish). I thank them all (really hoping that I am not forgetting anyone: Jacques, Paul, Valerio, Jiawei, Jorge, Zonglin, Lauren, James, Filip, Richard, Fiona, Jake, Petros, Imogen, Luna, Guillermo, Ricardo, Paola, Mark, and Matthew). I thank Lauren N. DeLong, in particular, for making me discover neurosymbolic AI. Lauren, Jacques, and I, with the help of others, even wrote a paper about it ("*Neurosymbolic AI for reasoning over knowledge graphs: a survey*")! She made the paper happen, which was no easy task, and I am very happy to have collaborated with her.

Although our paths crossed quite late into my PhD journey, I am honored to have had James H. Davenport and Ian Stark as my PhD examiners. They made the viva a stimulating research conversation, and I thank them for their careful reading and thoughtful comments.

I also thank Iain Murray and Steve Tonneau, who helped me stir the project in the right direction in the yearly reviews.

I would like to thank the LFCS and the School of Informatics for creating a vibrant and stimulating research environment. Being a part of this community has been fantastic. It has been great to interact with so many people working on related areas. There was even a small group of us working on Lean. For a while, we had bi-weekly "Lean lunches", and it was nice to see them become more and more popular (unfortunately, we stopped, although now there is a weekly student-run Lean club, which is wonderful and better organized than our lunches). Those lunches actually greatly inspired some of the work in this thesis. It was in these lunches that I first learned about e-graphs and realized they could be useful for `CvxLean`, which I worked on together with Andrés Goens and Siddharth Bhat, with Tobias Grosser's advice. Working with them has been a real privilege.

I also feel extremely privileged to have worked with Jeremy Avigad and Alexander Bentkamp. They figured out how to make DCP produce proofs in Lean, and `CvxLean` would not exist without them. I thank Jeremy and the Hoskinson Center for Formal Mathematics at CMU for hosting me as a visitor during the Spring of 2022 and funding my stay. That was a very intellectually stimulating period, and I got to work side by side with some truly exceptional people, including Jeremy and Alex. I was lucky to share an office with Bartosz Piotrowski and Ed Ayers, with whom I became very close. As often happens in research, we decided to turn our friendship into a paper and wrote "*Machine-learned premise selection for Lean*", which was a wonderful experience.

I would like to thank the LFCS again, this time for their financial support, which allowed me to attend conferences and summer schools and covered the last six months of my stipend. The first 3 years were covered by the University of Edinburgh through the Principal's Career Development Scholarship. I thank the university for supporting my work.

Conferences and summer schools have been some of the most memorable experiences in my time as a PhD student. I have met lots of interesting people and made some outstanding friends. I have very fond memories of VTSA 2022 in Saarbrücken, CLAS 2022 in Tbilisi, ETAPS 2023 in Paris, and AITP 2023 in Aussois. But perhaps the most inspiring (and fun) event I attended was the Marktoberdorf 2023 summer school. Those two weeks were very special, and my "backseat core" friends Henry, Jonáš, Sofia, Denis, Dimitri, Djordje, Fred, Ioana, and Max were hugely responsible for that. I challenge them to tell me how far Paris is when they read this (and to an international game when we meet again).

Before becoming a PhD student, I was a Maths and Computer Science (JMC) student at Imperial College. There, I had some really fantastic professors who, perhaps unintentionally, made me realize that I wanted to stick around academia for a bit longer. I thank Kevin Buzzard, in particular, who I was fortunate to have as my Master's thesis supervisor. It is no coincidence that, after working with Kevin, Lean became my tool of choice. In my time as an undergraduate, I became fascinated by theorem-proving and formal verification in general. One of the experiences that sparked my interest was being a summer intern at Inria, supervised by Arthur Charguéraud, where I learned so much about writing formal proofs in Coq. After my time at Imperial, I spent a year at the Institute for Logic, Language, and Computation in Amsterdam. I thank everyone I met during that year for helping me take the plunge and pursue a PhD.

Outside of work, a number of friends have made everything better, and I feel so lucky to have them in my life. I thank Martí for his optimism, his kindness, and for all the (sometimes underwater) adventures. I thank Aris for his sense of humor, his empathy, and for all the (often absurd) conversations. I thank all my friends from home and from Imperial, whom I have always been able to count on despite being far away. To Els Nais, my Camino friends, my Hospi friends, and my ICSWP friends, I thank you all. I also thank all the new friends I made in Edinburgh; you have given so much life to my work-life balance. I send a special thank you to the Edinburgh water polo team, particularly the M1s (also known as the "shrimps"); remember when we won BUCS?

Lastly, I am forever grateful for my family. I could not have done any of it without their love and support. To my parents Mª Carmen and Juan Carlos, my siblings Èlia and Joan, my grandparents, aunts, uncles, and cousins: *gràcies*. To Neela: I deeply thank you for your love, for always believing in me, and for making my life as bright as ever.

Declaration

I declare that this thesis was composed by myself, that the work contained herein is my own except where explicitly stated otherwise in the text, and that this work has not been submitted for any other degree or professional qualification except as specified.

(*Ramon Fernández Mir*)

To my family

Quo facto quando orientur controversiae, non magis disputatione opus erit inter duos philosophos, quam inter duos computistas. Sufficiet enim calamos in manus sumere sedereque ad abacos, et sibi mutuo (accito si placet amico) dicere: calculemus.

– Gottfried Wilhelm Leibniz

Table of Contents

Chapter 1

Introduction

1.1 Motivation

Building hardware and software systems is, in part, an exercise in *trust*. We decide whether to trust an operating system, a library, a compiler, a microprocessor, etc. We may even convince ourselves that we trust our own code. But as systems grow and build on top of a myriad of layers of abstraction and hidden assumptions, it is clear that we must provide some form of evidence as to their reliability.

Simulation and testing can guarantee a high degree of trust for most applications. However, this is insufficient for systems with safety or mission-critical components. The 1994 Pentium floating-point division bug [Ede97] is a notable example of an inadequately thorough analysis. The lookup table the division algorithm relied on was missing some values, which led to inaccurate results for certain input pairs. It cost $\sim$ \$475M to replace all faulty devices. Another example is the 1996 Ariane 501 failure [Dow97], in which the software failed to handle an out-of-range value, which made the rocket explode shortly after the launch. The cost of this accident was $\sim$ \$370M.

To help prevent these errors, there is currently much interest in using formal methods. These allow for a rigorous analysis at different levels. First, they provide a way to write formal specifications, which can already help detect flaws in the design. Second, they often allow us to, in one way or another, supply formal proofs of properties of the mathematical model of the system. In the hardware industry, for example, formally verifying complex designs is now seen as an indispensable step before production. Aerospace engineering, cloud computing, and financial services are just a few other fields that also greatly benefit from formal

verification technology. Developing practical verification tools is hence necessary, although also highly challenging, making it a very relevant and exciting field of research. Researchers have developed a wide variety of approaches to assert that systems meet their specifications, each with its own merits. In this thesis, we consider *interactive theorem provers* (ITPs), in particular, the Lean theorem prover [dMU21].

ITPs, also called *proof assistants*, have been successfully used to formalize many deep mathematical results. Simply put, the idea is that users write definitions, theorems and proofs in a formal language, and the system checks that the proofs are correct, i.e., they follow the rules of a given proof calculus. While we can trace early developments back to the 1950s, it was not until much later that ITPs and their libraries were mature enough to be used for advanced mathematics [HUW14]. Famously, in the 2000s and early 2010s, Hales et al. verified key parts of the proof of the Kepler conjecture in HOL Light [HHM$^+$10], rebutting all doubts about the original proof. Around the same time, two celebrated formalization efforts were completed in Coq [CDT24], led by Gonthier: the Four-Color Theorem [Gon07], and the Odd Order Theorem [GAA$^+$13]. These and similar projects have been greatly responsible for popularizing ITPs. Even though the number of ITP users is still relatively small compared to other pieces of software, these communities have experienced unprecedented growth in the last few years. The number of professional mathematicians who have recently found these systems useful for everyday research is particularly remarkable. Notably, Prof. Peter Scholze and Prof. Terence Tao, both Fields medalists, have been recently involved in formalization projects using ITPs (Lean, in particular) and have praised the ability of these tools to handle modern mathematics.

Proof assistants have also been used in more applied domains. For example, verifying properties of continuous and hybrid systems is an area where ITPs have provided substantial benefits. Some good examples include work done in Isabelle/HOL [Imm15, IT16, Fos19, FyMS20] and the KeYmaera X prover [PQ08, MGVP17, JGK$^+$17, KGSJ17]. One reason is that fully automatic verification is unrealistic in these complex domains. A common setup in this line of work is using a proof assistant as a high-level modeling language, where a formal model of the system is defined together with formal specifications of its intended behavior. Verification is typically semi-automatic, with interactive guidance decomposing the structure of the problem and automatic procedures

proving lower-level correctness properties. These procedures often invoke an external tool such as an SMT solver or a computer algebra system. These external tools can either emit proof certificates, thus resulting in a fully verified proof, or not, and be used as trusted oracles, as is common with non-linear real arithmetic decision procedures.

The idea of using an ITP for high-level modeling, relegating low-level computations to a different, more suitable tool, is key to the philosophy of this thesis. This approach has many benefits even if the verification is not end-to-end. Mainly, it allows us to verify the correctness properties we are interested in while taking advantage of decades of research in tooling for automated theorem proving, computer algebra, numerical analysis, etc. In our case, we explore the benefits of linking Lean with *convex optimization* tools. The high-level modeling task consists of defining the problem and transforming it rigorously (with formal proofs of the correctness of the transformation steps) to a form in which it can be solved. The low-level component is a convex optimization solver, responsible for outputting a numerical solution.

Convex programming[1] [BV04] is concerned with the class of optimization problems that are either convex (convex objective function and feasible set) or equivalent to a convex problem. It generalizes linear programming, which is the case when the objective function and the constraints are linear. In the same way that Dantzig revolutionized linear programming by introducing the simplex algorithm in 1947, the development of *interior-point methods* [NN94], starting in the 1980s, provided a way to solve convex optimization problems efficiently. This makes the class of convex optimization problems particularly interesting, as it is expressive enough for many problems of interest while avoiding the high complexity (or, potentially, undecidability if we allow arbitrary transcendental functions) of general non-linear programming. Convex programming is widely used in finance, statistics, communications, and engineering, to mention a few. In this thesis, we focus mostly on engineering problems, as they are problems where we believe the extra rigor of our approach can add the most value. It turns out that designing or analyzing a system can often be re-phrased as a convex optimization problem.

[1] In the literature, convex programming and *convex optimization* are often used interchangeably. One subtlety is geometric programming, which is considered part of convex programming, even though geometric programs are typically not convex. We use "convex programming" to emphasize that the problems we deal with are not necessarily convex, but they can be transformed into convex problems and solved using convex optimization solvers.

Some examples include:

- Stability analysis via Lyapunov functions [SL91]. Given an autonomous system $\dot{x} = f(x(t))$, we can show that it is stable near its equilibrium points by finding an energy function V such that $V(x) \geq 0$ and $V(x) = 0$ if, and only if, $x = 0$; and $\nabla V \cdot f(x) \leq 0$ for all $x \neq 0$. If f is a polynomial, these inequalities can be solved using sum-of-squares [BPT12], so it becomes a convex feasibility problem.

- Safety analysis via barrier certificates [PJP07]. In the same context as above, the sum-of-squares approach can also be used to prove that a set of unsafe states is unreachable from an initial set of states. A barrier certificate is a function that witnesses safety in the same way that the Lyapunov function V witnesses stability.

- Electronic circuit design [BKVH07, §10.3]. Problems in this domain include gate sizing, wire sizing, and power optimization, all of which can be solved using geometric programming.

- Power control [BKVH07, §10.3]. In communications and network systems, resource allocation can often be expressed as a convex optimization problem.

- Structural analysis [BKVH07, §10.3]. Some mechanical problems, such as truss design, are also convex. In Section 6.4, we elaborate on one such example where we minimize the weight of the structure subject to some strength constraints.

- Avionics design [LLP17]. There are numerous convex problems involving the design of optimal shapes with some aerodynamic requirements. We consider one such example in Section 6.5, where we maximize the lift-to-drag ratio of a triangular-shaped object.

We point the reader to the bibliography section of Chapter 4 in Boyd and Vandenberghe [BV04] for more applications.

Regardless of the application domain, the workflow from defining to solving a problem is the same, illustrated in Figure 1.1. The user starts with the original problem in mathematical notation; we say that the problem is in *standard form* at that stage. There are few restrictions to writing problems in this form

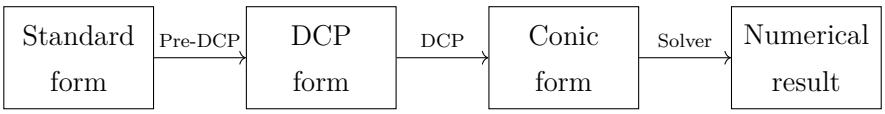


Figure 1.1: Convex programming workflow.

as long as they are mathematically convex. This is problematic from the point of view of automating their analysis and transformation. The approach taken by the convex optimization community has been to design restricted domain-specific languages in a way that problems defined using them are amenable to analysis. They preserve much of the familiar notation and only require the user to follow a relatively simple set of rules. One such framework is *disciplined convex programming (DCP)* [GBY06]. As a first step, a user must reformulate the problem into an equivalent DCP-compliant problem, i.e., a problem in *DCP form*. We call this process a *pre-DCP transformation*. The next step, referred to as the *DCP transformation*, is to transform the problem into *conic form*, a low-level form commonly accepted by convex optimization solvers. This transformation is done automatically by the framework. Finally, the problem is sent to the solver, and a solution is produced. Some popular solvers are `MOSEK` [AA00], `ECOS` [DCB13], `SDPA` [YFN$^+$10], `SeDuMi` [Stu99], and `Gurobi` [Opt20]. There are three potential failure points in this workflow:

(FP1) Pre-DCP transformations are done by hand, and some mistakes can be subtle and hard to spot. For example, one might fail to check every side condition when performing a change of variables. There is also evidence of more advanced transformations in semidefinite programming, where beginner mistakes are known to be common.[2]

(FP2) DCP transformations are less prone to error. However, when a developer of a DCP framework makes an update (e.g., extends the modeling language or simplifies some aspect of the transformation), the only assurance they have is a limited number of tests. We found one such error in an example, which we discuss in Section 6.5.

(FP3) Numerical errors introduced by the solver are a well-known issue in convex programming. There are several sources of inaccuracy, mainly: inexact termination of interior-point methods, non-strict feasibility, ill-

[2]See, for example, `https://yalmip.github.io/badsdps` (accessed 2024-02-05).

conditioning, and floating-point rounding errors [RVS18]. It is often hard to detect how significant these errors are, and most convex programming frameworks simply trust the solver.

In this work, we tackle points **(FP1)** and **(FP2)**. As explained, we are concerned with the correctness of high-level transformations. Improving on point **(FP3)** is an interesting research avenue that we discuss in Section 7.4.4. Although it would be complementary to our approach, it is, in essence, a different research project.

Apart from adding extra rigor to the modeling and transformation of optimization problems, there is a more ambitious vision that this work makes progress towards. That is the idea of *mathematizing programming*. Unquestionably, there is almost always some disparity between the ideal mathematical model programmers have in mind and what their code actually does. Rooted in Lean's philosophy is the idea that one should be able to write mathematically precise and unambiguous code without sacrificing performance or usability. We hope that this thesis can, to some extent, convince the reader that this vision is more than just a dream and that a future where Lean becomes way more than a specialized tool for theorem-proving experts is at reach.

1.2 Research questions

Our research questions essentially boil down to understanding how an interactive theorem prover can help make pre-DCP and DCP transformations more rigorous.

(RQ1) Is it possible to embed pre-DCP transformations in a proof assistant so the problem can be interactively transformed in a convenient way without introducing errors?

- **(RQ1.1)** How much of this process can be automated?
- **(RQ1.2)** Can we transform geometric, quasiconvex, and other non-convex programs without specialized frameworks such as DGP (see Section 2.2.3) or DQCP (see Section 2.2.4)?

(RQ2) Can we develop a *verified* DCP procedure with comparable capabilities to state-of-the-art tools such as `CVXPY` [DB16]?

- **(RQ2.1)** How do we make it user-friendly and extensible?

(RQ2.2) Are there any advantages to this approach beyond correctness?

We take a practical approach to answer these questions by developing and extending several components of `CvxLean`, a convex programming framework embedded in the Lean theorem prover. As we will show in great detail, `CvxLean` provides support for pre-DCP and DCP transformations, and we have successfully formalized a number of examples and case studies to support our claims.

1.3 Thesis structure and contributions

This dissertation comprises seven chapters, including this first introductory chapter, where we have presented the motivation and questions that led this research. In **Chapter 2**, we present some technical and mathematical preliminaries. We introduce the Lean theorem prover and explain some key concepts in convex optimization. Future work, open questions, and concluding thoughts are found in **Chapter 7**. We also include six appendices:

- **Appendix A** describes the library of primitive functions (hereafter, the *atom library*) that can be used to write a problem in DCP form. This library is the basis of the DCP procedure described in Chapter 4.
- **Appendix B** contains a technical proof used in Section 4.6 to justify that our DCP transformation step is verified.
- **Appendix C** is an extension of interval arithmetic with open intervals, which is needed for Chapter 5.
- **Appendix D** shows all the rewrite rules used in Chapter 5.
- **Appendix E** extends Section 5.7, where we evaluate the algorithm presented in Chapter 5.
- **Appendix F** is a short guide to install and navigate `CvxLean`.

The four chapters between the background and the conclusion are the core of the thesis. They describe the different components of `CvxLean` and discuss case studies. In this section, we briefly describe these chapters. Each of them includes a section on related work and the novelties of our approach, explaining

how the work was divided when the work resulted from collaboration. The main collaborators were Dr. Alexander Bentkamp and Prof. Jeremy Avigad, who originally proposed developing a system like `CvxLean` [BA21]. We worked together on the first version of the tool, which resulted in a publication at the International Conference on Tools and Algorithms for the Construction and Analysis of Systems 2023 (TACAS) [BFMA23]. After that, I led the project, and this thesis can be seen as an account of the second version of `CvxLean`, presenting substantial improvements on that first version. Some of the improvements are the result of collaboration with Dr. Andrés Goens, Siddharth Bhat, and Dr. Paul Jackson.

We summarize and highlight the contributions of the core chapters below.

Chapter 3

We give a first overview of `CvxLean`, designed to help the reader understand all the `CvxLean`-specific code in the rest of the thesis. We cover and motivate the notions of equivalence, strong equivalence, reduction, and relaxation. These relations give us the verification conditions that we ultimately need to prove. For each relation, we give examples of transformations that preserve it. We focus on equivalence-preserving transformations and show how they can be applied interactively.

Some of the main definitions are joint work with Bentkamp and Avigad, as presented in our TACAS paper.

Overall contribution The building blocks to define and manipulate optimization problems in Lean.

My contributions I refined and generalized the relations that we support. We only considered reductions in the form of backward solution maps in the first version. Now, we support equivalences, reductions, and relaxations, each with its accompanying "commands". These commands (called *tactics* in the ITP world) allow users to transform problems manually, automatically producing the corresponding proof certificate at every step. There have been substantial improvements in the ergonomics of these tactics, both for end-users and for developers who want to extend `CvxLean` with new tactics. I also added some new transformations, most notably conditional rewrites and changes of variables.

Chapter 4

We cover how the DCP transformation step is extended to produce proofs of equivalence. We first develop the theory behind it, being very precise about all technical considerations. Then, we show how it is implemented in `CvxLean`. The implementation also elaborates on how problems can be solved in our tool.

This chapter is mainly joint work with Bentkamp and Avigad and is heavily based on our TACAS paper. The core DCP algorithm was written by Bentkamp. The refinement of the theoretical framework is joint work with Jackson.

Overall contribution The first fully verified DCP transformation procedure.

My contributions I worked on:

- providing a rigorous mathematical foundation;
- automating and enhancing how side conditions are inferred;
- developing a new way to declare atoms using previously declared atoms, called *multi-level* atom declarations;
- expanding the atom library; and
- putting in place the machinery to call a solver and retrieve a solution.

Chapter 5

We present an approach to transform optimization problems into DCP form. As in the previous chapter, we divide the theoretical aspects of our work and the implementation. The procedure uses `egg` [WNW+21], an e-graph-based rewriting tool, to explore the space of equivalent problems and find one in DCP form.

This chapter is joint work with Goens, Bhat, and Jackson.

Overall contribution A novel system for automatically transforming optimization problems into DCP form based on *e-graphs*.

My contributions I designed and developed the rewrite system. This included the language, the rewrite rules (including conditional rewrites based on interval arithmetic), and the extraction mechanism. I extended a pre-existing interface

between Lean and `egg` and built proof reconstruction mechanisms to cast the procedure as an equivalence-preserving transformation, following the design principles of `CvxLean`. My responsibilities also included conducting extensive experiments.

Chapter 6

This chapter is dedicated to case studies. While many small examples are given throughout, we modeled and analyzed some larger problems that required more detailed explanations. These are fundamental to validate the usefulness of the tool.

I am the sole author of all the case studies presented.

Contributions The formalization of four case studies, carefully selected to showcase diverse scenarios in which `CvxLean` provided substantial value.

1.4 Code reproducibility

This thesis is based on `CvxLean`, which is publicly available at:

`https://github.com/verified-optimization/CvxLean`

More specifically, we discuss version 2.0, accessible under "Releases" on the GitHub page. We point the reader to Appendix F for installation instructions.

Throughout the thesis, we look at different parts of the codebase and show many examples. To make all the code snippets easy to follow and reproduce, we have created a separate repository that follows the order of the thesis.

`https://github.com/ramonfmir/CvxLean-thesis`

It includes links to definitions in the original `CvxLean` repository and formalizations of all the examples shown.

Chapter 2

Background

2.1 The Lean theorem prover

Lean [dMKA+15] is a programming language and proof assistant based on dependent type theory. Its formalism is highly expressive and allows users to define rich mathematical structures. Its standard library, mathlib [The20], is still more limited than Isabelle's [Pau89] or Coq's [CDT24], which are currently the two most popular ITPs apart from Lean, but is growing rapidly. Lean was designed with automation in mind from the beginning, and its main goal is to "bridge the gap between interactive and automated theorem proving," as stated in the system description.

It has primarily been used to formalize mathematical theorems but also for other more applied projects.[1] For example, Certigrad [SLD17] is a Lean project that made progress towards bug-free machine learning by developing a stochastic backpropagation algorithm that produces proofs about the relationship between the sampled gradients and the true mathematical gradients. Lean has also provided value in the blockchain world, for instance, to check properties of compiled Cairo code [AGL+22]. Much of the focus has been on tooling and automation. For example, `aesop` [LF23] provides infrastructure for customizing proof search strategies. There has also been progress on the interactivity side, most notably `ProofWidgets` [NAE23] allows users to define visually rich and interactive proof

[1] The official community website, `https://leanprover-community.github.io`, contains a more comprehensive list of projects under the "Papers" and "Projects" sections. Also, all Lean 4 projects are available in Lean's reservoir `https://reservoir.lean-lang.org/` (links accessed 2024-02-08). On a personal note, some ongoing projects that I am particularly excited about include `SciLean`, `lean-egg`, `duper`, `lean-smt`, `Saturn`, `ssa`, `lean4game`, and `PaperProof`.

states. More recently, there has been a lot of interest in using large language models for generating proof sources or finding useful lemmas; a good example is the `LeanDojo` [YSG+23] project.

Lean has been completely re-engineered with the release of Lean 4 [dMU21] in 2021. A rich macros language has been designed to develop domain-specific languages within Lean conveniently [UdM20]. The *metaprogramming* framework (see Section 2.1.4) has also changed considerably, making a lot of the state data more accessible to tactic developers. There have also been improvements in the ergonomics of writing monadic code, which is the basis of metaprogramming in Lean, mainly by allowing imperative programming-style constructs [UdM22]. Extensibility has been one of the main goals in this redesign, and many of the new features are steps towards allowing Lean developers to easily extend Lean within Lean. It also includes several computational optimizations such as faster arithmetic, efficient code generation, a new type-class resolution procedure [SUdM20] and smart memory management [UdM19].

2.1.1 Proof assistants and their architecture

Before delving into the specifics of Lean, it might be helpful to take a step back. We will discuss how proof assistants work in general, outlining some common design principles. Most of the ideas apply most strongly to ITPs such as Lean [dMU21], Isabelle [Pau89], Coq [CDT24], HOL4 [SN08], HOL Light [Har09], PVS [OSRSC99], and Nuprl [Jac95], which all have *tactic* languages for guiding proof. Some also apply in various other ways to other major ITPs such as ACL2 [KMM00], Mizar [GKN15], and Agda [Nor09]. We will point out some of the differences as needed. This section follows the background section of Ayers' thesis [Aye21, §2.1].

At the core of any mechanical proof assistant, we find a *foundational language* together with a set of *derivation rules* to express logical formulas, mathematical objects, and proofs. There are different choices for this foundation, the most common being first-order logic (FOL), higher-order logic (HOL), or some flavor of dependent type theory (DTT). The *kernel* is the component responsible for checking proofs in this language. There are broadly two classes of kernels. Some take as input a proof object[2] and check whether it is a valid proof, which is how

[2]For now, we may think of a proof object as some term or expression that represents a series of applications of the derivation rules. This approach is sometimes called de Bruijn's criterion.

Lean's kernel works, for instance. The other class is that of kernels based on the LCF architecture[3] [Mil72], which do not rely on storing proof objects. Instead, the kernel consists of partial functions corresponding to the derivation rules that check every proof step (without needing to remember previous steps in the same proof). Isabelle uses this approach. Interestingly, Isabelle is designed to work with multiple foundations, the most popular being Isabelle/HOL.

The (trusted) kernel decides whether a proof is valid or not. To be sure that there are no bugs in the kernel, proof assistant developers aim to keep it as small and simple as possible. This means that many desirable language features cannot be implemented in the kernel. Instead, these features are implemented in a higher-level language, sometimes called the *vernacular*. Some common features include type class inference or implicit arguments, both of which allow users to write much more readable definitions, theorems, and proofs. This language can be converted to the low-level machine-checkable foundational language through a process called *elaboration*.

To write a proof in this high-level language, one commonly uses special instructions called *tactics*. Tactics tell the system to apply one or several proof steps, ultimately building a proof (or proof term). They operate on the current proof state, which usually consists of one or more unproven *goals*, each made up of a collection of *hypothesis* formulas and a *conclusion* formula that needs proving from the hypotheses. With the Lean IDE, a panel called the *infoview* provides a visualization of the current proof state.

We give several examples of tactics in Section 2.1.3. Later on, in Section 2.1.4, we explain how one can build custom tactics. These sections, together with Section 2.1.5 and the upcoming Section 2.1.2, provide a concise introduction to Lean, with enough details to follow all the Lean code presented in this thesis and most of the code in `CvxLean`. While it will not be necessary to understand them in depth, some familiarity with the notation and vocabulary introduced will help. For a more extensive (and friendlier) introduction, we point the reader to the standard reference: *Theorem Proving in Lean 4* [AdMKU24].

[3]For further details, we point the reader to a blog post by Prof. Lawrence C. Paulson FRS where de Bruijn's criterion and the LCF architecture are compared: `https://lawrencecpaulson.github.io/2022/01/05/LCF.html` (accessed 2024-02-19).

2.1.2 Type theory

In this section, we give an overview of Lean's foundation, following the background section of Lewis' thesis [Lew18, §2.1]. Lean is based on the Calculus of Inductive Constructions (CIC), which is a variant of the Calculus of Constructions (CoC) [CH88] with inductive types [CP88]. It is a *dependent type theory*, meaning that types can depend on values. For example, consider writing a function that replicates a real number n times; in Lean, this function has type `(n : ℕ) → ℝ → Vector ℝ n`[4], where a term of type `Vector ℝ n` is a list of real numbers together with a proof that its length is n. This is an instance of a dependent arrow, or Π-type[5], where the type of the output of the function depends on the value of the input. Note that `n : ℕ` is a bound variable in the type expression.

There are essentially two ways to define new types: defining a Π-type or defining an inductive type. Inductive types are defined by a list of constructors. An example is `Nat`, or `ℕ`, defined below.

```
inductive Nat where
  | zero : Nat
  | succ (n : Nat) : Nat
```

For every inductive type, Lean generates a *recursor*, which determines how to use (or *destruct*) a term of said type. For example, to build `Vector.replicate`, one applies `ℕ`'s recursor specifying what happens in the `zero` and `succ` cases. When there is just one constructor, and the inductive type is non-recursive (i.e., the definition body does not involve the type being defined), we call the type a `structure`. In its most basic form, it behaves like C's `struct`. A difference is that a structure type can be made parametric in one or more arguments, in which case it defines a family of structures.

A parametric structure can be declared to be a *type class*. Type classes in Lean are similar to those in other theorem provers, such as Coq or Isabelle, and are an extension of the concept found in Haskell. A clear benefit is that they allow for polymorphic declarations. One key component is the *type class resolu-*

[4]The notation `t : T` is a typing judgment and means that *term* `t` has *type* `T`.

[5]We do not use the "Π" notation as it is no longer supported in Lean 4. However, we note that in Lean 3 and other systems, the type of `Vector.replicate` would be written as `Π (n : ℕ), ℝ → Vector ℝ n`.

tion mechanism, which provides an automatic way of synthesizing inhabitants of *instances* of type classes. For example, `Ring` is a type class, and we can declare particular *instances*, e.g., `Ring ℝ` (an inhabitant of the type `Ring ℝ` consists of terms for 0, 1, +, −, and ·, and proofs of the ring axioms). Any declaration over arbitrary rings can be applied to `ℝ`, and type class resolution will find the inhabitant of `Ring ℝ`. Moreover, instances can be chained, so, for example, if we define a `Ring` instance, we automatically have an `Add` instance (an inhabitant of `Add R` consists of a term of type `R → R → R` and the infix notation "`+`" is syntactic sugar for said term). Inference of such mappings is essential because these mappings are very frequently needed, and specifying them explicitly would add a huge amount of clutter to expressions. This enables compact notation for operators over algebraic structures, where the user does not have to specify the inhabitant explicitly; rather, it is inferred. If `a : ℝ` and `b : ℝ`, we can simply write `a + b`. Type classes are ubiquitous in mathlib; notably since it is the way in which algebraic hierarchies are built, as hinted by our example. However, relying too much on type classes also has its drawbacks, as surveyed by Baanen [Baa22] in the case of Lean 3.

Types are first-class citizens, meaning that they are terms in the language, which implies that they must also have a type. For example, `ℕ : Type`, where `Type` itself also has a type. Lean introduces a countable hierarchy of universes so that `Type : Type 1`, `Type 1 : Type 2`, and so on. We think of elements in `Type u` as data. The type of propositions is `Prop` and a term `p : P` where `P : Prop`[6] is called a proof term, or a *proof*. This is known as the *propositions-as-types* paradigm, which is an instance of the *Curry–Howard correspondence*. The type `Prop` is unique in that two proofs of the same proposition are regarded as the same term, known as *proof irrelevance*, and that terms built from propositions, e.g., `P → P`, remain in `Prop`, known as *impredicativity*. We note that `Prop` is defined as `Sort 0` and `Type u` is defined as `Sort (u + 1)`. Apart from types, terms can take three other forms in Lean. First, they can be *constants*; for instance, constructors in inductive definitions are constants, e.g., `Nat.zero` in the definition of `ℕ`. They can also be *lambda abstractions*, or λ-terms, of the form `(fun (a : A) ↦ b a) : (a : A) → B a`[7] where `b a : B a`. If `B`

[6]Note that `P` here is also a type, whereas `n` shown earlier is not. The proposition `P` is seen as true if it has an inhabitant (i.e., is a non-empty type).

[7]We can also write `λ` instead of `fun`, and `=>` instead of `↦`.

does not depend on `a`, and `b : B`, then the type is a non-dependent arrow, i.e., `(fun (a : A) ↦ b) : A → B`. If `B` is `A → Prop`, then we can also write `∀ a : A, b a`[8], which is the same as `(a : A) → b a`. Finally, terms can also be *applications*, such as `b a` above.

A series of reduction rules give terms computational meaning to terms and establish when two terms are *definitionally equal*. These include β-reduction as in λ-calculus, δ-reduction to unfold definitions, ζ-reduction for let binders, and ι-reduction for inductively defined terms.

When declaring a term of a Π-type, we can think of the left-hand side of the arrow as function arguments. This is clear when using the `def` command, where we can write declarations such as `def f (n : ℕ) : ℕ := ...`, where `f : ℕ → ℕ`. Writing `variable (n : ℕ)` before the definition has the same effect and is often used when multiple definitions take the same arguments. One subtlety is that users can instruct the elaborator how to infer these arguments. By default, they must be provided explicitly, which is indicated by `()`. We may also mark them as *implicit*, indicated by `{}`, which tells Lean to infer them from the other arguments. For example, `def g {n : ℕ} (h : 1 ≤ n) : ℕ := ...`, which means that we can write `g p` where `p` is a proof of `1 ≤ n` and Lean will find `n` for us. Even if an argument is set to be explicit, we may choose to skip it, effectively making it implicit, by writing `_`. We can also indicate when an argument is to be inferred by type class resolution with `[]`. If we have `def h {R} [Ring R] (x : R) : R := ...` (a polymorphic declaration), we can write `h (0 : ℝ)` and type class resolution will find (or, in general, build) a term of type `Ring ℝ`.

There are some differences with other provers based on CIC, such as Coq. For example, quotient types are, to some extent, built in.[9] We point the reader to Carneiro's MSc thesis [Car19] for a detailed account of these differences and a thorough analysis of Lean 3's type theory. The type system has changed slightly in Lean 4, which Ullrich explains in his PhD thesis [Ull23, §3.2]. In particular, the kernel has been extended with more primitive operations. For example, while

[8] In fact, this is also true for any `Sort u`, but we will only use `∀` for propositions. This is a good example of the propositions-as-types paradigm: a proof of a universally quantified proposition over `x : A` is a function with domain `A`.

[9] More precisely, and in mathematical notation, consider an equivalence relation R over a type A, and a function $f : A \to B$ that respects R (for all x and y, if $R(x, y)$, then $f(x) = f(y)$). In that case, we can "lift" f to $f' : A/R \to B$. Then, $f'(\llbracket x \rrbracket)$ is *definitionally equal* to $f(x)$, where $\llbracket x \rrbracket$ is the equivalence class of x.

the definition of `ℕ` remains the same, a new conversion rule allows for computations with natural numbers to be performed at the system level using an arbitrary-precision arithmetic library. There is now also support for η-conversion for structures so that, for example, given `p : A × B`, `(p.fst, p.snd)` is definitionally equal to `p`. The kernel now also supports mutual inductive types.

2.1.3 Tactics overview

This section discusses how to write proofs in Lean using tactics. It is a fairly comprehensive overview which may be skipped on the first reading and referred back to later as needed.

Writing proof terms directly (*term mode*) is possible and, generally, makes elaborating and type-checking proofs faster. However, as soon as a few steps are required, writing proofs in *tactic mode* is preferable and more convenient. The `by` keyword indicates that we are entering tactic mode. In practice, a mixture of both modes is used, aiming for optimized but readable proofs.

The notation `x : A, h : H ⊢ G` indicates a goal. The proof (or tactic) state is given by a list of goals. The *local context* of the goal is `x : A, h : H`, where `x` and `h` are names and `A` and `H` are types. Conventionally, we use names starting with `h` as the names of those elements in the local context whose type is a proposition, which, in this case, would mean `H : Prop`. For convenience, we call any element in the local context a hypothesis, including those that are not propositional. A hypothesis `x : A` should be regarded as a declaration of a variable `x`, which binds occurrences of the variable in subsequent declaration types and in the conclusion type. The conclusion is `G`, which may be an arbitrary type.

Next, we will cover the main built-in tactics. We will also look at some tactics that are part of mathlib [The20]. In particular, we will focus on arithmetic tactics. Note that these might depend on mathlib-defined objects such as the real numbers.

Basic tactics If the conclusion is a Π-type, e.g., `⊢ ∀ x : ℕ, x + 0 = x` or `A → B`, we can use `intros n`, or variants of it, to place the left-hand-side of the arrow in the local context, analogously to ∀-introduction and →-introduction in natural deduction. For dependent types with bound variables, the occurrences of the variable will be replaced by the user-provided name. The first tactic state would become `n : ℕ ⊢ n + 0 = n`. The tactic `exact` takes an expression and

closes the goal with it if its type can be *unified*[10] with the conclusion. Following the same example, we could write `exact (Nat.add_zero n)`. If the conclusion matches a of the hypothesis `h` exactly, one can use `assumption` instead of `exact h`. In fact, `exact` is just a variant of `apply`, which generalizes *modus ponens*. Given `⊢ B` and `f : A → B`, `apply f` will turn the goal into `⊢ A`, opening several subgoals if the function applied has several arguments. Higher-order pattern matching and unification are used to make this work for applications of terms with arbitrary Π-types. It will also try to infer type class instances. Some tactics apply properties of relations, mainly `rfl`, `symm`, and `trans`. In the case of equality, `rfl` will succeed only if the terms are definitionally equal. Another tactic for equality goals is `congr` (or the more general `congr!`), which will iteratively apply congruence rules, e.g., it will turn `⊢ f (g a) = h (g b)` into `⊢ f = h` and `⊢ a = b`.

Tactics for inductive types If a hypothesis `x` is an element of an inductively defined type, `cases x` will "destruct" `x`, splitting the goal into as many constructors as the type has. It is related to the `induction` tactic, which works similarly but also adds the appropriate inductive hypothesis to the context. Note that both are applications of the recursor. If the conclusion is an inductive type, we can use the tactic `constructor`, which will apply the first constructor that matches the type.

Tactics to structure a proof First, note that there is syntax to structure a proof, particularly regarding organizing subgoals. Indentation together with `.` (or `|` if the subgoals are generated by destructing an inductive type) is used to focus a sequence of tactics on a subgoal. While these are technically tactics, they are rarely regarded as such. To avoid repetition, it is often useful to prove intermediate results, which can be achieved using the tactic `have h : H := ...`. To make proofs easier to follow, we can use `show G'` to replace a conclusion `G` with a definitional equal type `G'`. Tactic combinators allow us to make proof scripts more succinct. The `tac1; tac2` syntax indicates that tactics are applied in sequence, and `tac1 <;> tac2` does the same but applies `tac2` to all the subgoals generated by `tac1`. Finally, another way to structure proofs is using

[10] We say "unify" because expressions can have implicit arguments (or, in general, metavariables, which we discuss in Section 2.1.4). Although higher-order unification is undecidable, Lean provides a heuristic algorithm that works in many common cases.

a calculation proof style through the `calc` tactic, which allows us to split a relational goal into several steps.

Rewriter The `rewrite` tactic takes rewrite rules and substitutes them in the conclusion or a hypothesis expression, handling the necessary pattern-matching to find where the rewrite rule applies. Given a universally quantified equality formula such as `h : ∀ x. x + x = 2 * x`, the instruction `rewrite [h]` will change the first subterm that matches the pattern `_ + _`. Rewrites are directed, but one can reverse them by writing `← h`. Moreover, the `at` syntax allows us to specify which hypothesis to rewrite; by default, the rewrite is applied to the conclusion. To choose which subterm to rewrite, the user can either instantiate some of the quantifiers to build a new pattern or use `nth_rewrite` and specify the occurrence that needs to be rewritten. The tactic `rw` is the same as `rewrite` in combination with (a version of) `rfl`.

Simplifier Sometimes, instead of using carefully chosen rewrites to modify a term, one can use `simp`, which automates the rewrite process. A common use case would be to prove equations such as `0 + y * 1 + z * 0 = y`. There is a list of oriented lemmas (called the *simp set*) that the simplifier iteratively rewrites until it cannot make more progress. More lemmas can be added to the simp set by tagging them or specifying them in the tactic, i.e., `simp [h]`. It is also possible to define our own simp set, ignoring all the tagged lemmas with the construct `simp only`. The tactic works modulo commutativity and associativity, avoiding rewrite loops by maintaining an order of the arguments, a technique known as ordered rewriting. The idea is that the simp set is chosen to be as confluent as possible to direct the tactic towards a normal form, which may not always exist. It also works with conditional rewrites, using hypotheses and lemmas in the simp set to apply rewrites of the form `H → a = b`. It has some variants such as `simpa`, used for finishing a proof, usually applying one of the assumptions in the current context, and `dsimp`, which only uses definitional equalities.

Arithmetic tactics The tactics discussed so far are mostly part of Lean's core. There are also tactics specialized to mathlib-defined types that play an important role in our development. For example, `norm_num` simplifies numerical expressions and can prove inequalities of ground arithmetic terms using a `simp`-like mecha-

nism. The `ring` tactic, a re-implementation of the same tactic in Coq [GM05], normalizes expressions in commutative semirings into Holder normal forms, which is particularly useful to prove that two expressions are equal. This is often used in conjunction with `field_simp`, which clears the denominators in an equality or an inequality of terms in a field. Another useful tactic is `linarith`, which can prove linear arithmetic goals by setting up a proof by contradiction and using Fourier-Motzkin variable elimination. Lastly, `positivity` is specialized to positivity, nonnegativity, and nonzeroness goals. It is useful as it can support, to some extent, non-linear expressions and transcendental functions.

Conversion mode We conclude by discussing the so-called `conv` mode briefly. This allows users to travel inside the conclusion expression as needed. For example, we can focus on the left-hand side of an equality, omitting from the proof state the right-hand side. There is also support from pattern-matching to select a particular subterm. Once the desired expression is shown, users can apply a restricted set of tactics that work in this setting, usually `rw` or `simp` (note, for example, that `exact` would not make sense here).

2.1.4 Metaprogramming

Working at the *meta-level* essentially means being able to introspect and modify the data structures built during the compilation process.[11] In particular, this allows us to extend the features of the programming language itself, for instance, by adding new language constructs. It differs from regular (or object-level) programming, where code is written at the outermost level without changing any of the internal components of the language.

The reason why metaprogramming plays a more important role in theorem-proving systems than other systems is mainly because we often want to write custom tactics, which often need to introspect on the expression and goal structure. In many of these systems, meta-level code is written in a different language; for instance, Coq uses OCaml (and Ltac [Del00] for tactics), Isabelle uses ML[12], and Agda uses Haskell. That is not the case in Lean, where its metaprogramming

[11] The term metaprogramming is also used to refer to "code that writes code"; for example, in Ruby, it is possible to define new classes dynamically at runtime. We use the term in more generality. Declaring new data structures dynamically is one instance of metaprogramming.

[12] ML here stands for Meta-Language and has little to do with statistical algorithms.

language is Lean.[13]

The rest of this section focuses on Lean, following the Lean metaprogramming book [PTA+24]. The two compilation stages that we extend are parsing and elaboration.

- Parsing involves turning source code into syntax (`Syntax`).
- Elaboration involves turning syntax (`Syntax`) into expressions (`Expr`). Expressions can be type-checked and evaluated. For example, proof terms are `Expr`s that type-check to the theorem statement's type.

Syntax and macros The parser matches source code according to a set of rules to produce a Lean object of type `Syntax`. It is structured as a tree, including raw symbols such as parentheses and basic typing information. For instance, at this point, Lean knows if the corresponding code snippet is a term or a tactic. The syntax can be extended to define our own domain-specific languages, consequently extending Lean's parser. Macros are rules that go from `Syntax` to `Syntax`. When code is parsed, all available macro rules are applied (or expanded) iteratively. This is how notations (syntactic sugar) are handled, for instance.

Expressions and elaboration The elaborator turns `Syntax` objects into `Expr` objects (or a function that the compiler can work with). This is one of the largest components of Lean. It is responsible for inferring implicit arguments, instantiating metavariables, performing unification, resolving identifiers, etc. In this setting, *metavariables* are "holes" in expressions. For example, a goal in a proof is a metavariable, which will, hopefully, be instantiated by a proof term. Command and tactic syntax are elaborated into executable code that changes the environment (e.g., adding a definition) or proof state, respectively. Oftentimes, this involves composing, decomposing, and modifying expressions. Indeed, most of our work involves manipulating terms of type `Expr`, shown below.

```
inductive Expr where
  | bvar    : Nat → Expr
  | fvar    : FVarId → Expr
  | mvar    : MVarId → Expr
```

[13] Lean does rely on some C++ components (e.g., the kernel), but the front-end of the compiler is written in Lean, which is the part that we want to modify.

```
  | sort    : Level → Expr
  | const   : Name → List Level → Expr
  | app     : Expr → Expr → Expr
  | lam     : Name → Expr → Expr → BinderInfo → Expr
  | forallE : Name → Expr → Expr → BinderInfo → Expr
  | letE    : Name → Expr → Expr → Expr → Bool → Expr
  | lit     : Literal → Expr
  | mdata   : MData → Expr → Expr
  | proj    : Name → Nat → Expr → Expr
```

Most constructors are clear from our discussion about Lean's type theory; however, it might be useful to clarify some of them. Free variables refer to variables in the local context, whereas bound variables (with de Bruijn indices) appear inside a λ-term. The constructor `forallE` builds dependent arrows, i.e., Π-types. Literals hold natural numbers or strings and are there for optimization purposes. Metadata is a useful way to annotate expressions with custom data. Finally, structure projections are logically redundant but used for η-reduction of structures. We note that there is also a process called *delaboration*, which is how we can pretty-print expressions, including those generated at the meta-level.

Monadic programming Lean's metaprogramming framework [EUR+17] is built around a series of monads. Monads are a useful abstraction in functional programming to encode side effects. Notably, one can simulate having a mutable state. The following four state monads each extend each other and have access to different amounts of meta-level information:

- `CoreM` is the base monad and has access to the environment, which includes information about files, namespaces, imported files, and available declarations.
- `MetaM` is extended with the metavariable context, so one has access to the metavariables that have been assigned, the ones pending, and other related information.
- `TermElabM` gives access to the infrastructure used for elaboration; thus, working with syntax usually happens at this level. `CommandElabM` is related to it, used specifically for command syntax.

- `TacticM` also has access to the current proof goals.

2.1.5 Formalized mathematics

We conclude the Lean part of the background by giving a non-exhaustive overview of some mathematical theories formalized in Lean. Basic definitions concerning logic, sets, and the algebraic hierarchy can be found in the textbook *Mathematics in Lean* [AM24]. Regarding more advanced mathematics, Lean is mostly known for its number theory and algebraic geometry projects [Lew19, BCM20, BHL+22, CL21, dF22, BBCD23]. There has also been substantial progress on other parts such as analysis [Kud22, Gou22]. Some advanced results in combinatorics and graph theory [GMM21, DM22], and set theory [HvD20] have been successfully formalized too.

Some parts of mathlib[14] are directly relevant to this thesis, and it might be useful to give pointers to them, which we do here. First, in `Data/Real` and `Analysis/SpecialFunctions`, we find the theory of real numbers and results about exponentials, logarithms, and powers. The `Analysis/Convex` library defines convex sets and cones and presents results about convex hulls and extreme points. It also formalizes key results such as Carathéodory's convexity theorem, Riesz's extension theorem, and Jensen's inequality. Notions of strictness, properness, quasiconvexity, and conic duality are also defined. Finally, in `LinearAlgebra/Matrix`, we find some fundamental linear algebra, such as the theory of determinants, bases, or invertibility. Bentkamp extended this part of the library for this project, formalizing the spectral theorem and defining LDL decompositions.

2.2 Convex optimization

This section first covers some fundamental notions in convex optimization following Boyd and Vandenberghe [BV04]. They are mostly standard, although we are particularly careful with some terminology regarding the "forms" of the problems, which play a key role in this thesis. Even though this section is Lean-agnostic, some of the notation might be slightly unconventional, as it is designed to be consistent with the Lean formalization we will present later.

[14]The full documentation can be found in `https://leanprover-community.github.io/mathlib4_docs/index.html` (accessed 2023-03-23).

The rest of the chapter gives an overview of *disciplined convex programming (DCP)* in Section 2.2.2, *disciplined geometric programming (DGP)* in Section 2.2.3, and *disciplined quasiconvex programming (DQCP)* in Section 2.2.4.

2.2.1 Definitions

The first definition we need is that of a mathematical optimization problem in full generality, regardless of its shape and curvature properties.

Definition 2.2.1 (Minimization[15] problem)**.** Let D be a set, which we will call *domain*, and let R be a preordered set, which we will call *codomain* or *range*. A *minimization problem* P is defined by:

- $f : D \to R$, the *objective function*, and
- $cs : D \to \{\top, \bot\}$, the *constraints*.

We interpret cs as a predicate over D where $cs(x) = \top$ (or just $cs(x)$) means that x satisfies the constraints of P. We can define $P := (f, cs)$, or in the usual more descriptive format below:

$$\begin{aligned} P := \text{minimize} \quad & f(x) \\ \text{subject to} \quad & cs(x). \end{aligned}$$

Since $cs(x)$ is usually of the form $cs_1(x) \wedge \cdots \wedge cs_m(x)$, we can write

$$\begin{aligned} P := \text{minimize} \quad & f(x) \\ \text{subject to} \quad & cs_1(x) \\ & \vdots \\ & cs_m(x). \end{aligned}$$

We call $f, cs_1, \ldots, c_m$ the *components* of P.

Definition 2.2.2 (Feasible point predicate)**.** Given a minimization $P := (f, cs)$ over some domain D and range R, $x \in D$ is feasible if it satisfies the constraints:

$$\text{feasible}_P(x) := cs(x).$$

The set of all feasible points is called the *feasible set*.

[15] We can also phrase optimization problems as maximization problems. Usually, a maximization problem can be considered a minimization problem with the objective function negated. That is what all the instances of "maximize" will mean in this thesis. However, this is not the case in all settings; for instance, if a function f is positive in the domain, maximizing f could also mean minimizing $1/f$. For simplicity and to avoid any confusion, we will set up our definitions in terms of minimization problems.

Definition 2.2.3 (Optimal point predicate)**.** Given P as above, $x^\star \in D$ is an *optimal point* if it is a feasible point and for all other feasible points $x \in D$ we have $f(x^\star) \leq f(x)$. We define optimality as a predicate as follows:

$$\text{optimal}_P(x) := \text{feasible}_P(x) \wedge \forall y.(\text{feasible}_P(y) \Rightarrow f(x) \leq f(y)).$$

Definition 2.2.4 (Solution set)**.** The *solution set* of a minimization P, denoted sol_P, is the set of all optimal points:

$$\text{sol}_P := \{x \in D \mid \text{optimal}_P(x)\}.$$

In (real) convex optimization, D is a finite-dimensional real vector space and the codomain R is $\mathbb{R}$ or $\mathbb{R} \cup \{\infty\}$. The usual shape of D is a cartesian product of copies of $\mathbb{R}^n$ and $\mathbb{R}^{n \times n}$.

Fix D to be any finite-dimensional real vector space. We proceed by discussing the notion of convexity.

Definition 2.2.5 (Convex set)**.** A set $C \subseteq D$ is a *convex set* if for all $x, y \in C$ and $t \in [0,1]$ we have $tx + (1-t)y \in C$.

Recall that the definition of an *affine set* is as above but with $t \in \mathbb{R}$. Subsets of D can be non-affine or non-convex, but there is no notion of a concave set.

Definition 2.2.6 (Convex function)**.** A function $f : C \to \mathbb{R}$ is a *convex function* if C is a convex set and for all $x, y \in C$ and $t \in [0,1]$ we have

$$f\left(tx + (1-t)y\right) \leq tf\left(x\right) + (1-t)f\left(y\right).$$

The definition of a *concave function* is as above replacing "$\leq$" with "$\geq$". A function that is both convex and concave is an *affine function*. Generally, a function $f : C \subseteq D \to \mathbb{R}$ can be convex, concave, affine, or neither.

To see the link between functions and convex sets, note that a function is convex if, and only if, its *epigraph*,

$$\text{epi}(f) := \{(x,t) \mid f(x) \leq t\},$$

is a convex set. Similarly, a function is concave if, and only if, its *hypograph*,

$$\text{hypo}(f) := \{(x,t) \mid f(x) \geq t\},$$

is a convex set.

Definition 2.2.7 (Convex optimization problem)**.** A problem $P := (f, cs)$ is convex if f is a convex function and cs defines a convex feasible set.

As mentioned earlier, convex optimization problems can be represented in many forms, which we define here.

Definition 2.2.8 (Standard form[16], first definition)**.** Let functions $f_i : D \to \mathbb{R}$ for $i = 0, \ldots, m$ be convex and $h_i : D \to \mathbb{R}$ for $j = 1, \ldots, l$ be affine. A convex optimization problem in *simplified standard form* is the following:

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

Keeping everything the same, if $\tilde{f}_i : D \to \mathbb{R}$ for $i = 1, \ldots, m$ are concave, and $\tilde{h_j} : D \to \mathbb{R}$ for $j = 1, \ldots, l$ are affine, a problem in *standard form* is as follows:

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

Usually, the above is defined with $D := \mathbb{R}^n$. We require the extra generality to capture scenarios where the domain involves matrix spaces.

We have implicitly assumed that the functions are defined over all of D. However, many of the functions that we will use in our problems, such as logarithms and inverses, are partial functions. In those cases, we assume that their domain is artificially extended, and it will be our responsibility to check at a later stage that the problem is well-defined. To be more precise, if a partial function application occurs somewhere in the problem, we will need to show that the constraints imply that the argument is in the domain of the function. We note that working with extended-value functions is standard in convex analysis [Roc70, § 4] to simplify many of the derivations.

One issue with our definition above is that it does not allow for *positive semidefinite* constraints.

[16]Most authors do not distinguish between standard form and simplified standard form. Of course, from a mathematical point of view, it is obvious that the two forms are equivalent. They are, however, different syntactic objects, which matters in our analysis. We also point out that the word "standard" is not used uniformly across the literature.

Definition 2.2.9 (Positive semidefinite)**.** Given $A \in \mathcal{S}^n$, i.e., an $n \times n$ real symmetric matrix, we say that it is *positive semidefinite* and write $A \succeq 0$ if for all $x \in \mathbb{R}^n$, $x^T Ax \geq 0$. Equivalently, A has a *Cholesky decomposition* $A = L^T L$.

To obtain a definition of standard form that unifies all convex optimization problems that we are interested in, we extend definition 2.2.8 with constraints of the form $h_k(x) \succeq 0$ where each $h_k : D \to \mathbb{R}^{n_k \times n_k}$ is affine.

Having defined standard form, we now work towards defining *conic form*. First, we define what it means to be a *convex cone*.

Definition 2.2.10 (Convex cone)**.** A *convex cone* is a convex set $C \subseteq D$ that is also a cone, i.e., for every nonnegative scalar $s \in \mathbb{R}$ and $x \in C$, $sx \in C$.

Conic solvers do not work with arbitrary convex cones. Instead, there is a small set of useful and well-understood families of cones to choose from. While the selection may vary, these are some of the most frequent ones, all of which have been used in our problems:

- The *zero cone* $0^n := \{(0, \ldots, 0)\}$.
- The *nonnegative orthant cone*[17] $\mathbb{R}^n_+ := \{x \in \mathbb{R}^n \mid x_i \geq 0\}$.
- The *second-order (or quadratic) cone* $\mathcal{Q}^{n+1} := \left\{(t, x) \in \mathbb{R} \times \mathbb{R}^n \mid \sqrt{x^T x} \leq t\right\}$.
- The *second-order (or quadratic) rotated cone*

$$\mathcal{Q}^{n+2}_r := \left\{(v, w, x) \in \mathbb{R} \times \mathbb{R} \times \mathbb{R}^n \mid x^T x \leq 2vw \wedge 0 \leq v, w\right\}.$$

- The *exponential cone*

$$\mathcal{K}_{\exp} := \left\{(t, s, r) \in \mathbb{R}^3 \mid (t \geq s \cdot \exp(r/s) \wedge s \geq 0) \vee (t \geq 0 \wedge r \leq 0 \wedge s = 0)\right\}.$$

- The *positive semidefinite cone* $\mathcal{S}^n_+ := \{A \in \mathcal{S}^n \mid A \succeq 0\}$.

We conclude this subsection by discussing problems in *conic form*.

Definition 2.2.11 (Conic form)**.** Let $A : D \to \mathbb{R}^m$ and $c : D \to \mathbb{R}$ be linear maps, $F : D \to \mathbb{R}^k$ an affine map, $b \in \mathbb{R}^m$, and $\mathcal{K}$ an k-dimensional convex cone[18]. A

[17] Usually called the *positive orthant cone*, which we avoid as it can be misleading.

[18] Technically, the cone needs to be proper (convex, pointed, closed and with non-empty interior) to be compatible with interior-point algorithms. All the examples given earlier are proper cones, except for the zero cone, which is treated specially by solvers.

convex optimization problem in *conic form* over D is given by:

$$\begin{aligned} \text{minimize} \quad & c(x) \\ \text{subject to} \quad & A(x) = b \\ & F(x) \in \mathcal{K} \end{aligned}$$

We have stated this definition in full generality. Usually, this definition is specialized to $D := \mathbb{R}^n$, with objective function $c^T x$, and constraints are $Ax = b$ and $Fx + g \in \mathcal{K}$, where $c \in \mathbb{R}^n$, $A \in \mathbb{R}^{m \times n}$, $F \in \mathbb{R}^{k \times n}$, and $g \in \mathbb{R}^k$. As a small remark, if, in this setting, we restrict $\mathcal{K}$ to be products of 0^n and $\mathbb{R}^n_+$, then we obtain the definition of a linear program.

Note that we can remove $A(x) = b$ and write only one constraint of the form $F'(x) \in \mathcal{K}'$ where $F'(x) := (F(x), A(x) - b)$ and $\mathcal{K}' := \mathcal{K} \times 0^m$.

We point the reader to Section 4.7.3, where we show one way to define conic forms: *conic benchmark format*. It provides a concrete way to describe the maps c, A, and F as sums of dot products and inner products, thus allowing the problem to be defined solely from numerical data and the components of $\mathcal{K}$.

2.2.2 Disciplined convex programming

We have shown two very distinct ways to represent optimization problems. Standard form is the familiar mathematical notation we think of and usually phrase optimization problems. On the other hand, conic form should be thought of as a machine-level representation, which we never want to write directly. The natural next question is how these two are related. In particular, how can we transform a problem in standard form into an equivalent problem in conic form? The most popular approach is *disciplined convex programming* (DCP), which we discuss in this section.

DCP [Gra05, GBY06] is a framework to systematically transform problems into conic form. The key idea is to replace expressions not in conic form with equivalent optimization problems in conic form.

The framework has two key components:

- An *atom library*. Atoms are functions tagged with the following properties:
  - Domain and range.
  - Curvature: convex, concave, or affine.

  - Monotonicity: whether it is increasing, decreasing, or neither in each of its arguments.

  Most atoms have a corresponding *graph implementation* [GB08], which is a representation of the function as the solution of an optimization problem in conic form.

- A *curvature ruleset* that determines the curvature of combinations of atoms, variables, and numerical constants.

A rigorous logical treatment of DCP is given in Chapter 4. For now, we provide an example of an atom and some intuition on the curvature ruleset.

Consider the extended-value square root function $\sqrt{\cdot} : \mathbb{R} \to \mathbb{R}$. In its atom definition, we specify that its domain is $\{x \in \mathbb{R} \mid x \geq 0\}$. Its range is $\mathbb{R}$, and it is concave and monotonically increasing in its argument. The following conic program gives its graph implementation, parametrized over x:

$$\begin{aligned} &\text{maximize} && v \\ &\text{subject to} && (x, 0.5, v) \in \mathcal{Q}_r^3, \end{aligned}$$

where maximizing is the same as minimizing $-v$. From the definition of $\mathcal{Q}_r^{n+2}$ earlier, it follows that $(x, 0.5, v) \in \mathcal{Q}_r^3$ is equivalent to $x^2 \leq v$. In the case of convex atoms, the graph implementation is a minimization. The idea of the graph implementation is that its solution equals the value of the atom in the following sense. For any $x \geq 0$, we have:

$$\sqrt{x} = \sup\left\{v \in \mathbb{R} \mid (x, 0.5, v) \in \mathcal{Q}_r^3\right\}$$

The information above completely defines the square root atom.

Whenever we find an application of the square root, we would like to replace it with a new variable v and add the second-order constraint (we call this process *graph expansion*). However, we need to consider what happens if the argument is an arbitrary expression. For example, we could have $\sqrt{\log(x)}$. First, we need to make sure that we can infer its curvature. In this case, we can check that it is concave (as a function) if $x > 0$. We would proceed by first replacing $\log(x)$ with its graph implementation, introducing a new variable w and the constraint $(w, 1, x) \in \mathcal{K}_{\exp}$, equivalent to $\exp(x) \leq w$. Then, we would replace $\sqrt{w}$ as shown earlier.

$$\sqrt{\log(x)} \quad \xrightarrow{\text{expands to}} \quad v \text{ with constraints } (w, 0.5, v) \in \mathcal{Q}_r^3 \text{ and } (w, 1, x) \in \mathcal{K}_{\exp}.$$

To generalize and automate the argument above that led us to check that $\sqrt{\log(x)}$ is a concave expression, we require a curvature deduction calculus. In this case, we implicitly used a *composition rule* [GBY06, § 6.4].

Definition 2.2.12 (Composition rule for concave atoms)**.** Given an expression $e := f(a_1, \ldots, a_n)$ where f is a concave atom, we say that e is *DCP concave* if for each $i \in \{1, \ldots, n\}$, one of the following is true:

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

Similarly, we define a composition rule for convex atoms.

Definition 2.2.13 (Composition rule for convex atoms)**.** Given an expression $e := f(a_1, \ldots, a_n)$ where f is a convex atom, we say that e is *DCP convex* if for each $i \in \{1, \ldots, n\}$, one of the following is true:

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

Affine expressions can only be built by composing affine functions, variables, and numerical constants.

An expression is considered DCP-compliant if its curvature can be inferred from the composition rules. This has some limitations. For example, $\sqrt{\exp(x)}$ is not DCP-compliant, even though it is convex over all of $\mathbb{R}$. This means that DCP convex (respectively, concave) functions are a strict subset of convex (respectively, concave) functions. The reason why these rules are needed is that it is not easy, in general, to deduce the curvature of a given function. We cannot say much about composed functions that do not follow the composition rules, as exemplified by $\exp\left(-x^2\right)$, which is convex in $\left(-1/\sqrt{2}, 1/\sqrt{2}\right)$ and concave everywhere else.

These rules allow us to check the curvature of expressions. To check that a problem is DCP-compliant, there are also *top-level rules*[19]. They establish

[19]In Grant's initial exposition of DCP [GBY06], there are also product-free and sign rules. They ensure that only multiplication by constants is allowed and define how curvature might change depending on the sign of the constant. Strictly speaking, they are unnecessary and can be simulated by having several versions of the multiplication atom. For example, we can define $c \cdot x$ for constant $c \geq 0$ to be an affine atom (hence also convex and concave) with an increasing argument.

that the objective function must be a DCP convex or affine expression. Valid constraints follow one of the following rules:

- "DCP affine" = "DCP affine".
- "DCP convex" $\leq$ "DCP concave".
- "DCP affine" $\in \mathcal{K}$, where $\mathcal{K}$ is a convex cone.

If a problem follows these rules, we say that it is in *DCP form*. Note that any problem in conic form is also in DCP form.

DCP frameworks usually split the analysis[20] and *canonization* phases by first checking that the problem is in DCP form. A problem in DCP form is built from atom applications, which may be used to build an expression tree with variables and constants at its leaves, which we call an *atom tree*. The canonization procedure traverses this tree bottom-up, replacing atom applications with new variables and new conic constraints from the atom's graph implementation, as illustrated in our example earlier. Once this process finishes, the result is a problem in conic form, which should be *equivalent* (see definition 3.3.1) to the original problem.

This process requires checking curvature rules and domain conditions, which may be nontrivial. Even assuming the correctness of these checks, it is far from obvious that the resulting problem is equivalent. In Chapter 4, we will show how to prove that they are equivalent. Hence, we leave further discussions of the canonization algorithm and its correctness for then.

2.2.3 Disciplined geometric programming

Geometric programming is concerned with problems with a special structure that can be solved using convex optimization tools. It has been studied since the 1960s but has recently experienced a surge in popularity due to the development of specialized software and its applicability to engineering problems [BKVH07, §10.3]. As we will see, they are also a great source of examples for our work. Interestingly, geometric programs are usually not convex but can be turned into convex problems (DCP convex, to be more precise) after a simple transformation.

[20] This phase is also called the (curvature) verification phase. We avoid that terminology since, in this thesis, we focus on verifying the correctness of the transformation, not the curvature.

In this section, we define geometric programs and discuss the transformation steps required to solve them.

Fix n and let $x_1, \ldots, x_n$ be real positive variables. A *posynomial* is a function of the form

$$\sum_{i=1}^{K} c_i x_1^{a_{1i}} \cdots x_n^{a_{ni}},$$

where $c_i > 0$ and $a_i \in \mathbb{R}$. If $K = 1$, we call it a *positive monomial*. A *geometric program (GP)* is an optimization problem of the form

$$\begin{array}{ll} \text{minimize} & f_0(x) \\ \text{subject to} & f_i(x) \leq \tilde{f}_i(x) \quad i = 1, \ldots, m \\ & g_j(x) = \tilde{g}_j(x) \quad j = 1, \ldots, p, \end{array}$$

where each f_i is a posynomial and each $\tilde{f}_i$, g_j, and $\tilde{g}_j$ is a positive monomial.

Consider the following example:

$$\begin{array}{rl} \text{minimize} & y/x \\ \text{subject to} & x, y, z > 0, \\ & 2 \leq x \leq 3, \\ & x^2 + 3(y/z) \leq 5\sqrt{y}, \\ & xy = z^2. \end{array}$$

It is not DCP-compliant for several reasons. For example, xy does not follow the DCP curvature rules: the atoms for multiplication require one argument to be constant. To see that it is not mathematically convex, we can argue that the constraint $xy = z^2$ does not define a convex set. Trace a line segment between $\vec{a} := \left(2, 4, \sqrt{8}\right)$ and $\vec{b} := \left(3, 10, \sqrt{30}\right)$; looking at the middle point, we see that $2.5 \cdot 7 \neq \left(\left(\sqrt{8} + \sqrt{30}\right)/2\right)^2$. This argument works since there is more than one point in the feasible set; indeed, both $\vec{a}$ and $\vec{b}$ are feasible points.

We turn it into a DCP convex problem as follows. First, we apply the change of variables $(x, y, z) \mapsto (\exp(u), \exp(v), \exp(w))$, which works since the variables are positive. After the change of variables, we may remove the positivity constraints. Then, we apply logarithms to both sides of the first two inequalities and the equality, which is valid because, in all cases, both sides are positive. After

simplifying and rearranging the terms, we obtain:

$$\begin{aligned} \text{minimize} \quad & v - u \\ \text{subject to} \quad & \log(2) \leq u \leq \log(3), \\ & 0.2\exp(2u - 0.5v) + 0.6\exp(0.5v - w) \leq 1, \\ & u + v - 2w = 0. \end{aligned}$$

It is easy to see that this problem is in DCP form as everything is affine except the left-hand side of the second constraint, which is DCP convex. It suffices to note that the exponential applied to an affine function is DCP convex, and multiplication by positive scalars and addition preserve DCP convexity.

Disciplined geometric programming (DGP) [ADB19] aims at automating this process. It works in a similar fashion to DCP but with log-log curvature and a specialized atom library. Let $\mathbb{R}_{++} := \{x \in \mathbb{R} \mid x > 0\}$. Then $f : D \subseteq \mathbb{R}^n_{++} \to \mathbb{R}_{++}$ is log-log convex if $F : u \mapsto \log(f(\exp(u)))$ is convex. The definitions of log-log concave and log-log affine functions are analogous, and there are composition rules based on the monotonicity of the arguments, like in DCP. Atoms are tagged according to their log-log curvature, with graph implementation in DCP form. This is used to systematically canonize DGP problems to DCP form.

DGP can be used beyond geometric programs. It also allows fractional powers of posynomials such as $\sqrt{1+x^2}$. These are called *generalized geometric programs* (GGP). Note that, since $\sqrt{1+x^2}$ is an increasing function of $1+x^2$, we can add a new variable t, replace the expression by $\sqrt{t}$ and add the constraint $1+x^2 \leq t$ to obtain an equivalent program without the power on the sum. In fact, DGP can be extended to handle *log-log convex programs* (LLCP) beyond GGPs, as long as they are built from definable DGP atoms.

2.2.4 Disciplined quasiconvex programming

Another interesting class of non-convex problems that can be solved by convex methods is *quasiconvex programming*.

A function $f : D \subseteq \mathbb{R}^n \to \mathbb{R}$ is quasconvex if D is convex and for any $\alpha \in \mathbb{R}$, the set $\{x \in D \mid f(x) \leq \alpha\}$, known as the α-sublevel set, is convex. A *quasiconvex program* (QCP) minimizes a quasiconvex function over a convex set. The key observation is that a family of convex functions $\{\phi_\alpha \mid \alpha \in \mathbb{R}\}$ can represent a quasiconvex function f, in the sense that $f(x) \leq \alpha \Leftrightarrow \phi_\alpha(x) \leq 0$.

For instance, the ratio function $f(x,y) = x/y$ is quasilinear (quasiconvex and quasiconcave) when $y > 0$. It is represented by

$$\phi_\alpha(x) = \begin{cases} 0 & \text{if } x \leq \alpha y, \\ \infty & \text{otherwise.} \end{cases}$$

To see how this extends to full problems, consider the following example of a quasiconvex program:

$$\begin{aligned} \text{minimize} \quad & -x/y \\ \text{subject to} \quad & \exp(x) \leq y. \end{aligned}$$

It can be represented by a family of feasibility problems parametrized by t:

$$\begin{aligned} \text{find} \quad & x, y \\ \text{subject to} \quad & \exp(x) \leq y, \\ & -x \leq ty \end{aligned}$$

The idea of the solving procedure is to guess an initial interval for t and iteratively update the value of t by bisecting the interval until the minimal value of t that satisfies the constraints is found. The problem above is optimal at $x^* = 1, y^* = e$, and this value is attained at $t = -1/e$.

If we treat t as a parameter (not a variable), then the problem above is in DCP form. In the style of DGP, *disciplined quasiconvex programming* (DQCP) [AB20] is the framework responsible for canonizing QCPs to (a family of problems in) DCP form. Again, there is a DQCP atom library with the corresponding composition rules.

2.2.5 Summary of forms

This chapter has provided an overview of convex optimization, particularly focusing on the forms of problems. We use the word "form" to emphasize that we are talking about representations of problems, carefully making a distinction with the concept of "class", which has to do with the mathematical nature of the problem. Two problems in different forms might be equivalent, placing them, in some sense, in the same class.

We note that chaining problem transformations from one representation to another is a fundamental idea in convex optimization modeling software, as explained by Agrawal et al. [AVDB18] in the case of `CVXPY`.

It will be useful for the rest of our work to keep the forms that we consider in mind, which, as explained, do not always match exactly the definitions found in the literature. We summarize them here:

- *Standard form* corresponds to definition 2.2.8 with positive semidefinite constraints. Problems in standard form are a subset of convex optimization problems.
- *Conic form* corresponds to definition 2.2.11. They are also a subset of convex optimization problems.
- *DCP form* is used for problems defined with DCP atoms that follow the DCP curvature ruleset. Problems in DCP form are a subset of problems in standard form extended with arbitrary cone membership constraints $e \in \mathcal{K}$, where e is affine.
- *DGP form* is used for problems defined with DGP atoms that follow the DGP curvature ruleset. Problems in DGP form are a subset of LLCPs.
- *DQCP form* is used for problems defined with DQCP atoms that follow the DQCP curvature ruleset. Problems in DQCP form are a subset of QCPs.

Chapter 3

Transformations in CvxLean

3.1 Introduction

This chapter starts by giving an overview of `CvxLean`, particularly showing the core types and syntax to define optimization problems. Then, we formally introduce the notions of (regular and strong) equivalence, reduction, and relaxation. While these relations are mentioned often in the literature, we have not found a rigorous logical treatment, so we fill this gap here. As we will see, the two main transformations formalized in this thesis, introduced in Chapter 4 and Chapter 5, are equivalences or, more precisely, strong equivalences. Hence, it is important to understand what is required to verify equivalences formally.

These definitions are the foundation of `CvxLean`. The `equivalence` command gives an environment for users to transform their problems while proving equivalence, which is one of the main intended uses of the tool. There are also `reduction` and `relaxation` commands for cases when another relation is sought. Tactics are designed to construct equivalences, reductions, or relaxations explicitly, guided by the user. We will cover how all of this machinery works.

Moreover, we formalize the following common equivalence-preserving transformations [BV04, § 4.1.3]:

1. Changes of variables (see Section 3.3.2.1).
2. Rewriting the objective function or a constraint (see Section 3.3.2.2 for unconditional rewrites and Section 3.3.2.3 for conditional rewrites).
3. Pre-composing the objective function or a function involved in a constraint with another function (see Section 3.3.2.4).

4. Renaming variables or constraints (see Section 3.3.2.5).

5. Other basic transformations: reordering constraints, removing constraints, introducing slack variables, introducing equality constraints, eliminating equality constraints, and putting the problem in epigraph form (see Section 3.3.2.6).

While all of them correspond to lemmas (except for renaming, which is a meta-level transformation), some of them are conveniently wrapped as tactics for better usability. For example, `change_of_variables` tries to prove that a given change of variables is valid automatically and, if successful, applies the appropriate equivalence lemma.

The introduction will conclude with an overview of related work and an outline of contributions. Then, we will introduce CvxLean in Section 3.2. The rest of the chapter is split in three: Section 3.3 discusses equivalences, Section 3.4 discusses reductions, and Section 3.5 discusses relaxations. For every relation and transformation defined, we will show how it can be realized in CvxLean. Section 3.6 provides a summary of the properties of each transformation. While this chapter focuses on relations between optimization problems and relation-preserving transformations, it also serves as a first overview of CvxLean. By introducing some simple examples of problems and transformations, we aim to give the reader a practical how-to guide of the tool, which will be useful for the rest of the thesis.

3.1.1 Related work

This section considers related work from a broader perspective than transformations in convex optimization. First, we look at tools at the intersection of proof assistants and mathematical software, placing our tool in a larger context. Then, we discuss some convex optimization formalization work done using proof assistants.

There is a long history of work on building bridges between computer algebra systems (CAS) and interactive theorem provers. For example, interfaces with Maple have been built from Isabelle [BHC95], HOL [HT98], and Coq [DM05]. Another widely used system is Mathematica, which has been linked to HOL Light [SDAT15] and Lean 3 [LW22]. It is worth noting that both Maple's and Mathematica's features span well beyond computer algebra, and both support different flavors of numerical computation and statistical modeling. Ballarin and

Paulson give a nice overview of the beneficial synergies between the two components [BP99], also describing a connection between Isabelle and Sumit. A CAS provides efficient algorithms to manipulate algebraic expressions at the cost of potentially giving unsound results. Introducing a theorem prover in the loop can make some soundness concerns disappear as formal proofs are required. Factorization is a good example that clearly illustrates how these two systems may interact, where the (irreducible) factorization suggested by the CAS is then checked to equal the original expression in the ITP [Dav23, § 3]. More intricate levels of coordination are also possible, where reconstructing a proof may rely on several interactions with the CAS.

In the spirit of using ITPs in combination with efficient external tools, there is much interest in linking ITPs with automated theorem provers (ATPs). In this case, the motivation is not so much ensuring soundness but rather extending the proof automation capabilities of the proof assistant. Hammers [BKPU16], including Isabelle's Sledgehammer, are a clear example of this approach. They have become complex pieces of software that combine many different approaches, but the core idea originally was to use first-order provers such as Vampire to find a proof and reconstruct it in the ITP. Satisfiability Modulo Theories (SMT) solvers are another important class of ATPs that encode many theories of interest for ITP users (e.g., linear and non-linear arithmetic). An example of one such interface is `SMTCoq` [EMT+17] as well as the ongoing `lean-smt`[1] effort. All the interfaces discussed above require re-constructing a proof, which usually cannot be built directly from the ATP's output, or doing so would result in performance hits. Instead, one often thinks of the ATP as providing hints and/or the proof structure.

While the interface between `CvxLean` and external convex optimization solvers is guided by the philosophy of using an ITP to define the specification and to be the main orchestrator, delegating computationally expensive tasks to specialized tools, there is a fundamental difference in that the certificate given by the solver is not verified. We argue that there are many benefits to using an ITP despite not ensuring full end-to-end verification (see, for example, our answer to **(RQ2.2)** in Section 7.1). Defining the problems in Lean using formal mathematical functions gives us a well-defined mathematical object we can transform and reason about. Verifying the transformations to conic form guarantees full correctness in the

[1] `https://github.com/ufmg-smite/lean-smt` (accessed 2024-01-17).

modeling phase, so the conic solver is the only source of any error in the solution.

There is one project developed in Coq, ValidSDP [RVS18], where properties about the numerical output of a convex optimization solver are formally verified in the ITP. Their tool is specialized to semidefinite programs, their main application being solving real arithmetic inequalities via sum-of-squares. Their focus is not on creating an environment for users to model and transform problems but rather on numerical verification. We discuss their approach in more detail in Section 4.1.1.

Another related project is SciLean[2], a framework for scientific computing written in Lean 4, with support for automatic differentiation as well as a variety of symbolic and numerical methods. It does not rely on any external tool, but its design is similar to ours in that high-level transformations are verified, whereas low-level computations depend on an unverified translation from real to floating-point expressions.

Finally, another line of related work concerns formalizations related to convex optimization in interactive theorem provers. The Isabelle [NPW02] HOL-Analysis library 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 [Gre11] in the Archive of Formal Proofs includes properties of convex functions. Lean's mathlib [The20] includes a number of fundamental results, 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 [AK19]. We do not know of any project that has formalized the notions of reduction, equivalence, or relaxation between optimization problems.

3.1.2 Contributions

The main contribution of CvxLean described in this chapter is the infrastructure to transform and reason about optimization problems in Lean. To the best of our knowledge, there is no verified framework that allows users to model optimization problems at the degree of generality that we do. Our robust library of definitions allows us to encode many interesting problems and transform them

[2] https://github.com/lecopivo/SciLean (accessed 2024-01-17).

using some common transformations in convex optimization. Moreover, these transformations are verified to be relation-preserving and generate the appropriate correctness witness.

Some preliminary work that led to the development of this chapter can be found in our TACAS paper [BFMA23]. In particular, we discuss strong equivalences, "reductions", the `reduction` command, and some simple "user-defined reductions".

The focus of the paper is the verified DCP algorithm presented in Chapter 4, allowing us to solve problems. As I discuss in Section 4.1.2, my contributions at that point were the interface with the solver and the creation of an initial benchmark of problems, which also required extending other parts of the tool as needed. The formal definitions of optimization problems, solutions, and strong equivalences were Bentkamp's work, as well as the "user-defined reductions" section [BFMA23, § 6], which is introduced as further motivation for using an ITP in this domain. In this chapter, I elaborate on how I have taken these parts further. Since the TACAS paper, the following have been my main contributions:

- Re-defining reductions. Originally, reductions were defined as "solution maps", which could not be used directly for retrieval. Note that to use a map $\text{sol}_Q \to \text{sol}_P$, we need to *prove* that the point is a solution, which we cannot easily do in general.

- Related to the point above, I added support for *computable* retrieval maps. In this way, the solution point output by the low-level solver can seamlessly be mapped back to the original problem.

- Defining equivalences and relaxations. Each of these relations has an accompanying command and tactics. The way in which the commands work is new. All the tactics have been re-designed to be compatible with weaker relations. For example, any equivalence-preserving tactic can be used in the `reduction` command.

- The above required extensive work on the metaprogramming side and was developed in a way so that it is easy to define new relation-preserving transformations.

- Proving properties of these relations, particularly regarding how they relate to each other.

- Developing the `change_of_variables` tactic.

- Formalizing other transformations, such as some pre-compositions, rewrites, or those involving slack variables.

3.2 CvxLean overview

The type of optimization problems, corresponding to definition 2.2.1 is defined in Lean as follows.

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

This definition is intentionally as general as possible. Currently, `D` is always a real vector space, and `R` is always $\mathbb{R}$. However, a lot of the machinery could be applied in the future to other domains, for instance, integer problems.

Given `p : Minimization D R`, writing definitions 2.2.2 and 2.2.3 in Lean is straightforward.[3]

```
variable {D R : Type} [Preorder R] (p : Minimization D R)

def feasible (x : D) : Prop := p.constraints x
def optimal (x : D) : Prop :=
  p.feasible x ∧ ∀ y, p.feasible y → p.objFun x ≤ p.objFun y
```

Both `D` and `R` are implicit, as they can always be inferred from `p`. Note that we require `R` to be a preorder (reflexive and transitive relation) rather than, say, a partial or linear order. It is the minimal requirement to define optimality in a meaningful way. This will be clearer when we discuss equivalences, reductions, and relaxations and their desired properties.

The set of solutions (definition 2.2.4) is defined as the following type.

[3]The notation `p.feasible` means `feasible p`. It works because these definitions are in the `Minimization` namespace (which we do not show), so we could also write `Minimization.feasible p`. In general, if we have a definition of the form `T.a` that takes `t : T` as an argument, Lean supports writing `t.a` to mean `T.a t`.

```
structure Solution where
  point : D
  isOptimal : p.optimal point
```

It can be seen as the subtype of `D` consisting of terms that satisfy the optimality predicate for `p`.

Next, we show how problems are defined in the framework. Lean 4's rich macro system [UdM20] allows us to define complex notation and gives us full control over how syntax is elaborated. Consider the following linear problem `p`.

```
optimization (x y : ℝ)
  minimize 40 * x + 30 * y
  subject to
    c₁ : 12 ≤ x + y
    c₂ : 16 ≤ 2 * x + y
```

This is a term of type `Minimization (ℝ × ℝ) ℝ`. It is definitionally equal to:

```
Minimization.mk
  (fun p => 40 * p.1 + 30 * p.2)
  (fun p => 12 ≤ p.1 + p.2 ∧ 16 ≤ 2 * p.1 + p.2)
```

As we can see, our syntax allows users to name the different components of the domain and use the variable names in the objective function and constraints, which is much more intuitive than the definition with projections above.

These names are stored as metadata in the domain type. From `(x y : ℝ)`, we build `[ℝ, `x] × [ℝ, `y]`, where we are using custom notation to show the associated metadata, which is usually hidden. Importantly, variable identifiers must be unique. The "`` ` ``" is Lean's notation for names, which, for all our purposes, we can think of as strings. The custom elaborator is then responsible for replacing them with the appropriate projection (e.g., from the domain type, it can be inferred that `x` corresponds to the first projection). Constraint names are also stored as metadata in each constraint's expression, e.g., `[12 ≤ p.1 + p.2, `c₁]`. These will be useful later on when we want to transform specific constraints. We also define a delaborator so that problem expressions are shown to the user in our custom syntax.

In this case, we can solve the problem directly. It is easy to see that $x^* = 4$

and $y^* = 8$. Proving `p.optimal ⟨4, 8⟩` in Lean is an easy exercise (for instance, using `linarith`).

3.3 Equivalences

The next definition allows us to prove that two optimization problems are equivalent by finding maps between the domains of the problems with certain properties.

Definition 3.3.1 (Equivalence)**.** Let P be a minimization problem defined over a domain D and let Q be defined over E. We say that P and Q are *equivalent* if there exist maps $\varphi : D \to E$ and $\psi : E \to D$ such that:

(φ_{opt}) $\forall x.\ \text{optimal}_P(x) \Rightarrow \text{optimal}_Q(\varphi(x))$.

(ψ_{opt}) $\forall y.\ \text{optimal}_Q(y) \Rightarrow \text{optimal}_P(\psi(y))$.

We write $P \equiv_{\varphi,\psi} Q$, or simply $P \equiv Q$ if the maps are clear from the context.

Note that no algebraic properties are required on the maps. A natural question is: why are they not inverses of each other? That would not make sense in this context as equivalent problems may not have isomorphic domains; think, for example, of adding an extra variable. They also do not preserve, in general, any structural properties of the vector space domains, i.e., they are not linear maps. One could also consider monotonicity, assuming an order on the domains, which is already an issue as there could be several sensible possibilities (e.g., for $\mathbb{R}^2$, we can take the lexicographic or the product order). Regardless, they are not always monotone or antitone; for instance, $x \mapsto x^2$ could be a valid map.

They are only supposed to be solution-retrieval maps in both directions (i.e., *reductions*, which we explain in Section 3.4). This is exactly what convex optimization experts have in mind when they think of equivalence, as seen from the following excerpt from Boyd and Vandenberghe [BV04, § 4.1.3]:

> In this book we will use the notion of equivalence of optimization problems in an informal way. We call two problems equivalent if from a solution of one, a solution of the other is readily found, and vice versa. (It is possible, but complicated, to give a formal definition of equivalence.)

The formal definition that they do not show is definition 3.3.1, with one subtlety. When they say "readily found", they refer to the fact that these maps need to be

computable (in some sense) and polynomial-time. The fact that most real numbers are non-computable seems to be an issue. This can likely be circumvented by requiring suitable computable rational approximation procedures. This explains why it would be "complicated" to give a fully formal definition.

In our work, we ignore these issues. Our definition does not impose any restrictions on the computability or complexity of the maps, and we allow any function (as a Lean term) with the correct domain and range (and satisfying the conditions). For practical purposes, that is never an issue. Firstly, the way in which these maps are built is usually from a restricted set of well-understood functions. Secondly, in practical scenarios, when we want to compute with them, we use floating-point approximations so we lose any relevant algebraic properties anyway (floats are not even a pre-order).

A stronger notion is often used internally in `CvxLean`. Most transformations preserve this strong equivalence, and proving this stronger notion rather than the above one is often simpler. As shown below, strong equivalences also require the feasible sets to be mapped back and forth and assume a degree of similarity between the objective functions to ensure the bounding properties.

Definition 3.3.2 (Strong equivalence)**.** In the same setting as above with f and g the objective functions of P and Q, respectively, we say that P and Q are *strongly equivalent* if there exist maps φ and ψ such that:

$$(\varphi_{\text{feas}})\ \ \forall x.\ \text{feasible}_P(x) \Rightarrow \text{feasible}_Q(\varphi(x)).$$

$$(\psi_{\text{feas}})\ \ \forall y.\ \text{feasible}_Q(y) \Rightarrow \text{feasible}_P(\psi(y)).$$

$$(\varphi_{\text{bound}})\ \ \forall x.\ \text{feasible}_P(x) \Rightarrow g(\varphi(x)) \leq f(x).$$

$$(\psi_{\text{bound}})\ \ \forall y.\ \text{feasible}_Q(y) \Rightarrow f(\psi(y)) \leq g(y).$$

We write $P \equiv'_{\varphi,\psi} Q$ or $P \equiv' Q$.

Bentkamp came up with the definition above by examining equivalent problems generated by the DCP procedure. His goal was to distill verification conditions on φ and ψ that were always true, and that would result in simpler proof certificates. Another way to look at it is as relaxations in both directions, as proved in Section 3.5.

It would also be interesting to consider other notions of equivalence, for example, by requiring (φ_{feas}), (ψ_{feas}), (φ_{opt}), and (ψ_{opt}). Understanding better what

that would imply is, in great part, the purpose of our discussion in Section 3.6. As we will see there, most transformations we are interested in preserve strong equivalences. We will also look at cases where the "$\leq$" in (φ_{bound}) and (ψ_{bound}) could be replaced by "=", which can be seen as an even stronger notion.

We proceed by proving some relevant properties. First, it is easy to show that both notions of equivalence are indeed equivalence relations. We have proved that in Lean, as well as the rest of the arguments in this chapter.

Next, we show that $P \equiv'_{\varphi,\psi} Q$ implies $P \equiv_{\varphi,\psi} Q$. We need to argue that (φ_{opt}) follows from the strong equivalence conditions. Suppose x is optimal in P (hence, also feasible). Then, for any y feasible in Q:

- $g(\varphi(x)) \leq f(x)$, by (φ_{bound});
- $f(x) \leq f(\psi(y))$, by optimality of x applied to $\psi(y)$, which is feasible because of (ψ_{feas}); and,
- $f(\psi(y)) \leq g(y)$, by (ψ_{bound}).

Thus, $g(\varphi(x)) \leq g(y)$. Moreover, $\varphi(x)$ is feasible, which follows from (φ_{feas}), so it is optimal. A similar argument can be used to prove (ψ_{opt}).

Note that this means that, instantiating y with $\varphi(x)$, all the intermediate inequalities become equalities and, hence, $f(x) = g(\varphi(x))$. So if x is optimal, $\varphi(x)$ is not only optimal in Q, but it has the same value as x in P, which is one way in which this is a *stronger* notion. The other way is by requiring the maps to preserve feasibility.

An important fact is that equivalences do not preserve curvature properties. A convex, DCP convex, and non-convex problem could be equivalent, e.g.:

$$\begin{pmatrix} \min & x \\ \text{s.t.} & 1 \leq x \leq e \end{pmatrix} \equiv \begin{pmatrix} \min & \log(x) \\ \text{s.t.} & 1 \leq x \leq e \end{pmatrix} \equiv \begin{pmatrix} \min & \log(x)^2 \\ \text{s.t.} & 1 \leq x \leq e \end{pmatrix},$$

The first problem is linear and thus DCP convex, the second is not convex, and the third is convex but not DCP since it is a composition of a concave increasing and convex increasing function. We show their plots in Figure 3.1. Equivalences are clear since the unique solution to all problems is $x^* = 1$, so we may choose φ and ψ to be the identity map. This is also an instance of pre-composing the objective function with a monotonic function (see Section 3.3.2.4).

Interestingly, the first two problems are not strongly equivalent. Take the first equivalence. For it to be a strong equivalence, for any x such that $1 \leq x$, we

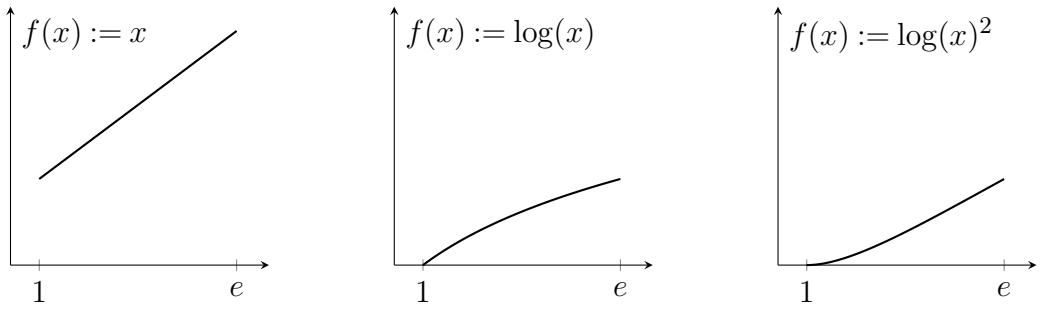


Figure 3.1: Plots of three equivalent problems.

would need $1 \leq \psi(x)$, by (ψ_{feas}). Also, by (ψ_{bound}), $\psi(x) \leq \log(x)$, which would imply $1 \leq \log(x)$, but $\log(1) = 0$, so we arrive at a contradiction. The second two problems are strongly equivalent but not through the identity maps. A possibility is $\varphi(x) := \exp\left(\sqrt{\log(x)}\right)$ and $\psi(x) := \exp\left((\log(x))^2\right)$.

The key difference is that strong equivalences are sensitive to the values of the respective objective functions. If we adjust the first problem slightly, we obtain the following chain:

$$\begin{pmatrix} \min & x \\ \text{s.t.} & 0 \leq x \leq 1 \end{pmatrix} \equiv' \begin{pmatrix} \min & \log(x) \\ \text{s.t.} & 1 \leq x \leq e \end{pmatrix} \equiv' \begin{pmatrix} \min & \log(x)^2 \\ \text{s.t.} & 1 \leq x \leq e \end{pmatrix},$$

The first strong equivalence can now be established via $\varphi(x) := \exp(x)$ and $\psi(x) := \log(x)$. In particular, this shows that strong equivalences do not preserve curvatures either.

3.3.1 An environment for user-guided transformations

Users of `CvxLean` can build equivalences directly by providing the maps and proving the proof obligations. This can be tedious, so we provide mechanisms to build equivalences conveniently. We also want to allow users to transform problems (preserving equivalence) without necessarily knowing the target problem beforehand. Moreover, in many use cases, one wants to combine several equivalences in successive steps. Every step corresponds to a tactic.

Conceptually, every equivalence-preserving tactic can be regarded as a map $P \mapsto (Q,\ \mathit{proof} : P \equiv Q)$. This structure is reminiscent of LCF-style tactics for goal-directed proofs [Pau83], where the proof is called a *validation* witnessing the correctness of the tactic. While we call it a "proof", note that the equivalence witness is actually a pair of optimality-preserving functions. Generally, a tactic

results in a list of subgoals; in our case, every tactic results in exactly one subgoal. Another unique aspect of our tactics is that they are *generative*, in the sense that they create a new term: the optimization problem Q. This differs from standard (or *refinement*) tactics, which refine a goal into zero or more subgoals.

We wish to chain these tactics so that applying them sequentially as follows

$$\begin{aligned} &tac_1 : P \mapsto (P,\ proof_1 : P \equiv P_1) \\ &tac_2 : P_1 \mapsto (P_2,\ proof_2 : P_1 \equiv P_2) \\ &\quad\vdots \\ &tac_n : P_{n-1} \mapsto (P_n,\ proof_n : P_{n-1} \equiv P_n) \end{aligned}$$

results in a new problem P_n and a proof $P_0 \equiv P_n$. Clearly, the proofs can be combined by transitivity of $\equiv$.

The `equivalence` command is the environment where a sequence of these user-guided equivalence-preserving tactics can be applied. It looks as follows.

```
equivalence eqv/q : p := by
  tac₁
  tac₂
  ...
  tacₙ
```

The user enters tactic mode when the cursor is placed right after the `by` keyword. The original problem is `p`, which exists in the environment, and `q` and `eqv` are user-provided names to identify the resulting problem and equivalence witness, respectively. The command runs `tac₁; ...; tacₙ` and, if successful, adds the terms `q` and `eqv : p ≡ q` to the environment. Before any tactic is applied, the goal to be solved looks as follows.

```
⊢ p ≡ ?q
```

The "`?`" symbol means that `?q` is a metavariable. The idea is that tactics build an equivalence from which the right-hand side can be inferred. Every built-in equivalence-preserving tactic application is internally preceded by a transitivity application so that as the tactic closes an equivalence goal, another equivalence goal with the transformed left-hand side is added. We show how this happens in one transformation step, i.e., one tactic application. Suppose `tac` proves `p ≡ r`

(this corresponds to $P \mapsto (R,\ proof : P \equiv R)$ in our conceptual setting earlier, where the "creation" of a new problem R is implicit from the proof).

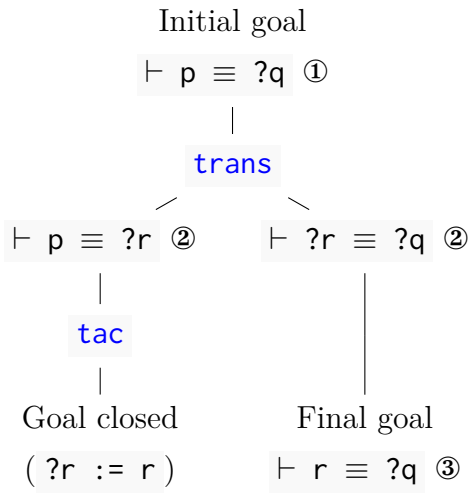


Figure 3.2: Tactic states in one equivalence step.

The strategy depicted in Figure 3.2 relies on the fact that proving the first subgoal opened by transitivity instantiates the metavariable `?r`. To be more precise, we indicate with number tags the order in which goals appear. The transition from ② to ③ is caused by applying the generative tactic `tac` to `⊢ p ≡ ?r`, which proves the equivalence and implicitly generates `r`, which replaces `?r`, thus also changing the other remaining goal. From the point of view of the user, the result is that the left-hand side of the equivalence changes as tactics are applied.

To see how several steps are combined, consider once again a sequence of tactics $\texttt{tac}_i$. We show, in comments, how the intermediate goals look like.

```
equivalence eqv/q : p := by
  -- ⊢ p ≡ ?q
  tac₁
  -- ⊢ p₁ ≡ ?q
  tac₂
  ...
  -- ⊢ pₙ₋₁ ≡ ?q
  tacₙ
  -- ⊢ pₙ ≡ ?q
```

The final goal at the end of the equivalence command ($\texttt{p}_n \equiv \texttt{?q}$) is automatically closed by reflexivity. This instantiates the right-hand side metavariable

(`?q`) with the last left-hand side term (in this example, `?q := p`$_n$). The term is added to the environment as a definition named `q`. The equivalence witness is readily obtained by elaborating the whole `by` block, which is a term of type `p ≡ q`. As explained, this is added to the environment as a definition named `eqv`.

If users want to apply tactics outside of the built-in ones defined in `CvxLean`, which take care of goal management and visualization, they can do so as follows:

```
equivalence eqv/q : p := by
  equivalence_step =>
    ... -- any tactic sequence
```

Preceding the tactic by `equivalence_step` ensures that transitivity is applied and the correct goal is shown. For example, suppose a user proves the following lemma: `one_step : p ≡ r`. Writing `apply one_step` inside the `equivalence` command will close the goal and will not allow the user to apply further steps, which might not be the intended behavior. Our proposed solution is to write `equivalence_step => apply one_step`, which will instead show a new goal `r ≡ ?q`. It is the user's responsibility to ensure that custom tactic blocks written in this way result in valid equivalence proofs and do not leave goals open. Otherwise, the step will fail.

Usually, the reason for establishing that a starting problem `p` is equivalent to `q` is to solve `q` and map back the solution to `p` (via ψ). Since `eqv` is in the environment, we also have `eqv.psi`, as `psi` is a field of the `Equivalence` structure. However, if `p` or `q` are defined over the real numbers, this map will be non-computable. Indeed, Lean cannot generate executable code for any expression involving $\mathbb{R}$. As explained in Section 4.7.3, we have a mechanism that turns real expressions into floating-point expressions (in an unverified way). We use this to build a computable backward map from `eqv.psi`. To instruct the system to build this map, a user can write `equivalence*` instead of `equivalence`. This will add the definition `eqv.backward_map` to the environment. Its type is the floating-point translation of the type of `eqv.psi`. For instance, if `eqv.psi : ℝ × ℝ → ℝ`, then `eqv.backward_map : Float × Float → Float`, where `Float` is the type of machine floats. It can be evaluated in the usual way, for example:

```
#eval eqv.backward_map ⟨1, 0⟩ -- computes a value
```

3.3.2 Equivalence-preserving transformations

The rest of the section comprises a series of equivalence-preserving transformations [BV04, §4.1.3]. For each of them, we provide an example and show how they can be used as a tactic.

3.3.2.1 Change of variables

One of the most common transformations when modeling optimization problems is performing a change of variables. We make the definition precise below.

Definition 3.3.3 (Change of variables function)**.** A function $c : E \to D$ is said to be a *change of variables function* if we have the following data:

- $c^{-1} : D \to E$,
- a predicate condition_c over D, and
- a proof of the following property: $\forall x.\ \text{condition}_c(x) \Rightarrow c\left(c^{-1}(x)\right) = x$.

In `CvxLean`, this is defined as a type class (see Section 2.1.2) where common change of variables functions are registered as instances. The key "type" of the type class is `E → D`, and instances are terms (functions), which is slightly unusual; it is more common that instances are also types. In this setting, type class resolution will pattern-match terms. We give an example of an instance below.

```
instance : ChangeOfVariables rexp :=
  { inv := log
    condition := fun x => 0 < x
    property := fun _ hx => exp_log hx }
```

Here, `rexp` is Lean's exp(·). There are also instances that instruct the system how to combine them (e.g., by composing them or by projecting them). While this approach works for simple expressions, there are cases in which pattern-matching fails. To mitigate that, in our implementation, we do some pre-processing before trying to infer the instance. Alternatively, one can always define a new instance, which is often straightforward.

The idea is that if the condition of the change of variables function is satisfied in the feasible set, it yields an equivalence.

Lemma 3.3.1 (Changes of variables are strong equivalences)**.** Let $P := (f, cs)$ be a minimization over D and $c : E \to D$ a change of variables function. Suppose that for all x feasible in P, we have condition$_c(x)$. Then $(f, cs) \equiv' (f \circ c, cs \circ c)$.

Proof. Let $\varphi := c^{-1}$ and $\psi := c$. It is immediate that ψ satisfies its feasibility and bounding properties (ψ_{feas}) and (ψ_{bound}). The requirements on φ follow from the change of variables property of c, which can be used as the condition is satisfied in the feasible set by assumption. □

Finally, we give a simple example that shows how this is realized as a tactic that can be used to build equivalences. As a small remark, notice that we provide the problem directly after " `:` " instead of an identifier for a problem; indeed, any term of type `Minimization D R` is a valid initial problem.

```
equivalence eqv₁/p₂ :
    optimization (x y : ℝ)
      minimize x * y
      subject to
        c₁ : 0 < x
        c₂ : 1 ≤ y := by
  change_of_variables! (u) (x ↦ rexp u)
  change_of_variables! (v) (y ↦ rexp v)
```

The tactic also takes an argument (`u` and `v` in the example above) to rename the optimization variable appropriately, as one usually does in pen-and-paper changes of variables. Moreover, a simplification step pushes function composition inside the expression. The " `!` " instructs the system to try to remove trivial constraints after the transformation. In this case, $0 < \exp(u)$ can be removed. This code snippet adds `p₂` , shown below, to the environment.

```
optimization (u : ℝ) (v : ℝ)
  minimize rexp u * rexp v
  subject to
    c₂ : 1 ≤ rexp v
```

3.3.2.2 Conversion mode

Equality and if-and-only-if rewrite rules applied to any of the subexpressions of the minimization problem result in a strongly equivalent problem. This is clear for unconditional rewrites, as they result in equal problems (after applying propositional extensionality and the appropriate congruence rules). In that case, a convenient way to apply these rewrites is through the tactic conversion (`conv`) mode. As explained in Section 2.1.3, it allows us to travel into any sub-expression, even those inside the body of λ-terms. The goal is replaced by said sub-expression, and rewrites can be applied. We have custom `conv` commands that allow us to navigate to the objective function or particular constraints. We modify p_2 from the previous section.

```
equivalence eqv₂/p₃ : p₂ := by
  conv_obj =>
    -- u : ℝ, v : ℝ | rexp u * rexp v
    rw [← exp_add]
    -- u : ℝ, v : ℝ | rexp (u + v)
  conv_constr c₂ =>
    -- u : ℝ, v : ℝ | 1 ≤ rexp v
    simp
    -- u : ℝ, v : ℝ | 0 ≤ v
```

The " `|` " notation indicates that we are in conversion mode. Hence, what we see is, technically, not a proof goal (which would be indicated with " $\vdash$ ") but a sub-expression that we can modify with a restricted set of tactics, usually rewrites or simplification tactics. The lemma `exp_add` (reversed) rewrites $\exp(v)\exp(v)$ into $\exp(u+v)$. Simplifying the constraint c_2 transforms $1 \leq \exp(v)$ into the equivalent $0 \leq v$. The resulting problem p_3 looks as follows:

```
optimization (u : ℝ) (v : ℝ)
  minimize exp (u + v)
  subject to
    c₂ : 0 ≤ v
```

3.3.2.3 Conditional rewrites

In many situations, rewrites only apply within the feasible set and need to use constraints to prove the conditions. In that case, one can use `rw_obj` and `rw_constr`. From the user's point of view, they behave similarly to `conv_obj` and `conv_constr` but also add some constraints as hypotheses.

```
equivalence eqv/q :
    optimization (x : ℝ)
      minimize sqrt (x ^ 2)
      subject to
        c₁ : 0 ≤ x := by
  rw_obj =>
    -- x : ℝ, c₁ : 0 ≤ x ⊢ sqrt (x ^ 2) = ?g x
    rewrite [rpow_two, sqrt_sq c₁]
    -- x : ℝ, c₁ : 0 ≤ x ⊢ x = ?g x
    rfl
  -- the objective function of the problem is x
```

The tactic above rewrites the objective function from $\sqrt{x^2}$ into x using the fact that $x \geq 0$. Note that `c₁` is in the local context. The target objective function is represented by the metavariable `?g`. Once the proof of the equality is finalized, it will be instantiated. We use `rewrite` instead of `rw` so that the result of the rewrite can be visualized; otherwise, since `rw` also uses a version of `rfl`, it would have immediately closed the goal.

Similarly, a tactic `rw_constr c` can be used for rewrites that depend on constraints other than `c`. More precisely, for a problem with constraints `c₁`, `c₂` and `c₃`, the goal after, say, `rw_constr c₁` will include `c₂` and `c₃` as hypotheses in the local context.

We can also specify the target term with the `into` keyword; for instance, we could have written `rw_obj into x => ...`. As we will see, conditional rewrites with a target term play a key role in Sections 5.3.4 and 5.6. For now, we consider the following example, where we have a constraint `c : log (y / x) ≤ 0` and the positivity of `x` and `y` can be established from the other constraints.

```
rw_constr c into (log (y / x) ≤ log 1) =>
  -- ... ⊢ log (y / x) ≤ 0 ↔ log (y / x) ≤ log 1
  simp
rw_constr c into (y / x ≤ 1) =>
  -- ... ⊢ log (y / x) ≤ log 1 ↔ y / x ≤ 1
  rw [log_le_log_iff] <;> positivity
rw_constr c into (y ≤ 1 * x) =>
  -- ... ⊢ y / x ≤ 1 ↔ y ≤ 1 * x
  rw [div_le_iff] <;> positivity
rw_constr c into (y ≤ x) =>
  -- ... ⊢ y ≤ 1 * x ↔ y ≤ x
  simp
```

As we can see, the difference is that `?g` has been replaced by the expression given by the user in each goal. This step-by-step approach is similar to `calc`. If the target expression is known, proving equality (or if-and-only-if) in this way is useful to take advantage of built-in automation as, sometimes, it suffices to simplify both sides. As seen in the example above, this is not always possible, and one needs to resort to applying specific lemmas.

3.3.2.4 Pre-compositions

Another common transformation targets function applications that appear in the problem and pre-composes them with other functions under some conditions. For example, we often want to re-scale functions, multiplying them by a positive scalar. We show how these pre-compositions (also called maps) yield equivalences and, later on, how they are realized in `CvxLean`.

Lemma 3.3.2 (Pre-composing the objective function)**.** Let $P := (f, cs)$ be a minimization over D with range R and $m : R \to R$. Suppose m is an order embedding in the image of f on the feasible set of P, i.e. for all P-feasible r and s, $f(r) \leq f(s)$ if, and only if, $m(f(r)) \leq m(f(s))$. Then $(f, cs) \equiv (m \circ f, cs)$

Proof. Let φ and ψ be identity maps. For (φ_{opt}), suppose x is optimal in P and take any y satisfying cs. We need to show that $m(f(x)) \leq m(f(y))$. Since x is optimal, we have that $f(x) \leq f(y)$, so the result follows from the property on m. Condition (ψ_{opt}) is proved in the same way by applying the condition on m in the opposite direction. □

This is a good example of a transformation that yields an equivalence but not a strong equivalence, as seen in our previous example with x, $\log(x)$, and $\log(x)^2$.

Lemma 3.3.2 has been formalized in CvxLean as well as particular recurrent instances. For example: applying the logarithm to the objective function given that the objective function is positive on the feasible set (which implies that the logarithm is an order embedding).

```
def map_objFun_log {f : D → ℝ} (h : ∀ x, cs x → f x > 0) :
  ⟨f, cs⟩ ≡ ⟨fun x => (log (f x)), cs⟩
```

Note that we use `def` instead of `lemma` (which is an alias for `theorem`). Morally, the reason is that equivalences live in `Type` and not `Prop`; they hold data. There is also a more subtle, although related, reason. To the kernel, the only difference between objects defined with the `theorem` keyword and objects defined with the `def` keyword is that the former are always non-computable. But, `map_objFun_log` is also non-computable because it depends on `ℝ`, so why use `def`? It turns out there is another difference in the elaborator. Theorems are always marked as irreducible; because of proof irrelevance, we never want to look unfold a theorem name to get the proof object. In our case, we do want to unfold equivalences. In particular, this happens when we build a computable backward map, where we simplify the expression `eqv.psi` to remove all the proof data and obtain a function.

It is also possible to map constraints or, more precisely, pre-compose functions involved in constraints. For example, if a problem has a constraint $f_i(x) \leq 0$, given a map g with the property that $g(x) \leq 0$ if, and only if, $x \leq 0$, we can replace the constraint by $g(f_i(x)) \leq 0$. The following lemma captures this equivalence.

```
def map_le_constraint_standard_form [Zero R]
  {cs' : D → Prop} {fi : D → R} {g : R → R}
  (hcs : ∀ x, cs x ↔ fi x ≤ 0 ∧ cs' x) (hg : ∀ x, g x ≤ 0 ↔ x ≤ 0) :
  ⟨f, cs⟩ ≡ ⟨f, fun x => g (fi x) ≤ 0 ∧ cs' x⟩
```

Note that instead of pattern-matching `cs` directly, we only require it to be provably equal to the required form. This is the usual way our lemmas are set up to provide more flexibility since $f_i(x) \leq 0$ could appear anywhere in $cs(x)$.

Similarly, constraints of the form $f_i(x) = 0$ can be mapped with maps satis-

fying $g(x) = 0$ if, and only if, $x = 0$.

```
def map_eq_constraint_standard_form [Zero R]
  {cs' : D → Prop} {hi : D → R} {g : R → R}
  (hcs : ∀ x, cs x ↔ hi x = 0 ∧ cs' x) (hg : ∀ x, g x = 0 ↔ x = 0) :
  ⟨f, cs⟩ ≡ ⟨f, fun x => g (hi x) = 0 ∧ cs' x⟩
```

3.3.2.5 Renaming

This is a unique kind of transformation as it does not change the underlying mathematical expression that a problem represents.

First, we have support to rename variables. For example, if we are in equivalence mode with a problem in `x`, `y` and `z`, we write `rename_vars [a, b, c]` to change the names of the optimization variables. This is achieved by changing the metadata attached to the domain type.

Similarly, we can rename constraints c_1 and c_2 to a_1 and a_2 applying `rename_constrs` $[a_1, a_2]$. Again, the tactic updates the label stored in the metadata of each expression representing each constraint.

This might be useful in situations where a transformation step has introduced variables and/or constraints with auto-generated names, and we want to rename them. Currently, this may only happen with DCP transformations.

3.3.2.6 Other basic transformations

Here, we list some other verified transformations currently used less frequently in our examples. However, they are often mentioned in the literature, so we report them here. Moreover, they are a useful point of reference for writing new tactics and proving other equivalences.

Let $P := (f, cs)$ over D with range R as usual. The following transformations preserve equivalences.

Re-ordering constraints First, we provide a convenient way to re-order constraints, which clearly leads to an equivalent problem. For a problem with constraints c_1 and c_2, we can write `reorder_constrs` $[c_2, c_1]$ to swap c_1 and c_2. We need to show $cs(x) \Leftrightarrow cs'(x)$, where the cs' is the re-ordered version of cs as instructed by the user. To do so, we use `tauto` internally, a tactic that proves propositional tautologies.

While the expression in the constraints technically changes, this transformation is more similar to renaming in that it is used for visualization purposes. In general, we want to offer users as much flexibility as possible in how the problem is displayed.

Removing constraints Removing a constraint is generally a relaxation, not an equivalence, as we explain in Section 3.5. Here, we refer to the situation in which a constraint is always true (e.g., $0 < \exp(u)$) or is implied by the other constraints. We can use the tactic `remove_constr` in those cases. Suppose a problem has constraints `c₁`, `c₂` and `c₃`. To remove `c₂`, we can write `remove_constr c₂ => ...` where the ellipsis is a formal proof that for any `x`, `c₂ x` can be proved from `c₁ x` and `c₃ x`. The tactic `remove_trivial_constrs` will attempt to remove each constraint using basic arithmetic tactics.

One clear scenario where this transformation is relevant is if the constraint being removed is not DCP-compliant. Any strict inequality, for example, needs to be removed if we want to solve the problem.

Eliminating equality constraints ([BV04, p. 132]) Suppose that there is some $g : E \to D$ such that for any x with $h(x) = 0$ there exists y such that $x = g(y)$. Assume that $cs(x)$ is provably equal to $h(x) = 0 \wedge cs'(x)$. Then P is equivalent to

$$\begin{aligned} &\text{minimize} && f(g(y)) \\ &\text{subject to} && cs'(g(y)), \end{aligned}$$

which is a problem over E.

Following the conditions above, the only way to build $\varphi : D \to E$ is by using the axiom of choice. For any x such that $h_i(x) = 0$, $\varphi(x)$ needs to be its g-inverse, so we need to extract an element from an existential proposition. Thus, φ is a noncomputable map regardless of the domain, so it cannot be translated into a computable map at the level of floats. This is, however, not an issue because, usually, we only care about ψ.

This transformation can be useful to discover further simplifications.

Introducing equality constraints ([BV04, p. 132]) Suppose $cs(x)$ is provably equal to $cs'(x, g(x))$ where $g : D \to E$ and cs' is a predicate over $D \times E$. Then P

is equivalent to

$$\begin{aligned} \text{minimize} \quad & f(x) \\ \text{subject to} \quad & y = g(x) \\ & cs'(x, y), \end{aligned}$$

defined over $(x, y) : D \times E$.

This is the reverse of the previous transformation, where a sub-expression $g(x)$ is factored out into a new equality constraint. It can be useful to make some constraints shorter and more readable, especially when $g(x)$ has a particular meaning in the problem.

Adding a slack variable ([BV04, p. 131]) Suppose $cs(x)$ is provably equal to $h(x) \leq 0 \wedge cs'(x)$. Then P is equivalent to

$$\begin{aligned} \text{minimize} \quad & f(x) \\ \text{subject to} \quad & 0 \leq s \\ & h(x) + s = 0 \\ & cs'(x), \end{aligned}$$

which is a problem over $(x, s) : D \times R$.

This transformation is used, for example, in the simplex method for linear programming. In general, it simplifies algebraic manipulation of linear constraints by only having to consider equalities.

Epigraph form ([BV04, p. 134]) It is easy to see that we can always introduce a new variable t and reformulate P into the equivalent following problem over $(x, t) : D \times R$:

$$\begin{aligned} \text{minimize} \quad & t \\ \text{subject to} \quad & f(x) \leq t \\ & cs(x). \end{aligned}$$

In this form, the objective function is linear, which can be useful in some cases to focus on transforming constraints.

3.4 Reductions

The notion of reduction of optimization problems is mentioned in Agrawal et al. [AVDB18, § 1.4], where they say reductions are the atomic rewriting unit in their

system to rewrite optimization problems. The word "rewrite" is used in the same sense as we use transformation here. They point to works in complexity theory, particularly on function problems [Pap07, § 10.3], where reductions are formally defined as follows.

> We say that a function problem A reduces to function problem B if the following holds: There are string functions R and S, both computable in logarithmic space, such that for any strings x and z the following holds: If x is an instance of A, then $R(x)$ is an instance of B. Furthermore, if z is a correct output of $R(x)$, then $S(z)$ is a correct output of x.

It is not immediately clear how this translates to optimization problems. First, note that A and B are classes of problems. For concrete optimization problems P and Q, we may think of them as classes with only one instance, i.e., $A = \{P\}$ and $B = \{Q\}$, so R simply maps P to Q. The more interesting map is S, which maps correct solutions of Q to correct solutions of P. There is room for interpretation as to what "correct" means, but we take it to mean that it maps optimal points to optimal points. This will also allow us to link reductions and equivalences. We have the following formal definition disregarding, as discussed earlier, computability and complexity concerns.

Definition 3.4.1 (Reduction)**.** Let P be a problem with domain D and range R and Q be a problem with domain E and range R. We say that P *reduces to* Q, or that Q is a *reduction of* P, if there exists $\psi : E \to D$ such that:

(ψ_{opt}) $\forall y.\ \text{optimal}_Q(y) \Rightarrow \text{optimal}_P(\psi(y))$.

We write $P \preceq_\psi Q$ or simply $P \preceq Q$.

The word "reduction" often has the connotation that the reduced problem is, in some sense, simpler. Our definition does not capture that; instead, we use "reduction" to mean that the reduced problem Q is at least as hard to solve as the original problem P. Practically, this means that if we have a solution of Q, we should be able to retrieve a solution for P. This is exactly our definition. In other words, we can lift ψ to a map $\text{sol}_Q \to \text{sol}_P$.

It is easily seen that $\preceq$ is reflexive and transitive. However, it is not symmetric; for instance, let P be any feasible problem and Q be infeasible, then $P \preceq Q$ but $Q \not\preceq P$.

The natural next question is how this relates to equivalences. Looking at definition 3.3.1, we can see that a reduction is just the "ψ part" of an equivalence. The proposition below follows immediately from this observation.

Proposition 3.4.1. $P \equiv Q$ if, and only if, $P \preceq Q$ and $Q \preceq P$.

Therefore, all equivalences give reductions (in both directions). However, an arbitrary reduction does not necessarily give an equivalence by the same argument that we used to justify that $\preceq$ is not symmetric and the proposition above.

Reductions are often what we care about in practice, as, in the end, one only solves the reduced problem. The solution of this problem must then be mapped back to, or reinterpreted in, the domain of the original problem. This is exactly what a reduction allows us to do. However, since any problem reduces to an infeasible problem, as argued above, one needs to be careful when proving that a problem reduces to another problem. In particular, if the solver tells us that the reduced problem is infeasible, that does not mean that the original problem is infeasible. For this reason, working at the level of equivalences is preferable. Nevertheless, in `CvxLean`, we support building reductions through the `reduction` command and let the users decide what relation they want.

```
reduction red/q : p := by ...
```

The command works the same way as `equivalence` and, as one would expect, any tactic applicable in equivalence mode also works here. In this case, the goal looks as follows.

```
⊢ p ≼ ?q
```

One good reason for using `reduction` instead of `equivalence` is that some transformations are reductions but not equivalences. We give an example below.

Recall Lemma 3.3.2 about mapping $m : R \to R$ to the objective function. If we only require m to be order-reflecting on the image of f in the feasible set of $P := (f, cs)$, i.e., for P-feasible r and s, if $m(f(r)) \leq m(f(s))$ then $f(r) \leq f(s)$. Then we obtain a reduction $(f, cs) \preceq (m \circ f, cs)$ that is not necessarily an equivalence. The proof follows immediately from the proof of Lemma 3.3.2.

3.5 Relaxations

In many cases, the initial problem is not convex and cannot be transformed into an equivalent convex problem. However, there are techniques that allow us to loosen some constraints and make them convex (or, simply, remove them), resulting in a problem with a larger feasible set that can be solved using convex optimization tools. This transformation is known as a *relaxation*. We define relaxations of optimization problems below, following the definition given by Klein and Young [KY09, 34-19].

Definition 3.5.1 (Relaxation)**.** Let P and Q be minimization problems with objective functions $f : D \to R$ and $g : E \to R$, respectively. We say that Q is a *relaxation* of P if there exists $\varphi : D \to E$ such that:

$$(\varphi_{\text{feas}}) \quad \forall x.\ \text{feasible}_P(x) \Rightarrow \text{feasible}_Q(\varphi(x)).$$

$$(\varphi_{\text{bound}}) \quad \forall x.\ \text{feasible}_P(x) \Rightarrow g(\varphi(x)) \leq f(x).$$

We write $P \succeq'_{\varphi} Q$ or $P \succeq' Q$.

One way in which this definition differs from other definitions of relaxation in the literature is that we do not require φ to be an embedding of feasible sets. The first reason is that Proposition 3.5.1 would not hold, so we would lose any link between (strong) equivalences and relaxations. Furthermore, we want to be able to say, for example,

$$\begin{pmatrix} \text{minimize} & x+y \\ \text{subject to} & x \geq 0 \\ & y \geq 0 \end{pmatrix} \quad \succeq' \quad \begin{pmatrix} \text{minimize} & x \\ \text{subject to} & x \geq 0 \end{pmatrix}.$$

Clearly, there are no injective maps between feasible sets.

Looking at definition 3.3.2, the following result is clear.

Proposition 3.5.1. $P \equiv' Q$ if, and only if, $P \succeq' Q$ and $Q \succeq' P$.

As with the other relations, we provide an environment in the form of a command to build relaxations.

```
relaxation rel/q : p := by ...
```

The goal displayed in the infoview is as expected.

```
⊢ p ≥′ ?q
```

We conclude this section by discussing some common relaxations. First, it is clear that removing any constraint always yields a relaxation. More interestingly, relaxations are often used to approximate non-convex objective functions or constraints. We focus on semidefinite relaxations (SDR) of quadratic constraints [LMS+10]. Let $C, A \in \mathcal{S}_+^n$, i.e., they are positive semidefinite $n \times n$ matrices, and $b \in \mathbb{R}$. We can show that

$$\begin{pmatrix} \text{minimize} & x^T C x \\ \text{subject to} & x^T A x \geq b \end{pmatrix} \quad \succeq' \quad \begin{pmatrix} \text{minimize} & \operatorname{Tr}(CX) \\ \text{subject to} & \operatorname{Tr}(AX) \geq b \\ & X \in \mathcal{S}_+^n \end{pmatrix}.$$

The optimization variables are $x \in \mathbb{R}^n$, and $X \in \mathbb{R}^{n \times n}$, respectively. The first problem is, in general, not convex (e.g., consider the set defined by $x^2 \geq 1$), but the second is convex. The relaxation is realized via the map $\varphi(x) = xx^T$. To see that this is valid relaxation, it suffices to use basic linear algebra identities and note that xx^T is positive semidefinite. Alternatively, we may note that we obtain a strong equivalence if we add $\operatorname{rank}(X) = 1$ as an extra constraint to the problem on the right-hand side. And the relaxation follows from removing said constraint.

Lagrangian relaxations are another important class of relaxations that we have not yet formalized. The idea is that constraints can be "brought into" the objective function multiplied by a penalty term (or Lagrange multiplier), thus resulting in a less constrained problem. For example, a linear constraint $ax \leq b$ may be removed, and the objective function $f(x)$ may be replaced by $f(x) + \lambda(ax - b)$, where $\lambda \geq 0$. This would also fit into our definition.

3.6 Properties of transformations

We have introduced several transformations and categorized them according to whether they are strong equivalences, equivalences, reductions, or relaxations. They are organized in a hierarchy as follows: $\equiv' \subseteq \equiv \subseteq \preceq$ and $\equiv' \subseteq \succeq'$. We have seen that strong equivalences do not preserve curvature properties; hence no relation preserves curvature properties. We have also seen that reductions do not preserve feasibility.

In this section, we analyze the forward and backward maps of each of the transformations we have presented in a fine-grained manner. For completeness, we also consider the DCP transformation from Chapter 4 and the automated pre-DCP transformation from Chapter 5.

We check whether they satisfy (φ_{opt}), (ψ_{opt}), (φ_{feas}), (ψ_{feas}), (φ_{bound}), and (ψ_{bound}). For the bounding properties, we distinguish between those that are always equalities and those that may be inequalities. We also specify whether the maps are identities. The summary is shown in Table 3.1, where the "✗" should be read as "not always".

Tr.	φ_{opt}	ψ_{opt}	φ_{feas}	ψ_{feas}	φ_{bound}		ψ_{bound}		φ	ψ
					$\leq$	$=$	$\leq$	$=$	id	id
COV	✓	✓	✓	✓	✓	✓	✓	✓	✗	✗
RW	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
$\mathbf{M}_{\text{obj}}^{\text{emb}}$	✓	✓	✓	✓	✗	✗	✗	✗	✓	✓
$\mathbf{M}_{\text{obj}}^{\text{ref}}$	-	✓	-	✓	-	-	✗	✗	-	✓
$\mathbf{M}_{\leq}$	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
$\mathbf{M}_{=}$	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
$\mathbf{E}_{=}$	✓	✓	✓	✓	✓	✓	✓	✓	✗	✗
$\mathbf{I}_{=}$	✓	✓	✓	✓	✓	✓	✓	✓	✗	✗
RN	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
RO	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
$\mathbf{RM}_t$	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
$\mathbf{RM}_n$	✗	-	✓	-	✓	✗	-	-	✓	-
SLK	✓	✓	✓	✓	✓	✓	✓	✓	✗	✗
EPI	✓	✓	✓	✓	✓	✓	✓	✗	✗	✗
SDR	✗	-	✓	-	✓	✗	-	-	✗	-
DCP	✓	✓	✓	✓	✓	✗	✓	✗	✗	✗
PRE	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓

Table 3.1: Properties of transformations.

The names of the transformations have been shortened as follows:

- **COV**: change of variables.
- **RW**: rewrites in the objective function or constraints (both conditional and

unconditional).

- $\mathbf{M}_{\text{obj}}^{\text{emb}}$: pre-composing the objective function with an order-embedding map.
- $\mathbf{M}_{\text{obj}}^{\text{ref}}$: pre-composing the objective function with an order-reflecting map.
- $\mathbf{M}_{\leq}$: pre-composing an inequality constraint in standard form with a non-positivity-preserving map.
- $\mathbf{M}_{=}$: pre-composing an equality constraint in standard form with a zero-preserving map.
- $\mathbf{E}_{=}$: eliminating an equality constraint.
- $\mathbf{I}_{=}$: introducing an equality constraint.
- **RN**: renaming variables or constraints.
- **RO**: re-ordering constraints.
- $\mathbf{RM}_t$: removing trivial constraints, that is, implied by other constraints.
- $\mathbf{RM}_n$: removing non-trivial constraints, hence a relaxation.
- **SLK**: introducing a slack variable.
- **EPI**: putting the problem in epigraph form.
- **SDR**: semidefinite relaxation.
- **DCP**: DCP transformation.
- **PRE**: pre-DCP transformation.

We can classify each transformation above according to the *strongest* relation it preserves as follows:

- Strong equivalence: **COV**, **RW**, $\mathbf{M}_{\leq}$, $\mathbf{M}_{=}$, $\mathbf{E}_{=}$, $\mathbf{I}_{=}$, **RN**, **RO**, $\mathbf{RM}_t$, **SLK**, **EPI**, **DCP**, and **PRE**.
- Equivalence: $\mathbf{M}_{\text{obj}}^{\text{emb}}$.
- Reduction: $\mathbf{M}_{\text{obj}}^{\text{ref}}$.
- Relaxation: $\mathbf{RM}_n$, and **SDR**.

As we can see, most of them preserve strong equivalences. Indeed, most of them would even work if the bounding properties were equalities. There are two exceptions: **EPI**, and **DCP**. These are actually closely related. Note that expanding a convex atom $f(x)$ usually introduces a new variable t (sometimes several variables are needed) and some constraints in conic form which are mathematically equivalent to $f(x) \leq t$. We can also see a pattern regarding those whose maps are the identity map. Those are precisely the transformations that do not "touch" the optimization variables. This is not the same as not changing the domain since, for example, a change of variables usually will not change the domain but gives the variables new meaning.

3.7 Summary

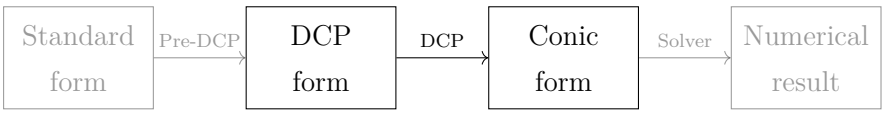
In this chapter, we have explained how problems are defined in CvxLean. We have defined equivalences, reductions, and relaxations and given examples showing how to construct each of them. Our focus has been on equivalences, showing a large number of common equivalence-preserving transformations, including changes of variables, rewrites, and maps. Equivalences will play a key role in the rest of the thesis, in particular, in chapters 4 and 5.

Chapter 4

A proof-producing DCP algorithm

4.1 Introduction

Recall Figure 1.1 from Chapter 1, used to introduce the different stages a convex optimization problem goes through. This chapter focuses on verifying the second step.



This is a crucial step in the pipeline. Indeed, verifying it was the initial motivation behind the `CvxLean` project.

The *canonization* to conic form is required to solve the problem. Let us begin by trying to understand why solvers take problems in conic form. There are many algorithms to solve convex optimization problems: gradient methods, ellipsoid methods, interior-point methods, etc. We consider only *interior-point methods* [NN94] here, which are the most popular. In particular, we focus on *primal-dual* interior-point methods [ART03, ADL$^+$11], which is what `MOSEK` [AA00], our chosen low-level solver, is based on. As the name indicates, these methods rely on the *dual problem*. The specific formulation is irrelevant to our work. The important observation is that the dual of a problem in conic form is much simpler, with a linear objective function and linear expressions in the cone membership constraints. This gives us a first hint as to why conic form is more suitable to compute with. We use the word "suitable" because, theoretically, one could formulate primal-dual methods to operate on problems in standard form. The first issue with that approach is that checking convexity is non-trivial, whereas

problems in conic form are convex by construction. Even if convexity could be somehow asserted, there are other reasons why conic form is preferred. To search for an optimal solution, primal-dual methods iteratively solve a set of (Newton) equations that characterize the so-called *central path*, which, under some conditions, is guaranteed to approach the optimal value. These equations come from perturbing the *Karush-Kuhn-Tucker (KKT) conditions* (or the *homogeneous model*), which may be regarded as intermediate optimization problems. We omit their definitions and only note that they are composed of expressions from the primal and dual problems. The point is that if the problem is in conic form, the equations solved in every iteration are linear, as well as the perturbation updates. This explains why solvers require the problems to be in conic form, as it makes the algorithms more efficient and robust.[1]

There is, however, a large gap between optimization problems stated in conventional mathematical notation and their equivalent conic forms. In some sense, this process resembles compiling a program to machine-level code. Our goal is to make this compilation step more rigorous, ensuring that the problem's semantics are preserved. Following the previous chapter, we will show that strong equivalence is preserved. Having the problem defined in a mechanical proof assistant such as Lean is an ideal setting to perform the necessary static analysis and check all side conditions that arise in the canonization process. The semantics of problems are precise and unambiguous since they are given by formal definitions. Our method analyzes the underlying expressions in the problem and uses this information to automatically transform the problem into conic form. It produces a proof of equivalence at the same time, which endows this fundamental transformation in convex optimization modeling with strong correctness guarantees.

Not every convex optimization problem can be immediately canonized to conic form. Some requirements must be met precisely to guarantee that the resulting cone program is equivalent to the original one. The approach that we use is disciplined convex programming (DCP) [GBY06], described in Section 2.2.2.

In the rest of this section, we cover some related work and outline our contributions. Then, we put forward some design principles that are unique to our work in Section 4.2, and that will help the reader understand many of the decisions that come after. Section 4.3 defines our version of the DCP atom library, extended

[1]For further details, Dr. Erling D. Andersen, the CEO of the company behind `MOSEK`, gave a presentation where he expands on why implementing algorithms on conic forms is preferred: `https://docs.mosek.com/whitepapers/func_versus_conic.pdf` (accessed 2024-03-04).

to include proofs of the necessary properties for each atom that can be used to prove equivalences. The associated curvature ruleset is defined in Section 4.4. In Section 4.5, we present the key procedure that transforms the problem. The verification of this algorithm is discussed in Section 4.6. All sections up to this point treat the transformation from a mathematical point of view. Section 4.7 covers how this is implemented in `CvxLean`. We also explain how the tool is connected to an external solver and a solution is retrieved, the verification of which is beyond the scope of this thesis and discussed as future work in Section 7.4.4.

4.1.1 Related work

DCP frameworks exist in many programming languages. The most popular of these is `CVXPY` [DB16], written in Python, which we use as our main point of reference. Their library includes a large number of atoms as well as canonizations for DGP and DQCP. It has been successfully used to encode many examples[2] from several domains. The original implementation of DCP was the MATLAB package `CVX` [GB, GB20], still widely used. Other implementations include `CVXR` [FNB20] in R and `Convex.jl` [UMZ+14] in Julia. A full list of related projects can be found on the `CVXPY` website.[3] Another popular framework is `YALMIP` [Lof04]. A lot of effort is put into these frameworks to try to ensure that the transformation is correct, however none of them construct formal proofs as we do in `CvxLean`. This gives us much higher confidence in the correctness of the transformation. While errors in the canonization step are rare, they may still occur (see Section 6.5). As these tools evolve and new features are added, the only assurance that no errors have been introduced is a limited number of tests. Moreover, a problematic canonization step might easily go unnoticed and only be detected when obtaining an unexpected solution. Having Lean check the canonization not only makes the process more robust but can also provide more meaningful and accurate explanations to the user if there is an issue with the original problem, e.g., a side condition cannot be proved, disallowing the canonization.

Even though numerical verification is not the focus of this chapter or this thesis, projects focusing on this aspect are related to ours as they are driven by the same goal: making convex optimization more reliable. For this reason, it is important to discuss some tools that motivate this line of work. `ValidSDP` [RVS18]

[2] `https://www.cvxpy.org/examples/index.html` (accessed 2023-12-14).
[3] `https://www.cvxpy.org/related_projects/index.html` (accessed 2023-12-14).

is a tool written in Coq that verifies the output of a semidefinite programming solver. The main goal is to prove real inequalities. The key idea is to use Rump's algorithm [Rum06] to verify positive semidefiniteness by reasoning about the maximum floating-point rounding error introduced. This allows them to prove the existence of a nearby exact solution without computing it explicitly. `VSDP` [Jan06, HJL12] is a MATLAB tool to perform rigorous computations of convex optimization results. It uses INTLAB [Rum98], a powerful framework for interval arithmetic which, amongst other features, allows for rigorous handling of floating-point errors. The tool is able to solve problems in conic form expressible with the positive semidefinite cone, the second-order cone, and the nonnegative orthant. They compute bounds on the objective function, verified solution enclosures, and verified infeasibility certificates. There exists some other work on using numerical certificates to prove real inequalities using sum-of-squares, notably some work done by Harrison in HOL Light [Har07].

4.1.2 Contributions

This chapter presents the first implementation of the DCP framework developed in a proof assistant and augmented to produce formal equivalence proof certificates. Our major contributions can be split as follows:

- *Extending atom declarations.* In `CvxLean`, every atom declaration includes proofs of relevant properties (see Section 4.3.3).
- *Building a relatively comprehensive atom library.* An overview of our atom library can be found in Appendix A. Developing it involved some formalization work to prove extra properties required for every atom.
- *Automating atom inference.* This allows users to define problems in a natural way, lifting the requirement to explicitly write problems using atoms, as discussed in Section 4.2.1.
- *Inferring and eliminating conditions.* In our setting, all side conditions[4] arising from this transformation need to be formally proved, which we do automatically. In some cases, these conditions are non-DCP-compliant (e.g., strict inequalities). One of our novelties is that we allow non-DCP-compliant constraints as long as we can "eliminate" them, i.e., prove that

[4]These come from domain restrictions of partial functions that appear in the problem.

ignoring them and canonizing the rest still yields an equivalent problem. See Sections 4.5.1 and 4.5.3 for further details.

- *Creating a proof-producing canonization procedure.* Its design is described in Section 4.5.2. Apart from outputting a problem in conic form, our canonization procedure seamlessly constructs complicated proof terms, combining the extra proofs attached to each atom in a carefully orchestrated way. This required extensive work on the metaprogramming side and is perhaps our most noteworthy contribution.

With our verified method, we have been able to canonize several examples from the literature, demonstrating that the extra rigor does not come at the cost of applicability.

This work is the focus of our TACAS paper [BFMA23]. Before that, Bentkamp and Avigad developed the first proof of concept for manually verifying DCP transformations [BA21]. When I joined the project, I worked mainly on connecting the tool to an external solver, `MOSEK` [AA00], in particular. I also formalized some atoms and built the first benchmark of examples, which required extending and refining some of the metaprogramming machinery. Bentkamp wrote the core code for the DCP algorithm. After the TACAS publication, I became the lead maintainer of `CvxLean` and improved the algorithm in several ways. The most relevant ones are the following:

- Improving automation for condition inference (see Section 4.5.3). The first version required an exact match, e.g., if `sqrt x` appeared in the problem, one of the constraints had to be `0 ≤ x`, whereas now the system tries to infer it from all the constraints so, for example, having the constraint `1 ≤ x` would also be valid.

- Enabling atom definitions to depend on previously defined atoms (see Section 4.6.3). This lifts the restriction on writing graph implementations of atoms in conic form. Instead, they can be written in a form canonizable to conic form using other atoms, making the definitions more concise and readable. This also involved generalizing the proof obligations required for each atom.

- Formalizing several new atoms, including `huber`, `quadOverLin`, `klDiv`, `geoMean`, `norm2`, `xexp`, `min`, `max`, `logSumExp`, `entr`, `powNegOne`,

and `powNegTwo`. This allowed us to encode more examples, which serve as evidence of the extensibility and applicability of the tool.

- Providing a detailed mathematical justification of the logic and correctness of the algorithm. While the TACAS paper provided some intuition and proof sketches, more work and space were needed for a fully rigorous exposition, which we present here.

4.2 Design principles

In this chapter, we will present our proof-producing DCP framework in great detail. This will involve several formal definitions, algorithms, and proofs. This section aims to prepare the reader for all the upcoming technical details by discussing two unique aspects of this work that have deeply influenced the design of the framework. Section 4.2.1 covers the role of expressions (and how they relate to atoms), and Section 4.2.2 covers our approach to proofs.

4.2.1 Atomizing expressions

As explained in Section 2.2.2, atoms are functions with known domain, range, curvature, and monotonicity at each argument.

In CvxLean, optimization problems are defined using expressions that involve pre-defined elementary arithmetic operations and definitions from mathlib. It is also possible to use custom definitions. This allows us to write problems conveniently, using familiar mathematical notation. One consequence of this is that we do not have full control over the language of expressions used. In other approaches, atoms are used as expression constructors, so the language of atoms is the language of expressions. In our case, the languages of atoms and expressions are not quite the same; often, an atom constructor corresponds directly to an expression constructor, but sometimes not. This means that there is not a one-to-one correspondence between an expression's AST and its *atom tree* (see Section 4.5.1). Atom trees are a fundamental building block in all DCP frameworks, and making one successfully ensures compliance with the DCP curvature rules.

For example, let us consider the expression $a \cdot \exp(a)$, where a is some other expression. It involves multiplication and the exponential function. However, it

also corresponds to a convex atom in our library (see Table A.5). Our approach makes use of discrimination trees in a crucial way to match expressions against atom expressions stored in the library. When traversing this expression, we need to do some extra work to find the atom it corresponds to. In this case, we will need to consider the following atoms:

- mul1, corresponding to the expression $c \cdot x$ when $c \geq 0$ is constant.
- mul2, corresponding to the expression $c \cdot x$ when $c \leq 0$ is constant.
- mul3, corresponding to the expression $x \cdot c$ when $c \geq 0$ is constant.
- mul4, corresponding to the expression $x \cdot c$ when $c \leq 0$ is constant.
- xexp, corresponding to the expression $x \cdot \exp(x)$.

Note that we distinguish the atom's name and the atom's expression. As we can see, there are four multiplication atoms (depending on the position and sign of the constant term). This is how we incorporate DCP's product-free rules into atom tree building. What is clear from this example is that depending on a and the context expression in which $a \cdot \exp(a)$ appears, different choices could be acceptable.

It will be useful to keep this distinction between atoms and expressions in mind as we explain how to declare an atom in Section 4.3 and, later on, when we show how to make atom trees in Section 4.5.1. Making atom trees will require the curvature calculation machinery from Section 4.4, which is designed to support atom tree *discovery* and is necessary since atom trees cannot be read off directly from the expression.

Automating atom tree discovery in this way is one of CvxLean's innovations.

4.2.2 Producing proofs

Another unique aspect of our DCP framework is that we aim to build proofs of (strong) equivalence between problems. To that end, there are certain properties that need to be proved per atom (see Section 4.3.3). We assume that proofs of these properties are stored (in the implementation, they are stored as proof terms). In Section 4.6, we justify the correctness of our approach, which, after defining φ and ψ as required by definition 3.3.2, involves proving four properties about them:

- (φ_{feas}), proved in Lemma 4.6.5.
- (φ_{bound}), proved in Lemma 4.6.6.
- (ψ_{feas}), proved in Lemma 4.6.8.
- (ψ_{bound}), proved in Lemma 4.6.9.

We present these proofs mathematically, assuming that atom proof obligations hold and using them at key moments. Conceptually, they justify the correctness of our approach and illustrate that the proof obligations that we require are sufficient to establish equivalence. However, it is important to remark that we never prove these lemmas in full generality in `CvxLean`. Instead, each of these lemmas corresponds to a function that, given a particular problem instance, will produce the appropriate proof term. The structure of these functions follows the reasoning patterns of the proofs.

4.3 Declaring an atom

This section defines formally all the information needed to declare an atom. One needs to give a concrete value to each of the definitions in this section to build an *atom declaration*. We will refer to each piece of information as a *field* in the atom declaration. We show a complete example in Section 4.3.4.

First, recall that an atom represents a function over some domain; this function is described in the atom's *signature* as explained in Section 4.3.1. Crucially, the signature includes its curvature and its monotonicity with respect to its arguments. One of the fundamental ideas of DCP is to associate a *graph implementation* (in conic form) to the atom. Section 4.3.2 explains how to give the atom an implementation. Finally, we define five *proof obligations* in Section 4.3.3 that are the key to making the DCP procedure produce proofs of equivalence. These are new and unique to our approach. Apart from the proof obligations, the rest of an atom declaration mostly follows Grant's atom library design [GBY06, § 7], and we will point out any differences as needed. As a first observation, the style in which we lay out the fields of an atom declaration is naturally more formal and verbose since they ultimately correspond to Lean expressions.

Fix an atom f of arity k, where f is the atom's name, which is treated as an identifier.

4.3.1 Signature

Definitions 4.3.1 to 4.3.7 form the atom *prototype* and *attributes*, as named in the original DCP formulation [GBY06, § 7.1]. We are particularly careful with the conditions on the input arguments. While domain restrictions are part of the original DCP formulation, we refine them by distinguishing between *background* and *variable* conditions. Using the same fields for both set and function atoms is also unique to our approach.

Definition 4.3.1 (Atom class)**.** $\mathsf{class}_f \in \{\textsc{Fn}, \textsc{Set}\}$. We distinguish between atoms that define a function and atoms that define a set. In practice, all atoms are functions, with atoms of class Set being those with range $\{\top, \bot\}$. However, the curvature rules with respect to the arguments are defined separately.

Atoms with $\mathsf{class}_f = \textsc{Set}$ are all represented using predicates and include "$=$", "$\leq$", and one for each of the cones listed in definition 2.2.10.

Definition 4.3.2 (Atom curvature)**.** If $\mathsf{class}_f = \textsc{Fn}$:

$$\mathsf{curv}_f \in \{\textsc{ConvexFn},\ \textsc{ConcaveFn},\ \textsc{AffineFn},\ \textsc{ConstantFn}\}.$$

The curvature labels can be ordered in an obvious way, corresponding to set inclusions of the sets that they represent: $\textsc{ConstantFn} \sqsubseteq \textsc{AffineFn} \sqsubseteq \textsc{ConvexFn}$ and $\textsc{AffineFn} \sqsubseteq \textsc{ConcaveFn}$, transitively closed. A special label UnknownFn will also be used in some definitions to indicate failure in deducing a curvature label; however, no atom should be labeled with it since it would never be used in the canonization. We have $t \sqsubseteq \textsc{UnknownFn}$ for all t, as expected.

If $\mathsf{class}_f = \textsc{Set}$, we only consider affine and convex sets; recall that there is no such thing as a concave set. Constant functions are fixed numerical values, and there is no analogous concept for sets. Therefore, in this case:

$$\mathsf{curv}_f \in \{\textsc{ConvexSet},\ \textsc{AffineSet}\}.$$

Again, it is useful to add a symbol UnknownSet to reason about failure. The order relation, in this case, is $\textsc{AffineSet} \sqsubseteq \textsc{ConvexSet} \sqsubseteq \textsc{UnknownSet}$.

Note that these are labels in an abstract domain that, a priori, tell us nothing about the curvature of the atom's underlying function or set (i.e., the atom expression as in definition 4.3.7). As it turns out, we never need to explicitly prove that expressions are convex, concave, or affine in order to build equivalent

problems in conic form. In a way, as we will see in Section 4.5.2, the canonization procedure checks convexity, but we do not emit a formal proof of it. Recall that our goal is to prove equivalence. An incorrect curvature label, however, will make the atom declaration fail. It determines the proof obligations for the atom, and these need to be synchronized with the curvature rules so that equivalence proofs work, as we make precise in Section 4.6.

Definition 4.3.3 (Atom domain)**.** An atom is defined over a domain as follows: $\mathsf{domain}_f := D_1 \times \cdots \times D_k = D$. In fact, D could be a dependent product, i.e., $(d_1 : D_1) \times (d_2 : D_2(d_1)) \times \cdots \times D_k(d_1, \ldots\ d_{k-1})$, so a component in the domain might depend on the other components.

For example, $(m : \mathbb{N}) \times \mathbb{R}^m$ would be the domain of an atom defined over real vectors. This is the way in which any vector size can be taken into account in the same definition. This setup is unique to the type theory our library is embedded in. In other approaches, $m : \mathbb{N}$ would not be explicitly part of the domain, and vectors would be internally represented as arrays.

Definition 4.3.4 (Atom formal arguments and monotonicity)**.** Let $\mathsf{args}_f : D := (\mathsf{args}_{f,1}, \ldots, \mathsf{args}_{f,k})$ be the atom's *formal arguments*, which is a list of variable names. We use them in further definitions as identifiers for the atom's input. For each $\mathsf{args}_{f,i}$, we associate one of the following labels:

$$\mathsf{inputKind}_f(i) \in \{\textsc{Increasing}, \textsc{Decreasing}, \textsc{Neither}, \textsc{Constant}\}.$$

To understand these labels, suppose f is a function atom. We treat the case $\mathsf{inputKind}_f(i) = \textsc{Constant}$ as a special case since it means that the i^{th} argument is treated as a *parameter* of the atom declaration and it may only be instantiated with a concrete numerical value. Otherwise, the labels encode the monotonicity of the function as follows:

- $\mathsf{inputKind}_f(i) = \textsc{Increasing}$: the function is monotone on its i^{th} argument.
- $\mathsf{inputKind}_f(i) = \textsc{Decreasing}$: the function is antitone on its i^{th} argument.
- $\mathsf{inputKind}_f(i) = \textsc{Neither}$: nothing is claimed about the function's monotonicity on its i^{th} argument.

The role of these labels will become clearer when we define the curvature rules in Section 4.4. In contrast with curvature labels, we will be required to prove that the input kinds are, in some sense, correct, as stated in definition 4.3.14.

From the above, it follows that these labels imply no strictness, so being increasing means being non-decreasing. Similarly, the notion of decreasing technically means non-increasing in our framework.

Definition 4.3.5 (Atom background conditions)**.** $\mathsf{bconds}_f : D \to \{\top, \bot\}$. They need to be derivable from facts independent of the problem and hence can only be applied to parameters. In other words, its definition can only involve $\mathsf{args}_{f,i}$ if $\mathsf{inputKind}_f(i) = \textsc{Constant}$.

To give an example, we consider the convex atom log-sum-exp corresponding to the expression $\log\left(\sum_{i=0}^{m-1} \exp(x_i)\right)$. This is an atom defined over real vectors, with formal arguments $(m, \vec{x}) : (m : \mathbb{N}) \times \mathbb{R}^m$, where m is a parameter. Its background conditions are $\mathsf{bconds}(m, \vec{x}) := m > 0$, which ensures that the sum is non-zero. It is a property of the expression, independent of the context problem.

The way in which we define the multiplication atoms (see Section 4.2.1) is another instance where bconds are used. They are affine with domain $\mathbb{R} \times \mathbb{R}$. One of the formal arguments in the multiplication expression must be a constant. We have four versions depending on the sign of the constant and its position. One case is $c \cdot x$ with $\mathsf{bconds}_{\mathtt{mul1}}(c, x) := c \geq 0$ (and $\mathsf{inputKind}(2) := \textsc{Increasing}$). Another case is $c \cdot x$ with $\mathsf{bconds}_{\mathtt{mul2}}(c, x) := c \leq 0$ (and $\mathsf{inputKind}(2) := \textsc{Decreasing}$). This also illustrates the interplay between different parts of the atom's signature.

Definition 4.3.6 (Atom variable conditions)**.** For problem-dependent conditions, we define $\mathsf{vconds}_f : D \to \{\top, \bot\}$. They are given by an expression that may involve any formal arguments. We can refer to all the conditions as conds_f defined as $\mathsf{conds}_f(\vec{x}) := \mathsf{bconds}_f(\vec{x}) \wedge \mathsf{vconds}_f(\vec{x})$.

For instance, the logarithm atom, defined over a real variable $x : \mathbb{R}$ has variable conditions $\mathsf{vconds}(x) := x > 0$.

All conditions can be understood as restrictions on the domain. A reference to an atom can only be replaced by its graph implementation (graph expansion) if its conditions (with the formal arguments replaced by the appropriate expressions) hold in the feasible set of the problem where they appear. An important remark is that affine function atoms and set atoms are not allowed to have variable conditions, i.e., there are no restrictions on their non-constant arguments. It would be technically possible to allow them, but it is not needed for any of the atoms we are interested in. Forbidding them also makes sense from a philosophical

point of view since set atoms behave as restrictions on the feasible set themselves, and although affine maps can be defined over arbitrary vector spaces, we usually think of affine maps over copies of $\mathbb{R}^n$ and $\mathbb{R}^{n\times m}$.

Since variable conditions encode information about the feasible set of the problem, we need to make sure that, after graph expansion, this information is not lost. As we will see in definition 4.3.16, we will need to prove that the atom's implementation constraints imply the variable conditions. This is not needed for background conditions since they are independent of the problem, hence the distinction.

Definition 4.3.7 (Atom expression)**.** $\mathsf{expr}_f : (\vec{x} : D) \to \mathsf{range}_f(\vec{x})$. This is the mathematical expression (given as a function) corresponding to the name f. In our examples, $\mathsf{range}_f(\vec{x})$ will be $\mathbb{R}$, $\mathbb{R}^n$, $\mathbb{R}^{n\times m}$, or $\{\top, \bot\}$. The latter is used for sets. We will write $R(\vec{x})$ for short and when the atom is clear from the context.

Following the examples given in this section, we have $\mathsf{expr}_{\mathtt{mul1}}(c, x) := c \cdot x$, $\mathsf{expr}_{\mathtt{log}}(x) := \log(x)$, and $\mathsf{expr}_{\mathtt{logSumExp}}(m, \vec{x}) := \log\left(\sum_{i=0}^{m-1} \exp(x_i)\right)$. We use typewriter font to emphasize that the subscript is an identifier and to distinguish it from the atom expression.

4.3.2 Graph implementation

Definitions 4.3.8 to 4.3.10 correspond to the *graph implementation* of the atom. At the end of this section, we make it precise how these definitions relate to the informal presentation of graph implementations in Section 2.2.2. There, we discussed how for all $x \geq 0$, $\sqrt{x} = \sup\left\{v \in \mathbb{R} \mid (x, 0.5, v) \in \mathcal{Q}_r^3\right\}$. Here, we show how we formally define the *implementation variable* v, the *implementation objective* v, and the *implementation constraint* $(x, 0.5, v) \in \mathcal{Q}_r^3$.

First, we remark that the implementation objective is not always a single variable, and, in general, it can be any affine expression. We consider as an example our arguably most complex atom: the log-det atom [BFMA23, § 5], which is concave. For any positive *definite* matrix $X \in \mathbb{R}^{n\times n}$, we have:

$$\log(\det(X)) = \begin{pmatrix} \text{maximize} & \sum_{i=1}^{n} t_i \\ \text{subject to} & (t, 1, y) \in \mathcal{K}_{\exp}^n \\ & \begin{pmatrix} D & Z \\ Z^T & X \end{pmatrix} \in \mathcal{S}_+^{2n} \end{pmatrix}.$$

The problem is defined over $t \in \mathbb{R}^n$ and $Y \in \mathbb{R}^{n \times n}$. We set $y \in \mathbb{R}^n$ to the diagonal of Y, $D \in \mathbb{R}^{n \times n}$ to Y with all the off-diagonal entries set to 0, and $Z \in \mathbb{R}^{n \times n}$ to Y with all the entries below the diagonal set to 0. This example not only shows a more interesting objective function but also illustrates the richness in definitions of graph implementations. It was formalized by Bentkamp and required proving several results that were not in mathlib at the time.

We proceed by showing how to describe, in general, a graph implementation by specifying the implementation variables, objective function, and constraints. Regarding terminology, we note that we will use the phrase "graph implementation" also to refer to the optimization problem associated with the graph implementation.

Definition 4.3.8 (Atom implementation variables)**.** The extended domain of the implementation variables is $\mathsf{impDomain}_f(\vec{x}) := E_1(\vec{x}) \times \cdots \times E_m(\vec{x}) = E(\vec{x})$, defined over $\vec{x} : D$. Implementation variables are given by a list of *fresh* variable names with respect to the formal arguments args_f. We define $\mathsf{impVars}_f : E(\vec{x}) := (\mathsf{impVars}_{f,1}, \ldots, \mathsf{impVars}_{f,m})$. The *full domain* is $(\vec{x} : D) \times E(\vec{x})$.

Once again, the extended domain depends on $\vec{x}$ because of atoms where the range or the domain includes $\mathbb{R}^n$ or $\mathbb{R}^{n \times m}$. In those cases, it is possible that the implementation variables are also in $\mathbb{R}^n$ or $\mathbb{R}^{n \times m}$. We assume that E can only depend on parameters.

Definition 4.3.9 (Atom implementation objective)**.** The objective function of the graph implementation is an expression

$$\mathsf{impObj}_f : (\vec{x} : D) \times E(\vec{x}) \to R(\vec{x})\,.$$

Definition 4.3.10 (Atom implementation constraints)**.** We require a list of constraints $\mathsf{impConstrs}_f := \left[\mathsf{impConstrs}_{f,1}, \ldots, \mathsf{impConstrs}_{f,l}\right]$, where each has type

$$\mathsf{impConstrs}_{f,i} : (\vec{x} : D) \times E(\vec{x}) \to \{\top, \bot\},$$

meaning that it is a predicate on the full domain. We write $\mathsf{impConstrs}_f(\vec{x}, \vec{v})$ as a shortcut for the conjunction of all constraints, i.e., $\bigwedge_{i=1}^{l} \mathsf{impConstrs}_{f,i}(\vec{x}, \vec{v})$.

We assume that graph implementations are in conic form, i.e., impObj_f is affine (or a cone membership constraint in the case of set atoms), and each $\mathsf{impConstr}_{f,i}$ is a cone membership constraint. We will show how this requirement can be relaxed in Section 4.6.3.

The graph implementation should be set up in a way such that for all $\vec{x} : D$ satisfying $\mathsf{conds}_f(\vec{x})$,

$$\mathsf{expr}_f(\vec{x}) = \inf_{\vec{v}:E(\vec{x})} \left\{ \mathsf{impObj}_f(\vec{x},\vec{v}) \mid \mathsf{impConstrs}_f(\vec{x},\vec{v}) \right\}$$

for convex atoms and

$$\mathsf{expr}_f(\vec{x}) = \sup_{\vec{v}:E(\vec{x})} \left\{ \mathsf{impObj}_f(\vec{x},\vec{v}) \mid \mathsf{impConstrs}_f(\vec{x},\vec{v}) \right\}.$$

for concave atoms. If that is the case, we say that the graph implementation is *correct*. To understand why these are minimization and maximization problems, respectively, we need to go back to the core idea of graph implementations [GB08]. As phrased by Grant and Boyd, it is to:

> (...) exploit the relationship between convex and concave functions, and their epigraphs and hypographs, respectively.

To make it clearer, consider an atom f for a function $\mathbb{R} \to \mathbb{R}$. If it is convex, then its epigraph is the convex set of points (x,t) with $\mathsf{expr}_f(x) \leq t$. In this set, t is *minimized* when it is equal to $\mathsf{expr}_f(\vec{x})$ (at the function's graph). Similarly, if it is concave, then its hypograph is the convex set of points (x,t) with $\mathsf{expr}_f(x) \geq t$ and t is *maximized* at the function's graph. In this simplified setting, t plays the role of the implementation variable (and objective), and the inequality is the implementation constraint. If we rewrite inequality into an equivalent cone membership constraint, then we have found a way to represent the atom's expression in conic form. The fact that we are able to replace an instance of $\mathsf{expr}_f(x)$ in an optimization problem by t and add the corresponding conic constraints (graph expansion) is less obvious and depends on the context the expression is in. The DCP curvature rules ensure that replacing atoms in this way always yields an equivalent problem.

We have only considered convex and concave function atoms, but affine function atoms and set atoms are also atoms, so they should also have a "graph implementation" in our unified atom declaration setting. This departs from the original idea of graph implementations. In all these cases, we do not need implementation variables or implementation constraints. However, the implementation objective is still required since it is used to canonize *any* expression, as seen in Section 4.5.2. For these atoms, the graph implementation is *correct* provided that for all $\vec{x} : D$ satisfying $\mathsf{conds}_f(\vec{x})$,

$$\mathsf{expr}_f(\vec{x}) = \mathsf{impObj}_f(\vec{x}).$$

For affine function atoms, their implementation objective is simply $\mathsf{expr}_f(\vec{x})$. This makes sense since graph expansion should not replace affine operations. For set atoms, it is perhaps even more unusual to talk about graph implementations. We are only able to do so because we consider them functions with range $\{\bot, \top\}$ with $\bot \leq \top$. With that in mind, the implementation objective is the "cone membership version" of the expression described by the set atom, for instance, $\mathsf{impObj}_{\mathsf{1e}}(x, y) = (y - x) \in \mathbb{R}_+$.

4.3.3 Proof obligations

Definitions 4.3.11 to 4.3.16 imply that the graph implementation is correct. As a matter of fact, they do more than that. These properties are used to assert that the DCP procedure yields a strong equivalence, which we prove in Section 4.6.

The following may be seen as *forward properties* and will be used to construct the forward map φ and prove its key properties.

Definition 4.3.11 (Atom solution)**.** We have $\mathsf{sol}_f := \left(\mathsf{sol}_{f,1}, \ldots, \mathsf{sol}_{f,m}\right)$, where $\mathsf{sol}_{f,i} : (\vec{x} : D) \to E_i(\vec{x})$. To build φ, we need to assign each $\mathsf{impVars}_{f,i}$ to an expression built from the original domain variables in D, which is exactly what is captured here. We write $\mathsf{sol}_f(\vec{x})$ to mean $\left(\mathsf{sol}_{f,1}(\vec{x}), \ldots, \mathsf{sol}_{f,m}(\vec{x})\right) : E(\vec{x})$.

Definition 4.3.12 (Atom solution correctness)**.** $\mathsf{solEqAtom}_f$ is a proof of the following statement:

$$\forall \vec{x} : D.\ \mathsf{conds}_f(\vec{x}) \Rightarrow \mathsf{impObj}_f\left(\vec{x}, \mathsf{sol}_f(\vec{x})\right) = \mathsf{expr}_f(\vec{x}).$$

This property relates the atom's implementation objective, the solution, and the expression in the expected way by ensuring that the solution of the graph implementation equals the expression. Note that this is required for the graph implementation to be correct.

Definition 4.3.13 (Atom solution feasibility)**.** $\mathsf{feasibility}_f$ is a proof of the solution satisfying the constraints if the conditions are met. Logically, this translates to:

$$\forall \vec{x} : D.\ \mathsf{conds}_f(\vec{x}) \Rightarrow \mathsf{impConstrs}_f\left(\vec{x}, \mathsf{sol}_f(\vec{x})\right)$$

This implies that the solution is feasible in the graph implementation.

Analogously, the following are *backward properties* and will be used to prove the key properties of the backward map ψ. They also ensure that $\mathsf{sol}_f(\vec{x})$ is optimal in the graph implementation.

Definition 4.3.14 (Atom monotonicity)**.** First, for $\vec{a}, \vec{b} : D$ define $\vec{a} \triangle \vec{b}$ as follows:

$$\bigwedge_{1 \le i \le n} \begin{cases} a_i \ge b_i & \text{if } \mathsf{inputKind}_f(i) = \textsc{Increasing} \\ a_i \le b_i & \text{if } \mathsf{inputKind}_f(i) = \textsc{Decreasing} \\ a_i = b_i & \text{otherwise} \end{cases}$$

Then, define the monotonicity condition as follows:

$$\mathsf{monoCond}_f(\vec{x}, \vec{y}) := \begin{cases} \mathsf{expr}_f(\vec{x}) \ge \mathsf{expr}_f(\vec{y}) & \text{if } \mathsf{class}_f = \textsc{Fn} \\ \mathsf{expr}_f(\vec{y}) \Rightarrow \mathsf{expr}_f(\vec{x}) & \text{if } \mathsf{class}_f = \textsc{Set}, \end{cases}$$

Finally, $\mathsf{monotonicity}_f$ is a proof of the statement below. In other words, it establishes that the labels given by $\mathsf{inputKind}_f$ are correct with respect to the atom's expression expr_f, assuming the conditions on the domain.

$$\forall \vec{x}, \vec{y} : D.\ \mathsf{conds}_f(\vec{x}) \Rightarrow \mathsf{conds}_f(\vec{y}) \Rightarrow \vec{x} \triangle \vec{y} \Rightarrow \mathsf{monoCond}_f(\vec{x}, \vec{y}).$$

Note the position of $\vec{x}$ and $\vec{y}$ in the set case of $\mathsf{monoCond}_f$, respecting the order relation on $\{\bot, \top\}$ discussed earlier.

Definition 4.3.15 (Atom bounds)**.** In order to show that $\mathsf{sol}_f(\vec{x})$ is optimal in the graph implementation, we need to show that $\mathsf{expr}_f(\vec{x})$ (which, by solution correctness, is the value of impObj_f at $(\vec{x}, \mathsf{sol}_f(\vec{x}))$) is a bound to $\mathsf{impObj}_f(\vec{x}, \vec{v})$ in the right way, depending on the atom's curvature. That is precisely what $\mathsf{boundsCond}_f$ captures.

$$\mathsf{boundsCond}_f(\vec{x}, \vec{v}) := \begin{cases} \mathsf{expr}_f(\vec{x}) \le \mathsf{impObj}_f(\vec{x}, \vec{v}) & \text{if } \mathsf{curv}_f = \textsc{ConvexFn} \\ \mathsf{expr}_f(\vec{x}) \ge \mathsf{impObj}_f(\vec{x}, \vec{v}) & \text{if } \mathsf{curv}_f = \textsc{ConcaveFn} \\ \mathsf{expr}_f(\vec{x}) = \mathsf{impObj}_f(\vec{x}, \vec{v}) & \text{if } \mathsf{curv}_f = \textsc{AffineFn} \\ \mathsf{expr}_f(\vec{x}) \Leftrightarrow \mathsf{impObj}_f(\vec{x}, \vec{v}) & \text{if } \mathsf{class}_f = \textsc{Set} \end{cases}$$

In fact, we only need this bound to hold in the feasible set of the graph implementation. We define bounds_f as a proof of the following statement:

$$\forall \vec{x} : D.\ \forall \vec{v} : E.\ \mathsf{impConstrs}_f(\vec{x}, \vec{v}) \Rightarrow \mathsf{boundsCond}_f(\vec{x}, \vec{v}).$$

Given $\mathsf{sol}_f(\vec{x})$ and the proof obligations above, it is easy to see how we can prove that the graph implementation is correct. Assume $\mathsf{conds}_f(\vec{x})$. Then, by $\mathsf{feasibility}_f$, $\mathsf{sol}_f(\vec{x})$ is feasible, and by bounds_f and $\mathsf{solEqAtom}_f$ it is optimal and

its value at the optimum equals $\mathsf{expr}_f(\vec{x})$, as required. Note that we did not use $\mathsf{monotonicity}_f$, which plays a role when graph expansion happens in the context of a problem.

There is one further consideration regarding variable conditions.

Definition 4.3.16 (Atom variable condition elimination)**.** $\mathsf{vcondElim}_f$ is a proof of the fact that implementation constraints imply variable conditions. For convex functions, we need to prove:

$$\forall \vec{x}, \vec{y} : D.\ \forall \vec{v} : E.\ \mathsf{impConstrs}_f(\vec{x}, \vec{v}) \Rightarrow \vec{x} \mathbin{\triangle} \vec{y} \Rightarrow \mathsf{vconds}_f(\vec{y}).$$

For concave function atoms, we need to prove:

$$\forall \vec{x}, \vec{y} : D.\ \forall \vec{v} : E.\ \mathsf{impConstrs}_f(\vec{x}, \vec{v}) \Rightarrow \vec{y} \mathbin{\triangle} \vec{x} \Rightarrow \mathsf{vconds}_f(\vec{y}).$$

For affine function atoms or set atoms, recall that there are no variable conditions, hence $\mathsf{vcondElim}_f$ is trivial.

This property enforces graph implementations to encode domain restrictions. It is not strictly needed to prove equivalences, as we could potentially canonize all variable conditions to conic form. This is not always possible, as some are strict inequalities, which cannot be canonized directly but are captured by the cones. In our framework, strict inequalities are allowed as long as they can be eliminated. Since this elimination process removes constraints, we need evidence that they are redundant, which is precisely what $\mathsf{vcondsElim}_f$ tells us.

For instance, consider once again the log atom with formal arguments $x : \mathbb{R}$, labeled as INCREASING. We have $\mathsf{vconds}_{\mathsf{log}}(x) := 0 < x$ (a strict inequality). Its implementation is given by an optimization variable $t : \mathbb{R}$, implementation objective t, and implementation constraint $(t, 1, x) \in \mathcal{K}_{\text{exp}}$. By the definition of $\mathcal{K}_{\text{exp}}$ (see definition 2.2.10), it is easy to see that the implementation constraint implies $1 \cdot \exp(t/1) \leq x$, and, therefore, for all $y \geq x$, we have $0 < y$. This allows us to eliminate the constraint $0 < x$ from a problem involving $\log(x)$ since it shows that, after graph expansion, the positivity constraint on x is not lost. The reason why it is phrased in terms of $\vec{y}$ is that this elimination procedure needs to work for, say, expressions of the form $\log(e)$ where e is an expression made up of other atoms that also need to be expanded. In those cases, if e is canonized to e', all we know is that $e' \mathbin{\triangle} e$ (as seen from the proof of Lemma 4.6.8).

4.3.4 Example: `sqrt`

To give a concrete example of a full atom declaration, we show the atom declaration fields associated with the square root atom `sqrt`, including proofs of the key properties. We omit the `sqrt` subscript.

- $\mathsf{class} := \textsc{Fn}$.
- $\mathsf{curv} := \textsc{ConcaveFn}$.
- $\mathsf{domain} := \mathbb{R}$, and $\mathsf{args} := (x : \mathbb{R})$ with $\mathsf{inputKind}(1) = \textsc{Increasing}$.
- $\mathsf{bconds}(x) := \top$.
- $\mathsf{vconds}(x) := 0 \leq x$.
- $\mathsf{expr}(x) := \sqrt{x}$.
- $\mathsf{impDomain}(x) := \mathbb{R}$, and $\mathsf{impVars} := (v : \mathbb{R})$.
- $\mathsf{impObj}(x, v) := v$.
- $\mathsf{impConstrs} := \left[\lambda(x, v).\ (x, 0.5, v) \in \mathcal{Q}_r^3\right]$.
- $\mathsf{sol}(x) := \sqrt{x}$.
- $\mathsf{solEqAtom} : \forall x : \mathbb{R}.\ \sqrt{x} = \sqrt{x}$ is obvious.
- $\mathsf{feasibility} : \forall x : \mathbb{R}.\ 0 \leq x \Rightarrow (x, 0.5, \sqrt{x}) \in \mathcal{Q}_r^3$ follows from the definition of the rotated quadratic cone, $\mathcal{Q}_r^3 = \{(x, y, z) \in \mathbb{R}^3 \mid 2xy \leq z^2\}$, and the fact that $(\sqrt{x})^2 = x$ for $0 \leq x$.
- $\mathsf{monotonicity} : \forall x, y : \mathbb{R}.\ 0 \leq x \Rightarrow 0 \leq y \Rightarrow x \geq y \Rightarrow \sqrt{x} \geq \sqrt{y}$ is obvious since $\sqrt{\cdot}$ is a monotone increasing function on $\mathbb{R}_+$.
- $\mathsf{bounds} : \forall x, v : \mathbb{R}.\ (x, 0.5, v) \in \mathcal{Q}_r^3 \Rightarrow v \leq \sqrt{x}$ can be proved as follows. By the definition above, $(x, 0.5, v) \in \mathcal{Q}_r^3$ is the same as $v^2 \leq x$, from which it follows immediately that $v \leq \sqrt{x}$.
- $\mathsf{vcondElim} : \forall x, y, v : \mathbb{R}.\ (x, 0.5, v) \in \mathcal{Q}_r^3 \Rightarrow y \geq x \Rightarrow 0 \leq y$ is clear as $0 \leq v^2 \leq x$ by the same argument as above and $x \leq y$.

4.3.5 Atom library overview

We conclude this section where we have shown how to declare an atom by summarizing the contents of CvxLean's atom library at the time of writing.

Our library consists of 107 atom declarations. These actually correspond to 49 classes of atoms, categorized in a sensible way. For example, we count the four versions of the multiplication atom as one class. Also, many atoms defined over $\mathbb{R}$ have element-wise versions in $\mathbb{R}^n$ and $\mathbb{R}^{n\times m}$, which we put in the same class.

There are 10 classes of set atoms, including all the cones listed in definition 2.2.10, "=", and "$\leq$". The 39 classes of function atoms are split as follows:

- 22 classes of affine atoms, including elementary operations ($+$, $-$, $\cdot$, and $/$) and various operations to manipulate vectors and matrices.
- 11 classes of convex atoms: absolute value, exponential, Huber loss, positive inverse, Kullback-Leibler divergence, log-sum-exp, max, ℓ^2-norm, some powers (2, -2, and -1), quadratic-over-linear, and $x \cdot \exp(x)$.
- 6 classes of concave atoms: entropy, geometric mean, logarithm, log-det, min, and square root.

We provide a more comprehensive overview in Appendix A.

4.4 Curvature rules

In this section, we operationalize the curvature deduction calculus by defining two executable methods. We define a generic method that outputs *expected* curvatures (exptdCurv) and an atom-dependent method that outputs *computed* curvatures given the curvatures of the instantiated arguments (curvRule). The need for this machinery is related to our discussion in Section 4.2.1. Because of how we handle expressions, multiple atoms could be considered when making atom trees (see Figure 4.2). These methods will help us choose a correct one and ensure that the problem is convex.

The definitions are based on the composition rules for DCP concave and DCP convex functions from definitions 2.2.12 and 2.2.13, respectively, using the monotonicity and curvature labels introduced in the previous section. We also make precise the role of affine functions and show how curvature rules for sets are

defined in this setting, mimicking the so-called top-level rules. We refer to these rules collectively as *curvature rules* or *DCP rules*.

First, we define a method that outputs the expected curvature of a_j, given the assigned curvature to an expression $g(a_1, \ldots, a_k)$ and the monotonicity of the function (or predicate) g at its j^{th} argument.

Consider expressions defined with function atoms. Let κ be the assigned curvature, with $\kappa \in \{\textsc{ConvexFn}, \textsc{ConcaveFn}, \textsc{AffineFn}, \textsc{ConstantFn}\}$. We define $\mathsf{exptdCurv}$ as follows, where the other input is a monotonicity label and the output is a curvature label.

$$\mathsf{exptdCurv}(\kappa, \textsc{Increasing}) = \kappa.$$

$$\mathsf{exptdCurv}(\kappa, \textsc{Decreasing}) = \begin{cases} \textsc{ConvexFn} & \text{if } \kappa = \textsc{ConcaveFn} \\ \textsc{ConcaveFn} & \text{if } \kappa = \textsc{ConvexFn} \\ \kappa & \text{otherwise.} \end{cases}$$

$$\mathsf{exptdCurv}(\kappa, \textsc{Neither}) = \begin{cases} \textsc{ConstantFn} & \text{if } \kappa = \textsc{ConstantFn} \\ \textsc{AffineFn} & \text{otherwise.} \end{cases}$$

$$\mathsf{exptdCurv}(\kappa, \textsc{Constant}) = \textsc{ConstantFn}.$$

Note that constant and affine target curvatures require the curvature of the expressions in their arguments to be constant and affine, respectively. For convex and concave target curvatures, the definition corresponds to the expectations set by the composition rules. When nothing is claimed about the monotonicity of a function at an argument, then the composition rules expect the expression at that argument to be affine (or constant).

The expected curvature for arguments of set atoms is as follows:

$$\mathsf{exptdCurv}(\kappa, \textsc{Increasing}) = \begin{cases} \textsc{ConcaveFn} & \text{if } \kappa = \textsc{ConvexSet} \\ \textsc{AffineFn} & \text{if } \kappa = \textsc{AffineSet} \end{cases}$$

$$\mathsf{exptdCurv}(\kappa, \textsc{Decreasing}) = \begin{cases} \textsc{ConvexFn} & \text{if } \kappa = \textsc{ConvexSet} \\ \textsc{AffineFn} & \text{if } \kappa = \textsc{AffineSet} \end{cases}$$

$$\mathsf{exptdCurv}(\kappa, \textsc{Neither}) = \textsc{AffineFn}.$$

$$\mathsf{exptdCurv}(\kappa, \textsc{Constant}) = \textsc{ConstantFn}.$$

This definition is better understood by considering the only set atom that allows non-affine arguments: "$\leq$". We say that an expression $x \leq y$ is decreasing in x in

the following sense: if $x \leq x'$ then $x' \leq y \Rightarrow x \leq y$, where "$\Rightarrow$" is seen as "$\leq$" in $\{\top, \bot\}$ (cf. definition 4.3.14). Therefore, this tells us that the left-hand side is expected to be convex, and the right-hand side is expected to be concave. The rest of the set atoms (equality and cones) require affine arguments. Cones are a special case because even when they happen to be monotonic on one or more of their arguments (e.g., $\mathbb{R}_+$), we set all input kinds to NEITHER so that the expression a in $a \in \mathcal{K}$ is expected to be affine.

Using $\mathsf{exptdCurv}$, we formally define $\mathsf{curvRule}$, the curvature computation rule for atoms. We aim to compute the curvature for an expression $\mathsf{expr}_f(a_1, \ldots, a_k)$ where the curvatures of $a_1, \ldots, a_k$ are known. This method clarifies how curvatures are calculated in a bottom-up fashion.

We begin with function atoms. We need to distinguish between affine atoms and the rest. The reason is that an expression with an affine atom at the top level might not be affine, e.g., $-\exp(x)$ is concave. Let $\vec{\kappa} = (\kappa_1, \ldots, \kappa_k)$ be a vector of curvatures where k is the arity of of f and $\mathsf{curv}_f = \text{AFFINEFN}$, then:

$$\mathsf{curvRule}(f, \vec{\kappa}) := \begin{cases} \text{CONSTANTFN} & \text{if } \forall i.\ \kappa_i \sqsubseteq \mathsf{exptdCurv}(\text{CONSTANTFN}, \mathsf{inputKind}_f(i)) \\ \text{AFFINEFN} & \text{if } \forall i.\ \kappa_i \sqsubseteq \mathsf{exptdCurv}(\text{AFFINEFN}, \mathsf{inputKind}_f(i)) \\ \text{CONVEXFN} & \text{if } \forall i.\ \kappa_i \sqsubseteq \mathsf{exptdCurv}(\text{CONVEXFN}, \mathsf{inputKind}_f(i)) \\ \text{CONCAVEFN} & \text{if } \forall i.\ \kappa_i \sqsubseteq \mathsf{exptdCurv}(\text{CONCAVEFN}, \mathsf{inputKind}_f(i)) \\ \text{UNKNOWNFN} & \text{otherwise.} \end{cases}$$

The cases of $\mathsf{curvRule}$ need to be considered one after the other (they are not mutually exclusive). We take advantage of the order relation $\sqsubseteq$ induced on the curvature labels as discussed in definition 4.3.2. It is easy to see that $\kappa_i \sqsubseteq \mathsf{exptdCurv}(\text{CONSTANTFN}, \mathsf{inputKind}_f(i))$ is the same as $\kappa_i = \text{CONSTANTFN}$, as expected. Similarly, in the affine case, $\kappa_i \sqsubseteq \mathsf{exptdCurv}(\text{AFFINEFN}, \mathsf{inputKind}_f(i))$ is equivalent to $\kappa_i \sqsubseteq \text{AFFINEFN}$. The convex and concave cases, however, do depend on the monotonicity of the arguments.

The story is different for non-affine atoms as composition rules imply that the curvature of a non-constant expression with an atom f at the top level with

known curvature can only be successfully labeled with curv_f.

$$\mathsf{curvRule}(f, \vec{\kappa}) := \begin{cases} \textsc{ConstantFn} & \text{if } \forall i.\ \kappa_i = \textsc{ConstantFn} \\ \mathsf{curv}_f & \text{if } \forall i.\ \kappa_i \sqsubseteq \mathsf{exptdCurv}(\mathsf{curv}_f, \mathsf{inputKind}_f(i)) \\ \textsc{UnknownFn} & \text{otherwise.} \end{cases}$$

To see curvRule in action, consider $-\exp(x)$ once again. As we will show, our algorithm starts by labeling variables and constants, so x will be labeled with AffineFn. Since exp is a convex atom with one increasing argument, we have $\mathsf{curvRule}(\mathsf{exp}, \textsc{AffineFn}) = \textsc{ConvexFn}$ because $\textsc{AffineFn} \sqsubseteq \textsc{ConvexFn}$, where $\textsc{ConvexFn} = \mathsf{exptdCurv}(\textsc{ConvexFn}, \textsc{Increasing})$. Negation is a decreasing affine atom. Going through the cases of $\mathsf{curvRule}(\mathsf{neg}, \textsc{ConvexFn})$ in order, we see that the first case that works is ConcaveFn since $\textsc{ConvexFn} = \mathsf{exptdCurv}(\textsc{ConcaveFn}, \textsc{Decreasing})$. This shows how, using curvRule, we can calculate the curvature of any expression composed of atoms, variables, and constants.

The following proposition follows directly from the definition of curvRule.

Proposition 4.4.1 (Relationship between curvRule and curv_f)**.** If, for some i, we have $\kappa_i \neq \textsc{ConstantFn}$, then $\mathsf{curv}_f \sqsubseteq \mathsf{curvRule}(f, \vec{\kappa})$ for any function atom f.

For set atoms, we state the rules for each of them directly, following the top-level rules (T4)-(T7) [GBY06, § 6.1]. First, we cover atoms for equality and inequality:

$$\mathsf{curvRule}(\mathsf{eq}, \kappa_1, \kappa_2) := \begin{cases} \textsc{AffineSet} & \text{if } \forall i.\ \kappa_i \sqsubseteq \textsc{AffineFn}, \\ \textsc{UnknownSet} & \text{otherwise.} \end{cases}$$

$$\mathsf{curvRule}(\mathsf{le}, \kappa_1, \kappa_2) := \begin{cases} \textsc{AffineSet} & \text{if } \forall i.\ \kappa_i \sqsubseteq \textsc{AffineFn}, \\ \textsc{ConvexSet} & \text{if } \kappa_1 \sqsubseteq \textsc{ConvexFn} \\ & \text{and } \kappa_2 \sqsubseteq \textsc{ConcaveFn}, \\ \textsc{UnknownSet} & \text{otherwise.} \end{cases}$$

For a cone membership atom f, recalling our remark about all input kinds being Neither, we define:

$$\mathsf{curvRule}(f, \vec{\kappa}) := \begin{cases} \textsc{ConvexSet} & \text{if } \forall i.\ \kappa_i \sqsubseteq \textsc{AffineFn}, \\ \textsc{UnknownSet} & \text{otherwise.} \end{cases}$$

For the DCP transformation, we do not need the AffineSet label and could use ConvexSet in its place in the rules above and still ensure compliance with the DCP rules. We add more fine-grained set curvature rules because of the pre-DCP transformation from Chapter 5, where a constraint labeled with AffineSet will be preferred over an equivalent constraint labeled with ConvexSet.

Apart from serving as a helper function to define curvRule, exptdCurv will serve another important purpose. It will be used as a top-down inference rule to drive atom discovery. In that case, the first input of exptdCurv will be seen as the *target* curvature. Imagine f is a candidate atom for the root expression of the objective function (with target curvature ConvexFn). Then, exptdCurv will be used to compute the expected curvatures of the arguments of f, given its input kinds, and recursively build sub-trees. See Section 4.5.1 for further details.

4.5 Transformation to conic form

This section explains how an optimization problem is canonized to conic form using the atom library and curvature rules. The first step is to make an *atom tree*, as described in Section 4.5.1. Section 4.5.2 shows how to use it to canonize the problem. How atom conditions are inferred is discussed in Section 4.5.3.

Throughout this section, fix the following problem.

$$\begin{aligned} P := \text{minimize} \quad & -\sqrt{x-y} \\ \text{subject to} \quad & y = 2x - 3 \\ & x^2 \leq 2 \\ & 0 \leq x - y \end{aligned}$$

4.5.1 Atom trees

There is a need to define various functions that recurse over the syntax of problems, both to describe the transformation and to show it is equivalence-preserving. Atom trees provide this syntactic representation. The concrete representation of atom trees will be made precise by mkAtomTree (see Figure 4.2), but we may define them simply as trees with atom identifiers at the internal nodes and variable names or numerical constants at the leaves.

We call the objective function and each constraint a *component* of the opti-

mization problem and identify them with e_o and $e_1, \ldots, e_p$. For each component, we aim to make an atom tree from the corresponding expression. The most important requirement is that it must follow the DCP curvature rules. This is the point where we check that the objective function is DCP convex and the constraints are DCP-compliant convex sets (originally called the verification phase [GBY06, § 8] in DCP). We show the atom tree for the objective function of P in Figure 4.1. We have also included the computed curvature labels and numbered the nodes following a post-order traversal. This number will be useful to uniquely identify every node (since the same atom might be used multiple times). The labels at the edges correspond to the input kinds of the parent atom.

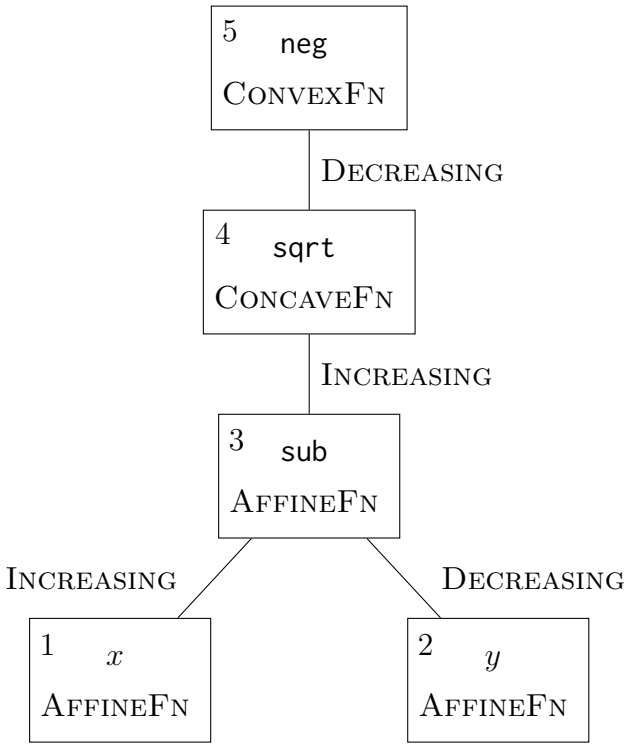


Figure 4.1: Atom tree for the objective function of P (T_{e_o}).

The atom tree for each component is *discovered* in a top-down fashion. As we traverse the sub-expressions, the core step is finding some atom that can be used for the top-level structure of the sub-expression, following our discussion in Section 4.2.1. For the objective function, for instance, the top node should be labeled as convex, which means that we should only consider affine or convex atoms (that match the expression). As we go down, the method exptdCurv is used to determine which curvatures to consider. However, building the whole sub-tree is required in order to be sure that the correct atom was chosen. That is

because the curvature can ultimately only be calculated bottom up. Variables are labeled with AffineFn and numerical constants with ConstantFn and then, using curvRule, we can go up the tree labeling every node as follows.

$$\mathsf{compCurv}(n) := \begin{cases} \textsc{ConstantFn} & \text{if } n \text{ is a constant leaf;} \\ \textsc{AffineFn} & \text{if } n \text{ is a variable leaf;} \\ \mathsf{curvRule}(f, \vec{\kappa}) & \text{if } n \text{ is an internal node with atom } f, \\ & \text{children } c_1, \ldots, c_k \text{ and } \kappa_i = \mathsf{compCurv}(c_i). \end{cases}$$

Recall that multiple atoms might match the same expression, as discussed in Section 4.2.1. We store the expr_f (see definition 4.3.7) of every atom in the library in a discrimination tree [McC92] to enable fast identification of atoms that match some sub-expression of a component. Given any expression, we can pattern-match against the discrimination tree and get a list of atom names whose atom expression matches. The pseudocode in Figure 4.2 shows precisely how this happens, as well as how curvatures are taken into account.

The inputs of mkAtomTree are an expression *expr*, a target curvature $tgtCurv$, and a counter $lastID$ used to identify the nodes (where node means either a leaf "Leaf" or an internal node "INode"). If it fails, it returns a failure token "Failure". If it succeeds, it returns an atom tree, a computed curvature for the whole atom tree, and an updated counter. The computed curvature is exactly $\mathsf{compCurv}(n)$, where n is the root of the tree; and, importantly, it is ensured to be $\sqsubseteq tgtCurv$. Assuming that $tgtCurv$ is known (not UnknownFn or UnknownSet), this implies that all of the subsequently computed curvatures in a successful run are also known. Any atom tree returned by mkAtomTree is said to be *valid*. In the pseudocode, successful recursive calls are checked by an if-statement where mkAtomTree returns a triple. In the atom tree, the value at the internal nodes is a tuple (f, s) where f is an atom identifier, and the value at the leaves is a tuple $(expr, s)$ where *expr* is either a variable name or a constant; s is a number in both cases.

Before proceeding with our discussion, it might be useful to summarize the curvatures involved in every call:

- $tgtCurv$: the target curvature at the current procedure call.
- $exptdArgCurv_j$: the expected curvature for a_j.
- $compArgCurv_j$: the computed curvature for a_j.

```
function mkAtomTree(expr, tgtCurv, lastID):
    if expr is constant, i.e., has no optimization variables then
        return (Leaf(expr, lastID + 1), ConstantFn, lastID + 1);
    end
    if tgtCurv == ConstantFn then return "Failure";
    if expr is an optimization variable then
        return (Leaf(expr, lastID + 1), AffineFn, lastID + 1);
    end
    for (f, [a_i, ..., a_k]) in atomLibrary.match(expr) do
        if curv_f ⋤ tgtCurv then continue;
        Ts, compArgCurvs := [], [];
        i := lastID;
        failed := false;
        for j in {1,...,k} do
            exptdArgCurv_j := exptdCurv(tgtCurv, inputKind_f(j));
            if (T_j, compArgCurv_j, lastID_j) :=
              mkAtomTree(a_j, exptdArgCurv_j, i) then
                Ts := Ts ++ [T_j];
                compArgCurvs := compArgCurvs ++ [compArgCurv_j];
                i := lastID_j;
            else
                failed := true; break;
            end
        end
        if failed then continue;
        compCurv := curvRule(f, compArgCurvs);
        if bconds_f(a_1, ..., a_k) && compCurv ⊑ tgtCurv then
            return (INode((f, i + 1), Ts), compCurv, i + 1);
        end
    end
    return "Failure";
```

Figure 4.2: Procedure to try to make a valid atom tree from an expression.

- *compCurv*: the computed curvature for $\mathsf{expr}_f(a_1, \ldots, a_k)$ ($\mathsf{compCurv}(n)$).

We will now look into the procedure in more detail. The base cases are straightforward. The main for loop goes through all the matches from the atom library. Every match consists of f, the atom's identifier, and a list of expressions $a_i, \ldots, a_k$. This means that $expr = \mathsf{expr}_f(a_1, \ldots, a_k)$. As discussed, there could be several matches, some of which might fail, so we need to go through them until we succeed. Note that, from f, we are able to access all the fields in the atom declaration, as described in Section 4.3, which is why we can use curv_f, $\mathsf{inputKind}_f$, bconds_f, and curvRule (which is defined in terms of class_f, curv_f and $\mathsf{inputKind}_f$). First, looking at curvRule as defined in Section 4.4 and Proposition 4.4.1, we can see that, for non-constant target curvatures, if $\mathsf{curv}_f \not\sqsubseteq tgtCurv$, then the computed curvature will also be $\not\sqsubseteq tgtCurv$, so we can discard it right away. Otherwise, we go through each expression a_j, which instantiates one of the atom's arguments. We use exptdCurv to compute the expected curvature of a_j given the target curvature for $\mathsf{expr}_f(a_1, \ldots, a_k)$ (i.e., $tgtCurv$), and proceed to try to make an atom tree for a_j. A recursive call to mkAtomTree might fail, in which case we can try with the next match, if any. If none of the matches are successful, then the current procedure call fails. For a call to mkAtomTree to be successful where *expr* is not a constant or a variable, we need:

- A match $(f, [a_1, \ldots, a_k])$ with a successful call of mkAtomTree for each a_j, given the expected curvature at that argument.
- Background conditions to hold, i.e., $\mathsf{bconds}_f(a_1, \ldots, a_k)$.
- The computed curvature for $\mathsf{expr}_f(a_1, \ldots, a_k)$ to follow the curvature rules with respect to the current target curvature, i.e.,

$$\mathsf{curvRule}(f, compArgCurvs) \sqsubseteq tgtCurv.$$

In that case, we return an atom tree with f at the top node and Ts, the atom trees for the a_js, as its children. As a final remark, it is easy to see how the counter is used to uniquely identify nodes and leaves.

Now, the idea is to apply this procedure to every component. For the objective function, the target curvature is ConvexFn, and for each of the constraints, the target curvature is ConvexSet. If making the atom tree for the objective function fails, then the whole process fails: the problem is not in DCP form.

However, for constraints, we intentionally allow it to fail at this stage. This is because of the way in which we deal with variable conditions. Note that we have not checked variable conditions yet. Following our discussion about variable condition elimination (see definition 4.3.16), we allow some constraints to be non-DCP-compliant as long as they can be eliminated. In particular, this situation arises with strict inequalities. For example, as we saw, the log atom has $\mathsf{vconds}_{\mathtt{log}}(x) := 0 < x$ so, if $\log(x)$ appears in the problem, it is allowed for one of the constraints to be $0 < x$. At this point, we keep track of the components for which mkAtomTree failed and address elimination later. We show the pseudocode to make atom trees for all components (mkAtomTrees) in Figure 4.3.

The pseudocode also demonstrates how all the nodes and leaves in the resulting set of trees are uniquely identified. In our running example, there are no failed constraints, and the result of mkAtomTrees is shown in Figure 4.4, where we only show atoms, variable names, and constants, with their node number circled. Note that the curvature and monotonicity labels shown in Figure 4.1 are not needed past this point; the curvature has been checked. One can follow the procedure step by step to see how the trees are made for this simple example. An interesting observation is that the multiplication node is `mul1`, which is the multiplication atom for non-negative constants on the left. In this case, it is the only instance where there were multiple matches to consider from the atom library.

Before proceeding to the main canonization algorithm, we show how variable conditions are checked. For now, we assume that for any node (f, s) in T_{e_o} or one of the Ts with non-trivial variable conditions (i.e., $\neq \top$), we have that $\mathsf{vconds}_f(a_1, \ldots, a_k)$, where a_js are the expressions that matched in mkAtomTree, is equal to one of the constraints. Going back to the running example of this section, looking at the objective function, we have that $\mathsf{vconds}_{\mathtt{sqrt}}(x - y) := 0 \leq x - y$, which is equal to the third constraint (e_3). We consider the case when vconds are inferrable but do not equal any constraint in Section 4.5.3 (this includes the case when vconds is defined by a conjunction of conditions).

This information about which constraints hold variable conditions and, therefore, are eliminable is stored in a hashmap vcondElimMap. After mkAtomTrees terminates successfully, we perform another pass as follows. We iterate through every node (f, s) in the atom trees and check that the expression of each of its variable conditions equals a constraint e_j. If that is not the case, the algorithm fails. Otherwise, we store $\{s \mapsto e_j\}$ in vcondElimMap. We make sure that all

```
function mkAtomTrees(e_o, e_1, ..., e_p):
    if (T_{e_o}, _, lastID) := mkAtomTree(e_o, ConvexFn, 0) then
        Ts, failedConstraints := [], [];
        i := lastID;
        for j in {1,...,p} do
            if (T_{e_j}, _, lastID_j) := mkAtomTree(e_j, ConvexSet, i) then
                Ts := Ts ++ [T_{e_j}];
                i := lastID_j;
            else
                failedConstraints := failedConstraints ++ [e_j];
            end
        end
        return (T_{e_o}, Ts, failedConstraints);
    end
    return "Failure";
```

Figure 4.3: Procedure to try to make atom trees from a whole problem.

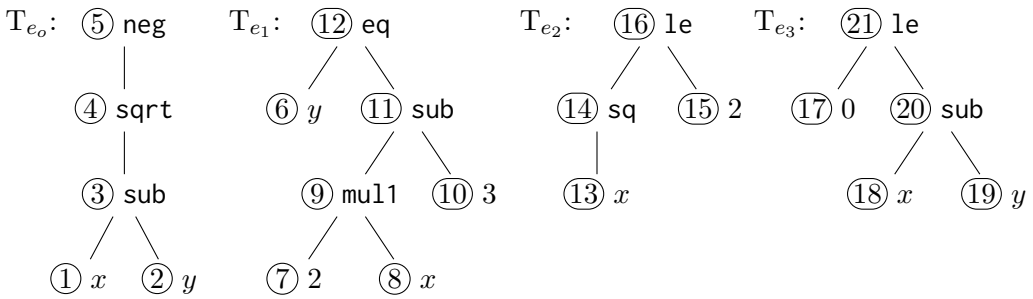


Figure 4.4: T_{e_o}, T_{e_1}, T_{e_2}, and T_{e_3} (P's atom forest).

the constraints in $failedConstraints$ are used for condition elimination; if not, we fail. In our running example, we have vcondElimMap $:= \{4 \mapsto e_3\}$. This is straightforward, but for completeness, we provide pseudocode in Figure 4.5. As previously, we indicate with $a_1, \ldots, a_k$ the sub-expressions of the atom.

```
function mkVcondElimMap(T_{e_o}, Ts, failedConstraints):
    vcondElimMap := {};
    for T in T_{e_o}::Ts do
        for (f, s) in T do
            if vconds_f(a_1, ..., a_k) ≠ ⊤ and equals an e_j then
                vcondElimMap[s] := e_j;
                if e_j in failedConstraints then
                    remove e_j from failedConstraints;
                end
            else return "Failure";
        end
    end
    if failedConstrs ≠ [] then return "Failure";
    return vcondElimMap;
```

Figure 4.5: Procedure to make vcondElimMap given atom trees and failed constraints.

When executing mkAtomTree, we return the first tree that is DCP-compliant (and satisfies all background conditions). Therefore, we could technically pick the wrong atom if, at some point, there were multiple options for which we could successfully make an atom tree, but variable conditions only held for one of them. This limits the way in which we declare some atoms. For example, the square atom, with $\mathsf{expr}_{\mathsf{sq}}(x) := x^2$ is not declared in full generality. We define it with no variable conditions and set $\mathsf{inputKind}_{\mathsf{sq}}(1) :=$ NEITHER, i.e., we do not claim anything about its monotonicity. In particular, this means that we only allow e^2 if e is affine. However, we could have an increasing version with $\mathsf{vconds}(x) := x \geq 0$, and a decreasing version with $\mathsf{vconds}(x) := x \leq 0$. This would currently be problematic as we could potentially pick the wrong version in mkAtomTree. To circumvent this, we could wrap the expression in a definition with a distinct name (which is exactly what other implementations of DCP do anyway, as argued in Section 4.2.1). So, it is a limitation on the automation front, but it does not impose any restrictions on the problems that can be handled. We leave adding

full support for this as future work (see Section 7.4.5).

4.5.2 Canonization

Fix a problem and let $\mathsf{domVars}$ be the names given to the optimization variables. Throughout this section, we will use $\vec{x}$ to denote the variables corresponding to such names. Let D be their type, i.e., the problem's domain. Let R be the problem's range, which we may assume to be $\mathbb{R}$. Note that we are re-using $\vec{x}$ and D here, which were previously used for the arguments and the domain of atoms. We will make sure that this distinction is clear. To fix some more notation, we will use n to refer to the unique identifier (a number) of a node in an atom tree. If it holds an atom, this will be denoted by f. Its children will be denoted by $c_1, \ldots, c_k$, where each c_i is also an identifier.

Once the atom trees are in place and all conditions have been proved or accounted for in $\mathsf{vcondElimMap}$, we can start building the canonized problem. This involves using the graph implementation of the atom at each node. In essence, we will replace every atom with its implementation objective and add the implementation constraints. This process introduces new optimization variables (the implementation variables). The first problem we face is that we need to ensure that the new variables we introduce are fresh. Luckily, we have a unique identifier for each node, which will allow us to rename the variables as needed. More precisely, we define the following version of $\mathsf{impVars}_f$:

$$\mathsf{impVars}^*_n := \left(v + \text{“.”} + n \mid v \text{ in } \mathsf{impVars}_f\right).$$

This renames the implementation variables at each node by adding the node's identifier as a suffix. Recall that $\mathsf{impVars}_f$ can be parametrized by the constant arguments of f, which we may omit here, as they have already been instantiated at this point.

We collect the implementation variables introduced up to a node n as follows, where “up to n” means in all the nodes in the sub-tree rooted at n.

$$\mathsf{newVars}_{\downarrow n} := \mathsf{impVars}^*_n +\!\!+ \mathsf{newVars}_{\downarrow c_1} +\!\!+ \cdots +\!\!+ \mathsf{newVars}_{\downarrow c_k},$$

where “$+\!\!+$” denotes concatenation. Note that suffixes ensure that all variable names are unique. We use the notation $\mathsf{newVars}_e$, where e is a component name to mean $\mathsf{newVars}_{\downarrow n}$, where n is the root of component e. To make it clearer, let

us look at e_o in the running example of this section with atom tree T_{e_o} as shown in Figure 4.1. Leaves never introduce new variables. We have $\mathsf{newVars}_{\downarrow 3} = ()$, $\mathsf{newVars}_{\downarrow 4} = (v.4)$, and $\mathsf{newVars}_{\downarrow 5} = (v.4)$, hence $\mathsf{newVars}_{e_o} = (v.4)$.

The full set of new variables is the collection of the new variables introduced by each component.

$$\mathsf{newVars} := \mathsf{newVars}_{e_o} +\!\!+ \mathsf{newVars}_{e_1} +\!\!+ \cdots +\!\!+ \mathsf{newVars}_{e_p}.$$

The second constraint, $x^2 \leq 2$ is the only other instance where a new variable is introduced. In particular, it is introduced by the sq atom, which, as seen in Figure 4.4, is in node number 14. Assume that the implementation variable of the sq atom is named w (note that it could also be named v). Then, we have $\mathsf{newVars} = (v.4, w.14)$. This means that the optimization variables for the canonized problem will be x, y, $v.4$, and $w.14$.

Let E be the domain of $\mathsf{newVars}$. This is formed by the extended domain of the atoms, so we will call it the extended domain of the problem. We use $E_{\downarrow n}$ to refer to the domain of $\mathsf{newVars}_{\downarrow n}$.

One difficulty with the definitions that follow is going back and forth between variable names and variables. Given $\vec{v} : E$, seen as variables in the extended domain, it will be useful to pick out only those variables in the vector corresponding to those introduced at a particular node. There is a one-to-one correspondence between $\vec{v}$ and the names in $\mathsf{newVars}$. Suppose that we want to pick out those that correspond to $\mathsf{impVars}^*_n$. We use $\pi_{\mathsf{impVars}^*_n}(\vec{v})$ to denote exactly that, which makes sense since $\mathsf{impVars}^*_n$ appear in $\mathsf{newVars}$.

We proceed by defining one of the main ingredients of the canonization procedure. We map each sub-tree to its corresponding *canonized expression*. For leaf nodes, $\mathsf{canonExpr}_n$ is just the corresponding variable or constant they hold. To be more precise about the variable case, we can re-use the π notation as follows.

$$\mathsf{canonExpr}_n(\vec{x}, \vec{v}) := \pi_{(\mathsf{domVars}_i)}(\vec{x}),$$

where n holds the i^{th} variable name in $\mathsf{domVars}$.

For internal nodes n with atom f, they are built recursively using impObj_f.

$$\begin{aligned}&\mathsf{canonExpr}_n(\vec{x}, \vec{v}) := \\ &\quad \mathsf{impObj}_f\left(\left(\mathsf{canonExpr}_{c_1}(\vec{x}, \vec{v}), \ldots, \mathsf{canonExpr}_{c_k}(\vec{x}, \vec{v})\right), \pi_{\mathsf{impVars}^*_n}(\vec{v})\right).\end{aligned}$$

It is defined over $D \times E$, where $D \times E$ is the full domain of the canonized problem. As usual, we also define $\mathsf{canonExpr}_e$ for a component e to be $\mathsf{canonExpr}_n$ where n is the root of the component e.

In our example, we have:

$$\mathsf{canonExpr}_1(x, y, v, w) := x$$
$$\mathsf{canonExpr}_2(x, y, v, w) := y$$
$$\mathsf{canonExpr}_3(x, y, v, w) := \mathsf{impObj}_{\mathsf{sub}}(x, y) = x - y$$
$$\mathsf{canonExpr}_4(x, y, v, w) := \mathsf{impObj}_{\mathtt{sqrt}}(x - y, v) = v$$
$$\mathsf{canonExpr}_5(x, y, v, w) := \mathsf{impObj}_{\mathsf{neg}}(v) = -v$$

We use v and w instead of $v.4$ and $w.14$ to simplify the presentation but also to remark that the way in which $\mathsf{canonExpr}_n$ is defined, using $\pi_{\mathsf{impVars}^*_n}$, allows us to do so.

Note that this also defines canonized expressions for constraints, as set atoms also have (propositional-valued) atom objective functions. Specifically, they are cone membership expressions; recall that $\mathsf{impObj}_{\mathtt{le}}(x, y) := (y - x) \in \mathbb{R}_+$. For the constraints in our running example, we have:

$$\mathsf{canonExpr}_{e_1}(x, y, v, w) := (2x - 3 - y) \in 0$$
$$\mathsf{canonExpr}_{e_2}(x, y, v, w) := (2 - w) \in \mathbb{R}_+$$

The third constraint is marked for elimination in $\mathsf{vcondElimMap}$ (it is in the set of values of the map), so it is not canonized.

The canonized expression for a given node n can only involve new variables in $E_{\downarrow n}$. For this reason, we define a restricted version $\mathsf{canonExpr}^{\restriction}_n$ with domain $D \times E_{\downarrow n}$. The leaf cases are defined in the same way as $\mathsf{canonExpr}_n$, otherwise:

$$\mathsf{canonExpr}^{\restriction}_n\left(\vec{x}, \vec{v}^{(\downarrow n)}\right) :=$$
$$\mathsf{impObj}_f\left(\left(\mathsf{canonExpr}^{\restriction}_{c_1}\left(\vec{x}, \vec{v}^{(\downarrow c_1)}\right), \ldots, \mathsf{canonExpr}^{\restriction}_{c_k}\left(\vec{x}, \vec{v}^{(\downarrow c_k)}\right)\right), \pi_{\mathsf{impVars}^*_n}(\vec{v})\right),$$

where $\vec{v}^{(\downarrow n)} := \pi_{\mathsf{newVars}_{\downarrow n}}(\vec{v})$, which picks out the v_is corresponding to the variables introduced up to n. As we will see in Section 4.6, this version is needed to argue that the definition of the forward map is well-founded.

It will be useful to also keep track of the *orignal expression* corresponding to every sub-tree. For leaf nodes, it is simply the corresponding variable or constant they hold, just like canonized expressions. For internal nodes, it is defined as follows:

$$\mathsf{ogExpr}_n(\vec{x}) := \mathsf{expr}_f\left(\mathsf{ogExpr}_{c_1}(\vec{x}), \ldots, \mathsf{ogExpr}_{c_k}(\vec{x})\right).$$

These are straightforward to understand as they are exactly the sub-expressions of our atom tree. For example, for T_{e_o} as in Figure 4.1, we have:

$$\begin{aligned}
&\mathsf{ogExpr}_1(x,y) := x \\
&\mathsf{ogExpr}_2(x,y) := y \\
&\mathsf{ogExpr}_3(x,y) := \mathsf{expr}_{\mathsf{sub}}(x,y) = x - y \\
&\mathsf{ogExpr}_4(x,y) := \mathsf{expr}_{\mathtt{sqrt}}(x-y) = \sqrt{x-y} \\
&\mathsf{ogExpr}_5(x,y) := \mathsf{expr}_{\mathsf{neg}}\left(\sqrt{x-y}\right) = -\sqrt{x-y}
\end{aligned}$$

The graph implementation of each of the atoms may introduce new constraints. Indeed, it always introduces new constraints for non-affine function atoms. These constraints are collected in an analogous way to newVars, i.e.,

$$\begin{aligned}
&\mathsf{newConstrs}_{\downarrow n}(\vec{x},\vec{v}) := \\
&\quad \mathsf{newConstrs}^*_n(\vec{x},\vec{v}) +\!\!+ \mathsf{newConstrs}_{\downarrow c_1}(\vec{x},\vec{v}) +\!\!+ \cdots +\!\!+ \mathsf{newConstrs}_{\downarrow c_k}(\vec{x},\vec{v}),
\end{aligned}$$

where we re-use "$+\!\!+$", to denote list concatenation in this case, and

$$\begin{aligned}
&\mathsf{newConstrs}^*_n(\vec{x},\vec{v}) := \\
&\quad \mathsf{impConstrs}_f\left(\left(\mathsf{canonExpr}_{c_1}(\vec{x},\vec{v}),\ldots,\mathsf{canonExpr}_{c_k}(\vec{x},\vec{v})\right), \pi_{\mathsf{impVars}^*_n}(\vec{v})\right).
\end{aligned}$$

As usual, $\mathsf{newConstrs}_e$ means $\mathsf{newConstrs}_{\downarrow n}$, where n is the root node of T_e. We define newConstrs by concatenating the constraints of every component.

Returning to the running example, the following new constraints are added. For the objective function, `sqrt` requires an implementation constraint:

$$\mathsf{newConstrs}_{e_o}(x,y,v,w) = \left[((x-y),0.5,v) \in \mathcal{Q}^3_r\right].$$

The first constraint is affine, so it does not add new constraints, hence:

$$\mathsf{newConstrs}_{e_1}(x,y,v,w) = [].$$

The second constraint, however, requires one new constraint for sq; thus:

$$\mathsf{newConstrs}_{e_2}(x,y,v,w) = \left[(w,0.5,x) \in \mathcal{Q}^3_r\right].$$

Again, since the third constraint is marked for elimination, it is skipped.

Putting everything together, we define the canonize procedure, shown in Figure 4.6. It takes a problem P with components $e_o, e_1, \ldots, e_n$ over variables domVars. It returns the canonized optimization problem. See Figure 4.3 for the

function canonize($e_o, e_1, \ldots, e_p$)**:**
  (T_{e_o}, Ts, failedConstraints) := mkAtomTrees($e_o, e_1, \ldots, e_p$);
  vcondElimMap := mkVcondElimMap(T_{e_o}, Ts, failedConstraints);
  newVars := $\mathsf{newVars}_{e_o}$;
  newConstrs := $\mathsf{newConstrs}_{e_o}$;
  canonConstrs := {};
  for e in $e_1, \ldots, e_n$ **do**
    if e is not marked for elimination in vcondElimMap **then**
      newVars := newVars ++ $\mathsf{newVars}_e$;
      newConstrs := newConstrs ++ $\mathsf{newConstrs}_e$;
      canonConstrs := canonConstrs ++ $\mathsf{canonExpr}_e$;
    end
  end
  $\vec{x}$:= domVars ++ newVars;
  $f := \lambda\vec{x}.\ \mathsf{canonExpr}_{e_o}(\vec{x})$;
  $cs := \lambda\vec{x}.\ (\bigwedge(\mathsf{canonConstrs} +\!\!+ \mathsf{newConstrs}))\,(\vec{x})$;
  return (f, cs);

Figure 4.6: Main canonization procedure.

definition of mkAtomTrees, and Figure 4.5 for the definition of mkVcondElimMap. We say that e is not marked for elimination in vcondElimMap if it is not in the set of values of the map. As expected, the resulting problem is an optimization over $D \times E$. It is clear that the objective function is affine, and all constraints are cone membership constraints since all these expressions are built from applications of impObj and impConstrs to variables and constants. Therefore, the problem is in conic form (see definition 2.2.11).

Applying canonize to the running example in this section yields the following canonized problem:

$$\begin{aligned} \text{minimize} \quad & -v \\ \text{subject to} \quad & (2x - 3 - y) \in 0 \\ & (2 - w) \in \mathbb{R}_+ \\ & ((x - y), 0.5, v) \in \mathcal{Q}_r^3 \\ & (w, 0.5, x) \in \mathcal{Q}_r^3. \end{aligned}$$

Call the components of the canonized problem $r_o, r_1, \ldots, r_4$. There is a correspon-

dence between these and the components of the original problem e_o, e_1, e_2, and e_3, which splits the canonization in three as follows:

- $\{e_o, e_3\}$ corresponds to $\{r_o, r_3\}$: minimizing $-\sqrt{x-y}$ given $0 \leq x-y$ is the same as minimizing $-v$ subject to $((x-y), 0.5, v) \in \mathcal{Q}_r^3$.

- $\{e_1\}$ corresponds to $\{r_1\}$: $y = 2x-3 \Leftrightarrow (2x-3-y) \in 0$.

- $\{e_2\}$ corresponds to $\{r_2, r_4\}$: $x^2 \leq 2 \Leftrightarrow w \leq 2 \wedge (w, 0.5, x) \in \mathcal{Q}_r^3$.

This hints at why the canonization is correct, at least for this particular example. We prove that is always the case in Section 4.6.

4.5.3 Condition inference

In the example above, the only atom with non-trivial vconds was `sqrt`, which requires its argument to be non-negative. Its argument was $x-y$, whose non-negativity was witnessed by the third constraint $0 \leq x-y$. However, in many cases, the condition does not appear as a constraint but is deducible from the other constraints. For example, we could have $1 \leq x-y$ or more complicated combinations such as $0 \leq x \wedge y \leq 2x$.

To enable that, we extend vcondElimMap to hold either constraint identifiers (i.e., $e_1, \ldots, e_p$) or proof terms of $\mathsf{vconds}_f(a_1, \ldots, a_k)$. These proof terms are built by assuming all the constraints (except the current one if the atom appears in a constraint) and trying to prove the variable conditions in that context. In other words, we aim to show that the variable conditions hold in the feasible set. We can keep track of when the assumed constraints are used in the proof term by their identifiers. Note that, in a way, vcondElimMap already held proof terms, albeit they were trivial. We adjust mkVcondElimMap accordingly. In particular, we replace "$\mathsf{vconds}_f(a_1, \ldots, a_k) \neq \top$ and equals an e_j" in mkVcondElimMap (see Figure 4.5) by a proof procedure (in the implementation, this is done using arithmetic tactics).

When it comes to canonize, we need to adjust it to eliminate only those con-

straints that equal a variable condition exactly. Consider the following example:

$$\begin{array}{lll} & & \text{minimize} \quad -v \\ \text{minimize} \quad -\sqrt{x-y} & & \text{subject to} \quad (2x-3-y) \in 0 \\ \text{subject to} \quad y = 2x-3 & \xrightarrow{\text{canonize}} & \quad\quad\quad (2-w) \in \mathbb{R}_+ \\ \quad\quad x^2 \leq 2 & & \quad\quad\quad (x-y-1) \in \mathbb{R}_+ \\ \quad\quad 1 \leq x-y & & \quad\quad\quad ((x-y), 0.5, v) \in \mathcal{Q}_r^3 \\ & & \quad\quad\quad (w, 0.5, x) \in \mathcal{Q}_r^3. \end{array}$$

The example above uses constraint $1 \leq x-y$ to check the variable condition for $\sqrt{x-y}$. We cannot eliminate $1 \leq x-y$ since $(x-y, 0.5, v) \in \mathcal{Q}_r^3$ implies $0 \leq x-y$ (by 4.3.16, for example), but not $1 \leq x-y$. In general, when a variable condition has been established from a non-trivial proof term, there is nothing to eliminate, regardless of the constraints used in the proof term. That is because the constraints used might be stronger than what the implementation constraints capture. Therefore, in this case, $1 \leq x-y$ is treated as a regular constraint and canonized accordingly.

The way in which conditions are handled by our canonization procedure is unique to our approach. In other frameworks, such as `CVXPY`, the constraint $0 \leq x-y$ can be omitted completely, and the canonization is still valid. There is also no notion of condition inference. The approach is to simply include variable conditions as extra constraints before solving the problem.

Because of the formal environment our framework is defined in, we need to know, at the DCP transformation step, that function applications in our problem expression are well-defined. Even though real functions in Lean are usually artificially extended to all of $\mathbb{R}$, i.e., $\sqrt{\cdot} : \mathbb{R} \to \mathbb{R}$ with value 0 for $x < 0$, any formal proof of equivalence between a problem involving $\sqrt{x}$ and a problem in conic form will require proving that $x \geq 0$ holds in the feasible set.

Requiring all domain restrictions to be proved has an advantage: it allows for detecting infeasible and unbounded problems early on. We illustrate it with a simple example, which one can extrapolate to more intricate situations. Imagine a user is trying to minimize $-\sqrt{x}$ with $x \leq -1$. This is a perfectly valid problem in `CVXPY`, which can be canonized and sent to a solver. The solver promptly outputs that the problem is infeasible, albeit without further explanation. By checking the conditions, we know exactly why the problem is infeasible and can tell the user that we failed to show $x \geq 0$.

4.6 Verifying the transformation

Fix a problem $P = (f, cs)$ and let D be the domain of the original optimization variables, i.e., the domain of P. Suppose the canonize procedure (see Figure 4.6) succeeds and outputs a canonized problem $Q = (g, ds)$ over $D \times E$, where newVars : E. All the results in this section assume that canonization was successful.

We show that $P \equiv' Q$, i.e., they are strongly equivalent as in definition 3.3.2. The first step is to define the maps φ and ψ.

Definition 4.6.1 (Backward map)**.** The *backward map* $\psi : D \times E \to D$ is simply a projection:

$$\psi(\vec{x}, \vec{v}) := \vec{x}.$$

The definition of the forward map $\varphi : D \to D \times E$ is a bit more involved. The second part of the output is to be seen as an *interpretation* of the new variables using the original ones, which will be built from the atom solutions (see definition 4.3.11).

We define functions $\varphi_v : D \to E_v$ where $v : E_v$ is in newVars. The subscript may also be a collection of variables, in which case, $\varphi_{(v_1,\ldots,v_l)} : D \to E_{v_1} \times \cdots \times E_{v_l}$. Suppose v is the j^{th} implementation variable introduced in node n, then:

$$\varphi_v(\vec{x}) := \\ \mathsf{sol}_{f,j}\left(\mathsf{canonExpr}^{\restriction}_{c_1}\left(\vec{x}, \varphi_{\mathsf{newVars}_{\downarrow c_1}}(\vec{x})\right), \ldots, \mathsf{canonExpr}^{\restriction}_{c_k}\left(\vec{x}, \varphi_{\mathsf{newVars}_{\downarrow c_k}}(\vec{x})\right)\right).$$

Recall that $\mathsf{sol}_{f,j} : D_f \to E_v$ where D_f is the domain of the arguments of atom f. The recursion is well-founded as new implementation variables in a node are fresh with respect to that of its children. As a small remark, we note that there might not be any new variables in a node, in which case we simply ignore the recursive call to φ, e.g., if c_1 does not introduce new variables, the first argument of $\mathsf{sol}_{f,j}$ in the definition above becomes $\mathsf{canonExpr}^{\restriction}_{c_1}(\vec{x})$.

Combining the solutions of all $\mathsf{impVars}^*_n$ we have:

$$\varphi_{\mathsf{impVars}^*_n}(\vec{x}) := \\ \mathsf{sol}_f\left(\mathsf{canonExpr}^{\restriction}_{c_1}\left(\vec{x}, \varphi_{\mathsf{newVars}_{\downarrow c_1}}(\vec{x})\right), \ldots, \mathsf{canonExpr}^{\restriction}_{c_k}\left(\vec{x}, \varphi_{\mathsf{newVars}_{\downarrow c_k}}(\vec{x})\right)\right).$$

By definition, newVars is composed of $\mathsf{impVars}^*_n$s, so we also have $\varphi_{\mathsf{newVars}}$.

Definition 4.6.2 (Forward map)**.** We define the *forward map* as follows:

$$\varphi(\vec{x}) := (\vec{x}, \varphi_{\mathsf{newVars}}(\vec{x})).$$

In the definition of φ, we could not use $\mathsf{canonExpr}_n$ directly as its arguments are expressions in the extended domain, which is exactly what we were building. However, once we have φ, we can prove the following equality, which is easier to understand and more convenient to use.

Lemma 4.6.1 (Characterization of the forward map in terms of $\mathsf{canonExpr}_n$)**.** Let n be a node in a valid atom tree, then:

$$\varphi_{\mathsf{impVars}_n^*}(\vec{x}) = \mathsf{sol}_f\left(\mathsf{canonExpr}_{c_1}(\varphi(\vec{x})), \ldots, \mathsf{canonExpr}_{c_k}(\varphi(\vec{x}))\right)$$

Proof. We show by induction on the structure of the atom tree rooted at n that

$$\mathsf{canonExpr}_n(\varphi(\vec{x})) = \mathsf{canonExpr}_n^{\restriction}(\vec{x}, \varphi_{\mathsf{newVars}_{\downarrow n}}(\vec{x})).$$

The base case, when n corresponds to a leaf, is trivial. For the inductive step, let f be the corresponding atom and proceed as follows:

$\mathsf{canonExpr}_n(\varphi(\vec{x}))$

$\ldots = \mathsf{impObj}_f\left(\left(\ldots, \mathsf{canonExpr}_{c_i}(\varphi(\vec{x})), \ldots\right), \pi_{\mathsf{impVars}_n^*}(\varphi_{\mathsf{newVars}}(\vec{x}))\right)$

by definition of $\mathsf{canonExpr}_n$

$\ldots = \mathsf{impObj}_f\left(\left(\ldots, \mathsf{canonExpr}_{c_i}^{\restriction}(\vec{x}, \varphi_{\mathsf{newVars}_{\downarrow c_i}}(\vec{x})), \ldots\right), \pi_{\mathsf{impVars}_n^*}(\varphi_{\mathsf{newVars}}(\vec{x}))\right)$

by the inductive hypothesis

$\ldots = \mathsf{impObj}_f\left(\left(\ldots, \mathsf{canonExpr}_{c_i}^{\restriction}(\vec{x}, \varphi_{\mathsf{newVars}_{\downarrow c_i}}(\vec{x})), \ldots\right), \pi_{\mathsf{impVars}_n^*}\left(\varphi_{\mathsf{newVars}_{\downarrow n}}(\vec{x})\right)\right)$

by definition of $\pi_{\mathsf{impVars}_n^*}$, since $\mathsf{newVars}_{\downarrow n}$ contains $\mathsf{impVars}_n^*$

$\ldots = \mathsf{impObj}_f\Big($

$\left(\ldots, \mathsf{canonExpr}_{c_i}^{\restriction}\left(\vec{x}, \varphi_{\mathsf{newVars}_{\downarrow n}}(\vec{x})^{(\downarrow c_i)}\right), \ldots\right), \pi_{\mathsf{impVars}_n^*}\left(\varphi_{\mathsf{newVars}_{\downarrow n}}(\vec{x})\right)\Big)$

by definition of $\varphi_{\mathsf{newVars}_{\downarrow n}}(x)^{(\downarrow c_i)}$, which is short for $\pi_{\mathsf{newVars}_{\downarrow c_i}}(\varphi_{\mathsf{newVars}_{\downarrow n}}(x))$, and equal to $\varphi_{\mathsf{newVars}_{\downarrow c_i}}(x)$

$\ldots = \mathsf{canonExpr}_n^{\restriction}(\vec{x}, \varphi_{\mathsf{newVars}_n}(\vec{x}))$

by definition of $\mathsf{canonExpr}_n^{\restriction}$.

This will allow us to only use $\mathsf{canonExpr}_n$ and ignore $\mathsf{canonExpr}_n^{\restriction}$. □

The following lemmas tell us that conditions hold in original and canonized expressions under certain requirements, and they will be useful later on. Recall that conditions, referred to with conds_f, are the conjunction of background conditions (see definition 4.3.5) and variable conditions (see definition 4.3.6).

Lemma 4.6.2 (Atom conditions hold for original expressions)**.** Let n be a node in an atom tree with atom f and children $c_1, \ldots, c_k$. Pick $\vec{x} : D$ and assume $cs(\vec{x})$, then:

$$\mathsf{conds}_f\,(\vec{o})\,,$$

where $\vec{o} := \Big(\mathsf{ogExpr}_{c_1}(\vec{x}), \ldots, \mathsf{ogExpr}_{c_k}(\vec{x})\Big)$.

Proof. By construction, the mkAtomTree procedure (see Figure 4.2) only succeeds if the background conditions are met for the original expressions instantiating the atom arguments for each atom f. These are actually independent of $cs(\vec{x})$. Therefore, we only need to consider vconds_f.

Variable conditions are checked by mkVcondElimMap (see Figure 4.5), as discussed in Section 4.5.3. The proof terms stored in vcondElimMap are exactly what we require here, so the result follows immediately. □

Lemma 4.6.3 (Atom conditions hold for canonized expressions)**.** Let n be a node in an atom tree with atom f and children $c_1, \ldots, c_k$. Pick $\vec{x} : D$ and $\vec{v} : E$ and assume $ds(\vec{x}, \vec{v})$, then:

$$\mathsf{conds}_f\,(\vec{r})\,,$$

where $\vec{r} := \Big(\mathsf{canonExpr}_{c_1}(\vec{x}, \vec{v}), \ldots, \mathsf{canonExpr}_{c_k}(\vec{x}, \vec{v})\Big)$.

Proof. First, background conditions hold since canonization does not modify any constants; recall that background conditions can only be defined on constants. So it is clear from Lemma 4.6.2. We focus on vconds_f.

We restate the definition of $\mathsf{newConstrs}^*_n(\vec{x}, \vec{v})$ below (new constraints added at node n):

$$\mathsf{impConstrs}_f\left(\vec{r}, \pi_{\mathsf{impVars}^*_n}(\vec{v})\right).$$

It is contained $\mathsf{newConstrs}_{\downarrow n}(\vec{x}, \vec{y})$, and, therefore, appears in $ds(\vec{x}, \vec{v})$, which holds by assumption. Hence, we can apply $\mathsf{vcondElim}_f$ (definition 4.3.16), on $\vec{r}$, $\vec{r}$, and $\pi_{\mathsf{impVars}^*_n}(\vec{v})$, which works in all cases since $\vec{r} \triangle \vec{r}$. From it, we deduce $\mathsf{vconds}_f(\vec{r})$, and we are done. □

In what follows, we will use $\vec{o}$ and $\vec{r}$ to refer to the children original and canonized expressions, respectively.

Having defined φ and ψ (and proved some basic lemmas about conditions), we proceed to show that (φ, ψ) yields a strong equivalence between P and Q.

4.6.1 Forward map properties

In this section we show (φ_{feas}) and (φ_{bound}) as defined in definition 3.3.2. First, we show the following key result about φ.

Lemma 4.6.4 (Relation between original and canonized expressions via φ)**.** Let n be a node in an atom tree from canonize. Pick $\vec{x} : D$ and assume $cs(\vec{x})$. Then:

$$\mathsf{canonExpr}_n(\varphi(\vec{x})) = \mathsf{ogExpr}_n(\vec{x}).$$

Proof. We proceed by induction on the atom tree. The base case is trivial as if n is a leaf, then $\mathsf{canonExpr}_n$ and ogExpr_n are defined in the same way. Now fix an inner node n and proceed as follows:

$$\mathsf{canonExpr}_n(\varphi(\vec{x}))$$

$$\ldots = \mathsf{impObj}_f\left(\left(\ldots, \mathsf{canonExpr}_{c_i}(\varphi(\vec{x})), \ldots\right), \varphi_{\mathsf{impVars}_n^*}(\vec{x})\right)$$

by definition of $\mathsf{canonExpr}_n$

$$\ldots = \mathsf{impObj}_f\left(\left(\ldots, \mathsf{canonExpr}_{c_i}(\varphi(\vec{x})), \ldots\right), \mathsf{sol}_f\left(\ldots, \mathsf{canonExpr}_{c_i}(\varphi(\vec{x})), \ldots\right)\right)$$

by Lemma 4.6.1

$$\ldots = \mathsf{impObj}_f\left(\left(\ldots, \mathsf{ogExpr}_{c_i}(\vec{x}), \ldots\right), \mathsf{sol}_f\left(\ldots, \mathsf{ogExpr}_{c_i}(\vec{x}), \ldots\right)\right)$$

by the inductive hypothesis

$$\ldots = \mathsf{expr}_f\left(\ldots, \mathsf{ogExpr}_{c_i}(\vec{x}), \ldots\right)$$

by $\mathsf{solEqAtom}_f$ (definition 4.3.12) and Lemma 4.6.2

$$= \mathsf{ogExpr}_n(\vec{x})$$

by definition of ogExpr_n.

When the canonized and original expressions are propositional, the "=" can be replaced by "⇔". □

Lemma 4.6.5 (Forward map feasibility (φ_{feas}))**.**

$$\forall \vec{x} : D.\ cs(\vec{x}) \Rightarrow ds(\varphi(\vec{x}))$$

Proof. Assume $\vec{x}$ is a feasible point in P. In other words, assume $cs(\vec{x})$ holds. Ignore all failed constraints that could not be canonized. We obtain a conjunction of constraints $\bigwedge_{j=1}^{m} cs_j(\vec{x})$. Recall that the canonize procedure returns two

types of constraints: canonized constraints and new constraints introduced by the atoms. There are m canonized constraints corresponding to each successful original constraint. We can write

$$ds(\varphi(\vec{x})) = \bigwedge_{j=1}^{m} ds_j^{\text{canon}}(\varphi(\vec{x})) \wedge \bigwedge_{j=1}^{l} ds_j^{\text{new}}(\varphi(\vec{x})),$$

where ds_j^{canon}s are the canonized constraints are ds_j^{new}s are the new constraints.

Since $cs_j(\vec{x}) = \mathsf{ogExpr}_{e_j}(\vec{x})$ and $ds_j^{\text{canon}}(\varphi(\vec{x})) = \mathsf{canonExpr}_{e_j}(\varphi(\vec{x}))$, it follows that

$$cs_j(\vec{x}) \Leftrightarrow ds_j^{\text{canon}}(\varphi(\vec{x}))$$

for all $j \in \{1, \ldots, m\}$ by Lemma 4.6.4, which is in fact stronger than what is required.

For new constraints, suppose ds_j^{canon} is the r^{th} constraint introduced by atom f at node n. We proceed as follows

$ds_j^{\text{new}}(\varphi(\vec{x}))$

$\ldots = \mathsf{impConstrs}_{f,r}\left(\left(\ldots, \mathsf{canonExpr}_{c_i}(\varphi(\vec{x})), \ldots\right), \varphi_{\mathsf{impVars}_n^*}(\vec{x})\right)$

by definition of $\mathsf{newConstrs}_{\downarrow n}$

$\ldots = \mathsf{impConstrs}_{f,r}\left(\left(\ldots, \mathsf{canonExpr}_{c_i}(\varphi(\vec{x})), \ldots\right), \mathsf{sol}_f\left(\ldots, \mathsf{canonExpr}_{c_i}(\varphi(\vec{x})), \ldots\right)\right)$

by Lemma 4.6.1

$\ldots = \mathsf{impConstrs}_{f,r}\left(\left(\ldots, \mathsf{ogExpr}_{c_i}(\vec{x}), \ldots\right), \mathsf{sol}_f\left(\ldots, \mathsf{ogExpr}_{c_i}(\vec{x}), \ldots\right)\right)$

by Lemma 4.6.4

The result now follows by $\mathsf{feasibility}_f$ (definition 4.3.13) together with the fact that the conditions are met for original expressions as established by Lemma 4.6.2. □

Lemma 4.6.6 (Forward map bounding property (φ_{bound}))**.**

$$\forall \vec{x} : D.\ cs(\vec{x}) \Rightarrow g(\varphi(\vec{x})) \leq f(\vec{x})$$

Proof. We show that for any feasible $\vec{x}$, $g(\varphi(\vec{x})) = f(\vec{x})$, which clearly implies the optimality property. Indeed, it suffices to note that $f(\vec{x}) = \mathsf{ogExpr}_{e_o}(\vec{x})$ and $g(\varphi(\vec{x})) = \mathsf{canonExpr}_{e_o}(\varphi(\vec{x}))$ by definition of $\mathsf{canonize}$ (see Figure 4.6). Then the result follows from Lemma 4.6.4. □

The feasibility and bounding property proofs for the forward map above use two of the proof obligations required for every atom: $\mathsf{solEqAtom}_f$ and $\mathsf{feasibility}_f$.

These were applied at key moments in the proofs and were, in some sense, the parts that captured the notion of equivalence. We will use the remaining properties for the backward map: $\mathsf{monotonicity}_f$, bounds_f, and $\mathsf{vcondElim}_f$.

4.6.2 Backward map properties

In this section, we show (ψ_{feas}) and (ψ_{bound}), as defined in definition 3.3.2. In a similar spirit as the previous section, there is one key property about ψ that will be useful to prove both. The idea is that canonized expressions bound original expressions in the correct way, depending on the curvature computed by mkAtomTree. The proof is long and technical and involves considering all possible combinations of curvatures and monotonicities, so we have relegated it to Appendix B. The key steps of the proof require $\mathsf{monotonicity}_f$ and bounds_f.

Lemma 4.6.7 (Canonized expressions bound original expressions)**.** Fix $\vec{x} : D$ and $\vec{v} : E$, and assume $ds(\vec{x}, \vec{v})$. Then, for a node n in an atom tree resulting from canonize, we have that

$$\mathsf{canonExpr}_n(\vec{x}, \vec{v}) \mathbin{\square} \mathsf{ogExpr}_n(\vec{x}),$$

where

$$a \mathbin{\square} b := \begin{cases} a = b & \text{if } \mathsf{compCurv}(n) \sqsubseteq \textsc{AffineFn} \\ a \geq b & \text{if } \mathsf{compCurv}(n) = \textsc{ConvexFn} \\ a \leq b & \text{if } \mathsf{compCurv}(n) = \textsc{ConcaveFn} \\ a \Leftrightarrow b & \text{if } \mathsf{compCurv}(n) = \textsc{AffineSet} \\ a \Rightarrow b & \text{if } \mathsf{compCurv}(n) = \textsc{ConvexSet}. \end{cases}$$

Recall that compCurv matches the curvature labels computed in mkAtomTree (see Figure 4.2). It is important to note that this does not always equal curv_f, in particular when an affine atom is at the root of a convex or concave expression.

Proof. See Lemma B.1. □

Lemma 4.6.8 (Backward map feasibility (ψ_{feas}))**.**

$$\forall \vec{x} : D.\ \forall \vec{v} : E.\ ds(\vec{x}, \vec{v}) \Rightarrow cs(\psi(\vec{x}, \vec{v}))$$

Proof. Assume $(\vec{x}, \vec{v})$ is feasible in the canonized problem Q. This means that $\mathsf{newConstrs}(\vec{x}, \vec{v})$ holds as well as $\mathsf{canonExpr}_{e_j}(\vec{x}, \vec{v})$ for every constraint component

e_j. Recall that ψ is just a projection. Hence, we need to show $cs(\vec{x})$, which involves showing $cs_j(\vec{x})$ for all j. Take any j. There are two possibilities:

- If e_j is marked for elimination, i.e. $\mathsf{vcondElimMap}[n] = e_j$ for some n, then $cs_j(\vec{x})$ is equal to the variable conditions for node n. Suppose n holds atom f and has children $c_1, \ldots, c_k$. More precisely, it must be the case that

$$cs_j(\vec{x}) = \mathsf{vconds}_f\left(\mathsf{ogExpr}_{c_1}(\vec{x}), \ldots, \mathsf{ogExpr}_{c_k}(\vec{x})\right) = \mathsf{vconds}_f(\vec{o}).$$

  Recall that only convex and concave function atoms can have variable conditions. The result will be established by variable condition elimination. First, we need to prove $\vec{r} \triangle \vec{o}$ in the convex case and $\vec{o} \triangle \vec{r}$ in the concave case, where "$\triangle$" is as in definition 4.3.14. If all o_i are constant, then all c_i will have been labeled with ConstantFn so it follows from Lemma 4.6.7 that $\vec{r} = \vec{o}$. Otherwise, it follows from Proposition 4.4.1 that $\mathsf{compCurv}(n) \sqsubseteq \mathsf{curv}_f$. In a valid atom tree, the computed curvature can never be UnknownFn so, since f is convex or concave, we have $\mathsf{compCurv}(n) = \mathsf{curv}_f$. We proceed as follows:

  - Suppose $\mathsf{curv}_f = $ ConvexFn, and consider the possible monotonicity kinds of the inputs of f (see definition 4.3.4). By curvRule, every c_i for an increasing input i of f is labeled with $\sqsubseteq$ ConvexFn, every c_i for a decreasing input i of f is labeled with $\sqsubseteq$ ConcaveFn and the rest are labeled with $\sqsubseteq$ AffineFn. Together with Lemma 4.6.7, this implies that $\vec{r} \triangle \vec{o}$.
  - Suppose $\mathsf{curv}_f = $ ConcaveFn, and consider the possible input kinds. By curvRule, every c_i for an increasing input i of f is labeled with $\sqsubseteq$ ConcaveFn, every c_i for a decreasing input i is labeled with $\sqsubseteq$ ConvexFn and the rest are labeled with $\sqsubseteq$ AffineFn. Together with Lemma 4.6.7, this implies that $\vec{o} \triangle \vec{r}$, as required.

  We know $\mathsf{impConstrs}\left(\vec{r}, \pi_{\mathsf{impVars}_n^*}(\vec{v})\right)$ holds as it is part of $\mathsf{newConstrs}(\vec{x}, \vec{v})$. Therefore, we can use $\mathsf{vcondElim}_f$ (see definition 4.3.16) with $\vec{r}$, $\vec{o}$, and $\mathsf{impVars}_n^*(\vec{v})$ in either the convex and concave case to obtain $\mathsf{vconds}_f(\vec{o})$.

- If e_j is not marked for elimination, then we use Lemma 4.6.7 directly. We apply it to the top node of e_j, which must have been labeled with curvature

$\sqsubseteq$ ConvexSet and, therefore,

$$\mathsf{canonExpr}_{e_j}(\vec{x}, \vec{v}) \Rightarrow \mathsf{ogExpr}_{e_j}(\vec{x}).$$

Since $cs_j(\vec{x}) = \mathsf{ogExpr}_{e_j}(\vec{x})$ and $\mathsf{canonExpr}_{e_j}(\vec{x}, \vec{v})$ holds by assumption, the result follows. □

Lemma 4.6.9 (Backward map bounding property (ψ_{bound}))**.**

$$\forall \vec{x} : D.\ \forall \vec{v} : E.\ ds(\vec{x}, \vec{v}) \Rightarrow f(\psi(\vec{x}, \vec{v})) \leq g(\vec{x}, \vec{v})$$

Proof. Assume that $(\vec{x}, \vec{v})$ is a feasible point in Q, as in the previous lemma. We invoke Lemma 4.6.7 on the top node of e_o, the objective function component. Since it must have curvature $\sqsubseteq$ ConvexFn, it follows that

$$\mathsf{canonExpr}_{e_o}(\vec{x}, \vec{v}) \geq \mathsf{ogExpr}_{e_o}(\vec{x}).$$

We have that $f(\psi(\vec{x}, \vec{v})) = f(\vec{x}) = \mathsf{ogExpr}_{e_o}(\vec{x})$. Also, $g(\vec{x}, \vec{v}) = \mathsf{canonExpr}_{e_o}(\vec{x}, \vec{v})$, by definition of canonize, so we are done. □

We have shown (φ_{feas}), (φ_{bound}), (ψ_{feas}), and (ψ_{bound}). Therefore, we have that $P \equiv' Q$. One remarkable aspect of this work is that the proofs of Lemmas 4.6.5, 4.6.6, 4.6.8, and 4.6.9 are generated automatically given P and Q. Our procedure is constructive, with functions that build proof terms for each of them. The structure of these functions mirrors the steps of the proofs.

4.6.3 Multi-level atom declarations

This section explains how the atom library is extended to allow for *multi-level atom declarations* [GB08, § 4.2]. These lift the restriction of expressing graph implementations in conic form. Instead, previously defined atoms can be used, simplifying the definitions considerably. The verification phase discussed so far in this section is crucial to enable declaring atoms in this way. The user provides evidence (the proofs of proof obligations in the atom declaration) of the correctness of the graph implementation (not in conic form). This graph implementation is then canonized to conic form, and our system generates evidence of these two problems being equivalent. We must chain these two steps in order to produce proofs of the proof obligations of the corresponding atom declaration in conic form.

We assume that we are declaring either a convex or a concave function atom. We will refer to the initial atom declaration as the *uncanonized atom declaration* and the adjusted atom declaration in conic form as the *canonized atom declaration.*

As a motivating example, we consider the unit-halfwidth Huber loss function:

$$h(x) := \begin{cases} x^2 & \text{if } |x| \leq 1 \\ 2x - 1 & \text{if } |x| \geq 1. \end{cases}$$

It is used for robust estimation in statistics as an alternative to the squared error loss to avoid over-penalizing outliers, as it grows linearly outside the interval $[-1, 1]$. The function is convex and can be represented by the following graph implementation over v and w:

$$\begin{aligned} \text{minimize} \quad & 2v + w^2 \\ \text{subject to} \quad & |x| \leq w + v \\ & 0 \leq w \leq 1 \\ & 0 \leq v \end{aligned}$$

An important remark is that it is a problem *parametrized* over x. To give some intuition about why it is correct, we claim that the following are solutions to the problem. If $|x| \leq 1$, the minimum is attained when $w = |x|$ and $v = 0$, and if $|x| \geq 1$, then it is attained when $w = 1$ and $v = |x| - 1$. It is clear that the objective function equals $h(x)$ at the solution.

To state it in this form in the atom library, we must know that it can be canonized to conic form, which involves expanding the absolute value and square atoms and expressing inequalities as memberships to $\mathbb{R}_+$. In CvxLean, this happens when Lean first processes the atom declaration, which ensures that all the atom implementations are stored in conic form. To achieve that, we reuse the canonize procedure (see Figure 4.6), which was designed to be applied to optimization problems but can be re-purposed to work with graph implementations, using only the atoms defined at that point in time.

Applying the canonize procedure to the components of the graph implementation will introduce new variables and constraints. Therefore, the implementation and proof obligations must be adjusted accordingly.

Let f be a new atom with *uncanonized* atom declaration fields class, curv, args (naming $\vec{x} : D$), bconds, vconds, expr, impVars (naming $\vec{v} : E$), impObj, impConstrs, sol, solEqAtom, feasibility, monotonicity, bounds and vcondElim.

First, note that the signature for the canonized atom remains the same:

$$\mathsf{class}' := \mathsf{class}, \quad \mathsf{curv}' := \mathsf{curv}, \quad \mathsf{args}' := \mathsf{args}, \quad \mathsf{bconds}' := \mathsf{bconds},$$
$$\mathsf{vconds}' := \mathsf{vconds}, \quad \text{and} \quad \mathsf{expr}' := \mathsf{expr}.$$

Next, we apply canonize to the components impObj and impConstrs over the variables impVars, which corresponds exactly to canonizing the optimization problem associated with the graph implementation. One adjustment is needed. We need to set the target curvature for the objective curvature in canonize to curv′ so that it works for concave atoms. Recall that graph implementations for convex atoms are minimization problems and so require a convex objective function, whereas those for concave atoms are maximization problems and require a concave objective function (see our discussion about graph implementations in Section 4.3.2).

Note that args are not considered optimization variables; they are treated as parameters. This can be seen from our discussion about graph implementations or from the Huber loss function example earlier. However, we need to assign the appropriate curvature label to them. For each atom argument at position i, if $\mathsf{inputKind}(i) = \textsc{Constant}$, we label it with ConstantFn (and call it a constant parameter); otherwise, we label it with AffineFn (and call it a non-constant parameter). This is important as an uncanonized constraint could involve, say, $\exp(x)$ where x is a non-constant parameter, and we want to make sure that this is expanded and not seen as a constant expression by mkAtomTree (see Figure 4.2). By instructing the system to treat non-constant parameters as affine expressions, we ensure that the resulting implementation is in conic form. Note that if $\exp(x)$ occurs in the implementation and x is a constant parameter, it is acceptable not to expand it since, in that case, x needs to be given a concrete numerical value before solving the problem.

The canonize procedure outputs an optimization problem Q over a set of variables. Using the machinery introduced earlier, we have an equivalence $\left(\varphi^f, \psi^f\right)$ between the graph implementation and Q. See definition 4.6.2 and definition 4.6.1, respectively, for the definitions of these maps. We split Q by its components and call the objective function canonObj, and the *new* variables canonVars (naming $\vec{z} : F$). The resulting constraints are split into canonConstrs and newConstrs as usual. The list canonVars contains the implementation variables needed for the atoms that appear in the graph implementation; for instance, an extra variable

t is needed to replace the square in the objective function of the Huber function atom.

These can be used to define the new graph implementation as follows:

$$\begin{aligned}
\mathsf{impVars}' &:= \mathsf{impVars} +\!\!+ \mathsf{canonVars}, \\
\mathsf{impObj}'(\vec{x}, \vec{v}, \vec{z}) &:= \mathsf{canonObj}(\vec{x}, \vec{v}, \vec{z}), \\
\mathsf{impConstrs}'(\vec{x}, \vec{v}, \vec{z}) &:= \mathsf{canonConstrs}(\vec{x}, \vec{v}, \vec{z}) +\!\!+ \mathsf{newConstrs}(\vec{x}, \vec{v}, \vec{z}).
\end{aligned}$$

Because of our variable naming convention, we know that there is no overlap between impVars and canonVars: user-provided variable names cannot contain dots.

Figure 4.7 summarizes the relationship between the uncanonized atom declaration, the uncanonized problem (which we will call P), the canonized atom declaration (so far), and the canonized problem Q. Next, we proceed to declare sol′, solEqAtom′, feasibility′, monotonicity′, bounds′, and vcondElim′, as defined in Section 4.3.3.

Regarding the solution for the canonized atom, we must find appropriate expressions for the canonVars ($\vec{z} : F$) fields of Q. The map $\varphi^f_{\vec{z}} : D \times E \to F$ gives us values for the $\vec{z}$ fields of Q given values for the $\vec{x}$ and $\vec{v}$ fields of P. Recall that the image of the forward map in the extended domain can be seen as an *interpretation* of the new variables in terms of the old ones. The user-provided sol gives us an interpretation of $\vec{v}$ as expressions in $\vec{x}$. With this in mind, we define

$$\mathsf{sol}'(\vec{x}) := \mathsf{sol}(\vec{x}) +\!\!+ \varphi^f_{\vec{z}}(\vec{x}, \mathsf{sol}(\vec{x})).$$

The first part of the new solution is simply sol, and the second part is the forward map adjusted by replacing each occurrence of v_i by $\mathsf{sol}_i(\vec{x})$. Composing $\varphi^f_{\vec{z}}$ with sol gives a function $D \to F$ defined by an expression in $\vec{x}$. All in all, the new solution sol′ is a well-defined function $D \to E \times F$.

We are only missing the proof data for the new atom. In CvxLean, the canonize procedure outputs proof terms, and these are combined with the user-provided proofs into new proof terms. Here, we take a more mathematical approach and give proof sketches of the five requirements given the signature and implementation above.

First, solEqAtom′ (see definition 4.3.12) follows immediately from solEqAtom.

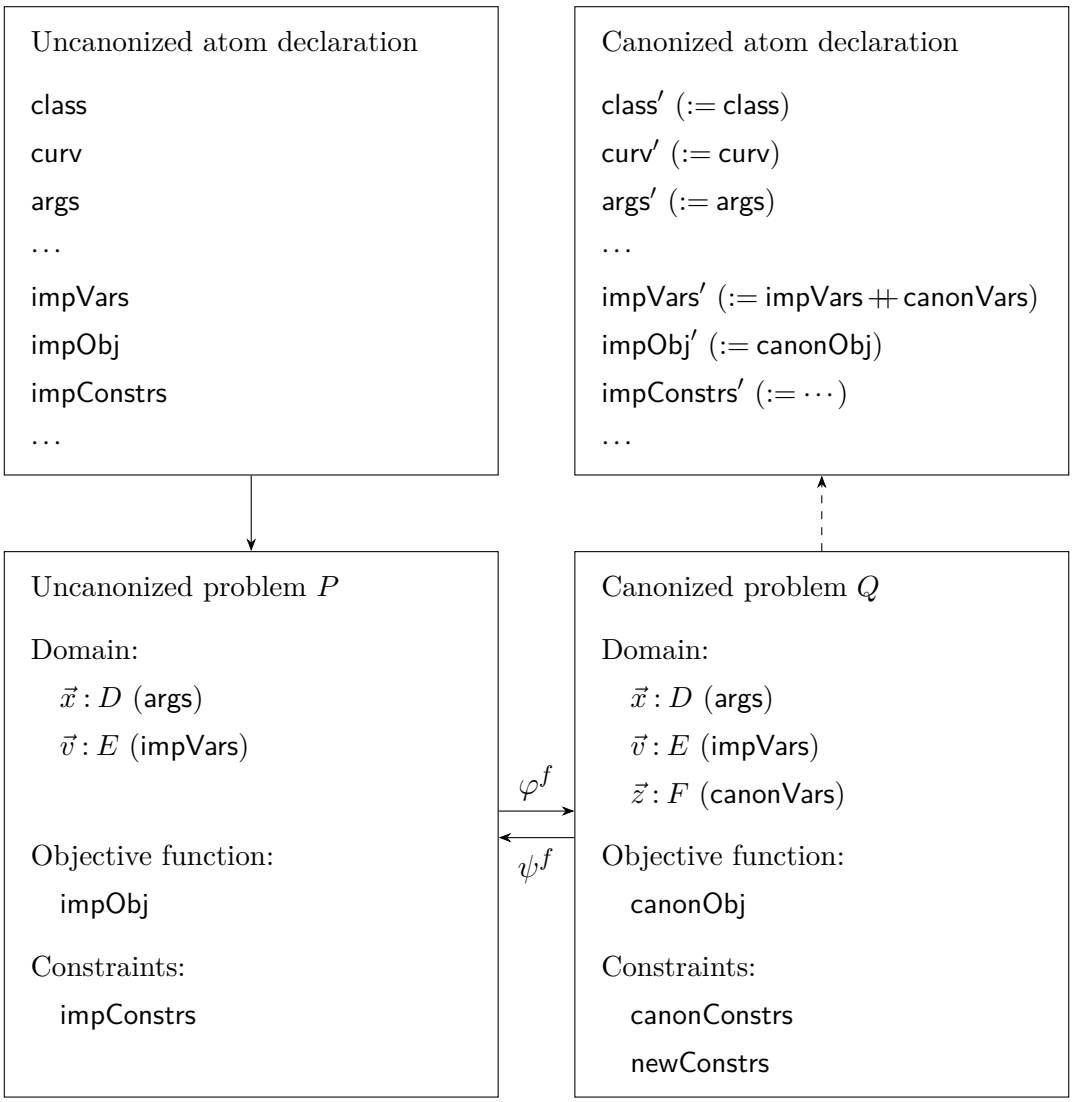


Figure 4.7: Canonizing an atom declaration.

It suffices to note that

$$\begin{aligned}
\mathsf{impObj}'(\vec{x}, \mathsf{sol}'(\vec{x})) &= \mathsf{canonObj}\left(\vec{x}, \mathsf{sol}(\vec{x}), \varphi^f_{\vec{z}}(\vec{x}, \mathsf{sol}(\vec{x}))\right) && \text{by definition,} \\
&= \mathsf{canonObj}\left(\varphi^f(\vec{x}, \mathsf{sol}(\vec{x}))\right) && \text{by definition,} \\
&= \mathsf{impObj}(\vec{x}, \mathsf{sol}(\vec{x})) && \text{by Lemma 4.6.4.}
\end{aligned}$$

Regarding $\mathsf{feasibility}'$ (see definition 4.3.13), for $\vec{x}$ feasible in the graph implementation such that $\mathsf{conds}(\vec{x})$ holds, we must show that

$$\mathsf{impConstrs}'\left(\vec{x}, \mathsf{sol}(\vec{x}), \varphi^f_{\vec{z}}(\vec{x}, \mathsf{sol}(\vec{x}))\right).$$

By assumption, $\mathsf{impConstrs}(\vec{x}, \mathsf{sol}(\vec{x}))$ holds. Thus, we can apply Lemma 4.6.5 on forward map feasibility to $(\vec{x}, \mathsf{sol}(\vec{x}))$ with the forward map φ^f. This gives us the desired result.

The proof of $\mathsf{monotonicity}'$ (see definition 4.3.14) is simply $\mathsf{monotonicity}$ since args and expr remain the same.

For bounds' (see definition 4.3.15), fix $\vec{x}, \vec{y} : D$ and assume $\mathsf{impConstrs}'(\vec{x}, \vec{v}, \vec{z})$ holds. Our goal is to show

$$\mathsf{boundsCond}'(\vec{x}, \vec{v}, \vec{z}),$$

where $\mathsf{boundsCond}'$ is defined as $\mathsf{boundsCond}$ with impObj replaced by impObj'. Note that the atom expression remains unchanged. Since $\mathsf{impConstrs}'(\vec{x}, \vec{v}, \vec{z})$ holds by assumption, we can apply Lemma 4.6.8 on backward map feasibility to obtain $\mathsf{impConstrs}(\vec{x}, \vec{v})$. By bounds, we obtain $\mathsf{boundsCond}(\vec{x}, \vec{v})$. By Lemma 4.6.7 on the relationship between canonized expressions and original expressions, we have that

$$\mathsf{impObj}'(\vec{x}, \vec{v}, \vec{z}) \sqsupseteq \mathsf{impObj}(\vec{x}, \vec{v}). \qquad (\star)$$

It is routine to show that, for $\mathsf{curv}_f = \textsc{ConvexFn}$ and $\mathsf{curv}_f = \textsc{ConcaveFn}$, $(\star)$ combined with $\mathsf{boundsCond}(\vec{x}, \vec{v})$ by transitivity gives $\mathsf{boundsCond}'(\vec{x}, \vec{v}, \vec{z})$, as required.

Finally, $\mathsf{vcondElim}'$ is straightforward. By the same argument as above, from $\mathsf{impConstrs}'(\vec{x}, \vec{v}, \vec{z})$ we deduce $\mathsf{impConstrs}(\vec{x}, \vec{v})$ by backward map feasibility. Knowing that, and since vconds are the same for the canonized atom declaration, we can simply use $\mathsf{vcondElim}$.

We have shown how the restriction of declaring the atom's graph implementation in conic form can be lifted by using the canonization procedure and adjusting all the fields based on its output. As presented, it may look like a two-level declaration since it only requires one extra canonization step to go from the uncanonized declaration to the canonized declaration. However, in the canonization step, we could use graph implementations of other atoms that were not declared in conic form by the user and had to go through the same process before being stored in the library. It is in that sense that what we have presented here allows for multi-level atom declarations.

4.7 Implementation

In this section, we look at the implementation details and show how atoms are declared, how the DCP procedure can be invoked, and how it is ultimately used to call the solver and retrieve a solution.

4.7.1 Declaring atoms

We start with atom declarations and the `declare_atom` command. The definitions in Section 4.3 represent quite faithfully what a user needs to provide in order to declare an atom. However, for historical reasons, there is one key difference in our Lean implementation. Instead of requiring $\mathsf{monotonicity}_f$ and bounds_f, we require a derived property, which we call $\mathsf{optimality}_f$. We opted to present $\mathsf{monotonicity}_f$ and bounds_f instead since they have a clearer mathematical interpretation. Crucially, all atoms satisfy all three properties, and the proof of Lemma 4.6.7 can be adjusted to use $\mathsf{optimality}_f$.

First, we state an auxiliary predicate (cf. boundsCond in definition 4.3.15).

$$\mathsf{optCond}_f(\vec{x},\vec{y},\vec{v}) := \begin{cases} \mathsf{impObj}_f(\vec{x},\vec{v}) \geq \mathsf{expr}_f(\vec{y}) & \text{if } \mathsf{curv}_f = \textsc{ConvexFn} \\ \mathsf{impObj}_f(\vec{x},\vec{v}) \leq \mathsf{expr}_f(\vec{y}) & \text{if } \mathsf{curv}_f = \textsc{ConcaveFn} \\ \mathsf{impObj}_f(\vec{x},\vec{v}) = \mathsf{expr}_f(\vec{y}) & \text{if } \mathsf{curv}_f = \textsc{AffineFn} \\ \mathsf{impObj}_f(\vec{x},\vec{v}) \Rightarrow \mathsf{expr}_f(\vec{y}) & \text{if } \mathsf{curv}_f = \textsc{ConvexSet} \\ \mathsf{impObj}_f(\vec{x},\vec{v}) \Leftrightarrow \mathsf{expr}_f(\vec{y}) & \text{if } \mathsf{curv}_f = \textsc{AffineSet} \end{cases}$$

We define the $\mathsf{optimality}_f$ as follows, depending on the curvature of the atom. For convex function atoms:

$$\forall \vec{x},\vec{y} : D.\ \forall \vec{v} : E.\ \vec{x} \triangle \vec{y} \Rightarrow \mathsf{impConstrs}_f(\vec{x},\vec{v}) \Rightarrow \mathsf{optCond}_f(\vec{x},\vec{y},\vec{v}).$$

For concave function or convex set atoms:

$$\forall \vec{x},\vec{y} : D.\ \forall \vec{v} : E.\ \vec{y} \triangle \vec{x} \Rightarrow \mathsf{impConstrs}_f(\vec{x},\vec{v}) \Rightarrow \mathsf{optCond}_f(\vec{x},\vec{y},\vec{v}).$$

For affine atoms, we need to prove both.

With this in mind, we show how the square root atom is declared in `CvxLean`, following Section 4.3.4. The custom `declare_atom` command is used as follows.

```
declare_atom sqrt [concave] (x : ℝ)+ : Real.sqrt x :=
vconditions (cond : 0 ≤ x)
implementationVars (t : ℝ)
implementationObjective (t)
implementationConstraints
  (c1 : rotatedSoCone x (1/2) ![t])
solution (t := Real.sqrt x)
solutionEqualsAtom by ...
feasibility
  (c1 : by ...)
optimality by ...
vconditionElimination
  (cond : by ...)
```

The first line defines the signature of the atom. After `declare_atom`, the name of the atom is given, which is treated as an identifier. In square brackets, we indicate its curvature, which can be one of `convex`, `concave`, `affine`, `convex_set`, and `affine_set`. This defines curv_f in the expected way. Then, we indicate the atom arguments with their monotonicities, which determine args_f and $\mathsf{inputKind}_f$ as follows:

- "`+`" means it is Increasing,
- "`-`" means it is Decreasing,
- "`?`" means it is Neither, and
- "`&`" means it is Constant.

After the semi-colon, we write the atom expression expr_f. In this case, it is the square root as defined in mathlib.

It is easy to see how the next five lines define vconds_f, $\mathsf{impVars}_f$, impObj_f, and $\mathsf{impConstrs}_f$. Background conditions can also be specified in the same way as variable conditions using the keyword `bconditions` before `vconditions`.

The remaining part of the declaration corresponds to the solution and proof obligations of the atom. The command automatically generates the required fields and expected types. The user must provide a solution for each implementation variable, which must be an expression in the atom's arguments. The user must

also provide proofs in place of the "`...`". For example, the optimality proof obligation corresponds to the following goal in the infoview:

```
x t : ℝ
c1 : rotatedSoCone x (1 / 2) ![t]
⊢ ∀ (x_1 : ℝ), x ≤ x_1 → t ≤ sqrt x_1
```

Affine atoms have a trivial graph implementation: there are no implementation variables or constraints, and the implementation objective is simply the atom's expression. Therefore, no solutions are required, and there are no solution correctness or feasibility proof obligations. The proof of optimality, on the other hand, needs to be provided explicitly; it is usually easy to prove using monotonicity properties of affine operations. To ensure that the expression is affine, we also require:

- A proof of homogeneity:

$$\forall \vec{x} : D.\ \forall \kappa : \mathbb{R}.\ \kappa \mathsf{expr}_f(\vec{x}) + \mathsf{expr}_f(\vec{0}) = \mathsf{expr}_f(\kappa \vec{x}) + \kappa \mathsf{expr}_f(\vec{0}).$$

- A proof of additivity:

$$\forall \vec{x}, \vec{y} : D.\ \forall \kappa : \mathbb{R}.\ \mathsf{expr}_f(\vec{x}) + \mathsf{expr}_f(\vec{y}) = \mathsf{expr}_f(\vec{x} + \vec{y}) + \mathsf{expr}_f(\vec{0}).$$

Recall that a function $A : \mathbb{R}^n \to \mathbb{R}^m$ is affine if there exists a linear function $L : \mathbb{R}^n \to \mathbb{R}^m$ and $\vec{b} \in \mathbb{R}^m$ such that $A(\vec{x}) = L(\vec{x}) + \vec{b}$. Thus, to show that A is affine, it suffices to show that $A(\vec{x}) - A(\vec{0})$ is linear. This justifies the two proof obligations above.

These proofs are used only as a sanity check and are not stored as they are not needed for canonization. As a simple example, consider the definition of the trace operation:

```
declare_atom Matrix.trace [affine]
  (m : Type)& (hm : Fintype m)& (A : Matrix m m ℝ)+ : Matrix.trace A :=
homogeneity by ...
additivity by ...
optimality by ...
```

Note that we are able to specify that `m` is any finite type by setting an instance `Fintype m` as a constant argument, which our procedure will try to infer using type class resolution when looking for matches in the library.

4.7.2 Canonizing and solving problems

In this section, we discuss how the machinery presented in Section 4.5 is realized in `CvxLean`. We briefly discuss some implementation details and then focus on how users can canonize and solve problems.

4.7.2.1 Back-end: the metaprogramming side

As discussed, making the proof-producing canonization procedure work seamlessly behind the scenes required extensive meta-level work. Our goal here is to give the reader some intuition on how this is achieved, focusing on the proof generation aspect.

The method is given the problem instance. It makes atom trees (see Section 4.5.1) and builds a canonized problem (see Section 4.5.2). The key idea is that, for each atom tree, we make other trees with the same structure that hold proof terms from the atom declaration (instantiated with the appropriate arguments in the given problem). For example, we have a solution correctness proof tree, which holds instantiated $\mathsf{solEqAtom}_f$ proof terms (see definition 4.3.12). Recall the key result about the forward map φ, Lemma 4.6.4. The proof worked by induction on the atom tree and used $\mathsf{solEqAtom}_f$ in the inductive step. In the implementation, this corresponds to a bottom-up traversal of the solution correctness proof tree, which, in some sense, folds together all the solution correctness proofs. This generates a proof term that, in combination with all the instantiated $\mathsf{feasibility}_f$ proof terms (see definition 4.3.13), generates yet another proof term of (φ_{feas}), corresponding to Lemma 4.6.5 for this particular problem instance. A similar methodology is used to generate the rest of the proof terms needed to build a strong equivalence.

One drawback of this approach is that it relies on patching together proof terms (`Expr`s). It is difficult to understand exactly what the resulting `Expr` will look like by inspecting the code alone, which poses maintainability challenges. To alleviate that, we provide extensive documentation and have split the functionality into small logical units as much as possible. Devising a more modularized and easy-to-follow strategy would be very valuable, and we hope to see some work in that direction in the future.

4.7.2.2 Front-end: what users see

The implementation of the canonize procedure can be used in tactic mode through the `dcp` tactic. Given an initial problem `p`, if successful, it generates a new problem `q` and a proof term of type `p ≡ q` (i.e., it is an equivalence-preserving transformation like the ones in Section 3.3.2). As expected, it works in both the `reduction` and `equivalence` commands.

The running example of this chapter can be defined as follows:

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

We can canonize it using `dcp` as follows:

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

This adds the term below to the environment, under the name `q`.

```
optimization (x y t.0 t.1 : ℝ)
  minimize -t.0
  subject to
    c₁' : zeroCone (2*x - 3 - y)
    c₂' : nonnegOrthCone (2 - t.1)
    c₃' : rotatedSoCone t.1 0.5 ![x]
    c₄' : rotatedSoCone (x - y) 0.5 ![t.0]
```

This corresponds exactly to the canonized problem shown in Section 4.5.2. As a small remark, the names `c₁'`, `c₂'`, `c₃'`, and `c₄'` are only there for presentation purposes and not generated by the procedure, which currently does not give names to canonized constraints. Also, note that new variables are `t.0` and `t.1`, which differs from the naming convention described in Section 4.5.2. That is because, in the implementation, we do not use atom tree node numbers. Instead, we ensure freshness by updating a counter every time a new variable is introduced.

One rarely wants to canonize the problem into conic form and stop there. Our ultimate goal is to solve the problem. The `solve` command takes a problem, canonizes it, sends it to the solver, and retrieves the solution. We show its usage below.

```
solve p

#print p.conicForm -- shows the canonized problem, in conic form
#eval p.status     -- "PRIMAL_AND_DUAL_FEASIBLE"
#eval p.value      -- -2.101003
#eval p.solution   -- (-1.414214, -5.828427)
```

As we can see, the canonized problem, status, value, and solution are added to the environment. Note that `p.solution` is the solution to `p`, and not `p.conicForm`, which also includes the auxiliary DCP variables in its domain. Similarly, `p.value` is the value of `p`'s objective function at `p.solution`. Interestingly, because they are strongly equivalent, the value of `p` and `p.conicForm` is the same at the optimum (which we proved after definition 3.3.2).

We have yet to explain how the solver is used once the problem is in conic form, leading us to the next section.

4.7.3 Trusted connection to the solver

Once a problem has been canonized 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 numerical computation with floating-point numbers. We traverse our symbolic expressions (`Expr`s) and replace functions defined over $\mathbb{R}$ with the corresponding functions on floats. For example, we replace `rexp` with the floating point exponential function `Float.exp`. We call this the *real-to-float* procedure. Our implementation makes it easy to declare such associations with the following syntax:

```
add_real_to_float : rexp := Float.exp
```

Note that this is unverified and gives us no correctness guarantees.

Let P be the initial problem, with domain D and range $\mathbb{R}$. We apply `dcp` to obtain the canonized problem Q, with domain $D \times E$ and range $\mathbb{R}$. We also

obtain an equivalence that contains a backward map $\psi : D \times E \to D$. Applying the real-to-float procedure to these Lean objects yields three other Lean objects:

- $P_{\mathbb{F}}$, with domain $D_{\mathbb{F}}$ (e.g., if $D = \mathbb{R} \times \mathbb{R}^n$, then $D_{\mathbb{F}} = \mathbb{F} \times \mathbb{F}^n$) and range $\mathbb{F}$.
- $Q_{\mathbb{F}}$, with domain $D_{\mathbb{F}} \times E_{\mathbb{F}}$.
- $\psi_{\mathbb{F}} : D_{\mathbb{F}} \times E_{\mathbb{F}} \to D_{\mathbb{F}}$.

These are computable objects, so we can proceed to extract the required numerical data. Note that $\psi_{\mathbb{F}}$ is the same as the `eqv.backward_map` added to the environment under the `equivalence*` command, described in Section 3.3.1. In the case of DCP, $\psi_{\mathbb{F}}$ is just a projection, but this same method, based on the real-to-float translation, can be used in general to make any backward map computable.

We use `MOSEK` [AA00]. It admits several input formats. One of the most versatile ones is *conic benchmark format* (CBF) [Fri14]. The representation of a problem in CBF is shown below.

$$\begin{aligned} &\text{min / max} && g^{obj} \\ &\text{subject to} && g_i \in \mathcal{K}_i, && i \in \mathcal{I} \\ &&& G_i \in \mathcal{S}_+^{n_i}, && i \in \mathcal{I}^{PSD} \\ &&& x_j \in \mathcal{K}_j, && j \in \mathcal{J} \\ &&& X_j \in \mathcal{S}_+^{n_j}, && j \in \mathcal{J}^{PSD} \end{aligned}$$

$$g^{obj} = \sum_{j \in \mathcal{J}^{PSD}} \left\langle F_j^{obj}, X_j \right\rangle + \sum_{j \in \mathcal{J}} a_j^{obj} x_j + b^{obj}$$

$$g_i = \sum_{j \in \mathcal{J}^{PSD}} \langle F_{ij}, X_j \rangle + \sum_{j \in \mathcal{J}} a_{ij} x_j + b_i$$

$$G_i = \sum_{j \in \mathcal{J}} x_j H_{ij} + D_i$$

Here, the $\mathcal{K}_i$s are cones. The possibilities include $\mathbb{R}$, $\mathbb{R}_+$, $\mathcal{Q}^{n+1}$, $\mathcal{Q}_r^{n+2}$, and $\mathcal{K}_{\text{exp}}$. Recall that $\mathcal{S}_+^n$ is the cone of $n \times n$ positive semidefinite matrices.

We will show how to put $Q_{\mathbb{F}}$ in this form. First, note that we only work with minimization problems. We must determine the number and dimensions of scalar variables, which we can read directly from $E_{\mathbb{F}}$. Note that variables can be members of any cone. We assume they are all in the free cone of the appropriate dimension, vectorizing matrices as needed. In particular, that means that there are no positive semidefinite variables ($\mathcal{J}^{PSD} = \emptyset$), although there can be positive

semidefinite constraints. Having established the domain of the variables $(\vec{x}, \vec{v})$, we proceed as follows:

1. The objective function g^{obj} is determined by a floating-point vector a^{obj} and a floating-point number b. We extract these from $g_{\mathbb{F}}$, the objective function of $Q_{\mathbb{F}}$. We use the fact that it is affine to compute:

$$b^{obj} := g_{\mathbb{F}}(0) \quad \text{and} \quad a_i^{obj} := g_{\mathbb{F}}(\mathbf{e}_i) - b^{obj},$$

   where $\mathbf{e}_i$ is standard basis vector with a 1 in the i^{th} position. Since all constraints are of the form $e(\vec{x}, \vec{v}) \in \mathcal{K}$, where e is affine, we can use the same *coefficient extraction* trick for them.

2. Each (n-dimensional) positive semidefinite constraint corresponds to a G_i constraint, which is determined by a list of floating-point matrices H_i and a floating-point matrix D_i. We proceed with coefficient extraction, with the only caveat that we need to reshape H_i and D_i afterward. Going through the PSD constraints allows us to define $\mathcal{I}^{PSD}$ as well.

3. The rest of the constraints are determined by the cone, a vector a_i, and a number b_i. We extract the coefficients a_i and b_i and store the cone to define $\mathcal{I}$ and $\mathcal{K}_i$ for $i \in \mathcal{I}$.

The resulting numerical data is a^{obj}, b^{obj}, a, b, H, and D. For example, for our running example (see the end of Section 4.5.2), we have:

$$a^{obj} = \begin{pmatrix} 0 & 0 & -1 & 0 \end{pmatrix}, \quad b^{obj} = 0 \quad a = \begin{pmatrix} 2 & -1 & 0 & 0 \\ 0 & 0 & 0 & -1 \\ 1 & -1 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 \end{pmatrix}, \quad b = \begin{pmatrix} -3 \\ 2 \\ 0 \\ 0.5 \\ 0 \\ 0 \\ 0.5 \\ 0 \end{pmatrix}.$$

Note that H and D are not defined as we do not have any positive semidefinite constraints. Let the variables be x, y, v, and w. Clearly, a^{obj} and b^{obj} correspond to $-v$. We can see how matrices a and b correspond to the constraints by noting the following:

- a_1 and b_1 define $2x - 3 - y$.
- a_2 and b_2 define $2 - v$.
- a_3, a_4, a_5 and b_3, b_4, b_5 define $((x - y), 0.5, v)$.
- a_6, a_7, a_8 and b_6, b_7, b_8 define $(w, 0.5, x)$.

This information is written in a sparse coordinate form and, together with the domain and the cones, is put into a `.cbf` file. We call `MOSEK` from Lean as an external subprocess. It writes the result in a `.sol` file.

The solution file contains the status, which is readily added to the Lean environment. If it is anything other than primal and dual feasible, we report the status back to the user and stop. Otherwise, we have a (approximate) solution, i.e., a concrete numerical value for each variable. After re-shaping these values into the correct vector and matrix form, we obtain a point $(x^*, v^*) \in D_{\mathbb{F}} \times E_{\mathbb{F}}$. We use the backward map, which is a projection, and obtain $x^* := \psi_{\mathbb{F}}(x^*, v^*) \in D_{\mathbb{F}}$. This value is put into the environment with the name `p.solution`. Using the objective function of $P_{\mathbb{F}}$, we compute its value at x^* and save it as `p.value`.

This process is unverified, i.e., we do not prove that x^* is a solution. Indeed, it is seldom a solution of the real-valued P, as it is a floating-point approximation (see Section 7.4.4 for a more detailed discussion). By verifying the canonization, however, we can be sure that any numerical errors come from the solver but not the modeling framework.

4.8 Summary

In this chapter, we have shown how the DCP framework can be extended to produce formal proofs of equivalence. This requires proving certain conditions for each atom in the library and then combining them appropriately as we expand the graph implementation of each atom. We also discussed improvements in discharging side conditions and defining complex atoms. Some implementation details are discussed, focusing on the main reason why one wants the problem in conic form: to invoke an external conic solver and obtain a numerical solution.

Making the DCP procedure proof-producing required a deep understanding of canonization and extensive work on the metaprogramming side. Verifying this step was the original motivation for developing `CvxLean` and successfully doing so

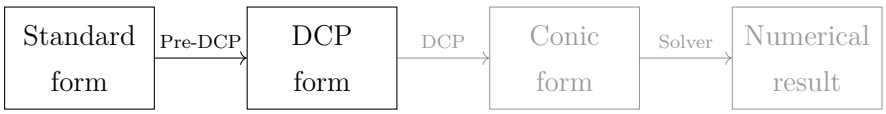
at the degree of generality and user-friendliness that we have is one of our main achievements.

Chapter 5

Rewriting into DCP form

5.1 Introduction

This chapter focuses on the first step of the workflow shown in Figure 1.1, particularly how to automate it, preserving equivalences.



Throughout this chapter, we will use the following running example.

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

This problem is not in DCP form, as the right-hand side of the second inequality, $\exp(x)$, is convex. Now consider the problem below.

$$\begin{aligned} Q := \text{minimize} \quad & x \\ \text{subject to} \quad & 0.001 \leq x \\ & \exp(-x) \leq \sqrt{x} \end{aligned}$$

Problem Q 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, as explained in Figure 5.1. It is not hard to see that P and Q are equivalent, as it is a rewrite (see Section 3.3.2.3) on the second constraint. The challenge is finding the sequence of rewrites (or pre-compositions like the ones in Section 3.3.2.4) to automatically

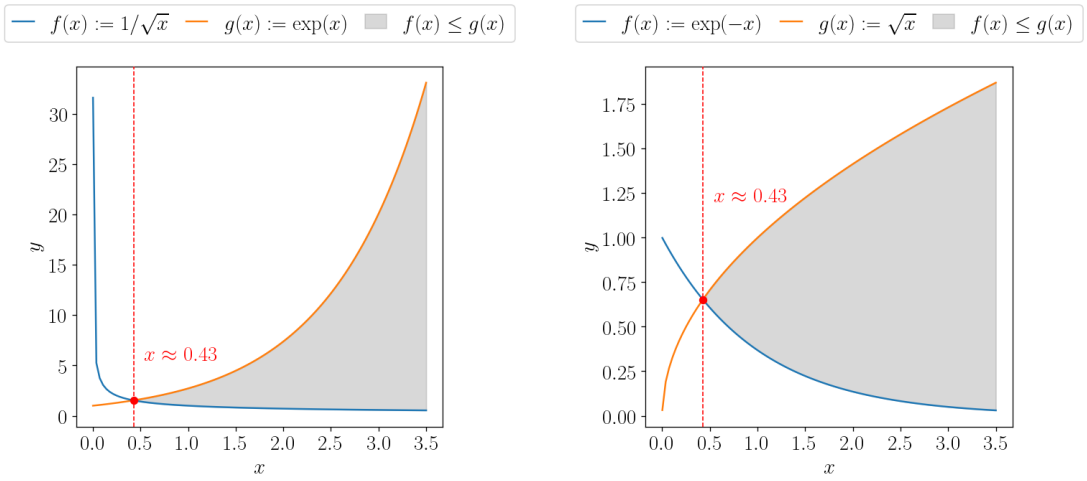


Figure 5.1: Geometric interpretation of the second constraint of P and Q, respectively. Note that both define the same convex set, $\approx \{x \mid 0.43 \leq x\}$. The shaded region can be understood in general as follows: if it is a convex set, then its projection to the x axis is convex; otherwise, it might not be (cf. the DCP top-level rule for "$\leq$").

transform an arbitrary problem in standard form into an equivalent problem in DCP form. Even for our small example above, several steps might be needed.[1]

$$\frac{1}{\sqrt{x}} \leq \exp(x) \quad \rightsquigarrow \quad 1 \leq \exp(x)\sqrt{x} \quad \rightsquigarrow \quad 1 \leq \sqrt{x}\exp(x)$$
$$\cdots \quad \rightsquigarrow \quad \frac{1}{\exp(x)} \leq \sqrt{x} \quad \rightsquigarrow \quad \exp(-x) \leq \sqrt{x}$$

Moreover, some rewrites can be conditional, with non-trivial conditions. In this case, the first step is only valid because the first constraint, $0.001 \leq x$, implies that x is positive and, therefore, $\sqrt{x}$ is positive. An important remark is that the algorithm will not have a target expression to rewrite into, which means that, going back to the example, it will be given P but not Q.

The set of potentially useful rewrites is large and complex. Ideally, we would like this set to be easily extensible so that as the system evolves and can handle more problems, one can simply add the necessary rewrite rules. This means that, without further work, the resulting rewrite system is unlikely to have desirable properties such as the existence of normal forms (see Section 5.3.1).

In some complex rewrite systems like the one at hand, it is sometimes possible to define a heuristic that decreases as one rewrites towards some desired end form, which then can guide the selection and orientation of rewrites [BBHI05].

[1] We use $\rightsquigarrow$ to indicate the intended direction of the rewrite. Later on, we will use $\leftrightsquigarrow$ for bi-directional rewrite rules.

However, in this case, there is no such obvious metric. Situations without a convenient strategy for applying rewrite rules are exactly when *e-graphs* and *equality saturation* are useful.

As in the previous chapter, we will first provide an implementation-agnostic description of the system and only discuss the implementation details in the last section. Section 5.2 serves as a background on e-graphs. In Section 5.3, we show how to build e-graphs that represent sets of equivalent optimization problems. The procedures to extract the term from the e-graph in DCP form, if any, as well as the sequence of rewrites, are explained in Section 5.4. At that point, we will have all the ingredients to construct proofs of equivalence, which we do in Section 5.5 focusing on our running example. In Section 5.6, we show the implementation in `egg` [WNW+21] and how it is verified in `CvxLean`. Finally, we evaluate the algorithm in Section 5.7, showing some performance metrics.

5.1.1 Related work

As discussed in Section 2.2.3 and Section 2.2.4, there exist extensions to DCP that can transform some classes of problems into DCP form. The two most notable extensions are disciplined geometric programming (DGP) [ADB19] and disciplined quasiconvex programming (DQCP) [AB20], both implemented in `CVXPY` [DB16]. The DGP framework transforms geometric programs into equivalent problems in DCP form. This is achieved by designing a library of log-log convex atoms analogous to the DCP atom library, which, instead of specifying canonizations to conic form, specifies canonizations to DCP form. Mathematically, this transformation is equivalent to performing the change of variables $x \mapsto e^u$ and applying logarithms to the objective function and both sides of the inequalities and equalities in the constraints. DQCP also relies on a specialized atom library of quasiconvex atoms. They are transformed into a parametrized problem in DCP form, i.e., a family of problems. The solving procedure needs to be adjusted to search for the optimal parameter. Our approach is not specialized to a particular class of problems and is designed to be extensible so that one can incorporate any rewrite-based transformation that can be used to transform a problem into DCP form. Compared to DGP, our procedure can, in principle, handle any geometric problem after performing the appropriate change of variables, which is straightforward in our system (see Section 3.3.2.1). Generalized geometric programs, however,

often require introducing new variables, which is currently beyond the scope of the algorithm. Our procedure can handle some problems in DQCP form, but they need to be mathematically convex and not only quasiconvex as, otherwise, there is no equivalent DCP form.

The key angle taken by this chapter is using e-graphs to find the appropriate transformation. E-graphs were originally designed for theorem proving [NO80, DNS05] and have since been used effectively for automated theorem proving, most notably in SMT solvers [dMB07]. In this setting, it is crucial to generate *explanations* that justify the equality found by the e-graph [NO05, FCW+22] (see Section 5.2.4). In our work, we use these explanations to build formal proofs of equivalence between optimization problems.

In the context of interactive theorem provers, there has been a lot of interest in implementing *congruence closure* algorithms. For example, Coq has a `congruence` tactic, although it is restricted to the simply-typed subset of the underlying type theory. Thus, much work has focused on making it work for more complex type theories such as Lean 3's [SdM16] or even homotopy type theory in cubical Agda [GS20]. The key functionality of the Lean 3 implementation (which was accessible through the `cc` tactic) was implemented in C++ and has not yet been ported to Lean 4.

These works focus on verification through congruence closure, witnessing the equivalence (or congruence) of terms in a purely syntactic way. In our work, we will use both e-class analysis and extraction based on cost minimization [WNW+21] to find representatives of the equivalence class in DCP form. In this sense, our work is more closely related to other uses of equality saturation that come from program optimization [TSTL09, STL11], where equivalence is necessary, but a particular property of the equivalence class is sought. For example, Herbie [PSWT15] is a tool that uses equality saturation to rewrite arithmetic expressions into mathematically equivalent expressions with improved floating-point accuracy. Another example is SPORES [WHS+20], which optimizes linear algebra expressions by finding rewrites that exploit sparsity and common subexpressions and reduce the number of operations. Diospyros [VNL+21] also optimizes linear algebra expressions in the context of digital signal processors. These techniques have also been applied in deep learning with Tensat [YPW+21] by finding optimizations in tensor computation graphs. They all use e-graphs and equality saturation to optimize a performance metric. This means that the

transformation to a minimum cost term is always acceptable and is seen as an optimization. As shown in Section 5.4, that is not the case for us, as the inspection of the minimum cost term in the e-graph tells us whether or not a transformation is even acceptable.

5.1.2 Contributions

The main contribution of the work reported in this chapter is automating the pre-DCP transformation step. We present an extensible framework for automatically transforming optimization problems into DCP form. Doing this manually is time-consuming and error-prone since many transformation steps might be required, often requiring side conditions to be proved. Organizing these steps and orienting equalities in some simple automated strategy for putting problems in DCP form is challenging. Instead, we automate the search for a useful sequence of steps by making use of equality saturation as implemented in `egg` [WNW+21], which allows us to use e-graphs to represent equivalence classes of optimization problems and their components efficiently. The equality saturation approach allows for the easy automatic handling of a wide range of transformation steps, including those that can work in two opposite directions. The procedure is fully verified in `CvxLean`, ensuring that the proof obligations at every step are met. We show that we are able to transform a wide range of optimization problems into DCP form.

A generic interface between `egg` and Lean was initially developed by Siddharth Bhat and Dr. Andrés Goens, advised by Dr. Tobias Grosser. That work led to interesting discussions about use cases for equality saturation, in particular, to rewrite optimization problems into DCP form. I specialized their interface to the language described in Section 5.3.2, implemented the e-class analysis logic based on interval arithmetic, developed the infrastructure to build proofs of equivalence, and formalized a large number of problems, refining the set of rewrite rules along the way. The design of the extraction mechanisms and the mathematical justification of the algorithm is joint work with Dr. Paul Jackson.

5.2 Understanding e-graphs

In this section, we first show what an e-graph for the second constraint of the running example (see Section 5.1) might look like. The goal is to develop an intuitive

understanding of the result of applying rewrites. Then, we cover some necessary background: we define e-graphs formally and present an abstract framework to reason about equality saturation. At the end, we discuss explanations.

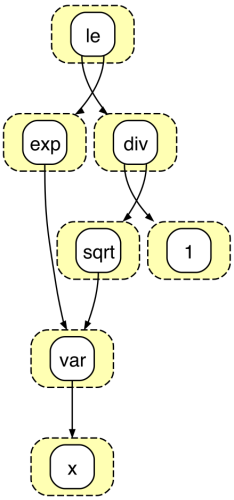
5.2.1 Intuition

We begin by showing how an e-graph is built, ignoring some definitions and technical details for now. An e-graph can be built for an initial single term and then iteratively extended by applying rewrite rules to terms represented by the e-graph. Each application of a rule generates new terms that are then added to the e-graph. Consider the steps in Figure 5.2, which apply rewrites to $1/\sqrt{x} \leq \exp(x)$. The dotted boxes represent *e-classes*. These contain *e-nodes* labeled with a function symbol[2], with arrows pointing to the e-class corresponding to each of its arguments. We can see how successive rewrites add new e-nodes and merge e-classes as appropriate.

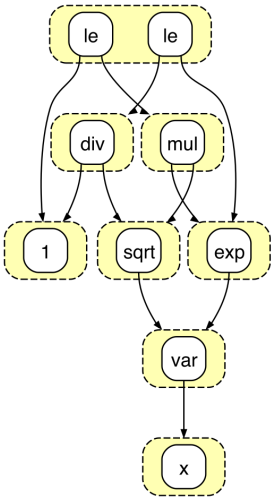
In step (a), each e-class in this e-graph has exactly one e-node. If we begin with the e-node in the class at the top of the diagram, the one corresponding to the whole inequality, we can recursively descend and recover the whole abstract syntax tree (AST) of the inequality. Notice that the variable x appears twice in the inequality, yet the e-class (and e-node) representing it only appears once in the e-graph: one standard e-graph invariant forbids two distinct e-classes representing the same set of terms. Using a well-known fact for real numbers, we can also rewrite the term $1/\sqrt{x} \leq \exp(x)$ to $1 \leq \sqrt{x}\exp(x)$, since we know $\sqrt{x} > 0$ from the other constraint. We explain how such inference works in general in Section 5.3.5. Instead of destructively changing the term, the e-graph stores both equivalent terms by extending the e-classes accordingly. The e-class for the original inequality in (b) has two e-nodes; following each of them will give the two equivalent terms. The more we apply rewrites, the more we grow the graph. For example, with commutativity of multiplication in (c), we add a new e-node on the e-class of the subterm representing $\sqrt{x}\exp(x)$; this new multiplication e-node has the children in a different order, thus representing $\exp(x)\sqrt{x}$ instead.

The rewrite rules that we have shown are specialized to the concrete terms in the e-graph. In reality, we usually use patterns, e.g., commutativity would be $ab \rightsquigarrow ba$, where a and b are patterns. Given an e-graph and a pattern-based

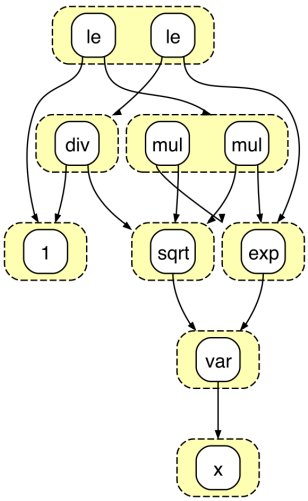
[2]As usual, we treat constants as function symbols with 0-arity.



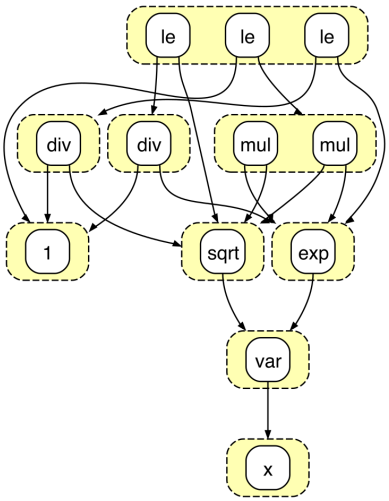
(a) Initial e-graph representing $1/\sqrt{x} \leq \exp(x)$.



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



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



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

Figure 5.2: E-graph building steps for the running example.

rule, a technique called *e-matching* [dMB07] is used to return the e-class where the left-hand-side applies, if any, as well as a substitution of the patterns with concrete e-nodes. We will treat e-matching as a black box in this work.

So far, we have shown how e-graphs support the incremental breadth-first exploration of the space of terms reached by applying rewrite rules to some initial term. Moreover, a clever use of data structures allows us to do so in a space and time-efficient way.

Once an e-graph has been built, a process called *extraction* allows us to select a term in the set of terms represented in the e-graph. As hinted earlier, this is usually achieved by specifying a *cost* function and picking the term that minimizes it. In Section 5.4, we will explain this process in detail. For now, we can assume that there is some way to select a term and that it is equivalent to the initial term modulo the set of rewrite rules given.

5.2.2 Definitions

Fix some term language $\mathcal{L}$ where expressions are of the form $f(e_1, \ldots, e_k)$ and f is a function symbol.

An equivalence relation R over a fixed term language is closed under congruence if for all f, $R(f(a_1, \ldots, a_n), f(b_1, \ldots, b_n))$ whenever $R(a_i, b_i)$ for all i. We usually think of R as a set of equalities. Formally, a congruence closure algorithm is a decision procedure for the quantifier-free theory of equality with uninterpreted function symbols [NO80], which deduces equalities from a set of equalities by computing its congruence closure.

An e-graph [NO80, NO05], or equality graph, is a data structure that compactly represents a set of terms drawn from a fixed language and a congruence relation on that set. At its core, it uses the Union-Find data structure [Tar75] to store the equivalence relation.

An *e-node* n is of the form $f(c_1, ..., c_k)$ where c_i points to an e-class. We call the c_is *e-class IDs*, and we may take them to be in $\mathbb{N}$. An *e-class* $\mathcal{C}$ is a set of e-nodes $\{n_1, \ldots, n_m\}$ or, more precisely, each set of nodes marked as equivalent by the Union-Find data structure forms an e-class.

Definition 5.2.1 (E-graph, formal definition)**.** An e-graph $\mathcal{G}$ over $\mathcal{L}$ is defined by a set of e-classes $\mathfrak{C} = \{\mathcal{C}_1, \ldots, \mathcal{C}_l\}$ that partitions a set e-nodes $\mathcal{N}$. It has a root, which we may take to be $\mathcal{C}_1$.

We may assume that there exists a map H, usually called the *hashcons*, that, given an e-node, returns the e-class ID of the class it belongs to.

The core idea is that of *representing* terms [WNW+21, Definition 2.3]. Let $ts(n)$ be the term set corresponding to e-node n and $ts(\mathcal{C})$ the term set corresponding to e-class $\mathcal{C}$. For an e-graph $\mathcal{G}$, we say that:

- An e-class $\mathcal{C} \in \mathcal{G}$ represents a term t if any $n \in \mathcal{C}$ represents it. In other words, $t \in ts(\mathcal{C})$ if $t \in ts(n)$ for some $n \in \mathcal{C}$.
- An e-node $n = f(c_1, \ldots, c_k)$ represents a term $t = f(e_1, \ldots, e_k)$ with the same function symbol if for all i, the e-class $\mathcal{C}_i$ that c_i points to represents e_i. Alternatively, we have $t \in ts(n)$ if $e_i \in ts(\mathcal{C}_i)$ for all i.

This gives us a concrete way of understanding the equivalence relation that an e-graph stores. To be more precise, we say that t_1 and t_2 are equivalent in $\mathcal{G}$, and write $t_1 \cong_{\mathcal{G}} t_2$ if there exists $\mathcal{C}$ in $\mathcal{G}$ such that $t_1, t_2 \in ts(\mathcal{C})$.

Definition 5.2.2 (Congruence invariant)**.** We say that an e-graph $\mathcal{G}$ satisfies the *congruence invariant* if $\cong_{\mathcal{G}}$ is a congruence relation.

As mentioned earlier, our work is based on `egg`. Definition 5.2.1 of an e-graph differs from theirs [WNW+21, Definition 2.1], which follows `egg`'s implementation. In theirs, the equivalence relation is stored over e-class IDs. They turn out to be mathematically equivalent after adjusting the main functions needed to build and extract from e-graphs accordingly.

The main difference is that two e-nodes with the same function symbol but different e-class IDs in their arguments might be regarded as the same if the e-class IDs are equivalent. They introduce the concept of the *canonical* e-node, where each e-class ID is replaced by the representative in the equivalence relation. An invariant on the hashcons map H ensures that the e-class an e-node belongs to corresponds to that of its canonical e-node. Therefore, in that sense, e-classes partition e-nodes in the same way as in our definition.

We will use our definition for reasoning abstractly about e-graphs and regard theirs as a particular implementation modified for efficiency. The reason for working at the level of e-class IDs is their novel approach based on *rebuilding* [WNW+21, § 3]. Traditionally, every time two e-classes are merged, we must look at the parent e-classes for two reasons: to make them point at the newly merged e-class and to check whether some of them need to be merged to maintain

the congruence invariant, known as *upward merging*. In the rebuilding paradigm, this is not done immediately after each merger, which minimizes the time spent traversing the e-graph. Several rewrites triggering mergers of e-class IDs can be applied before re-establishing the congruence invariant (rebuilding). In the next section, we explain merging in detail.

5.2.3 Reasoning about equality saturation

Here, we provide an abstract framework to reason about *equality saturation*. This refers to the process of building an e-graph. Technically, we say that an e-graph is saturated if no more rewrites apply. However, we will use the phrase "equality saturation" even in cases where we do not reach that point and instead stop early. Indeed, in our use of equality saturation in `CvxLean`, a suitable endpoint for rewriting is typically found before saturation is achieved.

Given an initial term t, we build an e-graph as follows. First, put every node of the AST of t in an e-class, reusing the same e-class for common sub-expressions. At this point, the root e-class only represents t. While the e-graph is not saturated or we reach a user-specified node, iteration, or time limit, execute iterate, defined in Figure 5.3.

```
function iterate():
    for rw in rewrite rules do
        for (σ,C) in G.match(rw.lhs) do
            n := rw.rhs.subst(σ);
            if n in some C' then
                merge(C,C');
            else
                add new e-class {n};
                merge(C,{n});
            end
        end
    end
```

Figure 5.3: One iteration of the equality saturation loop.

As mentioned earlier, e-matching gives us a substitution (σ) consisting of e-nodes in the e-graph that match the pattern on the left-hand side of the rewrite

rule. It also gives us the e-class $\mathcal{C}$ that contains rw.lhs.subst(σ). Note that a rewrite applies if, and only if, such an e-class exists. In reality, it does more than that. Instead of looking through the list of rewrites one by one, as shown in the abstract pseudocode above, e-matching returns all matches at once by using discrimination trees [dMB07].

We substitute σ into the right-hand side of the rewrite to obtain a new e-node n. This e-node might already exist, in which case we merge $\mathcal{C}$ with the e-class that contains it. Otherwise, we add a new singleton e-class and merge it with $\mathcal{C}$.

Next, we will give pseudocode for merge. The equality saturation loop that we have presented is *not* based on rebuilding. Therefore, the congruence invariant must be established after each iteration of the inner for loop. In particular, this means that our merge will also take care of upward merging. We assume that we can retrieve the parent e-nodes of an e-class, together with the e-class they belong to. With that in mind, we define the merging operation in Figure 5.4.

```
function merge(C, C'):
    if C and C' have the same e-class ID then
        return;
    end
    worklist := {};
    for (n'_p, C'_p) in C'.parents do
        make n'_p point to C instead;
        worklist := worklist ∪ {(n'_p, C'_p)};
    end
    update G so that C := C ∪ C';
    remove C' from G;
    for (n'_p, C'_p) in worklist do
        if there is (n_p, C_p) in C.parents s.t. n_p = n'_p then
            merge(C_p, C'_p);
        end
    end
```

Figure 5.4: Merging two e-classes.

We illustrate how merge works with an example. Suppose we have an e-graph with 5 e-classes:

$$\mathcal{C}_1 := \{g(2,3)\},\ \mathcal{C}_2 := \{f(4)\},\ \mathcal{C}_3 := \{f(5)\},\ \mathcal{C}_4 := \{x\},\ \text{and}\ \mathcal{C}_5 := \{y\}$$

The root is $\mathcal{C}_1$, which represents the term $g(f(x), f(y))$. Suppose the rewrite rule $x \rightsquigarrow y$ is applied. This results in a call to merge($\mathcal{C}_4$,$\mathcal{C}_5$). The first loop sets $\mathcal{C}_3 := \{f(4)\}$ and worklist $:= \{(f(4), \mathcal{C}_3)\}$. Then, the e-graph is updated, removing $\mathcal{C}_5$ and setting $\mathcal{C}_4 := \{x, y\}$. At this point, we have an issue, as $\mathcal{C}_2$ and $\mathcal{C}_3$ contain the same e-node. This is our way of indicating that, in order to maintain congruence, they should be merged. The last loop does exactly that, resulting in a call to merge($\mathcal{C}_2$,$\mathcal{C}_3$). Note that although they contain the same elements, they are regarded as different e-classes (different e-class IDs). This removes $\mathcal{C}_3$ and sets $\mathcal{C}_2 := \{f(4)\}$. Both $\mathcal{C}_2$ and $\mathcal{C}_3$ have the same common parent, so the merger terminates after calling merge($\mathcal{C}_1$,$\mathcal{C}_1$). Finally, the e-graph looks as follows:

$$\mathcal{C}_1 := \{g(2,2)\},\ \mathcal{C}_2 := \{f(4)\},\ \text{and}\ \mathcal{C}_4 := \{x, y\}.$$

The terms represented by $\mathcal{C}_1$ are now $g(f(x), f(x))$, $g(f(x), f(y))$, $g(f(y), f(x))$, and $g(f(y), f(y))$.

To see that merge terminates, it suffices to look at the number of e-classes in the e-graph before and after each recursive call. The only way in which the number of e-classes remains unchanged is if the base case is hit; otherwise, it will decrease by at least one.

We reiterate that this algorithm is different from egg's. However, the result of one call to iterate should theoretically be the same regardless of when the congruence invariant is maintained. This allows us to use the (simpler) algorithm that we have presented as the point of reference to reason about equality saturation.

Going back to our running example, imagine that all the rewrite rules are those shown in Figure 5.2 and an extra one, namely $1/\exp(x) \rightsquigarrow \exp(-x)$. After applying this last rule, the equality saturation loop halts, as calls to iterate make no further progress. The resulting e-graph is shown in Figure 5.5. The root e-class represents the following terms, which we split by the terms represented by each of the three e-nodes (n_1, n_2, and n_3 from left to right):

$$\begin{aligned}
ts(n_1) &:= \{1/\sqrt{x} \leq \exp(x)\} \\
ts(n_2) &:= \{\exp(-x) \leq \sqrt{x},\ 1/\exp(x) \leq \sqrt{x}\} \\
ts(n_3) &:= \{1 \leq \sqrt{x}\exp(x),\ 1 \leq \exp(x)\sqrt{x}\}
\end{aligned}$$

The sought term is $\exp(-x) \leq \sqrt{x}$. We proceed by discussing how to instruct the system to extract it.

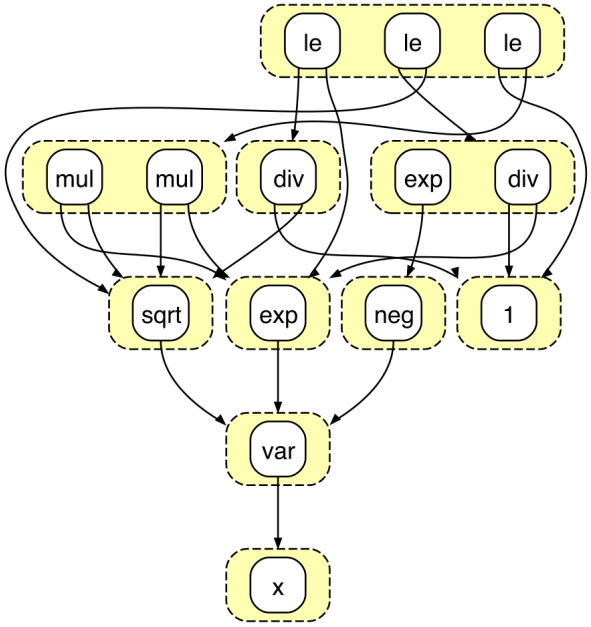


Figure 5.5: Saturated e-graph after the four rules are applied.

In general, extraction works by defining a cost function $\mathsf{cost} : \mathcal{N} \to \mathbf{C}$, where $\mathbf{C}$ is a well-order. This function assumes access to some pre-computed cost map CM that stores the cost of e-classes, which may be initialized to ∞. Each iteration traverses the e-classes and updates the cost map by selecting the minimum cost from the e-nodes in the e-class using cost and CM from the previous iteration. We assume that cost is monotonic with respect to CM: for all e-nodes n, if $CM \leq CM'$ then $\mathsf{cost}_{CM}(n) \leq \mathsf{cost}_{CM'}(n)$. Therefore, the process eventually reaches a fixpoint since $\mathbf{C}$ is a well-order. At that point, we can select an e-node in the root e-class that minimizes the cost. We recursively look into its children e-classes and do the same until we find a term. For this to terminate, we need to assume that the e-nodes that minimize the cost of each e-class do not induce loops, which is not obvious. Loops in e-graphs are associated with infinite families of terms[3], e.g., the e-class induced by $a \cdot 0 \rightsquigarrow 0$. Therefore, it is a reasonable assumption that the cost function should be designed to avoid picking e-nodes that lead to cycles.

Alternatively, extraction can happen "on the fly" by detecting that a term satisfying a required property has been found, stopping the equality saturation

[3]Yihong Zhang gives some nice examples of this and discusses other issues regarding termination in equality saturation in a blog post: `https://effect.systems/blog/ta-completion.html` (accessed 2024-02-25).

loop, and returning such a term. We discuss these two approaches applied to our domain in Section 5.4.

5.2.4 Explanations

In this section, we aim to understand, at a high level, how, given two terms represented by the same e-class, we can find the sequence of rewrites that justifies why they are regarded as equivalent. Such a sequence is called an *explanation*. The standard method to produce explanations was devised by Nieuwenhius and Oliveras [NO05], and versions of it are used in modern e-graph-based systems, including egg [FCW+22].

To discuss explanations properly, we need to look into the implementation of e-classes. So far, we have thought of e-classes as sets of e-nodes. In reality, they are implemented in the Union-Find data structure as spanning trees, where every node is an e-node and every edge in the tree comes from a union operation (which, in turn, comes from merge). The root of the spanning tree serves as the representative of the equivalence class. Unioning two e-classes is achieved by combining two spanning trees into a new one with a new edge. The critical insight to make e-graphs proof-producing is to note that this new edge indicates the reason for the union. There are two possible reasons. It can be an application of a rewrite rule or the result of congruence propagation, i.e., upward merging. Disregarding implementation details, we can imagine that the trees are extended with annotations on the edges. In the case of edges created by propagation, the annotation must somehow point to the edge that caused it.

Given two terms represented in the same e-class, we first take the e-nodes where they are represented. We can now look at the (unique) path[4] between the two e-nodes in the spanning tree and follow the annotations. We start with the most recently added edge in the path, which is the highest one in the tree. If it comes from a rule, we keep track of the rule; if it comes from propagation, we follow the annotation to find the rule that caused it. This effectively splits the path in two, so we proceed recursively, producing explanations for each path. All recursive traces follow a reverse chronological order, so it is clear that the process terminates. In the end, we have a sequence of rewrites, which can be extended

[4] An important remark is that the length of the path depends on the representative chosen at every union. There are heuristic techniques that aim at finding shorter paths. Finding the shortest possible path is known to be NP-complete [FCW+22].

with some extra work to indicate the sub-expression where they were applied.

5.3 E-graphs for optimization problems

In this section, we focus on building e-graphs that encode sets of equivalent optimization problems. We start by providing some more motivation

5.3.1 Why are e-graphs useful in this domain?

First, note that we need bi-directional rewrites. We often want to rewrite inequalities, multiplying and dividing through as needed. For example, we would like to have the rewrite rule $a/b \leq c \leftrightsquigarrow a \leq bc$ for $b > 0$ in our system:

- In the forward direction, we may have $1/\log(x) \leq 1$ with $x > 1$, which we would like to turn into $1 \leq \log(x)$ so that it is DCP-compliant.
- In the backward direction, we may have $x^3 \leq x$ with $x > 0$, which we would like to turn into $x^2 \leq 1$ so that it is DCP-compliant.

Note that in the examples above, we have implicitly used other rewrites, such as multiplying by one; these will also need to be included in the rule set.

Moreover, it is unclear whether we want to multiply and divide or add and subtract instead. Sometimes, indeed, we may want to do both. How to choose which rule to apply has been studied from the early days of computer algebra systems; the PRESS system [SBB$^+$89] is a good example of how several rewriting strategies must be combined in the context of solving symbolic equations. In our case, similar issues arise with the key difference that the acceptable end forms are quite varied as they can be expressions involving several operations. Take, for example, $2 \leq xy + 1$ with $x > 0$, which we would like to transform into $1/x \leq y$. This is the desired form as $1/x$ is DCP convex for $x > 0$. This shows that multiplying through to cancel all denominators, which is a sensible simplification step in many domains, will not work here as it will not ensure compliance with the product-free rules in DCP. Furthermore, we cannot use the number of product-free expressions directly as an optimization metric to guide the rewrites since some expressions involving products might correspond to atoms and are therefore acceptable, such as $x\exp(x)$, which is DCP convex or the entropy function $-x\log(x)$, which is DCP concave.

We have shown thus far that rewrites in both directions are needed and that one cannot simply normalize inequalities by "moving everything to one side", as there are several ways of doing so. Furthermore, it follows from our running example that the complexity of the resulting expression (in terms of, e.g., the number of operations) is insufficient to decide which rule to apply.

Having rewrites in both directions means that the system has no normal forms. With the rules we have shown so far, it is also easy to see that the order in which they are applied matters. Consider the following sequences for $0 < x \leq y$:

$$\begin{aligned} x \leq y \quad &\rightsquigarrow \quad x^2 \leq y^2 \quad \rightsquigarrow \quad 0 \leq y^2 - x^2, \\ x \leq y \quad &\rightsquigarrow \quad 0 \leq y - x \quad \rightsquigarrow \quad 0 \leq (y-x)^2. \end{aligned}$$

All these expressions capture the exact same information, $x \leq y$; however, the resulting expressions on the right-hand sides are syntactically very different.

In conclusion, we have a system where a large number of rewrites apply at any given point, which may lead to infinite non-normalizing chains, and we do not have a simple strategy for picking a rewrite rule. This justifies the need for equality saturation, which allows for an efficient breadth-first search exploration of the space of terms.

5.3.2 Language

Figures 5.2 and 5.5 hint at the existence of an underlying language to encode optimization problems, which we make precise here. Indeed, the first step in setting up any rewrite system is to fix a language of expressions. The formal grammar below describes optimization problems over real variables with arithmetic operations, including logarithms, exponentials, and powers.

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

This language allows us to encode a number of interesting optimization problems. Its function symbols correspond to DCP atoms (see Appendix A). There are several simple unary atoms (u) and binary atoms (b). We distinguish them from composed unary atoms (cu) and composed binary atoms (cb), as these can be expressed in terms of simple atoms. By giving them a constructor, we will ease the curvature analysis needed for extraction. For example, $\log(\exp(x)+\exp(y))$ is expressed using $\mathrm{lse}(x,y)$ (log-sum-exp), and we know immediately that it is a convex function. Without them, if we were to look at the child e-class of a "log" e-node, we would need to look through its term set and check if one of them is of the form $\exp(x)+\exp(y)$ in order to correctly detect if it is convex. The rest are defined follows: $\mathrm{xexp}(x)$ corresponds to $x\exp(x)$, $\mathrm{entr}(x)$ corresponds to $-x\log(x)$, $\mathrm{qol}(x,y)$ (quadratic over linear) corresponds to x^2/y, $\mathrm{geo}(x,y)$ (geometric mean) corresponds to $\sqrt{xy}$, and $\mathrm{norm2}(x,y)$ corresponds to $\sqrt{x^2+y^2}$.

It is clear that there are three types of terms:

- *Problem terms*, generated by the non-terminal prob.
- *Propositional terms*, generated by the non-terminal constr.
- *Real terms*, generated by the non-terminal expr.

Rewrite rules can only operate within these types. In particular, this implies that all terms represented by an e-class will have the same type. We say that an e-class is problem-valued, set-valued, or real-valued, depending on the type of terms it represents.

One of the main limitations of our language is that we do not currently allow vector and matrix expressions, which are common in optimization problems. Doing so is technically possible, and we discuss it as future work in Section 7.4.3.

5.3.3 Realizing ground-term rewrites as logical rules

The language we have just described is a language of ground terms. However, optimization problems as in definition 2.2.1 are described by two first-order expressions of the form $\lambda\vec{x}.f(\vec{x})$ and $\lambda\vec{x}.cs(\vec{x})$. In this case, we assume that the domain is $\mathbb{R}^n$. This section addresses how ground rewrites are lifted to first-order rewrites.

First, we note that the terms built using the "var" constructor are not variables from a logical point of view. Technically, we treat them as Skolem constants.

Given an optimization problem (f, cs), seen as $(\lambda\vec{x}.f(\vec{x}), \lambda\vec{x}.cs(\vec{x}))$, we first extract the variable names from $\vec{x}$, which we assume to be unique. These are used to replace each x_i in the body of the λ-term by $\text{var}(n_i)$, where n_i is the name of the variable x_i. This gives us two ground terms

$$\mathsf{ground}(f) := f(\vec{x})[x_i/\text{var}(n_i)] \text{ and } \mathsf{ground}(cs) := cs(\vec{x})[x_i/\text{var}(n_i)].$$

We use the substitution notation to be very precise about the fact that we are replacing every occurrence of an optimization variable in the body of the λ-terms. The constraints are a conjunction of single constraints, so we split them into

$$\mathsf{ground}(cs_1) := cs_1(\vec{x})[x_i/\text{var}(n_i)],\ \ldots,\ \mathsf{ground}(cs_n) := cs_n(\vec{x})[x_i/\text{var}(n_i)].$$

After replacing the operations in the expressions with the appropriate language constructors, we may think of $\mathsf{ground}(f)$ as a real term and every $\mathsf{ground}(cs_i)$ as a propositional term in our language. Thus, the initial term representing our problem is

$$(\mathsf{ground}(f),\ \mathsf{ground}(cs_1)\ \cdots\ \mathsf{ground}(cs_n)).$$

Next, we must consider what happens when a rewrite rule is applied to this ground term. Assume that the rewrite is applied to a subexpression in the objective function. To be more precise, we consider an example. Let $\mathsf{ground}(f)$ be

$$\text{var}(\text{“}x\text{”}) + \text{var}(\text{“}y\text{”}).$$

Suppose that we apply commutativity and obtain the ground term

$$\text{var}(\text{“}y\text{”}) + \text{var}(\text{“}x\text{”}).$$

Call this term $\mathsf{ground}(g)$, where $g := \lambda(x, y).y + x$. From the point of view of the rewrite system, all we know is that

$$\mathsf{ground}(f) = \mathsf{ground}(g).$$

We intentionally use equality and not “$\rightsquigarrow$” as, ultimately, regardless of the direction of the rewrite rule, the e-graph stores equalities over ground terms.

An important assumption is that rewrite rules cannot introduce new variables or rename variables (they cannot modify the argument of terms built using the “var” constructor). Therefore, we can infer from $\mathsf{ground}(f) = \mathsf{ground}(g)$ that for all x and y,

$$\mathsf{ground}(f)[\text{var}(\text{“}x\text{”})/x][\text{var}(\text{“}y\text{”})/y] = \mathsf{ground}(g)[\text{var}(\text{“}x\text{”})/x][\text{var}(\text{“}y\text{”})/y],$$

by, essentially, ∀-introduction. Note that the equation above is the same as saying that, for all x and y,

$$f(x,y) = g(x,y),$$

which, in turn, is the same as

$$x + y = y + x.$$

This shows we can lift a ground rewrite rule to a first-order rewrite as required by instantiating the optimization variables with Skolem constants, rewriting, and re-introducing the optimization variables.

The example that we have shown is a rewrite of a real term. While we applied it to the objective function term, the same idea can be applied to any real subexpression to lift a ground rewrite to a first-order rewrite. The same idea applies to propositional terms, where "=" is interpreted as "⇔".

Rewrites at the level of problem terms require special consideration. If a rewrite rule tells the rewrite system that

$$(\mathsf{ground}(f),\ \cdots) = (\mathsf{ground}(g),\ \cdots),$$

it does not mean that $\mathsf{ground}(f) = \mathsf{ground}(g)$ but rather that $(f, cs) \equiv_{\text{id,id}} (g, cs)$. In this case, it is less clear how Skolem constants may be treated as logical variables. However, if we unfold "$\equiv_{\text{id,id}}$" (see definition 3.3.1), noting that problem-level rewrites do not change the domain of the problem, we can see that it involves universally quantified statements over the domain of the problem where occurrences of applications of f and g are of the form $f(\vec{x})$ and $g(\vec{x})$. Thus, using the same strategy we used earlier, we can replace the Skolem constants and lift the ground terms to first-order expressions.

In the next section, we assume that ground-term rewrites are lifted to first-order rewrites as described above. In particular, this allows us to ignore Skolem constants altogether and express rewrite rules in terms of logical variables.

5.3.4 Rewrite rules

With a language in place, the next step is to define the rewrite rules of our system. While the "⇝" notation gives an intuitive understanding of how a term is rewritten, in this section, we aim to be more precise and express these transformations in a way that justifies their mathematical correctness. This will be

very important in Section 5.5 when proofs of equivalence need to be constructed from a sequence of rewrites. To that end, we define rewrites as inference rules as follows:

$$\frac{\Gamma_0;\Sigma_0 \vdash \phi_0 \quad \cdots \quad \Gamma_{n-1};\Sigma_{n-1} \vdash \phi_{n-1}}{\Gamma_n;\Sigma_n \vdash \phi_n}\ \mathsf{name}_{\vec{p}}$$

Each of the entailments is to be read as a deduction in sequent calculus, where Σs denote sets of hypotheses and ϕs are logical formulae. We also add free variable contexts Γs. As discussed in the previous section, these are the variables that appear in the problem terms, which we can treat as logical variables. We assume that the set of free variables in any ϕ_i, denoted $\mathrm{fv}(\phi_i)$, is contained in Γ_i.

The formula ϕ_n in the conclusion is always an equivalence relation in our setting ($\equiv$, $\Leftrightarrow$, or $=$). Every rule has a name, and it may be parametrized by metavariables $\vec{p}$, which can be instantiated by mathematical expressions.[5] Hence, every parametrized rule defines a family of rule instantiations. Recall that, by assumption, if these expressions contain free variables, they must be contained in the appropriate context. We will ignore the parameters when they are clear.

Our rewrite system consists of transformations on optimization problems that preserve equivalences of problems by transforming individual constraints or the objective function using, at the time of writing, one of 68 rules (51 of which are bi-directional). The full list can be found in Appendix D.

There are essentially three types of rules, depending on whether they rewrite problems, propositions, or real-valued expressions. In the rules that follow, the optimization variables are assumed to be fixed and indicated by $\vec{x}$.

Problem equivalence rewrites These operate on problem terms. All such rewrites we currently consider only change the objective function and do not touch the constraints. In fact, they are technically pre-compositions, or maps, as defined in Section 3.3.2.4. They are given by the following rule:

$$\frac{\vec{x};cs(\vec{x}) \vdash \forall \vec{y}.\ cs(\vec{y}) \Rightarrow (f(\vec{x}) \leq f(\vec{y}) \Leftrightarrow m(f(\vec{x})) \leq m(f(\vec{y})))}{\vdash (f,cs) \equiv (m \circ f, cs)}\ \mathsf{map\_objFun}_{f,cs,m}$$

Essentially, we require m to be an order embedding on the image of the objective function restricted to the feasible set. We do not have map_objFun in our ruleset, but we have two instances of it. In those cases, the order embedding property usually follows from simpler facts.

[5] Again, note that we depart from our language of ground terms here and rely on the correspondence between ground-term rewrites and first-order rewrites as discussed in Section 5.3.3.

For logarithms, we have the following more straightforward derived rule.

$$\frac{\vec{x}; cs(\vec{x}) \vdash 0 < f(\vec{x})}{\vdash (f, cs) \equiv (\lambda \vec{x}. \log(f(\vec{x})), cs)}\ \mathsf{map\_objFun\_log}_{f,cs}$$

The other rule involves squaring the objective function, as shown below.

$$\frac{\vec{x}; cs(\vec{x}) \vdash 0 \leq f(\vec{x})}{\vdash (f, cs) \equiv (\lambda \vec{x}. (f(\vec{x}))^2, cs)}\ \mathsf{map\_objFun\_sq}_{f,cs}$$

If-and-only-if rewrites These operate on propositional terms, i.e., constraints. Constraints are formed by a list of $\wedge$-ed propositions. Suppose cs consists of n equality or inequality constraints. We assume our rewrite rules target a specific constraint. First, define:

$$cs[i \mapsto cs'_i] := \lambda \vec{x}. \left(\left(\bigwedge_{j=0}^{i-1} cs_j(\vec{x}) \right) \wedge cs'_i(\vec{x}) \wedge \left(\bigwedge_{j=i+1}^{n} cs_j(\vec{x}) \right) \right).$$

In other words, the i^{th} constraint is replaced by cs'_i. With this notation, we define the following rule:

$$\frac{\vec{x}; \{cs_j(\vec{x})\}_{j \neq i} \vdash cs_i(\vec{x}) \Leftrightarrow cs'_i(\vec{x})}{\vdash (f, cs) \equiv (f, cs[i \mapsto cs'_i])}\ \mathsf{rw\_constr}_{f,cs,i,cs'_i}$$

We usually omit all parameters but i. Any if-and-only-if rewrite on the i^{th} constraint results in a problem equivalence by combining it with $\mathsf{rw\_constr}_i$.

An important remark is that to rewrite a specific constraint; we can assume that the other constraints hold. This is crucial as that is how proof obligations in conditional rewrites can be established, as discussed in Section 3.3.2.3.

Our running example uses the following if-and-only-if rewrite:

$$\frac{\Gamma; \Sigma \vdash 0 < b}{\Gamma; \Sigma \vdash a/b \leq c \Leftrightarrow a \leq b \cdot c}\ \mathsf{div\_le\_iff}_{a,b,c}$$

It uses it twice, once in each direction (we will add -rev to the rule name to indicate when the direction is reversed). In one of the applications, it requires the first constraint ($0.001 \leq x$) to be in the context to deduce $0 < \sqrt{x}$.

As a final remark, the $\mathsf{rw\_constr}_i$ rule can be seen as a derived rule of having $\vdash (f, cs) \equiv (f, cs')$ assuming $\vec{x}; \vdash cs(\vec{x}) \Leftrightarrow cs'(\vec{x})$. It is easy to see how it can be combined with the fact that, in propositional logic, $A \wedge B \Leftrightarrow A \wedge C$ is equivalent to $A \Rightarrow (B \Leftrightarrow C)$.

Equality rewrites These operate on real terms. Firstly, we can rewrite the objective function. A rewrite in the objective function yields an equivalence in combination with the following rule.

$$\frac{\vec{x}; cs(\vec{x}) \vdash f(\vec{x}) = f'(\vec{x})}{\vdash (f, cs) \equiv (f', cs)}\ \mathsf{rw\_objFun}_{f,f',cs}$$

We can apply equality rewrites to any sub-expression by using $\mathsf{rw\_constr_i}$ and rw_objFun in combination with the following congruence rules:

$$\frac{\Gamma; \Sigma \vdash a = a' \quad \Gamma; \Sigma \vdash b = b'}{\Gamma; \Sigma \vdash a \triangle b \Leftrightarrow a' \triangle b'}\ ,$$
$$\frac{\Gamma; \Sigma \vdash a_1 = a_1' \quad \cdots \quad \Gamma; \Sigma \vdash a_n = a_n'}{\Gamma; \Sigma \vdash f(a_1, \ldots, a_n) = f(a_1', \ldots, a_n')}\ .$$

Here, $\triangle \in \{=, \leq\}$ and $f : \mathbb{R}^n \to \mathbb{R}$. In our derivations, we will indicate any combination of these rules with congr.

A sub-class of equality rewrites rules is the class of rules that fold and unfold composed atoms. These are better understood with an example:

$$\frac{}{\Gamma; \Sigma \vdash \sqrt{ab} = \mathrm{geo}(a, b)}\ \mathsf{geo\_fold}_{\mathrm{a,b}}$$

Finally, we note that our running example uses the following equality rewrites:

$$\frac{}{\Gamma; \Sigma \vdash \exp(-a) = 1/\exp(a)}\ \mathsf{exp\_neg\_eq\_one\_div}_a, \quad \frac{}{\Gamma; \Sigma \vdash a \cdot b = b \cdot a}\ \mathsf{mul\_comm}_{a,b}.$$

5.3.5 E-class analysis

The remaining ingredient to start building e-graphs is to extend e-classes with the necessary semantic information to apply rewrites. Recall that an e-class is a syntactic object that represents a set of terms. It is often desirable to attach extra attributes that can be used in the building and extraction phases. The collection of attributes attached to an e-class is referred to as an *e-class analysis* [WNW+21]. A good way to understand its usefulness is to consider its role in *conditional rewrites*, where apart from matching the left-hand side, a user-specified property must be satisfied for e-matching to decide that the rewrite applies. For example, in our system, the rewrite $a/a \rightsquigarrow 1$ requires $a \neq 0$. Here, a is a metavariable, and the left-hand side will match an e-node with the function symbol "/", such that both arguments point to the same e-class, say, $\mathcal{C}$. Thus, we are only able to assert the condition if $\mathcal{C}$ is somehow tagged to be non-zero.

E-class analyses may also be used to add new nodes as done in constant folding, where the data is a numerical constant that allows the system to know the value of, e.g., the term $2+2$ and add a new node 4 to the e-class. Finally, they are used in dynamic rewrites, in which the right-hand side of the rewrite rule can depend on the data in the analysis. This differs from "static" rewrites in which the right-hand side is a fixed syntactic pattern. An interesting example of a dynamic rewrite can be found in egg's example of a partial evaluator for λ-calculus, where they keep track of the free variables in the e-class (in the e-class analysis) to avoid capture. The substitution rule is phrased as a dynamic rewrite where the left-hand side is $(\lambda x.b)[e/y]$ and the right-hand side is $\lambda x.b[e/y]$ if x is not free in e or $\lambda z.(b[z/x])[e/y]$ if x is free in e, where z is a fresh variable. Note that dynamic rewrites may also be conditional.

To extend an e-graph with e-class analyses, we first fix an abstract domain D. The analysis of e-class $\mathcal{C}$ is denoted by $d_{\mathcal{C}} \in D$. We require some structure on D so that we can ensure that e-class analyses behave well with the rewrite rules. More precisely, D must be a partially ordered set, where we understand the ordering as an "information ordering" so that if $d_1, d_2 \in D$ and $d_1 \leq d_2$, then if a rewrite applies to e-class $\mathcal{C}$ with $d_{\mathcal{C}} = d_1$, it must also apply to $\mathcal{C}$ with $d_{\mathcal{C}} = d_2$. Technically, this is needed so that equality saturation is monotonically increasing [TSTL09] (which we may understand as non-destructive). Combining two analyses when two e-classes are merged must respect the ordering. In other words, we need to define a join-semilattice $(D, \leq, \sqcup)$, where $\sqcup : D \times D \to D$ is associative, commutative and idempotent. Moreover, we need to define an operation $\mathsf{make} : \mathcal{N} \to D$, which instructs the system how to build $d_{\{n\}} \in D$ for an e-node $n \in \mathcal{N}$.This function is only invoked when a new e-node is added to the e-graph (first if statement in merge). There is also an optional operation modify, which can update an e-class based on its analysis (e.g., to add a node in constant folding). Our system does not use modify, so we will ignore it. Given an e-graph $\mathcal{G}$ with e-class analyses over D, make must satisfy the following property:[6]

$$\forall \mathcal{C} \in \mathcal{G}.\ d_{\mathcal{C}} = \bigsqcup_{n \in \mathcal{C}} \mathsf{make}(n),$$

called the *analysis invariant*. This condition ensures that $d_{\mathcal{C}}$ always exists, disallowing some bad scenarios, such as taking the maximum size of the term in an

[6] As a small remark, we deviate from the unorthodox notation in the egg paper [WNW+21, § 4.1] where the symbol $\bigwedge$ is used for joins instead of $\bigvee$ or $\bigsqcup$.

e-graph representing infinite sets of terms. It also says that the order in which the mergers that created $\mathcal{C}$ were performed is irrelevant. Lastly, it implies that $\mathsf{make}(n) \leq d_{\mathcal{C}}$ for all e-nodes $n \in \mathcal{C} \in \mathcal{G}$.

For supporting checking of conditions of conditional rewrites, we let $D := \mathbb{EIF}$, the *extended interval set*. The set $\mathbb{EIF}$ consists of intervals that may be closed or open in their endpoints. Each interval is of the form ${}_l\lfloor a, b \rceil_r$ where $a, b \in \mathbb{F} \setminus \{\text{NaN}\}$ (the set of non-NaN floating-point numbers) and $l, r \in \{o, c\}$ (signifying "open" or "closed"). We usually write (a, b), $(a, b]$, $[a, b)$ and $[a, b]$ instead of specifying l and r, all of which are elements of $\mathbb{EIF}$. A precise definition of $\mathbb{EIF}$ and its operations is given in Appendix C. Before building the e-graph, we initialize the e-class analyses for constant nodes and potentially some variable nodes. If c is a constant, we have $d_{\{c\}} = [c, c]$. For variables, we traverse the constraints of problem detecting any normalized linear constraints on a single variable (hereafter, *simple constraints*) and translate them as follows, where x is technically var("x"):

$$\begin{aligned}
x \leq c &\quad : \quad d_{\{x\}} := (-\infty, c], \\
x < c &\quad : \quad d_{\{x\}} := (-\infty, c), \\
c \leq x &\quad : \quad d_{\{x\}} := [c, \infty), \\
c < x &\quad : \quad d_{\{x\}} := (c, \infty], \\
x = c &\quad : \quad d_{\{x\}} := [c, c].
\end{aligned}$$

We assume that users indicate restrictions on variable domains in this way, which is common practice when modeling optimization problems. For variables that appear in multiple simple constraints, we take the intersection. If the result is empty, we can immediately say that the problem is infeasible, so we may assume that no analysis for any variable e-class is empty. If no simple constraint is found for a variable, its analysis data is set to $(-\infty, \infty)$.

The initial e-graph is a DAG, and every e-class contains exactly one e-node and represents exactly one term. Variables and constants are at the leaves, which we have already covered. The rest of the initial analyses for real-valued e-classes are computed in a bottom-up fashion using the corresponding operations in $\mathbb{EIF}$. There is no e-class analysis for set-valued or problem-valued e-classes, or, rather, it may be ignored.

It is easy to see how to define $\mathsf{make}(n)$: simply apply the corresponding $\mathbb{EIF}$ operation to n's children e-class analyses. We provide some clarifying examples below where $\mathcal{C}_1$ and $\mathcal{C}_2$ are real-valued e-classes with e-class IDs c_1 and c_2, re-

spectively.

- If $d_{\mathcal{C}_1} = (1, \infty)$ and $d_{\mathcal{C}_2} = [2, 5)$, then, $d_{\{c_1 + c_2\}} = (3, \infty)$.
- If $d_{\mathcal{C}_1} = [-1, 1]$ and $d_{\mathcal{C}_2} = (0, \infty)$, then, $d_{\{c_1 / c_2\}} = (-\infty, \infty)$.
- If $d_{\mathcal{C}_1} = (4, 16)$, then, $d_{\{\sqrt{c_1}\}} = (2, 4)$.

All of the examples above are exact computations, but this might not always be the case; for example, if $d_{\mathcal{C}_1} = [2, 2]$, then $d_{\{\sqrt{c_1}\}}$ is not a singleton but rather an interval $[a, b]$ with $a \leq \sqrt{2} \leq b$, where a and b are floats. In Lemma 5.3.1, we show in full generality that e-class analyses are indeed floating-point approximations of the ranges of the terms they represent.

The examples above only consider computing e-class analyses of e-classes with a single e-node. Once rewrites are applied and e-classes are merged, this data needs to be combined appropriately. We associate a join-semilattice structure to $\mathbb{EIF}$ as follows:

$$I_1 \leq I_2 := I_2 \subseteq I_1, \quad I_1 \sqcup I_2 := I_1 \cap I_2.$$

It may seem counter-intuitive at first, but the idea is that the smaller the interval, the larger the information it contains. We should never join two e-classes whose analyses are disjoint; if that happens, the algorithm fails. Assuming that the operations on $\mathbb{EIF}$ are correct approximations and that all the rewrite rules for real-valued expressions are sound, it will never be the case as established by Lemma 5.3.3 at the end of Section 5.3.5.1.

Finally, we can specify how the rewrite system uses this information. Suppose the left-hand side of a conditional rewrite involves a metavariable a. If an e-node matches said left-hand side, a is assigned an e-class $\mathcal{C}$. Suppose that the rule specifies a condition on a. The system will retrieve $d_{\mathcal{C}}$ to try to prove that the condition is satisfied, and hence the rule applies. In our rewrite system, this involves checking one of the following conditions:

- Positivity: $d_{\mathcal{C}} \subseteq (0, \infty)$.
- Negativity: $d_{\mathcal{C}} \subseteq (-\infty, 0)$.
- Nonnegativity: $d_{\mathcal{C}} \subseteq [0, \infty)$.
- Nonpositivity: $d_{\mathcal{C}} \subseteq (-\infty, 0]$.

- Nonzeroness: $[0,0] \not\subseteq d_{\mathcal{C}}$.
- Being in $\mathbb{N}$: $d_{\mathcal{C}} = [n,n]$ and $n \in \mathbb{N}$.

It is also used in a crucial way by the extraction mechanism to calculate the curvature of expressions, which we discuss in Section 5.4.

5.3.5.1 E-class analyses are never empty

Proving this property of e-class analyses is slightly technical and will require reasoning about equality saturation as outlined in Section 5.2.3. First, we present some useful terminology.

In Appendix C, we define an operation $\rho : \mathbb{EIF} \to \mathcal{P}(\mathbb{R})$, called a *realization* that returns the real interval represented by an interval in $\mathbb{EIF}$ in an obvious way. For every function symbol f in our language, $\widehat{f}$ is its lifted counterpart in $\mathbb{EIF}$.

Definition 5.3.1 (Real range of a term)**.** Let t be a term in our language with free variables in $\vec{x} = (x_1, \ldots, x_n)$. Assume each x_j is restricted to an interval $\rho(I_j)$ where $I_j \in \mathbb{EIF}$:

$$r(t) := \left\{ t[i_j/x_j] \mid i_j \in \rho(I_j) \text{ for all } j \text{ and } \vec{i} \in \mathrm{dom}(\lambda \vec{x}.t) \right\} \subseteq \mathbb{R},$$

where $t[i_j/x_j]$ means, as usual, substituting each x_j by the concrete value i_j. We abuse notation slightly since t is a symbolic expression; however, it is clear how each function symbol corresponds to a real function.

The above assumes that the variables are restricted to intervals defined using floating-point numbers. This is precisely how our initial analyses are built. Note that $r(t)$ may only be empty if one of the variable intervals is empty or it is undefined everywhere.

Definition 5.3.2 ($\mathbb{EIF}$ interval of a term)**.** In the same setting as definition 5.3.1, we define:

$$i(t) := \begin{cases} [c,c] & \text{if } t \text{ is a constant } c \text{ (a float)}, \\ I_j & \text{if } t \text{ is a variable } x_j, \\ \widehat{f}(i(a_1), \ldots, i(a_k)) & \text{if } t = f(a_1, \ldots, a_k). \end{cases}$$

This is exactly how analyses are initialized.

The following proposition follows from the key property of $\mathbb{EIF}$.

Proposition 5.3.1 (Real ranges and intervals)**.** For all real terms t, $r(t) \subseteq \rho(i(t))$.

Recall the merge operation introduced in Section 5.2.3, where $\mathsf{merge}(\mathcal{C},\mathcal{C}')$ merges the two e-classes into $\mathcal{C}''$ and repairs the rest of the e-classes to maintain congruence. We assume the following property: after the merge $d_{\mathcal{C}''} := d_{\mathcal{C}} \sqcup d_{\mathcal{C}'}$, which, in our case, is $d_{\mathcal{C}} \cap d_{\mathcal{C}'}$.

Lemma 5.3.1 (Real ranges and e-class analyses)**.** If t_1 and t_2 are represented by a real-valued e-class $\mathcal{C}$, then $r(t_1) = r(t_2) \subseteq \rho(d_{\mathcal{C}})$.

Proof. We show that every call to merge with real e-classes in the equality saturation loop maintains this invariant. That is sufficient since it is the only place where e-classes are updated. The base case (before merge is ever called) follows from Proposition 5.3.1 and the fact that every e-class in the initial e-graph represents exactly one term. There are three cases to consider:

- Adding a new e-node n and merging $\mathcal{C}$ with $\{n\}$. A real equality rewrite must trigger the merger, and there must be a term t_1 represented by $\mathcal{C}$ and a term t_2 represented by n such that $t_1 \rightsquigarrow t_2$ through said rewrite. Since rewrite rules are equalities in $\mathbb{R}$, t_1, and t_2, regarded as real functions, are equal in the current variable context. In particular, that means that $r(t_1) = r(t_2)$. By assumption, $r(t_1) = r(t_1')$ for all t_1' represented by $\mathcal{C}$ and $r(t_2') = r(t_2)$ for all terms represented by n. Therefore, the real range of all terms represented by $\mathcal{C} \cup \{n\}$ coincides. Moreover, $r(t_1) \subseteq \rho(d_{\mathcal{C}})$ and $r(t_2) \subseteq \rho(d_{\{n\}})$. We assume that intersection in $\mathbb{EIF}$ is a correct approximation, i.e., $\rho(d_{\mathcal{C}}) \cap \rho(d_{\{n\}}) \subseteq \rho(d_{\mathcal{C}} \cap d_{\{n\}})$. Thus, it follows that $r(t_1) = r(t_2) \subseteq \rho(d_{\mathcal{C}'})$ where $\mathcal{C}'$ is the result of merging $\mathcal{C}$ and $\{n\}$.

- Merging two existing e-classes $\mathcal{C}$ and $\mathcal{C}'$. The same argument replacing $\{n\}$ with $\mathcal{C}'$ works for this case.

- Upward merging to repair congruence. This case involves the recursive calls in merge that ensure congruence closure. The merger can only be triggered if two classes, $\mathcal{C}_p$ and $\mathcal{C}_p'$ contain the same e-node n of the form $f(c_i, \ldots, c_k)$. Thus, $ts(\mathcal{C}_p)$ and $ts(\mathcal{C}_p')$ overlap at least in $ts(n)$. Hence, once again, it is clear that terms represented by either e-class have the same real range, and the same argument works. □

Lemma 5.3.2 (Non-empty ranges)**.** If the initial problem is well-formed and feasible, then any real term t represented by the e-graph has $r(t) \neq \emptyset$.

Proof. If t is a variable x_j, then $r(t) = I_j$, and for the problem to be feasible, we must have $I_j \neq \emptyset$. If t is a constant c, then clearly $r(t) = \{c\} \neq \emptyset$. Finally, if t is a function application $f(a_1, \ldots, a_n)$, then proceed by contradiction. If $r(t) = \emptyset$ then $(r(a_1), \ldots, r(a_n)) \cap \mathrm{dom}(f) = \emptyset$, so the problem is not well-formed. □

Technically, there are many empty intervals in $\mathbb{EIF}$ (e.g., every interval of the form (a, a)). We say that $I \in \mathbb{EIF}$ is nonempty if it does not equal any empty interval or, equivalently, $\rho(I) \neq \emptyset$.

Lemma 5.3.3 (E-class analyses are never empty)**.** Assuming the original problem is feasible and well-formed, $d_{\mathcal{C}}$ is nonempty for all real e-classes $\mathcal{C}$.

Proof. This follows immediately from Lemma 5.3.1 and Lemma 5.3.2. □

5.4 Extraction mechanisms

We have considered two different extraction mechanisms. The first one, explained in Section 5.4.1, involves using a cost function, which is the standard approach described in Section 5.2.3. The second one, explained in Section 5.4.2, involves extending the e-class analysis domain to also include curvature information and the best term found so far and stopping as soon as the root e-class is labeled as convex.

5.4.1 Using a cost function

We show how to design a cost function to detect if a term in DCP form exists in the e-graph. In our discussion earlier, we assumed that its range (the cost type) is totally ordered. We will temporarily break this assumption, and later on, we will justify why this approach works. In particular, we define a partial order as follows.

Our cost type is $\mathbf{C} \times \mathbb{N} \times \mathbb{N}$ ordered lexicographically. The curvature type $\mathbf{C}$ is the disjoint union of $\mathbf{C}_f$, $\mathbf{C}_s$, and $\mathbf{C}_p$ shown in Figure 5.6. Note that these labels (except the problem labels) are not new and were already presented in Chapter 4, in particular in definition 4.3.2. The second is the number of variable occurrences with multiplicity so that, e.g., $2 \cdot x$ has a lower cost than $x + x$. The

third component represents the size of the expression. The last two allow us to break ties if there are several problems in DCP form. In that case, a sensible requirement is to select the one that is, in some sense, the simplest.

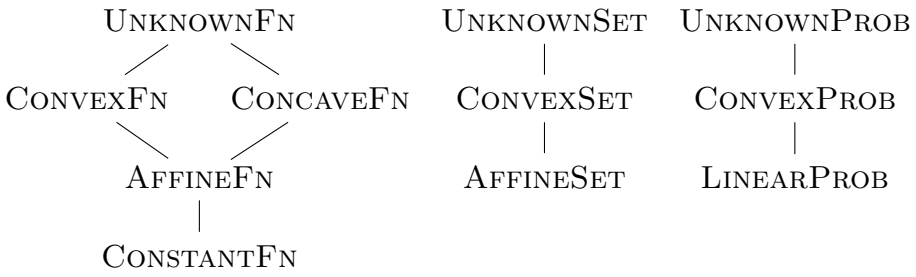


Figure 5.6: Ordering on curvatures of real terms ($\mathbf{C}_f$), propositional terms ($\mathbf{C}_s$), and problem terms ($\mathbf{C}_p$). Semantically, each label represents a set, and each ordering (visualized as a Hasse diagram) corresponds to set inclusion on the representations.

We present the part of our cost function from $\mathcal{N}$ to $\mathbf{C}$, ignoring variable occurrences and term size calculations, which are routine. Recall that this function is iterated until it reaches a fixpoint. At every iteration, it updates the cost map (CM) by computing the cost of every e-class. It is initialized to UnknownFn, UnknownSet, or UnknownProb depending on the type of the e-class.

First, define the following helper function to compute the costs of whole problems from the cost of the objective function (u_f) and the cost of the constraints ($u_1, \ldots, u_n$). As expected, it corresponds to the top-level DCP rule for problems.

$$\mathsf{probCost}(u_f, \{u_i\}_i) := \begin{cases} \textsc{LinearProb} & \text{if } u_f \sqsubseteq \textsc{AffineFn} \\ & \text{and } u_i \sqsubseteq \textsc{AffineSet} \text{ for all } i, \\ \textsc{ConvexProb} & \text{if } u_f \sqsubseteq \textsc{ConvexFn} \\ & \text{and } u_i \sqsubseteq \textsc{ConvexSet} \text{ for all } i, \\ \textsc{UnknownProb} & \text{otherwise.} \end{cases}$$

Next, we define cost on e-nodes n.

$$\mathsf{cost}(n) := \begin{cases} \mathsf{curvRule}(\mathsf{a}, CM[c_1], \ldots, CM[c_k]) & \text{if } n = \mathsf{a}(c_1, \ldots, c_k) \\ & \text{and } \mathsf{conds}_\mathsf{a}(d_{\mathcal{C}_1}, \ldots, d_{\mathcal{C}_k}), \\ \mathsf{curvRule}(\triangle, CM[c_1], CM[c_2]) & \text{if } n = c_1 \triangle c_2, \\ \mathsf{probCost}(CM[c_f], \{CM[c_{cs_i}]\}_i) & \text{if } n = (c_f, \{c_{cs_i}\}), \\ \textsc{UnknowFn} & \text{otherwise.} \end{cases}$$

We indicate with c e-class IDs. The case $\mathsf{a}(\mathcal{C}_1, \ldots, \mathcal{C}_k)$ corresponds to real e-nodes with function symbol a. Note that a is an atom in the atom library, and hence we can apply $\mathsf{curvRule}$ (see Section 4.4). It depends on the curvature of its arguments and is only defined if the atom conditions are met. Recall that each $d_{\mathcal{C}_i}$ is an interval in $\mathbb{EIF}$. We write $\mathsf{conds}_f(d_{\mathcal{C}_1}, \ldots, d_{\mathcal{C}_k})$ to mean $\forall \vec{x} \in (d_{\mathcal{C}_1}, \ldots, d_{\mathcal{C}_k}).\ \mathsf{conds}_f(\vec{x})$, where conds_f is as in definition 4.3.6. Similarly, $\mathsf{curvRule}$ is also defined for set atoms, in particular, `eq` and `le`. The notation $(c_f, \{c_{cs_i}\})$ indicates that it is a problem e-node with its respective objective function e-class ID and constraint e-class IDs. Note that each of the first three cases in the definition of cost can only output a label in $\mathbf{C}_f$, $\mathbf{C}_s$, and $\mathbf{C}_p$, respectively. The last case can only occur for atoms a where the conditions are not satisfied.

It might be helpful to remind how the curvature rules are applied. For example, the rules for negation are straightforward: they swap ConvexFn and ConcaveFn and leave the rest of curvatures the same. Others, such as multiplication (discussed after definition 4.3.5), are a little more subtle. In that case, we rely on the fact that there are four multiplication atoms depending on which argument is constant and whether said constant is non-negative or non-positive. For instance, if the first argument is constant and non-negative, the resulting curvature is the curvature of the second argument. To give an example of a non-affine atom, we can consider the square root (recall Section 4.3.4), assuming that the e-class analysis of its argument indicates that it is non-negative:

$$\mathsf{curvRule}(\texttt{sqrt}, u) := \begin{cases} \textsc{ConstantFn} & \text{if } u = \textsc{ConstantFn} \\ \textsc{ConcaveFn} & \text{if } u \sqsubseteq \textsc{ConcaveFn} \\ \textsc{UnknownFn} & \text{otherwise.} \end{cases}$$

Finally, set atoms are as follows. Both equalities and inequalities are labeled with AffineSet when both expressions are $\sqsubseteq$ AffineFn. Inequalities are labeled with AffineSet if both sides are $\sqsubseteq$ AffineFn, and with ConvexSet when the left-hand side is $\sqsubseteq$ ConvexFn, and the right-hand side is $\sqsubseteq$ ConcaveFn. Otherwise, their curvature is unknown, hence labeled with UnkwownSet.

Having defined cost, we discuss the overall cost calculation procedure. At each iteration, for each e-class $\mathcal{C}$ with e-class ID c, we look at its e-nodes and update CM as follows.

$$CM'[c] := \bigsqcap_{n \in \mathcal{C}} \mathsf{cost}(n).$$

For example, the first iteration only sets the variable e-classes to AffineFn and the constant e-classes to ConstantFn.

The first observation is that all e-nodes in the same e-class must be of the same type, which justifies why $\mathbf{C}$ can be split into three disjoint orders. In general, however, different e-nodes in the same e-class might have different curvatures (within the same curvature order). For example, $\log(\exp(x))$ and x can be represented by the same e-class (different e-nodes), but they should be labeled with UnknwonFn and AffineFn, respectively. This is, in fact, necessary; otherwise, we could not use this method to transform our problems into DCP form. In this case, the e-node that represents x will be preferred as it has a lower cost. Indeed, any e-node with "unknown" curvature will be avoided as much as possible as it can never lead to a problem in DCP form.

More interestingly, note that we take the meet and not the minimum[7], which is necessary since $\mathbf{C}_f$ is not total. We need to account for the case when the computed e-node costs of an e-class at an iteration contain ConcaveFn and ConvexFn (which we will call incomparable costs). To give an example, consider the terms $\text{geo}(2 \cdot x, x)$ (mathematically equal to $\sqrt{2 \cdot x \cdot x}$) and $\text{norm2}(x, x)$ (mathematically equal to $\sqrt{x^2 + x^2}$). It is easy to see how to rewrite one to the other and hence could be represented by the same e-class. The curvature rules for the geometric mean and the norm label them as ConcaveFn and ConvexFn, respectively, provided $x > 0$. Thus, the e-class will be labeled with AffineFn, following our calculation. This works under the assumption that our rewrite system contains enough rules so that the e-class also represents an affine term; in this case, $\sqrt{2} \cdot x$. Then, we know that, at some iteration, the cost of the e-class should be set to AffineFn. Of course, it makes sense mathematically: if a function is both convex and concave, then it is affine. In all of our examples, e-classes with incomparable costs at some iteration always represent an affine term.

To see that this process terminates, it suffices to note that the costs in CM decrease in every iteration and that the cost type is well-founded. Once it terminates, we inspect the root e-class and check whether its cost is $\sqsubseteq$ ConvexProb. If that is the case, we obtain a term that corresponds to a problem in DCP form. Note that in the cost calculation, we also keep track of the e-node that minimizes

[7] In our implementation, this calculation is done internally in `egg`, so we cannot easily replace the minimum by a meet. We simulate it by telling the system to, at this point, treat ConcaveFn and ConvexFn as equal in the order. This also relies on the fact that the e-class contains an affine e-node.

the cost of each e-class, which we have omitted to simplify the presentation. Therefore, obtaining the term is immediate (note that cycles are impossible since we minimize the term size).

Finally, going back to our example, we can see that the e-node that represents $\exp(-x) \leq \sqrt{x}$ is labeled with ConvexSet. The initial e-node for the other constraint representing $0.001 \leq x$ is labeled with AffineSet, and the initial objective function e-node representing x is labeled with AffineFn. Therefore, the term

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

which is represented by the root e-class, is labeled with ConvexProb. In this case, it is the only term in DCP form. This shows how we can use a cost function to extract a term in DCP form.

5.4.2 Stopping on success

We also consider on-the-fly extraction based on e-class analysis. Recall from Section 5.3.5 that we specify a domain D, which earlier we set to $\mathbb{EIF}$. We incorporate the cost type into D as follows: $D := \mathbb{EIF} \times \mathbf{C} \times \mathcal{T}$, where $\mathbf{C}$ is the ordered set of curvature labels from the previous section and $\mathcal{T}$ is the space of terms in our language. The idea is that the analysis data will hold the best curvature and the corresponding term found so far for that e-class. By doing so, we can check the root e-class after every call to iterate() (see Section 5.2.3) and stop if it has been labeled with ConvexProb.

We need to endow D with a join-semilattice structure, which we do as follows:

$$(I_1, u_1, t_1) \leq (I_2, u_2, t_2) := I_2 \subseteq I_1 \wedge u_2 \sqsubseteq u_1$$

$$(I_1, u_1, t_1) \sqcup (I_2, u_2, t_2) := \begin{cases} (I_1 \cap I_2, u_1, t_1) & \text{if } u_1 \sqsubseteq u_2, \\ (I_1 \cap I_2, u_2, t_2) & \text{otherwise.} \end{cases}$$

We think of "$\leq$" as an information ordering. As discussed earlier, a smaller interval is seen as more informative. A lower label in the curvature order corresponds to a more refined classification of the terms represented in the e-class and, hence, should also be considered more informative (it is clear that the "unknown" labels are the least informative). Regarding the join, we encounter the same issue as

with the cost function. The curvatures u_1 and u_2 might not be comparable. We do not take $u_1 \sqcap u_2$ as, morally, we do not want to associate the AffineFn label to a term that is not DCP affine, even if it is mathematically affine. Instead, we would just pick u_2 and t_2 if the comparison fails. Once again, we rely on the fact that, in that case, as rewrites are applied and the e-class grows, an e-node that the curvature rule sees as affine will be added to the e-class, and hence the whole e-class will be labeled as affine. As a small remark, we also take into account the size of the term and the number of variable occurrences and use them to break ties in $\sqcup_D$ when $u_1 = u_2$.

Note that incorporating the cost type into the analysis domain does not work for an arbitrary cost function. It is only possible because the curvature calculation for an e-node is local, i.e., it only depends on its children e-classes. This allows us to define $\mathsf{make} : \mathcal{N} \to D$ by making use of the cost calculation from the previous section. Indeed, we can re-use the definition of cost, where instead of getting costs from the cost map CM, we get them directly from the e-class analyses. Extending it to keep track of the best term is easy: simply apply the function symbol of the e-node to the terms found in each of its children's e-class analyses.

Applying this strategy to our running example results in the same extracted term. To see that, we can go back to Figure 5.2, where we showed the building steps. We aim to label every e-class (dotted box) with the "best" (lowest in the order) curvature found in its term set, as well as a term that has said curvature. We start by labeling the initial e-graph, which is easy since each e-class represents exactly one term. Then, every time a new e-node is added, we use make, and every time two e-classes are merged, we use $\sqcup$. At step (d), the root e-node is still labeled with UnknownSet, so we continue with the final step. This results in the e-graph shown in Figure 5.5. There, the e-class analysis of the root is

$$(\bullet,\ \textsc{ConvexSet},\ \exp(-x) \leq \sqrt{x}).$$

Recall we ignore the interval in set-valued e-class analyses. The other components were DCP-compliant, so the problem is labeled with ConvexProb and we stop.

5.5 Proof reconstruction

If a term in DCP form is found in the e-graph, the next step is to obtain a sequence of rewrite steps indicating the rule applied and the sub-expression modified, i.e.,

an explanation, as described in Section 5.2.4. However, more work is needed to build an equivalence of problems from these steps. The goal of this section is to fill that gap. The infrastructure laid out here translates directly to the proof replay component on the Lean side.

The inference rules defined in Section 5.3.4 are designed with proof reconstruction in mind. In order to combine them, we need the following transitivity rule:

$$\frac{\vdash (f, cs) \equiv_{\phi,\psi} (f', cs') \quad \vdash (f', cs') \equiv_{\phi',\psi'} (f'', cs'')}{\vdash (f, cs) \equiv_{\phi'\circ\phi,\psi\circ\psi'} (f'', cs'')}\ \mathsf{trans}$$

It says that the problem equivalence relation is transitive (composing the forward and backward maps in the right way). Its soundness follows immediately from the definition of equivalence. Recall that in the case of both map-based and rewrite-based equivalences (see Section 3.3.2.4 and Section 3.3.2.3, respectively), φ and ψ are the identity map so we may skip the subscript. Every rewrite step will be turned into a proof of equivalence between problems. These steps will then be combined by transitivity.

The proof of equivalence corresponding to the running example is shown in Figure 5.7, with the rewrite rules highlighted. The derivation for each equivalence step on the left is shown on the right. In general, every step starts with an application of $\mathsf{rw\_constr}_i$ or rw_objFun to navigate to the correct component (except the two problem-level rewrites map_objFun_log and map_objFun_sq, which are applied directly). For equality rewrites at sub-expressions, we apply congruence rules. Note that an explanation tells us which expression needs to be rewritten at every step, which means that we can always instantiate the parameters of the rules. All side conditions are discharged by arith.

The arith rule is a heuristic rule defined as follows. Let $\sim \in \{\leq, \geq, <, >, \neq\}$ and f be a real-valued function expressible in our language. Then,

$$\frac{}{\Gamma, \vec{x}; \Sigma \vdash f(\vec{x}) \sim 0}\ \mathsf{arith}$$

is valid if we can prove $f(\vec{x}) \sim 0$ using interval arithmetic from the intervals defined by the simple constraints in Σ.

On the e-graph side, that is precisely what happens. E-class analyses essentially store an interval that approximates $f(\vec{x})$, hence it is clear how the condition can be asserted. For example, $x; 0.001 \leq x \vdash 0 < \sqrt{x}$ is proved as follows. At the e-class initialization phase, we get that $A_{\{x\}} = [0.001, \infty)$. The e-class that rep-

$$\left(\begin{array}{ll} \min & x \\ \text{s.t.} & 0.001 \le x \\ & 1/\sqrt{x} \le \exp(x) \end{array}\right)$$

$$\ldots \overset{(\dagger_1)}{\equiv} \left(\begin{array}{ll} \min & x \\ \text{s.t.} & 0.001 \le x \\ & 1 \le \sqrt{x}\exp(x) \end{array}\right) \quad \dfrac{\dfrac{\dfrac{}{x; 0.001 \le x \vdash 0 < \sqrt{x}}\ \textsf{arith}}{x; 0.001 \le x \vdash 1/\sqrt{x} \le \exp(x) \Leftrightarrow 1 \le \sqrt{x}\exp(x)}\ \textsf{div\_le\_iff}}{\vdash (\dagger_1)}\ \textsf{rw\_constr}_2$$

$$\ldots \overset{(\dagger_2)}{\equiv} \left(\begin{array}{ll} \min & x \\ \text{s.t.} & 0.001 \le x \\ & 1 \le \exp(x)\sqrt{x} \end{array}\right) \quad \dfrac{\dfrac{\dfrac{}{x; 0.001 \le x \vdash \sqrt{x}\exp(x) = \exp(x)\sqrt{x}}\ \textsf{mul\_comm}}{x; 0.001 \le x \vdash 1 \le \sqrt{x}\exp(x) \Leftrightarrow 1 \le \exp(x)\sqrt{x}}\ \textsf{congr}}{\vdash (\dagger_2)}\ \textsf{rw\_constr}_2$$

$$\ldots \overset{(\dagger_3)}{\equiv} \left(\begin{array}{ll} \min & x \\ \text{s.t.} & 0.001 \le x \\ & 1/\exp(x) \le \sqrt{x} \end{array}\right) \quad \dfrac{\dfrac{\dfrac{}{x; 0.001 \le x \vdash 0 < \exp(x)}\ \textsf{arith}}{x; 0.001 \le x \vdash 1 \le \exp(x)\sqrt{x} \Leftrightarrow 1/\exp(x) \le \sqrt{x}}\ \textsf{div\_le\_iff-rev}}{\vdash (\dagger_3)}\ \textsf{rw\_constr}_2$$

$$\ldots \overset{(\dagger_4)}{\equiv} \left(\begin{array}{ll} \min & x \\ \text{s.t.} & 0.001 \le x \\ & \exp(-x) \le \sqrt{x} \end{array}\right) \quad \dfrac{\dfrac{\dfrac{}{x; 0.001 \le x \vdash 1/\exp(x) = \exp(-x)}\ \textsf{exp\_neg\_eq\_one\_div-rev}}{x; 0.001 \le x \vdash 1/\exp(x) \le \sqrt{x} \Leftrightarrow \exp(-x) \le \sqrt{x}}\ \textsf{congr}}{\vdash (\dagger_4)}\ \textsf{rw\_constr}_2$$

Figure 5.7: Full proof of equivalence for the running example.

resents $\sqrt{x}$ only represents one term, as seen in Figure 5.5. Therefore:

$$d_{\{\sqrt{x}\}} = \widehat{\text{sqrt}}([0.001, \infty)) = [0.0316227\ldots, \infty).$$

We do not show the full decimal expansion, but the important properties are that it is > 0 and $\leq \sqrt{0.001}$. Since $d_{\{\sqrt{x}\}} \subseteq (0, \infty)$, the system is able to infer that $\sqrt{x}$ is positive, which makes the div_le_iff rule applicable.

To reconstruct the proof, we may assume that a similar strategy is available. In reality, on the Lean side, the arith equivalent is strictly weaker than interval arithmetic, and there can be conditions that the e-graph system deems as true that Lean cannot verify due to the currently limited support for non-linear arithmetic. Importantly, this does not compromise the overall soundness of the system. This and other implementation details are discussed in the next section.

5.6 Implementation

In this section, we explain how the algorithm described so far is realized in egg and how the `pre_dcp` tactic uses the explanation to reconstruct a formal proof.

The first step to call egg is to encode optimization problems as S-expressions. We show our language below, which corresponds to the language described in Section 5.3.2.

prob ::=	(prob (objFun expr) (constrs {constr}*))	
constr ::=	(eq expr expr)	
|	(le expr expr)	
expr ::=	c	c is a non-NaN finite float
|	(var s)	s is a string
|	(u expr)	$u \in \{\texttt{neg}, \texttt{inv}, \texttt{abs}, \texttt{sqrt}, \texttt{log}, \texttt{exp}\}$
|	(b expr expr)	$b \in \{\texttt{add}, \texttt{sub}, \texttt{mul}, \texttt{div}, \texttt{pow}, \texttt{min}, \texttt{max}\}$
|	(cu expr)	$cu \in \{\texttt{xexp}, \texttt{entr}\}$
|	(cb expr expr)	$cb \in \{\texttt{qol}, \texttt{geo}, \texttt{lse}, \texttt{norm2}\}$

One subtlety is that, while we show three different sorts, when declaring a language in egg, there is only one sort, so it is our responsibility to make sure that

the rewrite rules operate within these sorts.[8]

To build a formal S-expression from an optimization problem in CvxLean, we use the already existing infrastructure to create atom trees (see mkAtomTree in Section 4.5.1). Recall that the implementation of mkAtomTree takes a Lean expression representing an optimization problem and uses a discrimination tree built from the functions in the atom library to look for a match. At that stage, curvature checks ensure that the tree can be expanded later on. We disable them since our current goal is to handle problems that do not follow the curvature rules specified by DCP. However, we still obtain an atom tree representing the given problem. Note that the atom library includes all the operations in our language, so it is able to handle all the expressions we are interested in.

We detect simple constraints and use them to build the domains in e-class analyses. We also split the objective function and each of the constraints before calling egg and tag each constraint. This will be useful later on for egg to indicate the location of the rewrite.

Our running example is written in CvxLean as follows:

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

After extracting the domain information and flattening the atom tree into a S-expression, it is encoded in JSON format as shown below:

```
{ "domains" : [["x", "0.001", "inf", "0", "1"]],
  "target" : {
    "obj_fun" : "(var x)",
    "constrs" : [[
      "h2", "(le (div 1 (sqrt (var x))) (exp (var x)))"]]}}
```

We read `["x", "0.001", "inf", "0", "1"]` as x being in the interval from

[8] As a small remark, a recent tool that extends egg, egglog [ZWF+23], would allow us to define a language with sorts such as the one above. Using egglog instead of egg could be an interesting avenue to explore.

0.001 to ∞ closed on the left and open on the right, i.e., $x \in [0.001, \infty)$. This represents exactly the constraint `h1`. We use JSON as a convenient representation language to send data between `egg` and Lean.

Our `egg` wrapper runs as a separate subprocess that we communicate with via standard input and output. When it receives the input, it builds the e-graph corresponding to the optimization problem using the 68 rewrite rules described in Section 5.3.4 and listed in Appendix D. The building strategy depends on the extraction strategy (stopping on success means that building stops when a problem in DCP form is found). For now, we assume that we use a cost function. The e-graph is incrementally extended until it reaches a specified maximum number of nodes (2500-80000), maximum building iterations (10-320), or time limit (5s-160s). It would also stop if it saturates, which never happens in practice. The limits are ranges, so assume we start with the lower end. Then, the extractor finds the problem term that minimizes the cost described in Section 5.4.1. If this term is not tagged with ConvexProb and, therefore, not DCP-compliant, all limits are doubled unless the current node limit is already 80000, in which case the procedure fails, claiming that no problem in DCP form was found. This "iterative limits" approach improves performance on small problems but still allows us to handle larger examples. If a problem in DCP form is found, we instruct `egg` to *explain* the equivalence by providing a sequence of steps.

Each step corresponds to a rewrite rule. It includes the component of the optimization problem where the rewrite needs to be applied, the source and target sub-expressions, and the "context" expression of these sub-expressions. For example, for the first step, this data is encoded in JSON as follows:

```
{ "rewrite_name": "div_le_iff",
  "direction": "Forward",
  "location": "h2",
  "subexpr_from": "(le (div 1 (sqrt (var x))) (exp (var x)))",
  "subexpr_to": "(le 1 (mul (exp (var x)) (sqrt (var x))))",
  "expected_term": "?" }
```

This tells us to rewrite $1/\sqrt{x} \leq \exp(x) \rsquigarrow 1 \leq \exp(x)\sqrt{x}$ in the second constraint using the rule `div_le_iff`. The value of `"expected_term"` corresponds to the "context" expression; we discuss below exactly how to interpret it. In this case, we rewrite the whole constraint, so there is no context. To show a less

trivial case, consider the last step:

```
{ "rewrite_name": "exp_neg_eq_one_div-rev",
  "direction": "Forward",
  "location": "h2",
  "subexpr_from": "(div 1 (exp (var x)))",
  "subexpr_to": "(exp (neg (var x)))",
  "expected_term": "(le ? (sqrt (var x)))" }
```

In this case, it tells us to rewrite $1/\exp(x) \rightsquigarrow \exp(-x)$ in the second constraint using the rule exp_neg_eq_one_div-rev.

Each step contains the name of the rewrite rule and whether it is applied forward or backward. Equivalence explanations generated by egg can sometimes involve rules oriented backward. These are rules oriented in the opposite direction to that in which they were applied when building the e-graph (which is not the same as "-rev" versions of bi-directional rules). In our case, however, backward applications are not strictly necessary. The main reason is that most of the rewrites in our rule set are already bi-directional, i.e., their reverse is included in the set, so we may just apply the reverse rewrite forward. Backward rewrites are most useful when e-graphs are used to prove that a term t_1 can be rewritten into t_2. Equality saturation applies rewrites to both t_1 and t_2, and if 10 rewrite steps are needed, perhaps 5 iterations are enough to merge the e-classes of both terms with backward rewrites. This is not our case, as we do not have a target term. Nevertheless, the explanation generator does not ensure finding the shortest sequence, which is known to be NP-complete [FCW+22], or that the rules in the sequence are in the forward direction in which they were applied. Instead, some heuristics are used to find any reasonably short sequence. In particular, this implies that the resulting sequence might involve rules applied backward.

The step also contains a `"location"`, which is the name of the constraint where the rule needs to be applied or whether it needs to be applied to the objective function. The location of the sub-expression is specified by a `"?"` in the expected term. In the steps shown earlier, the constraint before the rule is applied is the expected term replacing `"?"` by the value of `"subexpr_from"` and, the constraint after the rule as applied should be the expected term replacing `"?"` by the value of `"subexpr_to"`.

Each step is used to build a proof of equivalence, matching the structure of

the proof shown in Section 5.5. We focus on the first step for now. To make it clearer, consider the goal opened by running our example under the `equivalence` command. The goal looks as follows:

```
⊢ (optimization (x : ℝ)
    minimize x
    subject to
      h1 : 1 / 1000 ≤ x
      h2 : 1 / sqrt x ≤ rexp x
    ) ≡
  ?q
```

From the location, we know that the rewrite needs to be applied to the second constraint, so we essentially apply the following tactic, which was presented in Section 3.3.2.3:

```
rw_constr h2 into (1 ≤ rexp x * sqrt x)
```

This is the Lean equivalent of $\mathsf{rw\_constr}_2$. In reality, we call this tactic at the meta-level, providing an `Expr` that type-checks to `D → Prop`. Our example has only one optimization variable, but, in general, `D` is a product type, and we need to replace variable occurrences with projections when constructing expressions.

After applying the tactic above, the goal becomes:

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

At this point, `CvxLean` needs to know what rewrite corresponds to the name `"div_le_iff"`. Using a custom command, we associate the name with the tactic that needs to be applied, also specifying how to prove any side conditions. These are stored in an environment extension that tactics can access. In this case, we want to use the `mathlib` lemma about linear ordered semifields `div_le_iff`.

```
register_rule_to_tactic "div_le_iff" ;
  "(le (div ?a ?b) ?c)" => "(le ?a (mul ?b ?c))" :=
  apply div_le_iff (by arith);
```

The `arith` tactic is a custom tactic that uses a combination of pre-defined arithmetic tactics. First, it uses an extension of mathlib's `positivity` tactic with a pre-processing step that extends the local context with equivalent hypotheses that aid the proof search and contains a number of additional rules to handle several non-linear expressions. It also tries to use a combination of `linarith`, `field_simp` and `norm_num`. Ideally, an interval arithmetic tactic should be used here, mimicking the logic of e-class analyses. That is beyond the scope of this work, and, for now, with the strictly weaker `arith` procedure, we have been able to handle all the examples considered in this thesis.

For steps that rewrite proper sub-expressions, the story is slightly different. For example, consider the final step, after applying $\mathsf{rw\_constr}_2$ instantiating the parameters appropriately. The goal is as follows:

```
x: ℝ
h1: 1 / 1000 ≤ x
⊢ 1 / rexp x ≤ sqrt x ↔ rexp (-x) ≤ sqrt x
```

First, by propositional extensionality, we can replace `↔` with `=`. Then, we can use the expected term to build the expression `g := fun s => s ≤ sqrt x` in the current variable context, with type `ℝ → Prop`. One of Lean's congruence lemmas, `congrArg`, says that for all `a₁ a₂ : α` and `f : α → β`, if `a₁ = a₂`, then `f a₁ = f a₂`. Applying `congrArg` with `f` set to `g`, turns the goal into:

```
x: ℝ
h1: 1 / 1000 ≤ x
⊢ 1 / rexp x = rexp (-x)
```

This can now solved by applying one of the registered lemmas. Note that we never use `rw` or `simp`, which are not needed since we know exactly where the rewrite must take place. In a way, we have implemented our own simple version of `rw`.

Problem equivalence rewrites need to be considered separately. They are

stored in a similar way, including a flag that instructs the system to apply the tactic to the whole problem (without using rw_constr$_i$ or rw_objFun first).

```
register_objFun_rule_to_tactic "map_objFun_log";
  "(prob (objFun ?a) ?cs)" => "(prob (objFun (log ?a)) ?cs)" :=
  apply map_objFun_log (by arith);
```

Here, `map_objFun_log` is the lemma shown in Section 3.3.2.4.

The full procedure is implemented in the tactic `pre_dcp`. The tactic encodes the optimization problem, calls egg, and reconstructs the proof from its output. It is an equivalence-preserving tactic (like the ones from Section 3.3.2), so it can be called in both the `reduction` or `equivalence` commands. We show how it works by applying it to our running example below.

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

#print q
-- optimization (x : ℝ)
--   minimize x
--   subject to
--     h1 : 1 / 1000 ≤ x
--     h2 : rexp (-x) ≤ sqrt x
```

In this case, egg found the exact same rewrites as the ones described in the simplified setting from Section 3.3.2.3. The commented code shows the output of the value of `q`. Note that many more rules are applied here, so the e-graph representing this problem is considerably more complicated than the one in Figure 5.5; it has 2234 nodes.

5.7 Evaluation

In this section, we test the algorithm presented with several examples. Finding examples is challenging as the test suites found in convex optimization frameworks such as CVXPY mostly contain problems already in DCP form. We have gathered examples from geometric programming and quasiconvex programming. They illustrate that our algorithm works on interesting subclasses of problems that

require specialized procedures in other approaches. Another source of examples is the DCP quiz[9], an educational tool that generates random problems and asks the user to select their curvature. We chose expressions that were tagged as non-DCP, but that could be rewritten with our approach. We also found a few examples in exercises from the Convex Optimization course at Stanford. Finally, our test suite includes many artificial examples (unit tests) designed to test the system's robustness. These include the running example in this chapter.

Another challenge in evaluating the algorithm was finding alternative approaches to compare it against. First, we considered using a modified version of `simp` with stopping conditions on DCP terms. The lack of normal forms and the destructive nature of the simplifier made this idea succeed only on a very limited number of small examples. We also experimented with `aesop` [LF23], exploiting proof tree search to explore the set of equivalent problems. However, usually, many rewrite rules apply at any given point. This means that the size of the search tree increases very quickly, and our initial experiments could not transform any problems that required more than 2 or 3 steps in a reasonable time. The best approach would be to compare it with a heuristic rewrite system, fine-tuned to explore promising rewrite sequences that can lead to terms in DCP form. Such a system does not exist to the best of our knowledge, and it is unclear how it would be designed. Developing it would be, in fact, an interesting research project on its own. Lacking a suitable alternative for a fair comparison, we analyzed some examples qualitatively and gathered relevant statistics instead.

5.7.1 Two examples: a GP and a QCP

Our first example is a GP adapted from the main tutorial on the topic [BKVH07], which corresponds to the example shown in Section 2.2.3.

```
optimization (x y z : ℝ)
  minimize 1 / (x / y)
  subject to
    h1 : 0 < x
    h2 : 0 < y
    h3 : 0 < z
```

[9] `https://dcp.stanford.edu/quiz` (accessed 2024-02-27).

```
    h4 : 2 ≤ x
    h5 : x ≤ 3
    h6 : x ^ 2 + 3 * y / z ≤ 5 * sqrt y
    h7 : x * y = z ^ 2
```

The first step is to change the positive variables by exponentials. This can be done in CvxLean with the `change_of_variables` tactic (see Section 3.3.2.1). Then we can call `pre_dcp` to obtain:

```
optimization (u : ℝ) (v : ℝ) (w : ℝ)
  minimize v - u
  subject to
    h4 : log 2 ≤ u
    h5 : u ≤ log 3
    h6 : rexp (u * 2 - v * (1 / 2)) + 3 * rexp (v * (1 / 2) - w) ≤ 5
    h7 : u + v = 2 * w
```

It involves 51 rewrites, and the problem can be transformed into conic form and solved. It can also be canonized in CVXPY, setting the `gp` flag to `True`.

An interesting remark is that if we replace `h6` in the initial problem by:

```
h6' : x ^ 2 ≤ sqrt y - 5 * y / z
```

then DGP immediately fails as -5 is not a positive coefficient. However, our system accepts such constraints as it knows that `h6` and `h6'` are equivalent within our rule set.

Next, we consider an example that is a QCP and not a GP.

```
optimization (x y : ℝ)
  minimize -y
  subject to
    h1 : 1 ≤ x
    h2 : x ≤ 2
    h3 : 0 ≤ y
    h4 : sqrt ((2 * y) / (x + y)) ≤ 1
```

The key constraint is `h4`, which defines a set that is recognizably quasiconvex

by the rules of the DQCP framework. The square root is an increasing function, and the ratio $(2y)/(x+y)$ is quasilinear since the denominator is positive. The rules of DQCP state that constraints of the form $f(e) \leq c$ where f is increasing, e is quasiconvex, and c is a constant are quasiconvex. Instead, our system is able to detect that the inequality can be squared and rearranged (11 rewrites in total), resulting in a linear constraint. After applying `pre_dcp` we obtain:

```
optimization (x : ℝ) (y : ℝ)
  minimize -y
  subject to
    h1 : 1 ≤ x
    h2 : x ≤ 2
    h3 : 0 ≤ y
    h4 : y * 2 ≤ y + x
```

This problem is linear and, therefore, DCP-compliant. Again, in `CVXPY`, the original problem can be solved by setting the `qcp` flag to `True`. Interestingly, the operations that DQCP performs internally involve inverting monotone functions (such as the square root) on the left-hand side of inequalities and rearranging ratios, so the problem that is solved is essentially the same as found by our rewrite system. A notable difference is that they need to optimize a parameter, which takes 25 solver iterations. In either case, we find that $x^* = 2$ and $y^* = 2$ is the optimal solution.

5.7.2 Proof replay overhead

The rest of the benchmark consists of 145 problems to which `pre_dcp` applies. We show the sources of the problems in Table 5.1, as well as the number of problems in each category and their sizes. Unit tests are simple artificial tests designed to test particular rules. The rest come from the sources indicated at the beginning of this section. Term sizes are the size of the AST of the initial problem.

The following experiments are performed on a 2021 Macbook Pro with an Apple M1 Pro and 16GB RAM. All of the statistics that follow are averaged over 5 runs. First, we report some averages in Table 5.2 that summarize the performance of the procedure. We consider the runtime of the `egg` subprocess

Source	Count	Term size range
Unit tests	114	5-17
DCP quiz	10	6-14
Stanford	4	14-34
DGP	13	10-97
DQCP	4	17-36

Table 5.1: Benchmark composition.

and the total command time. The Lean time is simply the difference. We also consider the command time per step, i.e., per rewrite rule applied.

Average egg time	77.70 ms
Average Lean time	494.83 ms
Average command time	572.53 ms
Average command time per step	65.78 ms

Table 5.2: Performance summary.

As we can see, the verification overhead is significant in our benchmark, increasing the overall runtime by approximately a factor of ~ 7.37. That is indeed the main bottleneck and not the equality saturation workload. The mismatch between the representations of problems between the two components could be one of the main reasons, as it requires going back and forth between strings and Lean expressions. In particular, verifying a rewrite step relies on building the Lean expression for the expected term from a string before even starting the proof reconstruction process.

Next, we analyze the relationship between egg and command times in a finer-grained manner, showing the metrics for every instance. We removed some outliers, the 8 problems with command times above 1.75 seconds, to provide more informative plots.

We show the times with respect to the number of rewrite steps in Figure 5.8. Another suitable independent variable could be the term size, as a larger term size will require a larger e-graph, which would increase the egg time. However, as discussed, the main bottleneck is replaying the rewrite steps in Lean, which is what we analyze here. Therefore, we compare performance against the number

of steps, which is not directly correlated to the term size (a large problem could already be in DCP form).

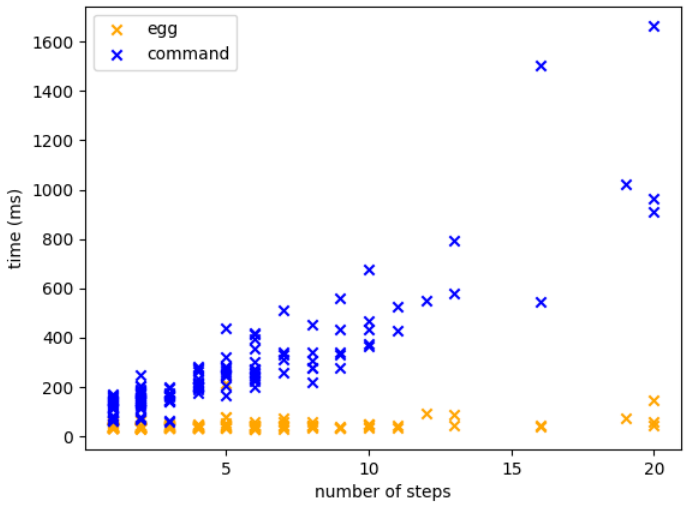


Figure 5.8: Command and egg times w.r.t. number of steps.

These results show that the command time grows roughly linearly with the number of steps, which gives us two directions to further improve the performance of our approach. First, we know that the sequence output by egg is not necessarily the shortest sequence. It uses a greedy strategy to optimize it to avoid any complexity overhead. However, in our case, it could be a beneficial trade-off to dedicate more resources to shortening the sequence. The second approach is to reduce the Lean time per step. Targeted rewrites (without re-constructing the right-hand side) instead of our guided step-by-step method could reduce the verification time at the expense of not knowing that the result of every step matches the expected term exactly. We could also consider grouping together all the rewrites that apply to a certain constraint, apply rw_constr$_i$ once, and use all these rewrites, avoiding reconstructing the whole problem at every step.

5.7.3 Comparison of extraction mechanisms

In this last part of the evaluation, we compare the two extraction approaches discussed in Section 5.4: using iterative node limits and a cost function; and stopping on success, that is, as soon as a problem in DCP form is found. Proof replay is ignored here, and we focus only on the egg side. We restrict our attention

to the problems from the DGP, DQCP, Stanford, and DCP Quiz sources. From the unit tests, we only consider our running example, which we call "main". There are 32 problems in total. Table 5.3 shows the total time, number of steps, and resulting term size per problem for each of the two extraction mechanisms. In Appendix E, we extend this table also including the number of e-nodes in the final e-graph and splitting the time into e-graph building time and explanation time.

On the one hand, the stop-on-success approach requires storing more data per e-class analysis. However, it often needs a smaller number of e-nodes to succeed, which explains why the overall time is lower in 29 out of the 32 examples.

An advantage of not stopping early is that we explore a more extensive set of problems. This allows us to potentially find several problems in DCP form and choose the simplest. This explains why the term sizes in the iterative approach are always equal or lower.

Regarding the number of steps in the explanation per approach, we would intuitively expect the stop-on-success approach to return shorter explanations. If we think of equality saturation as a breadth-first search in the space of terms, stopping on success means considering a lower breadth, where the breadth corresponds to the number of rewrite rules applied. However, that is not always the case. There are 6 problems for which the number of steps is lower in the iterative approach (in these cases, the number of steps never differs by more than 3). We inspect one of them, namely "stan2". The objective function of this example is

$$-(\sqrt{x}+\sqrt{y})^2.$$

We can see that it is not DCP-compliant since squaring is DCP convex and non-decreasing on non-negative arguments while the square root is DCP concave, so no composition rule applies. The trick here is to re-phrase this using the geometric mean. The iterative limit approach finds

$$-(x+(y+2\cdot\mathrm{geo}(x,y))),$$

whereas stopping on success finds

$$-(2\cdot\mathrm{geo}(x,y)+(x+y)).$$

As we can see, these terms are equal modulo associativity and commutativity. They both have the same term size and number of variable occurrences, so the

Problem	Stop on success			Iterative limits and cost function		
	Time	Steps	Term size	Time	Steps	Term size
gp1	5 ms	1	13	81 ms	7	11
gp2	2 ms	1	13	57 ms	6	12
gp3	5 ms	4	16	92 ms	16	12
gp4	5766 ms	47	43	5426 ms	46	43
gp5	5474 ms	38	45	5511 ms	37	45
gp6	603 ms	53	43	474 ms	51	43
gp7	193 ms	31	61	237 ms	35	61
gp8	4939 ms	79	72	7255 ms	180	72
gp9	31158 ms	123	104	51033 ms	175	104
agp1	2 ms	2	15	69 ms	10	11
agp2	2 ms	2	15	68 ms	9	12
agp3	4638 ms	29	46	5648 ms	35	46
agp4	9 ms	3	20	93 ms	10	19
qcp1	6 ms	6	12	85 ms	9	10
qcp2	3 ms	6	35	92 ms	16	28
qcp3	1 ms	1	15	59 ms	2	15
qcp4	1128 ms	17	19	699 ms	20	19
stan1	2 ms	1	17	75 ms	1	17
stan2	32 ms	9	16	104 ms	8	16
stan3	201 ms	20	31	262 ms	28	29
stan4	746 ms	38	48	813 ms	35	48
quiz1	1 ms	1	4	100 ms	1	4
quiz2	2 ms	3	11	74 ms	3	11
quiz3	1 ms	2	4	61 ms	2	4
quiz4	1 ms	2	6	91 ms	2	6
quiz5	1 ms	1	6	82 ms	1	6
quiz6	1 ms	1	10	88 ms	1	10
quiz7	33 ms	11	17	121 ms	10	10
quiz8	1 ms	1	6	92 ms	1	6
quiz9	227 ms	12	9	289 ms	12	9
quiz10	1 ms	1	3	83 ms	1	3
main	5 ms	3	15	65 ms	4	15

Table 5.3: Comparison of extraction mechanisms on selected problems.

cost-based extractor could have picked the latter. In that sense, the cost-based extractor was lucky to pick the former, which can be reached with one fewer rewrite rule in our rewrite system. The rewrite rules in both cases are the same, in a different order, but an extra application of the associativity rule for addition is needed in the stop-on-success case.

5.8 Summary

This chapter has covered the development of an algorithm to automatically put optimization problems in DCP form using e-graphs. In order to use the procedure described in Chapter 4, problems need to be in DCP form. We show how we can explore the space of equivalent problems (via rewrites) and, in many cases, find a term in DCP form. Compared to the DCP algorithm, this method's philosophy is very different; instead of statically analyzing the problem and systematically transforming it, we consider a large set of problems and pick the one in the correct form.

We have also covered how conditional rewrites are handled via e-class analyses, how extraction works, and how an explanation is turned into a formal proof of equivalence. The chapter concludes with a thorough evaluation where we have analyzed the scope and limitations of this approach.

Applying equality saturation for convex optimization modeling is a novel angle that, to the best of our knowledge, had not been explored before. Our results show that it is a promising method for transforming problems into equivalent DCP-compliant problems, setting the scene for further work in this direction. Moreover, our development demonstrates that `CvxLean` is an ideal environment to establish trust in complex transformation procedures by extending them to construct formal proofs.

Chapter 6

Case studies

6.1 Introduction

Throughout this thesis, we have provided several motivating examples. While many of them came from real-world problems, they served the purpose of showing specific features of `CvxLean`. In this chapter, we study four problems end to end.[1] These case studies have served to guide the development of the tool and better understand its current limitations, which we will point out.

The problems come from two sources. In Section 6.2 and Section 6.3, we find problems adapted from the additional exercises of the Stanford Convex Optimization course[2]. These were suggested by Parth Nobel, a PhD student specializing in convex optimization supervised by Prof. Stephen Boyd and Prof. Emmanuel Candès. We thank Nobel for the insightful conversation, which helped us understand what modeling exercises students find more challenging and get wrong due to incorrect transformations. Understanding common mistakes in transformation steps is very useful for the development of `CvxLean`, as our verified approach offers a way to prevent them. Sections 6.4 and 6.5 cover problems from quasiconvex and geometric programming, respectively.

6.2 Vehicle speed scheduling

This case study is adapted from exercise 4.10 in the additional exercises of the Stanford Convex Optimization course. We use some of the transformations dis-

[1] The code for all four of them is publicly available at:
`https://github.com/verified-optimization/CvxLean/tree/2.0/CvxLean/Examples`.

[2] `https://github.com/cvxgrp/cvxbook_additional_exercises` (accessed 2024-01-25).

cussed in Chapter 3, as well as the canonization to conic form and solving machinery discussed in Chapter 4.

Suppose we have a vehicle supposed to go through a fixed route that consists of $n > 0$ segments. The trajectory starts at time $t = 0$. For each segment i, where $0 \leq i < n$, we have:

- d_i, its distance,
- s_i^{min} and s_i^{max}, the speed bounds, and
- τ_i^{min} and τ_i^{max}, the time bounds for arriving at the end of the segment.

Moreover, for every i, we require $0 < d_i$, $0 < s_i^{min} \leq s^{max}$, and $0 < \tau_i^{min} \leq \tau^{max}$. A positive, monotonically increasing, convex function $F : \mathbb{R} \to \mathbb{R}$ represents the fuel consumption rate of the vehicle, given a speed.

The goal is to choose speeds s_i that minimize the total fuel used. The time taken to traverse segment i is d_i/s_i, therefore:

- the total time taken to arrive at the end of segment i is $\sum_{j=0}^{i}(d_j/s_j)$, and
- the total fuel used is $\sum_{i=0}^{n-1}(d_i/s_i)F(s_i)$.

This allows us to cast our problem as the following minimization over s:

$$\begin{array}{lll} \underset{s_0,\ldots,s_{n-1}}{\text{minimize}} & \displaystyle\sum_{i=0}^{n-1}(d_i/s_i)F(s_i) & \\ \text{subject to} & s_i^{min} \leq s_i \leq s_i^{max} & 0 \leq i < n \\ & \tau_i^{min} \leq \displaystyle\sum_{j=0}^{i}(d_j/s_j) \leq \tau_i^{max} & 0 \leq i < n \end{array}$$

In this form, the problem is not convex. The objective function is not convex for every admissible F; for instance, take $F(x) = x^{1.5}$ (with $F(x) = 0$ if $x < 0$). Furthermore, for $i > 0$, the constraint $\tau_i^{min} \leq \sum_{j=0}^{i}(d_j/s_j)$ is not convex, since $1/s_j$ is convex.

The key observation to turn it into a convex problem is that if the summands in the objective function were of the form $t_i F(d_i/t_i)$, then it would be convex since these are instances of the *perspective function*, as we will discuss shortly. To that end, we perform the change of variables $s_i \mapsto d_i/t_i$. This transformation requires $s_i \neq 0$ and $d_i \neq 0$ to be provable from the constraints, which is clear since

$0 < s_i^{min} \leq s_i$ and $0 < d_i$ by assumption. After performing the change of variables and simplifying, we obtain:

$$\begin{aligned} \text{minimize} \quad & \sum_{i=0}^{n-1} t_i F(d_i/t_i) \\ \text{subject to} \quad & d_i/s_i^{max} \leq t_i \leq d_i/s_i^{min} \quad 0 \leq i < n \\ & \tau_i^{min} \leq \sum_{j=0}^{i} t_j \leq \tau_i^{max} \quad 0 \leq i < n \end{aligned}$$

We explain how this first transformation is formalized in `CvxLean`. The initial problem is defined as follows:

```
def vehSpeedSched (n : ℕ) [Fact (0 < n)]
    (d τmin τmax smin smax : Fin n → ℝ) (F : ℝ → ℝ) :=
  optimization (s : Fin n → ℝ)
    minimize ∑ i, (d i / s i) * F (s i)
    subject to
      c_smin : ∀ i, smin i ≤ s i
      c_smax : ∀ i, s i ≤ smax i
      c_τmin : ∀ i, τmin i ≤ ∑ j in [[0, i]], d j / s j
      c_τmax : ∀ i, ∑ j in [[0, i]], d j / s j ≤ τmax i
```

The definition takes the parameters of the problem as arguments, so we have defined a parametrized family of problems. Recall that parameters are expected to be instantiated with a concrete value in order to solve the problem. The DCP transformation assumes that that is the case. However, we can leave them as parameters up until that point. As we will see, this means that we will also need to parametrize each transformation step.

A small subtlety in this definition is that we must indicate that $0 < n$ explicitly, which is a sensible assumption as otherwise the problem would be trivial. The reason is the way sums over finite ranges are defined. Note that in `c_τmin` and `c_τmax`, we sum over `[[0, i]]` (which corresponds to $\{0, \ldots, i\}$; note that it includes i). The bounds, `0` and `i`, are terms of type `Fin n`.[3] If `n` is zero, then `Fin n` has no inhabitants, and, in particular, we cannot cast `0` to an element

[3] A term of type `Fin n` is a tuple consisting of a natural number `x` and a proof of `x < n`. We can regard `Fin n` as $\{0, \ldots, n-1\} \subseteq \mathbb{N}$. It is used often in our development, in particular since it is how we define vectors: `Fin n → ℝ` can be regarded as $\mathbb{R}^n$.

of type `Fin n`. Using `Fact (0 < n)` is a technical detail that allows us to use type-class inference to deduce that `Fin n` has a zero element, allowing for nicer notation.

To perform the change of variables and the necessary simplification, we open an equivalence environment (see Section 3.3.1) as follows:

```
equivalence* eqv₁/vehSpeedSchedConvex (n : ℕ)
    (d τmin τmax smin smax : Fin n → ℝ) (F : ℝ → ℝ)
    (h_n_pos : 0 < n) (h_d_pos : StrongLT 0 d)
    (h_smin_pos : StrongLT 0 smin) :
    @vehSpeedSched n d τmin τmax smin smax F ⟨h_n_pos⟩ := by
  ...
```

The signature includes the necessary parameters to define the problem. These are added to the local context and are accessible by the equivalence-preserving tactics applied in the `by` block. For example, the positivity condition on `n` becomes a hypothesis in the context. We also include two extra properties (`h_d_pos` and `h_smin_pos`) that require further justification. Note that `StrongLT a b` means $a_i < b_i$ for all i, which is different from `a < b`, which means $a_i \leq b_i$ for all i and $a_i < b_i$ for some i. In other words, we assume that all distances and minimum speeds are positive. In particular, this implies that $s_i \neq 0$ and $d_i \neq 0$, which, as discussed earlier, are needed in order to perform the change of variables $s_i \mapsto d_i/t_i$. The change of variables function is $c : s_i \mapsto d_i/s_i$ before we rename s_i to t_i. Its right-inverse is $c^{-1} := c$, and $c(c^{-1}(s_i)) = s_i$ as long as $s_i \neq 0$ and $d_i \neq 0$. As a small remark, technically, we apply $s \mapsto d/t$, where division is elementwise. After the change of variables and some algebraic manipulation, `vehSpeedSchedConvex` looks as follows:

```
optimization (t : Fin n → ℝ)
  minimize Vec.sum (t * Vec.map F (d / t))
  subject to
    c_smin : smin ≤ d / t
    c_smax : d / t ≤ smax
    c_τmin : τmin ≤ Vec.cumsum t
    c_τmax : Vec.cumsum t ≤ τmax
```

Here we have removed the sums and the universal quantifiers and written the

problem in vector notation. That is because the current support for expressions with binding structures is limited, which we discuss in more detail in Section 7.4.2. Multiplication and division are lifted to operations on vectors elementwise, and `Vec.sum` and `Vec.map` do what their names indicate. The function `Vec.cumsum` stands for cumulative sum; it takes vector t and outputs t', where $t'_i := \sum_{j<i} t_j$.

In this form, whether we can reduce it to conic form only depends on F. If we assume that F is DCP convex, then the objective function is convex as it is a sum of perspective functions [BV04, § 3.2.6].[4]

We show that for any DCP convex F, $\Phi(F,k)(x) = kF(x/k)$ is canonizable to conic form provided $k > 0$. This requires providing a graph implementation for Φ. Consider the epigraph form of F. Let z be a new variable and canonize the expression $F(x) \leq z$ to conic form. Since F is DCP convex, the inequality can be canonized to a list of constraints of the form

$$a_i x + b_i z + c_i + \sum_j d_{ij} v_j \in \mathcal{K}_i,$$

where $\vec{v}$ are the new variables introduced by the canonization, $a_i, b_i, c_i, d_{ij} \in \mathbb{R}$ and each $\mathcal{K}_i$ is a convex cone (this is an instance of CBF, as shown in Section 4.7.3).

We claim that optimizing over z over the modified constraints:

$$a_i x + b_i z + c_i k + \sum_j d_{ij} v_j \in \mathcal{K}_i.$$

is the graph implementation of $\Phi(F,k)$. Note that we have only exchanged c_i by $c_i k$. We have that

$$F(x) = \inf\{z \mid F(x) \leq z\} = \inf\Big\{z \mid \forall i.\ a_i x + b_i z + c_i + \sum_j d_{ij} v_j \in \mathcal{K}_i\Big\}.$$

With that, we can proceed as follows:

$$\begin{aligned}
& kF(x/k) \\
= \ & \inf\{z \mid F(x/k) \leq z/k\} && \text{since } k > 0, \\
= \ & \inf\Big\{z \mid \forall i.\ a_i(x/k) + b_i(z/k) + c_i + \sum_j d_{ij} v_j \in \mathcal{K}_i\Big\} && \text{by the above}, \\
= \ & \inf\Big\{z \mid \forall i.\ a_i x + b_i z + c_i k + \sum_j d_{ij}(k v_j) \in \mathcal{K}_i\Big\} && \text{since } \mathcal{K}_i \text{ is a cone}, \\
= \ & \inf\Big\{z \mid \forall i.\ a_i x + b_i z + c_i k + \sum_j d_{ij} v'_j \in \mathcal{K}_i\Big\} && \text{changing } v_j \mapsto v'_j/k,
\end{aligned}$$

[4] In the same textbook [BV04, § 2.3.3], one finds a nice geometric interpretation of the action of the perspective on sets, using an analogy with a "pin-hole" camera.

Therefore, if we can obtain the conic form of F, we can multiply the constant terms in the constraints by k to obtain the conic form of $\Phi(F, k)$.

When declaring an atom in CvxLean, we do not have direct access to the canonization algorithm, which works at the meta-level. Therefore, atoms whose implementation depends on the canonized constraints output by canonize (see Section 4.5.2) are not currently supported. Note that this is very different from multi-level atom declarations (see Section 4.6.3), where the implementation can involve other atom *expressions*. We discuss some ideas on developing this feature in the future work section, Section 7.4.1.

Going back to optimal vehicle speed scheduling, despite not having perspective functions in full generality, we show how to proceed given a concrete F. Following the exercise, we consider the quadratic function

$$F(s) = as^2 + bs + c,$$

assuming $a \geq 0$ so that it is convex. Mathematically, replacing F and simplifying gives us the following objective function:

$$\sum_{i=0}^{n-1} ad_i^2(1/t_i) + bd_i + ct_i.$$

It is convex as every $1/t_i$ is DCP convex when $t_i > 0$ and it is multiplied by a positive scalar. The rest of the terms are linear or constant.

Once again, to arrive at this desired form in CvxLean, we enter the equivalence environment starting with `vehSpeedSchedConvex` with `F` replaced by the appropriate expression.

```
equivalence* eqv₂/vehSpeedSchedQuadratic (n : ℕ)
    (d τmin τmax smin smax : Fin n → ℝ) (a b c : ℝ)
    (h_n_pos : 0 < n) (h_d_pos : StrongLT 0 d)
    (h_smin_pos : StrongLT 0 smin) :
    vehSpeedSchedConvex n d τmin τmax smin smax
      (fun s => a • s ^ (2 : ℝ) + b • s + c)
      h_n_pos h_d_pos h_smin_pos := by
  ... /- 22 LOC -/
```

In this transformation step, we first add a new constraint `StrongLT 0 t`, i.e., the vector t is elementwise strictly positive. This fact is implied by `c_smin`, `h_d_pos`, and `h_smin_pos`, so it is valid to add it as a constraint. Storing

this fact as a constraint makes it easy to linearize `c_smin` and `c_smax`, as otherwise, we would need to prove it two separate times. More importantly, the DCP procedure will need this fact to expand `1 / t`. Since we are not applying `dcp` in this block, we do not need to require `0 ≤ a`, which is needed to ensure the problem is in DCP form, as together with `h_d_pos`, it implies that `a • d ^ 2` is nonnegative. We show `vehSpeedSchedQuadratic` below.

```
fun n d τmin τmax smin smax F h_n_pos h_d_pos h_smin_pos =>
  optimization (t : Fin n → ℝ)
    minimize Vec.sum (a • d ^ 2 * (1 / t) + b • d + c • t)
    subject to
      c_t : StrongLT 0 t
      c_smin : smin * t ≤ d
      c_smax : d ≤ smax * t
      c_τmin : τmin ≤ Vec.cumsum t
      c_τmax : Vec.cumsum t ≤ τmax
```

This time, we show the full definition, including the optimization parameters, some of which are proof obligations. We will need to instantiate the arguments of this λ-term, as the `solve` command expects a term of type `Minimization D R`.

Finally, we are in a position where we can solve the problem. We explain how the parameters are defined by showing an example of a parameter definition: s_{min}. We also show how to prove that it is positive.

```
@[optimization_param, reducible]
def smin_p : Fin n_p → ℝ :=
  ![0.7828, 0.6235, 0.7155, 0.5340, 0.6329,
    0.4259, 0.7798, 0.9604, 0.7298, 0.8405]

lemma smin_p_pos : StrongLT 0 smin_p := by
  intros i; fin_cases i <;> norm_num
```

In this case n_p is 10. The `@[optimization_param]` tag tells CvxLean that the definition holds a constant numerical value. In particular, the real-to-float procedure (see Section 4.7.3) is instructed to read the values of these definitions and turn them into floats to extract the coefficient of the problem. We also instruct the simplifier to unfold the definition of $\texttt{smin}_p$ with the `@[reducible]`

tag. The lemma is proved easily by looking at every index and using `norm_num` (which uses `simp`). Both $\texttt{smin}_p$ and $\texttt{smin}_p\texttt{\_pos}$ are required to instantiate `vehSpeedSchedQuadratic`.

We set $\texttt{a}_p$ `:` $\mathbb{R}$ and $\texttt{d}_p$ `: Fin` $\texttt{n}_p \to \mathbb{R}$ in a similar way. They are both strictly positive. As noted, we need `a * d ^ 2` to be nonnegative for the problem to be in DCP form. The strategy we use to derive atom conditions is limited and sometimes requires some guidance from the user. This is one of those cases. However, we can circumvent this issue by adding lemmas to the `simp` set as shown below.

```
@[simp]
lemma aₚdₚ2_nonneg : 0 ≤ aₚ * dₚ ^ 2 := ...
```

Having defined all of our parameters (which include numerical constants and properties of such constants), we can define a concrete instance of the problem and solve it.

```
def p : Minimization (Fin nₚ → ℝ) ℝ :=
  vehSpeedSchedQuadratic nₚ dₚ τminₚ τmaxₚ sminₚ smaxₚ aₚ bₚ cₚ
    nₚ_pos dₚ_pos sminₚ_pos

solve p
```

At this point, we are almost done, except for the fact that `p` is not the original problem. Therefore, we need to map the solution back to a solution of `vehSpeedSched`. Each of $\texttt{eqv}_1$ and $\texttt{eqv}_2$ generate computable backward maps to retrieve the solution. All the transformation steps that we performed were equivalences through identity maps except for the change of variables $s_i \mapsto d_i/t_i$. Note that it depends on d. In general, these maps may depend on any of the optimization parameters, so we must pass them as arguments.

```
def eqv₁.backward_mapₚ : (Fin nₚ → Float) → Fin nₚ → Float :=
  eqv₁.backward_map nₚ dₚ.float τminₚ.float τmaxₚ.float sminₚ.float
    smaxₚ.float (fun s => aₚ.float * s ^ 2 + bₚ.float * s + cₚ.float)
```

These maps operate at the level of floats, so the arguments should be the floating-point versions of the real arguments. We automatically add the floating-

point translation of each parameter to the environment, appending `.float` to its name.

After defining $\texttt{eqv}_2\texttt{.backward\_map}_p$ in a similar way, we conclude by recovering the solution to the original problem.

```
def sol := eqv₁.backward_mapₚ (eqv₂.backward_mapₚ p.solution)

#eval sol
-- ![0.955578, 0.955548, 0.955565, 0.955532, 0.955564,
--   0.955560, 0.912362, 0.960401, 0.912365, 0.912375]
```

6.3 Fitting a sphere to data

This case study is based on exercise 8.16 from the Stanford Convex Optimization course.

Given m points $x_1, \ldots, x_m \in \mathbb{R}^n$ we want to find $c \in \mathbb{R}^n$ and $r \in \mathbb{R}$ such that the sphere $\{x \in \mathbb{R}^n \mid \|x - c\| = r\}$ such that it minimizes

$$\sum_{i=1}^{m} \left(\|x_i - c\|_2^2 - r^2 \right)^2.$$

This will be the objective function of the optimization problem over c and r.

We require $r \geq 0$ for the problem to be well-defined. We formalize the problem in `CvxLean` as follows.

```
def fittingSphere (n m : ℕ) (x : Fin m → Fin n → ℝ) :=
  optimization (c : Fin n → ℝ) (r : ℝ)
    minimize (∑ i, (‖(x i) - c‖ ^ 2 - r ^ 2) ^ 2 : ℝ)
    subject to
      h₁ : 0 ≤ r
```

This problem is not convex. The issue is the expression $-r^2$, which is a concave function in r. As we have seen in the previous section, a common technique to turn problems into convex forms is to find an appropriate change of variables. In this case we change (c, r) to (c, t) where $t = r^2 - \|c\|_2^2$. It is designed so that the resulting problem is a least squares problem, which conic solvers can easily handle. In this case, by "least squares" we mean that the objective needs to be

of the following form.

$$\sum_{i=1}^{m}(k-a(c,t))^2,$$

where k is constant and $a(c,t)$ is an affine expression in c and t.

To formalize the change of variables in `CvxLean`, we need to build a map from (c,r), which is easily seen to be

$$(c,r)\mapsto\left(c,\sqrt{t+\|c\|_2^2}\right).$$

Its inverse is $(c,t)\mapsto(c,r^2-\|c\|_2^2)$, which corresponds exactly to the definition given earlier. It is only valid if $r\geq 0$, in which case we have that

$$\sqrt{r^2-\|c\|_2^2+\|c\|_2^2}=\sqrt{r^2}=r,$$

as required, where $r\geq 0$ is asserted by `h₁`.

This shows that the change of variables, as described in Section 3.3.2.1, is correct. Nevertheless, as we argue below, this is a special case, and that definition cannot be applied directly. Looking at the definition, we can see that no constraints are added to the resulting problem after the change. That works in most cases; however, here, we should only regard this change of variables as correct if $0\leq t+\|c\|_2^2$. This transformation should, therefore, add such inequality as a constraint. Otherwise, the only constraint would be $0\leq\sqrt{t+\|c\|_2^2}$, which is not provably well-formed and actually a trivial statement in Lean since Lean's $\sqrt{\cdot}$ is artificially extended to 0 for negative values. In `CvxLean`, we always aim to prove that the problem is well-formed. This is explicitly checked for the whole problem by the DCP transformation and for particular parts of the problem by pre-DCP transformations. In this case, well-formedness is needed to rewrite the objective function into a DCP-compliant form. To prove that adding such constraint still yields an equivalence, we must show that the inverse of the change of variables function is not only a right-inverse in the appropriate domain but also a left-inverse, which is clear for this problem. Even though this transformation does not work with our current change-of-variables set-up, we can always resort to proving our own equivalence lemmas as needed, which is what we do here.

As usual, we open an equivalence environment to apply it.

```
equivalence* eqv/fittingSphereT (n m : ℕ) (x : Fin m → Fin n → ℝ) :
    fittingSphere n m x := by
  ... /- 15 LOC -/
```

After further simplification and rewriting the problem in vector notation, the resulting problem, `fittingSphereT`, looks as follows.

```
optimization (c : Fin n → ℝ) (t : ℝ)
  minimize
    Vec.sum ((Vec.norm x ^ 2 - 2 * mulVec x c - Vec.const m t) ^ 2)
  subject to
    h₂ : 0 ≤ t + ‖c‖ ^ 2
```

We have also removed the $0 \leq \sqrt{t + \|c\|_2^2}$ constraint, which, as discussed, is vacuous. Regarding the simplified objective function and how the constraint is used, observe that, after applying the change of variables, r^2 becomes $\left(\sqrt{t + \|c\|_2^2}\right)^2$. To write the problem in the form above, we need to show that this expression equals $t + \|c\|_2^2$, which requires $0 \leq t + \|c\|_2^2$.

The constraint, however, is problematic to solve the problem. The right-hand side is DCP convex, which violates the top-level rule for "$\leq$".

We will show that we can *reduce* (see Section 3.4) `fittingSphereT` to a convex form `fittingSphereConvex`, where the latter is defined as follows.

```
def fittingSphereConvex:=
  optimization (c : Fin n → ℝ) (t : ℝ)
    minimize
      Vec.sum ((Vec.norm x ^ 2 - 2 * mulVec x c - Vec.const m t) ^ 2)
```

It is exactly `fittingSphereT` removing `h₂`.

To show that the reduction is valid, we first had to formalize the following well-known result about least squares problems.

Lemma 6.3.1 (Solution of the least squares problem)**.** Given $a \in \mathbb{R}^n$ with $n > 0$, let $z^* \in \mathbb{R}$ be the minimizer of $f(z) = \sum_{i=1}^n (a_i - z)^2$, then $z^* = \bar{a} = \frac{1}{n}\sum_{i=1}^n a_i$.

Proof. One possibility is to note that $f'(z) = -2n\bar{a} + 2nz$, so $f'(z) = 0$ if, and only if, $z = \bar{a}$. Since $f''(z) = 2n > 0$, it is clear that $\bar{a}$ is the minimum.

Instead, we follow our Lean formalization, a direct proof where we avoid

reasoning about derivatives. The key step is to write f as follows[5]

$$f(z) = n\left((z-\bar{a})^2 + \left(\bar{a^2} - \bar{a}^2\right)\right).$$

If z is optimal then for all y, $f(z) \leq f(y)$, which simplifies to $(z-\bar{a})^2 \leq (y-\bar{a})^2$. Thus, if we choose $y = \bar{a}$, we have $(z-\bar{a})^2 \leq 0$, which implies $z^* = \bar{a}$. □

This result is needed to define the following reduction.

```
def red (hm : 0 < m) :
    (fittingSphereT n m x) ≼ (fittingSphereConvex n m x) :=
  { psi := id,
    psi_optimality := ... }
```

We could use the `reduction` command, but there is not much benefit here, as we want to apply only one step, and we know the target problem exactly. To prove `psi_optimality`, we proceed as follows. First, recall that the objective function of both problems is

$$\sum_{i=1}^{m} \left(\|x_i\|^2 - 2x_i^T c - t\right)^2.$$

Suppose (c^*, t^*) is optimal in `fittingSphereConvex`. By Lemma 6.3.1 with $a_i := \|x_i\|^2 - 2x_i^T c$ and $z := t$,

$$t^* = \frac{1}{m}\sum_{i=1}^{m}\left(\|x_i\|_2^2 - 2x_i^T c^*\right),$$

and, therefore,

$$\begin{aligned} t^* + \|c^*\|_2^2 &= \frac{1}{m}\sum_{i=1}^{m}\left(\|x_i\|_2^2 - 2x_i^T c^* + \|c^*\|_2^2\right) \\ &= \frac{1}{m}\sum_{i=1}^{m}\|x_i - c^*\|_2^2 \\ &\geq 0. \end{aligned}$$

This shows that (c^*, t^*) is feasible in `fittingSphereT`. The second half of the optimality condition is obvious, as both problems have the same objective function. Therefore, it follows that the reduction is valid.

Finally, we can give concrete values to n, m, and x and solve the problem.

[5] Following Marty Cohen's answer to `https://math.stackexchange.com/questions/2554243` (accessed 2024-02-13).

```
solve fittingSphereConvex n_p m_p x_p
```

The solution to `fittingSphere` is retrieved as follows.

```
eqv.backward_map n_p m_p x_p.float fittingSphereConvex.solution
```

Note that we do not need to apply the backward map from the reduction step since it is the identity map.

We conclude this section with a visual representation of the result. Suppose we are given the following $x_1, \ldots, x_{10} \in \mathbb{R}^2$:

$$(1.82, -0.96),\ (1.35, 1.07),\ (0.70, 0.67),\ (0.76, 0.78),\ (2.39, -0.95),$$
$$(0.87, -0.86),\ (1.86, 1.28),\ (0.71, 0.53),\ (0.60, 0.07),\ (0.46, -0.46).$$

Then, CvxLean outputs $c^* \approx (1.66486312, 0.03193191)$ and $r* \approx 1.15903274$. The resulting circle and the given points are displayed in Figure 6.1.

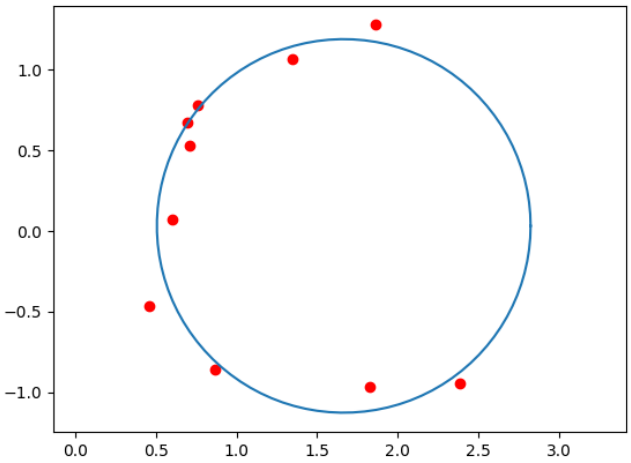


Figure 6.1: Result of fitting a sphere to data given $x_1, \ldots, x_{10} \in \mathbb{R}^2$.

6.4 Truss design

The example presented in this section is adapted from Boyd's tutorial on geometric programming [BKVH07]. We will transform it to a solver-compatible form using some basic transformations and, crucially, the pre-DCP procedure from Chapter 5.

Consider a two-bar truss with height $2h$ and width w as depicted in Figure 6.2. The two beams are cylinders with outer radius R and inner radius r. The variables of the problem are $h, w, R, r \in \mathbb{R}$. The total volume of the truss is given by

$$V := 2A\sqrt{w^2 + h^2},$$

where $A := 2\pi(R^2 - r^2)$ is the cross-sectional surface of the bars. The goal is to minimize the weight of the truss with the constraints that it is strong enough to withstand positive forces F_1 and F_2. Assuming a homogenous material, the weight is proportional to the volume, so it suffices to minimize V. The maximum allowed force in each bar depends on a given constant $\sigma > 0$, the maximum allowable stress coefficient. These constraints can be phrased as:

$$\frac{\sqrt{w^2 + h^2}}{2h} F_1 \leq \sigma A \quad \text{and} \quad \frac{\sqrt{w^2 + h^2}}{2w} F_2 \leq \sigma A.$$

Moreover, we impose the following additional constraints:

$$0 < h_{min} \leq h \leq h_{max}, \quad 0 < w_{min} \leq w \leq w_{max} \quad \text{and} \quad 1.1r \leq R \leq R_{max},$$

where h_{min}, h_{max}, w_{min}, w_{max} and R_{max} are fixed. It is clear that $h > 0$ and $w > 0$. We assume that $r > 0$, which implies that $R > 0$.

The full optimization problem in `CvxLean` is, therefore:

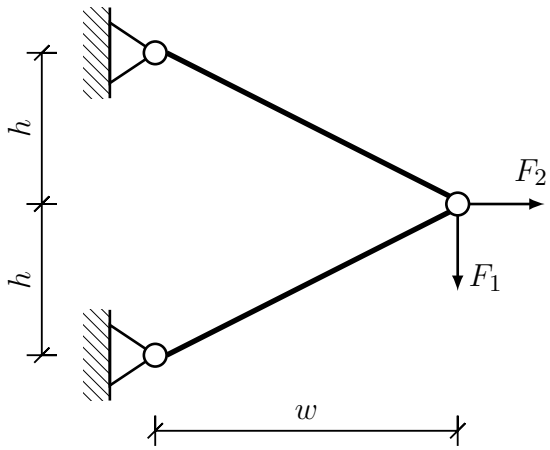


Figure 6.2: Two-bar truss.

```
def trussDesign :=
  optimization (h w r R : ℝ) with A := 2 * π * (R ^ 2 - r ^ 2)
    minimize 2 * A * sqrt (w ^ 2 + h ^ 2)
    subject to
      c_r    : 0 < r
      c_F₁   : F₁ * sqrt (w ^ 2 + h ^ 2) / (2 * h) ≤ σ * A
      c_F₂   : F₂ * sqrt (w ^ 2 + h ^ 2) / (2 * w) ≤ σ * A
      c_hmin : hmin ≤ h
      c_hmax : h ≤ hmax
      c_wmin : wmin ≤ w
      c_wmax : w ≤ wmax
      c_R_lb : 1.1 * r ≤ R
      c_R_ub : R ≤ Rmax
```

The syntax after `with` is parsed as a "let" declaration in the expressions where it appears; for example, the objective function is as follows.

```
fun (h, w, r, R) =>
  let A := 2 * π * (R ^ 2 - r ^ 2);
  2 * A * sqrt (w ^ 2 + h ^ 2)
```

In particular, that means that `A` is *not* an optimization variable.

In this form, the problem is not convex. The objective function, as well as constraints `c_F₁` and `c_F₂`, are problematic. It suffices to note that the expression $-r^2$ makes the objective function non-convex, an issue we already saw in the previous case study.

It is not a geometric program, either. All of its variables are positive, but once again, the negative sign in $-r^2$ means that the objective function is not a posynomial. The first step is to turn this problem into a geometric program.

The way in which we have written the problem, using A, gives us a good hint on how to do that. If A was a positive variable, then the objective function and constraints `c_F₁` and `c_F₂` would follow the rules of geometric programming. Since A depends on R and r, it is natural to perform a change of variables on one of them. From $A := 2\pi(R^2 - r^2)$, we deduce the following change.

$$R \mapsto \sqrt{\frac{A}{2\pi} + r^2}.$$

Note that this implies that A will now be an optimization variable. To perform this transformation step, we enter equivalence mode as follows.

```
equivalence* eqv₁/trussDesignGP (hmin hmax wmin wmax Rmax σ F₁ F₂ : ℝ) :
    trussDesign hmin hmax wmin wmax Rmax σ F₁ F₂ := by
  ... /- 34 LOC -/
```

After changing R to A, the problem is still not a geometric or generalized geometric program because of the constraint

$$1.1r \leq \sqrt{\frac{A}{2\pi} + r^2},$$

corresponding to `c_R_lb`.

Recall that the rules of disciplined geometric programming only allow the right-hand side of the inequality to be a positive monomial. With that in mind, we rewrite the constraint as follows:

$$\begin{aligned} 1.1r \leq \sqrt{\frac{A}{2\pi} + r^2} &\Leftrightarrow (1.1r)^2 \leq \left(\sqrt{\frac{A}{2\pi} + r^2}\right)^2 \\ &\Leftrightarrow 1.21r^2 \leq \frac{A}{2\pi} + r^2 \\ &\Leftrightarrow 0.21r^2 \leq \frac{1}{2\pi} A \end{aligned}$$

Crucially, this proof goes through because we know that $r > 0$ from `c_r`. The resulting problem `trustDesignGP` looks as follows, where we have also renamed the last two constraints:

```
optimization (h : ℝ) (w : ℝ) (r : ℝ) (A : ℝ)
  minimize 2 * A * sqrt (w ^ 2 + h ^ 2)
  subject to
    c_r : 0 < r
    c_F₁ : F₁ * sqrt (w ^ 2 + h ^ 2) / (2 * h) ≤ σ * A
    c_F₂ : F₂ * sqrt (w ^ 2 + h ^ 2) / (2 * w) ≤ σ * A
    c_hmin : hmin ≤ h
    c_hmax : h ≤ hmax
    c_wmin : wmin ≤ w
    c_wmax : w ≤ wmax
    c_A_lb : 0.21 * r ^ 2 ≤ A / (2 * π)
    c_A_ub : sqrt (A / (2 * π) + r ^ 2) ≤ Rmax
```

In CvxLean, we do not have a DGP library, but we achieve the same outcome by performing a change of variables and calling `pre_dcp` to rewrite the problem as needed to make it DCP-compliant. This transformation step requires knowing that all the parameters are positive. The change of variables

$$(h, w, r, A) \mapsto \left(\exp(h'), \exp(w'), \exp(r'), \exp(A')\right)$$

is only valid if we can prove that $h, w, r, A > 0$. The first two follow immediately from $h_{min} > 0$ and $w_{min} > 0$. The third one is exactly `c_r`. The last condition, $A > 0$, cannot be proved automatically with the current arithmetic tactics. Instead, we need to provide a proof manually, which consists of a short proof that follows from `c_A_lb` and `c_r`.

The rewrite rules needed to transform the problem into DCP form rely on the positivity of the parameters and will fail if we do not provide them. In fact, the problem would not be a valid geometric program without these assumptions, so it makes sense that our pre-DCP rewrite system requires them.

```
equivalence* eqv₂/trussDesignConvex
    (hmin hmax wmin wmax Rmax σ F₁ F₂ : ℝ)
    (hmin_pos : 0 < hmin) (hmin_le_hmax : hmin ≤ hmax)
    (wmin_pos : 0 < wmin) (wmin_le_wmax : wmin ≤ wmax)
    (Rmax_pos : 0 < Rmax) (σ_pos : 0 < σ) (F₁_pos : 0 < F₁)
    (F₂_pos : 0 < F₂) :
    trussDesignGP hmin hmax wmin wmax Rmax σ F₁ F₂ := by
  ... /- 14 LOC -/
```

The resulting problem, `trussDesignConvex`, follows the rules of DCP.

```
optimization (h' : ℝ) (w' : ℝ) (r' : ℝ) (A' : ℝ)
    minimize log (rexp (2 * h') + rexp (2 * w')) + 2 * (log 2 + A')
    subject to
      c_F₁ : 1 / 2 * log (rexp (2 * h') + rexp (2 * w'))
           ≤ log σ + A' - (log (F₁ / 2) - h')
      c_F₂ : 1 / 2 * log (rexp (2 * h') + rexp (2 * w'))
           ≤ log σ + (w' + A') - log (F₂ / 2)
      c_hmin : log hmin ≤ h'
      c_hmax : rexp h' ≤ hmax
      c_wmin : log wmin ≤ w'
```

```
c_wmax : rexp w' ≤ wmax
c_A_lb : log (21 / 100) + 2 * r' ≤ A' - log (2 * π)
c_A_ub : rexp A' ≤ (Rmax * Rmax - rexp (2 * r')) * (2 * π)
```

Starting with `trustDesignGP` after the exponential change of variables, our pre-DCP procedure found a sequence of 84 rewrite rules resulting in the problem above. The main objective of the procedure is to find a DCP-compliant problem. We have added some heuristics so that simpler problems are preferable, but the resulting problem is not necessarily the simplest. This explains why the problem above is rather convoluted. From the user's point of view, this is not an issue as this is the last step before solving it, so as long as the problem is in DCP form, the specific expressions involved are usually irrelevant.

Finally, we can give numerical values to the optimization parameters and use the `solve` command. We set $\texttt{hmin}_p$ `:= 1`, $\texttt{hmax}_p$ `:= 100`, $\texttt{wmin}_p$ `:= 1`, $\texttt{wmax}_p$ `:= 100`, $\texttt{Rmax}_p$ `:= 10` and σ_p `:= 0.5`. We also prove the necessary conditions on these parameters and proceed as follows.

```
solve trussDesignDCP hmin_p hmax_p wmin_p wmax_p Rmax_p σ_p F1_p F2_p
  hmin_p_pos hmin_p_le_hmax_p wmin_p_pos wmin_p_le_wmax_p Rmax_p_pos σ_p_pos
  F1_p_pos F2_p_pos
```

As usual, we also define the backward maps with respect to the parameters.

```
def eqv1.backward_map_p := eqv1.backward_map hmin_p.float hmax_p.float
  wmin_p.float wmax_p.float Rmax_p.float σ_p.float F1_p.float F2_p.float

def eqv2.backward_map_p := eqv2.backward_map hmin_p.float hmax_p.float
  wmin_p.float wmax_p.float Rmax_p.float σ_p.float F1_p.float F2_p.float
```

Finally, we show the solution below.

```
def sol := eqv1.backward_map_p
  (eqv2.backward_map_p trussDesignConvex.solution)

def h_p_opt := sol.1     -- 1.000000
def w_p_opt := sol.2.1   -- 1.000517
```

```
def r_p_opt := sol.2.2.1 -- 0.010162
def R_p_opt := sol.2.2.2 -- 2.121443
```

Note that both backward maps are non-trivial as they come from two separate changes of variables. The retrieval map looks as follows:

$$(h, w, r, R) := \left(\exp(h'), \exp(w'), \exp(r'), \sqrt{\frac{\exp(A')}{2\pi} + (\exp(r'))^2}\right),$$

where $(h', w', r', A', \dots)$ is the point returned by the solver, with "…" corresponding to the auxiliary variables introduced by the DCP transformation.

This case study shows how the novel e-graph-based pre-DCP algorithm from Chapter 5 is applied to a non-trivial textbook problem. It also provides evidence that it can be used as an alternative approach to DGP.

6.5 Hypersonic shape design

This case study is based on an example from `CVXPY`[6]. It is an instance of quasi-convex programming, which we will approach using the machinery for pre-DCP transformations presented in Chapter 5 in the style of the previous section.

A problem in aerospace design is maximizing an object's lift-to-drag ratio in a hypersonic flow. We assume the object's shape is a right triangle determined by its width Δx and its height Δy, with hypotenuse s. We consider the simple case where the flow is horizontal and determined by $\vec{v} = (1, 0)$ as shown in Figure 6.3.

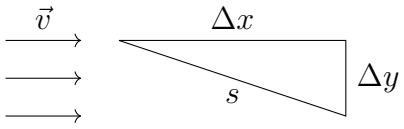


Figure 6.3: Triangular object in a hypersonic flow.

Let L be the lift and D be the drag. The goal is, therefore, minimizing

$$\frac{D}{L} = \frac{c_d}{c_l},$$

where c_d and c_l are the drag and lift coefficients, respectively. To derive the formulas for these coefficients, we first introduce the pressure coefficient c_p, which

[6] `https://www.cvxpy.org/examples/dqcp/hypersonic_shape_design.html` (accessed 2024-01-30).

follows Newton's sine-square law of air resistance. We have that $c_p = 0$ everywhere except on the surface along s, where it is given by $c_p = 2(\vec{v} \cdot \vec{n})^2$ where

$$\vec{n} = \left(-\frac{\Delta y}{s}, -\frac{\Delta x}{s} \right)$$

is the unit normal vector of the hypotenuse. A simple calculation gives us

$$c_p = \frac{2\Delta y^2}{s^2}.$$

Let c be the chord length of the body, which in this case is Δx. We assume that $0 < \Delta x$. It makes sense to assume that $0 \leq \Delta x$ since it is a distance. Moreover, we are not interested in cases when there is zero lift, so D/L is always well-defined. Clearly, $\Delta x = 0$ gives zero lift because of the symmetry of the problem, so it is reasonable to assume $\Delta x \neq 0$.

The lift and drag coefficients are given by:

$$c_d = \frac{1}{c} \int_0^s -c_p \vec{n}_x \mathrm{d}s = -\frac{s}{c} c_p \vec{n}_x = \frac{s}{c} \frac{2(\Delta y)^2}{s^2} \frac{\Delta y}{s},$$
$$c_l = \frac{1}{c} \int_0^s -c_p \vec{n}_y \mathrm{d}s = -\frac{s}{c} c_p \vec{n}_y = \frac{s}{c} \frac{2(\Delta y)^2}{s^2} \frac{\Delta x}{s}.$$

We resize the triangle so that $s = 1$ and, consequently, $\Delta y = \sqrt{1 - (\Delta x)^2}$. More precisely, we perform the change of variables $\Delta x \mapsto (\Delta x)'/s$ and $\Delta y \mapsto (\Delta y)'/s$. Finally, we obtain that

$$\frac{D}{L} = \frac{\Delta y}{\Delta x} = \frac{\sqrt{1 - (\Delta x)^2}}{\Delta x} = \sqrt{\frac{1}{(\Delta x)^2} - 1}.$$

This will be the objective function of our problem. The only constraint we impose is a lower bound on the size of the object. Following the `CVXPY` example development, this can be expressed as a convex constraint by requiring that a rectangle of height a and width b is contained inside the triangle, with one of the edges of length b lying along the hypothenuse. Putting everything together, we obtain the following optimization problem:[7]

$$\begin{aligned} \text{minimize} \quad & \sqrt{\frac{1}{(\Delta x)^2} - 1} \\ \text{subject to} \quad & 0 < \Delta x \leq 1 \\ & a\frac{1}{\Delta x} - (1 - b)\sqrt{1 - (\Delta x)^2} \leq 0. \end{aligned}$$

[7] Our formalization begins with this form. We leave it as future work to formalize the pen-and-paper transformations up to this point.

The constraint $\Delta x \leq 1$ comes from the geometry of the triangular object. This problem follows the rules of disciplined quasiconvex programming. The constraints are DCP-compliant and, therefore, DQCP-compliant. Inverses of positive variables are DCP convex, and $1-(\Delta x)^2$ is DCP concave, which is the argument of the square root, a DCP concave, and increasing function. The objective function is quasiconvex as monotone functions with convex real domains, such as the square root, are quasilinear [AB20, §2.3], and $1/(\Delta x)^2 - 1$ is DCP convex. Thus, `CVXPY` is able to canonize it and send it to the solver by setting the `qcp` flag to `True`. Interestingly, when we ran `CVXPY` in its `qcp` mode on this problem, we obtained an incorrect solution. The error came from one of the transformation steps, which we reported as a bug[8]. This makes this case study particularly relevant as it justifies one of our main motivations, which is the fact that there are few guarantees when updating unverified modeling frameworks such as `CVXPY`. Such an error in a transformation step would be impossible in `CvxLean` as it would break the chain of equivalences.

Currently, we do not support arbitrary quasiconvex programs. Fortunately, there is a way to solve this particular problem without using DCQP. First, we define the problem in `CvxLean` as follows:

```
def hypersonicShapeDesign :=
  optimization (Δx : ℝ)
    minimize sqrt ((1 / Δx ^ 2) - 1)
    subject to
      h₁ : 10e-6 ≤ Δx
      h₂ : Δx ≤ 1
      h₃ : a * (1 / Δx) - (1 - b) * sqrt (1 - Δx ^ 2) ≤ 0
```

We have replaced the positivity condition on Δx by $10^{-6} \leq \Delta x$. This is a common pattern when positivity constraints cannot be eliminated, as there is no cone for strict inequalities. Using the `equivalence*` command, we can invoke `pre_dcp` to make the problem DCP-compliant as follows.

[8] `https://github.com/cvxpy/cvxpy/issues/2165` (accessed 2024-01-30).

```
equivalence* eqv₁/hypersonicShapeDesignConvex
    (a b : ℝ) (ha : 0 ≤ a) (hb₁ : 0 ≤ b) (hb₂ : b < 1) :
    hypersonicShapeDesign a b := by
  pre_dcp
```

Requiring $0 \leq a$ and $0 \leq b < 1$ is needed to make the transformation step work. Recall that a and b enforce a minimum surface; they are the height and width of a rectangle contained inside the triangle, respectively. Certainly, they should be nonnegative. In this case, $0 \leq b$ is not used by any of the rewrites, but it serves as an extra sanity check. Since the triangle's hypotenuse is 1, it also makes sense that $b < 1$.

The pre-DCP procedure finds a sequence of 16 rewrites and transforms the problem into:

```
optimization (Δx : ℝ)
  minimize Δx ^ (-2) - 1
  subject to
    h₁ : 1 / 100000 ≤ Δx
    h₂ : Δx ≤ 1
    h₃ : sqrt a ^ 2 / Δx / (1 - b) ≤ sqrt (1 - Δx ^ 2)
```

The changes in the constraints are irrelevant and come from the fact that the sequence of rewrites is not necessarily optimal. More importantly, the objective function is squared and simplified. Squaring it is valid since $0 < \Delta x \leq 1$ from `h₁` and `h₂`, which implies that $1/(\Delta x)^2 - 1 \geq 0$. The pre-DCP e-class analysis based on interval arithmetic can readily assert this fact. Interestingly, this deduction cannot be replayed immediately in Lean, as there is limited support for non-linear arithmetic. For now, we have extended the `positivity` tactic directly, adding the lemma $\forall x.\, 0 < x \leq 1 \Rightarrow 1/x^2 - 1 \geq 0$ (in order to make `arith` work, which uses `positivity`). This approach does not scale, and better tactics would be highly desirable. In the future, we hope to replace this with a more general-purpose non-linear arithmetic tactic based, for example, on SMT, with proof reconstruction, or on interval arithmetic.

At this point, we are ready to solve the problem. We set `aₚ := 0.05` and `bₚ := 0.65` and run the `solve` command.

```
solve hypersonicShapeDesignConvex a_p b_p a_p_nonneg b_p_nonneg b_p_lt_one
```

As always, the proof obligations on `a` and `b` set when building the transformation need to be proved and passed as arguments for the particular instances $\mathtt{a}_p$ and $\mathtt{b}_p$. The following expression gives the width of the wedge.

```
def w_p_opt := eqv₁.backward_map a_p.float b_p.float
  hypersonicShapeDesignConvex.solution
```

In fact, since rewrites are equivalence-preserving via the identity map, we may also ignore the backward map.

```
hypersonicShapeDesignConvex.solution -- 0.989524
```

We can also retrieve the height and L/D ratios.

```
def h_p_opt := Float.sqrt (1 - w_p_opt ^ 2) -- 0.144368
def ldRatio_p_opt := 1 / (Float.sqrt ((1 / w_p_opt ^ 2) - 1)) -- 6.854156
```

This is the expected solution to the problem, and we avoided the several solving iterations used by `CVXPY`'s bisection algorithm for quasiconvex programs by simply calling `pre_dcp` and rewriting the problem into DCP form.

We can go one step further, noting that the main constraint can be written as follows:

$$
\begin{aligned}
& a\frac{1}{\Delta x} - (1-b)\sqrt{1-(\Delta x)^2} \le 0 \\
& \quad \Leftrightarrow a\frac{1}{\Delta x} \le (1-b)\sqrt{1-(\Delta x)^2} && \text{by arithmetic} \\
& \quad \Leftrightarrow \left(a\frac{1}{\Delta x}\right)^2 \le \left((1-b)\sqrt{1-(\Delta x)^2}\right)^2 && \text{because both sides are nonnegative} \\
& \quad \Leftrightarrow a^2\frac{1}{(\Delta x)^2} \le (1-b)^2(1-(\Delta x)^2) && \text{by arithmetic.}
\end{aligned}
$$

The objective can also be changed to $((\Delta x)^2)^{-1} - 1$. In this form, we can see that all occurrences of Δx are squared. Therefore, we can consider the change of variables $\Delta x \mapsto \sqrt{z}$, which is valid since $0 \le \Delta x$, to simplify the problem further. After this step, which has also been formalized in `CvxLean`, we obtain:

```
optimization (z : ℝ)
  minimize z⁻¹ - 1
  subject to
    h₁ : 1 / 10000000000 ≤ z
    h₂ : z ≤ 1
    h₃ : a ^ 2 * z⁻¹ ≤ (1 - b) ^ 2 * (1 - z)
```

The conic form of the problem above has 3 variables and 9 constraints, while the conic form of `hypersonicShapeDesignConvex` has 5 variables and 12 constraints. We see an improvement in solving time, which goes down from ~ 490ms to ~ 270ms. After mapping the solution back, which is needed here because of the change of variables, we obtain roughly the same solution `w_p_opt'`. It is likely, however, that the solution to the simplified problem is more accurate. While a more rigorous floating-point analysis would be required, an interesting observation is that the expression below is not ≤ 0 when checked with Lean's evaluator, which uses system-level floats.

```
a_p.float * (1 / w_p_opt) - (1 - b_p.float) * Float.sqrt (1 - w_p_opt ^ 2)
```

However, the same expression with `w_p_opt'` is indeed ≤ 0, as expected. It is worth noting that the expression above is ≤ 0.000001, so it is reasonably close to 0. We speculate that `w_p_opt` carries more round-off error since it results from a more complex problem.

This example shows one interesting potential use case for our tool: rigorously transforming problems into forms that can be solved more efficiently or more accurately. The interactive nature of `CvxLean` and the availability of manually-guided problem transformations such as those in Chapter 3 makes this exploration of guaranteed-correct reformulations relatively straightforward.

6.6 Summary

We have shown how the techniques developed in previous chapters are applied to real-world case studies. These are the key takeaways from each of them:

(6.2) The vehicle speed scheduling case study illustrates how to perform a change of variables to make the problem convex. We show how the problem is

rewritten and put in vector notation, which is needed due to some limitations regarding binders (see Section 7.4.2). The problem is specialized to a quadratic fuel function, simplified further, and solved. To approach this problem in full generality, we would need the perspective function atom, which would require considerably extending the atom declaration machinery, as we discuss in Section 7.4.1.

(6.3) We considered the problem of fitting a sphere to data. We discuss the need to be precise about the resulting constraints after each transformation step to ensure that the problem is well-formed, even before attempting to canonize it, which is usually never a concern in other tools. This case study is an example of a reduction that is not an equivalence. Moreover, proving the main property of the reduction required proving some results about least squares problems.

(6.4) We discuss a mechanical design problem regarding the measures and position of a two-bar truss. It can be turned into a geometric program, which we show we can tackle using the `pre_dcp` tactic. This example is a very useful stress test for the pre-DCP procedure as it requires a large number of rewrites. We also discuss the importance of backward maps, which are particularly complex in this example.

(6.5) We consider the aerospace design problem of finding the optimal proportions of a triangular object in a hypersonic flow. It can be formulated as a quasiconvex program. This case study is particularly relevant as `CVXPY` 1.4.2 gives an incorrect solution due to an erroneous transformation step. We show how the `pre_dcp` tactic can be immediately applied, yielding a DCP-compliant problem from which we obtain a correct solution. We discuss how further simplification steps may be beneficial to aid the solver.

These case studies show the potential of `CvxLean` and the applicability of its features. They are also helpful in understanding the tool's limitations, such as the ones discussed above or places where usability could be improved. These observations lead us to the concluding chapter of this thesis.

Chapter 7

Conclusion

We begin the conclusion of this thesis by reflecting on our work. Section 7.1 answers our initial research questions, and Section 7.2 places our achievements in a larger context. We address how to reach the convex optimization community in Section 7.3. Then, we turn our eyes to the future of `CvxLean` and lay out some concrete potential next steps in Section 7.4. In Section 7.5, we consider some less well-defined and more ambitious research questions that arose from our work. We give some final thoughts in Section 7.6.

7.1 Revisiting the research questions

In Section 1.2, we listed the questions that have guided our work. Here, we restate them and explain exactly how we have addressed them, serving also as a summary of contributions.

(RQ1) *Is it possible to embed pre-DCP transformations in a proof assistant so the problem can be interactively transformed conveniently without introducing errors?*

Developing commands and tactics for rigorous user-guided transformations, as accounted in Chapter 3, is how we have tackled this question. Besides producing correctness certificates, they are also interactive, showing users how problems look at every step and allowing for user-friendly problem exploration. These transformations are essentially manual with only small bits of automation (e.g., proving side conditions in changes of variables). Our biggest contribution on the automation front for pre-DCP transformations is discussed in the next answer.

(RQ1.1) *How much of this process can be automated?*
The answer to this question is found in Chapter 5, where we explore how to automate the conversion to DCP form. Using e-graphs is a novel approach in this domain which we have demonstrated to be a practical way to find a problem in DCP form in the space of equivalent problems. While the problems that the procedure can handle are still limited (mainly because of the current pre-defined language), we see our algorithm as the foundation for e-graph-based rewriting of optimization problems. We believe that this approach could be useful not only in our verified setting but also for other frameworks.

(RQ1.2) *Can we transform geometric, quasiconvex, and other non-convex programs without specialized frameworks such as DGP or DQCP?*
In our evaluation of the pre-DCP procedure (Section 5.7), we discuss to which extent we can transform problems in DGP and DQCP form. We argue that geometric (not generalized) programs can be rewritten into DCP form using our system after a manual exponential change of variables. Regarding quasiconvex programs, we can handle some instances where ratios and compositions with monotone functions can be simplified. The truss design problem in Section 6.4 is a good example of how to solve a geometric program without DGP. Similarly, the hypersonic shape design problem from Section 6.5 illustrates how a quasiconvex program can be solved without DQCP.

(RQ2) *Can we develop a verified DCP procedure with comparable capabilities to state-of-the-art tools such as* `CVXPY`*?*
Chapter 4 explains in detail the mathematics and implementation of a verified DCP canonization algorithm. Its capabilities are demonstrated through a large number of examples and tests, which are transformed into conic form and solved with a back-end solver. We even go beyond what other frameworks can do; notably because of atom inference and rigorously handling side conditions. There is still some work to do to be as complete as `CVXPY`. Our atom library is still limited, although its current scope gives us strong reasons to believe that most of the atoms in `CVXPY` can fit nicely in our design. Some technical exceptions that would require further research are discussed in Section 7.4.1.

(RQ2.1) *How do we make it user-friendly and extensible?*

The DCP algorithm is applied by `solve` without any further work from the user as long as the problem is in DCP form. It can be extended by adding more atoms to the atom library. Identifying and formalizing the key extra proof obligations needed for each atom was the main breakthrough that allowed us to set up an atom library analogously to `CVXPY` with just some additional fields. Users can conveniently declare new atoms with the `declare_atom` command, which sets up the proof goals automatically. The varied set of atoms we have formalized (see Appendix A) is good evidence of its extensibility.

(RQ2.2) *Are there any advantages to this approach beyond correctness?*
Working within our formal framework can give users a better understanding of the problem at hand. For example, the mathematical rigor with which atom conditions are checked can provide precise and meaningful error messages explaining why a DCP transformation might have failed. This also applies to the side conditions of any other transformation, which, as discussed, are often subtle and easy to miss. In other approaches, errors in manual pre-DCP transformations can result in unexpected values, and DCP transformations that do not satisfy all the conditions strictly can result in infeasible problems. Both these outcomes are not very informative and might even be incorrectly accepted by the user. There are also benefits in formally defining optimization problems in Lean. These are mathematical objects whose properties can be stated as lemmas, thus allowing users to reason about them as needed for their application (e.g., see Section 6.3). Finally, Lean's interactive features are a differentiating factor. Conveniently exploring how problems change by seeing the effects of transformations in Lean's infoview panel is one of `CvxLean`'s main advantages compared to other tools.

7.2 The bigger picture

In this section, we will try to place our work in a larger context. Some of our work, in particular regarding the novel pre-DCP transformations, can be seen as a contribution to convex optimization modeling. However, our main contribution is on the theorem-proving and proof engineering side. Our work takes the form of a software package embedded in a proof assistant rather than a formalization effort, which is what proof assistants are mostly known for. In that sense, `CvxLean` is part of the family of tools outlined in Section 3.1.1, where an ITP environment

wraps some piece of mathematical software such as a computer-algebra system. We propose a phrase that encapsulates this line of research: *proof-assisted mathematical software*. Here, by "proof-assisted", we mean using a mechanical proof assistant and not using proofs, which is only a part of it.

More broadly, our work is an instance of proof-assisted software development. It is related to software verification and correct-by-construction software development in that it is an approach to building rigorous software. In many ways, our methodology resembles that of *proof-carrying code* [Nec97], a technique for certified compilation where proofs are generated in parallel to the compiled program, witnessing adherence to some given safety rules for that particular instance.

7.3 Reaching out to the optimization community

We have made substantial progress toward making `CvxLean` easy to interact with. Our vision is that new users only need to learn about our syntax and a limited number of tactics and commands. If we want our tool to be useful for the convex optimization community, it is unrealistic to expect in-depth knowledge about Lean. Our examples show that we have successfully been able to hide a lot of Lean's complexity for simple problems. We consider an end-to-end example.

```
def p :=
  optimization (x y : ℝ)
    minimize -2 * x
    subject to
      c₁ : 0 ≤ x
      c₂ : 1 < y
      c₃ : log (y - 1) ≤ 2 * sqrt x + 1
      c₄ : 3 * x + 5 * y ≤ 10

equivalence* eqv/q : p := by
  change_of_variables! (v) (y ↦ v + 1)
  change_of_variables! (w) (v ↦ exp w)
  remove_constr c₂ =>
    field_simp; arith
  rw_constr c₃ into (w ≤ 2 * sqrt x + 1) =>
    field_simp
```

```
solve q

#eval eqv.backward_map q.solution -- (1.666667, 1.000000)
```

This illustrates the intended usage of the tool. Most of the code is CvxLean-specific. As mentioned, our design philosophy is to control the user's interaction with the system as much as possible. The idea is that users should be able to model simple problems by following CvxLean's examples and documentation.

However, requiring basic knowledge of Lean and mathlib definitions and tactics is unavoidable. We mainly expect users to know how to navigate a proof script and understand the infoview. Knowing how to apply basic tactics to discharge goals is also needed. In particular, arithmetic simplification tactics such as `field_simp` and `arith` above.

This kind of automation is useful to avoid knowing about the contents of mathlib in depth, which can be a considerable hurdle. One way we aid users in overcoming this difficulty is by automating key parts of the workflow. A clear example is the `pre_dcp` tactic, but also the DCP transformation happening behind the scenes in the `solve` command. Another way in which we help is by supporting guided transformations such as `rw_constr` c_3 `into ...`. Specifying target expressions and automatically discharging the proof obligations is more intuitive for non-specialists than surgically modifying expressions.

For more complicated transformations, such as the ones in our case studies, some steps require several fine-tuned tactic applications with specific lemmas. Moreover, some side conditions, such as the ones coming from changes of variables, can sometimes not be proved automatically. In those cases, the proof must be supplied explicitly by the user.

We also need to think about errors. A considerable amount of effort has been put into providing meaningful error messages. It is still possible, however, to see Lean errors that are out of our control and we have not caught. For example, there can be parsing errors or type errors, which can be hard to decipher. There might also be errors coming from the proofs, e.g., regarding failures in type-class inference. We are aware that these can be discouraging, so we aim to handle them ourselves as much as possible and point the user in the right direction.

We conclude this section by considering a radically different approach to reaching a wider audience. While improving and simplifying the interaction with Lean

is crucial, the reality is that Lean will not replace languages such as Python for convex optimization projects. Nevertheless, those projects could still benefit greatly from our verified transformations. Therefore, moving forward, an interesting project could be to develop, for example, an interface between `CVXPY` and `CvxLean` in a way in which our (automatic) transformations can be applied to problems defined in `CVXPY`. Lean would run in the background, and the interface would translate in both directions between `CVXPY` problems, which are Python objects. Of course, a major drawback with this approach is that we would lose all the interactive features.

7.4 Future work

This section is designed to be a fairly detailed roadmap for `CvxLean`. We discuss several projects concerning different parts of the tool. We justify why they are needed and give some ideas for approaching them.

7.4.1 More transparent atom dependencies

In the vehicle speed scheduling case study (Section 6.2), we discussed the difficulty of declaring the perspective function atom briefly. It is a DCP convex atom matching expressions of the form $sf(x/s)$ where s is a nonnegative constant, and f is DCP convex. Its implementation is unique in that it depends on canonizing f to conic form first. We can say that it is a *higher-order* atom as it takes a function composed of other atoms as an argument. For now, we consider the simple case when f is an atom. When an atom is declared using the `declare_atom` command, we expect to be able to canonize its implementation objective and constraints to conic form at "declaration time" and store it then for later use. That is not possible in the case of the perspective function. We suggest two separate ideas to enable higher-order atoms:

- **Defining an `Atom` type (or type class).** We do not have a formal definition of an atom. An atom declaration is a series of Lean expressions whose type is defined by a meta-level function. Imagine we had a generic type for an atom (with its signature, implementation, and proof obligations) in a way such that a term of said type would store the same information as an atom declaration currently does. It might be useful to split this type into

`ConvexAtom`, `ConcaveAtom`, and `AffineAtom`. Then, we could potentially define the perspective function as follows:

```
declare_atom perspective [convex] (f : ℝ → ℝ)? (x : ℝ)?
    (hf : ConvexAtom f)& (s : ℝ)& : s * f (x / s) := ...
```

The parameter `hf` could then be used in the declaration. Assuming that every atom declaration also adds the appropriate atom instance to the environment, the canonization procedure would be able to find it and replace uses of `hf` in the declaration of the perspective function appropriately. This might also require nontrivial simplifications to ensure the resulting expressions are in the desired form.

We have not defined these types because how to do so is unclear. The type of the implementation depends on the signature, and the type of the proof obligations depends on both the signature and the implementation. All of these depend on the parameters. These interdependencies would require a clever use of dependent types. Looking at the list of atoms that we have already declared, we can see that they are quite varied. Thus, it remains a challenge to make all these declarations elements of the same type. Having an `Atom` type would also be useful to verify the correctness of the DCP procedure itself, which we discuss in Section 7.5.

- **Allowing unsafe atom definitions**. This is a less principled approach, although it is likely easier to implement. Each field in atom declarations elaborates to an `Expr` of the type specified by the command. Instead, we may ask the user to write a *procedure* that returns an `Expr`, which in this case would be a program of type `MetaM Expr`. The idea then would be that the canonization procedure would *execute* the code stored in the atom declaration. In this approach, atom declarations could be dynamic objects rather than static expressions as they currently are. This would enable access to the expressions stored in the atom library at declaration time and the ability to combine them as needed. We call this approach "unsafe" as we would lose control over the types of the returned expressions, and it would be the user's responsibility to ensure that they are correct. An incorrect expression would make the resulting conic form and/or the equivalence witness fail to type-check, so the overall correctness of the DCP

transformation would not be compromised. The specifics of how this idea could be materialized still require further thought.

A related issue arises with *recursive* atoms. These are atoms whose implementation may involve an instance of themselves. We encountered this issue when trying to define the power atom in full generality. Currently, we only consider some instances of powers (-2, -1, 0.5, and 2), each with its own declaration. In `CVXPY`, the power atom is defined for arbitrary fractional powers. This utilizes rather intricate machinery. In particular, for the decimal part, one needs to take its *dyadic completion* to approximate it as sums of inverse powers of two. We will ignore that and focus on integer powers. In that case, we could potentially implement them by allowing recursive or mutually recursive atom declarations. It is clear how that would work for positive powers of two of the form x^{2^i}, which are convex everywhere. The base case is $i = 1$, which we have already defined. The case $x^{2^{i+1}}$ can be implemented with objective function t and constraints $x^{2^i} \leq v$ and $v^2 \leq t$. The declaration should detect that $x^{2^i} \leq v$ is a valid recursion. This idea can be generalized for arbitrary integers, with the caveat that the rest of positive powers require x to be nonnegative and negative powers require x to be positive. Note that the domain restrictions are needed for the functions to be convex. Take positive powers and consider an atom declaration of the form x^{2n-1} where $2n-1$ with $n \geq 1$ is not a power of two. We could implement x^{2n-1} with objective t and constraints $x^n \leq v$ and $v^2 \leq xt$. The latter can be expressed with the conic constraint $\left(x+t, (x-t, 2v)^T\right) \in \mathcal{Q}^3$. One could define the atom for x^{2n} similarly, changing the second implementation constraint to $v^2 \leq t$. These declarations would need to be in a mutually inductive block with powers of powers of two, and a proof of termination would be required.

Allowing declarations of this form would also require modifying how the canonization procedure uses atoms. Currently, we know that expanding an atom yields constraints in conic form, but with recursive atoms, we would need to iteratively expand until we reach the base case.

We point out that this is only one possible approach to generalized powers to illustrate where recursive definitions might be useful, but other approaches are possible. In the case of powers in particular, the best approach is likely to develop a custom mechanism, perhaps at the meta-level.

7.4.2 Better support for binders

Both vehicle speed scheduling (Section 6.2) and fitting a sphere to data (Section 6.3) involved higher-order operations such as sums. The latter also had universally quantified constraints in its original formulation. We intentionally rewrote the problem into a binder-free version, hiding the binders behind definitions. For sums, we used `Vec.sum`, and constraints of the form `∀ i, a i ≤ b i` were re-phrased to `a ≤ b`, using the definition of inequality for vectors.

We can handle simple cases, such as when the objective function is of the form `∑ i, x i` and `x` is an optimization variable. When the canonization procedure traverses the sum expression, it sees the following (where we have omitted the additive commutative monoid instance).

```
Finset.sum ℝ (Fin n) _ Finset.univ (fun i ↦ x i)
```

The only non-constant argument that needs to be considered is `fun i ↦ x i`, which, by η-reduction, becomes `x`. In this case, it detects that it is an optimization variable and successfully makes the atom tree. However, consider the expression `∑ i, (x i) ^ 2`. In this case, `fun i ↦ (x i) ^ 2` is problematic, even though it is definitionally equal to `x ^ 2`, for which we have an atom. This makes the canonization procedure fail, claiming it could not find an atom that matches the expression. The root of the issue is that we rely on first-order pattern matching for finding atom candidates. One needs at least second-order pattern-matching (e.g., as introduced by Huet and Lang [HL78]) in order to handle binding structures.

Writing expressions with binders is common when modeling optimization problems. Therefore, improving the matching mechanism would be highly desirable, which might involve also designing a custom expression reduction method and changing how atoms are stored.

It would also be interesting to explore the possibility of a vectorization tactic that would automatically turn, say, `∑ i, (x i) ^ 2`, into `Vec.sum (x ^ 2)`.

There are also issues with binders at the pre-DCD level, which leads us to the next section.

7.4.3 Extending the e-graph language

Currently, the pre-DCP procedure discussed in Chapter 5 only supports real variables and operations. A natural next step would be to extend it to support vectors and matrices. The first step would be considering first-order operations on vector and matrix-valued expressions, i.e., avoiding binders. These include dot products, matrix multiplication, scalar multiplication, element-wise operations, etc. A soft typing layer would be needed to ensure the terms are well-formed. In particular, we would need to keep track of dimensions, which could be achieved by storing that information in the e-class analysis. In this way, we can check, for example, that both expressions in a dot product are vectors of the same dimension. A considerable number of equalities might be needed to encode all the properties of the additional operations, which might affect performance. However, they should all be definable. We speculate on how one such rule could look.

```
rw!("mat_add_mul";
    "(matMul (matAdd ?a ?b) ?c)" =>
    "(matAdd (matMul ?a ?c) (matMul ?b ?c))"
    if is_mat("?a") if is_mat("?b") if is_mat("?c")
    if mat_dim_eq("?a", "?b")
    if mat_snd_dim_eq_fst_dim("?a", "?c"))
```

Note that we are not re-using the existing addition and multiplication, which could be an option as long as we are careful with the conditions in other rules; for example, we would need to disable the commutativity of matrix multiplication. One could argue that conditions such as the ones above are unnecessary if the initial expression is well-formed, but they are a rather inexpensive sanity check, and we have added them to illustrate how this data could be accessed from the e-class analysis.

Apart from real, vector, and matrix-valued expressions, we would need to consider the type of indices separately to allow for vector and matrix accesses. The next question would then be whether any index arithmetic is allowed.

A more ambitious extension would be adding full support for binders. It is well-known that that is a challenge in the context of e-graphs.[1] The first

[1] A blog post by Dr. Pavel Pancheckha gives nice and detailed explanations, with examples, of the difficulties of binding in e-graphs: `https://pavpanchekha.com/blog/egg-bindings.html` (accessed 2024-02-12)

issue is that α-equivalence needs to be handled explicitly. The most common approach is using de Bruijn indices. Indeed, proof assistants, such as Lean, have the same issue and use de Bruijn indices for λ-terms. This machinery would need to be replicated on the `egg` side, which is already a considerable amount of work, notoriously because of the intricacies of substitution in β-reduction with de Bruijn indices. With this in place, it should be possible to define rewrite rules on higher-order terms. Making this work together with the DCP procedure depends, of course, on successfully fixing the issues discussed in the previous section.

Other new constructs could be useful besides operations on matrices and vectors. For example, the positive semidefinite relation. If binders are supported, then universal quantification could also be an option.

7.4.4 Numerical verification

As discussed, the interface with the solver remains unverified. We propose an approach to produce verified solution enclosures and optimality bounds. It is adapted from Lange's work [Lan20], which is implemented in `VSDP` [Jan06].

The first step would be to develop an interval arithmetic tactic in Lean, which is a sizeable project in itself. To do it properly, one would first need to formalize floating-point arithmetic to some extent, perhaps following Harrison's work in HOL-Light [Har97, Har99] or the Flocq library [BM11] in Coq. For the first version, however, it might be enough to use rational or dyadic numbers. The tactic should support basic arithmetic operations, perhaps based on Appendix C, and some linear algebra. In particular, we assume that it can compute an interval containing the solution of a linear system.

First, we discuss feasibility. Let P be the following problem in conic form[2]:

$$\begin{aligned} \text{minimize} \quad & c^T x \\ \text{subject to} \quad & Ax = b \\ & x \in \mathcal{K}. \end{aligned}$$

The steps of the algorithm are as follows:

1. Obtain an approximate solution $\tilde{x}$ of P from the solver. Fail if the solver cannot find one.

[2]This is the usual definition of conic form, but note that it is a special case of definition 2.2.11.

2. Use the interval arithmetic tactic to obtain an interval $\mathbf{x}$ that solves the linear system $Ax = b$, using $\tilde{x}$ as a starting point. Fail if no interval can be found.

3. Use the interval arithmetic tactic to check that $\mathbf{x} \subseteq \mathcal{K}$. If successful, return $\mathbf{x}$; otherwise, compute a violation measure v.

4. Compute a perturbation vector $\xi_v \in \mathcal{K}$ parametrized by v.

5. Solve P with $b := b - A\xi_v$ to obtain $\tilde{x}'$. Fail if the solver fails. Otherwise, set $\tilde{x} := \tilde{x}' + \xi_v$ and go to step 2.

If the algorithm succeeds, the interval $\mathbf{x}$ contains a feasible point.

How violation measures and perturbations would be computed is not obvious, and it would probably require considerable experimentation to develop adequate heuristics.

To establish optimality bounds, we consider the dual problem of P, which we call D, defined below.

$$\begin{aligned} &\text{maximize} && b^T y \\ &\text{subject to} && c - A^T y \in \mathcal{K}^*. \end{aligned}$$

where $\mathcal{K}^* = \{z \mid \forall x \in \mathcal{K}.\ z^T x \geq 0\}$ is the dual cone.

The following algorithm is a version of the primal case and a simplified version of the one presented by Lange. We proceed as follows:

1. Obtain an approximate solution $\tilde{y}$ of D from the solver. Fail if the solver cannot find one.

2. Use the interval arithmetic tactic to obtain an interval $\mathbf{z}$ that overapproximates $c - A^T\tilde{y}$.

3. Use the interval arithmetic tactic to check that $\mathbf{z} \subseteq \mathcal{K}^*$. If successful, return $\tilde{y}$; otherwise, compute a violation measure v.

4. Compute a perturbation vector $\xi_v \in \mathcal{K}$ parametrized by v.

5. Solve D with $c := c - \xi_v$ and obtain $\tilde{y}'$. Fail if the solver fails. Otherwise, set $\tilde{y} := \tilde{y}'$ and go to step 2.

If both algorithms succeed, $(\mathbf{x}, \{\tilde{y}\})$ contains a primal-dual feasible pair. Therefore, by weak duality, it follows that:

$$b^T\tilde{y} \leq \hat{\rho}_D \leq \hat{\rho}_P \leq \sup c^T\mathbf{x},$$

where $\hat{\rho}_P$ and $\hat{\rho}_D$ are the optimal values of the primal and dual problems, respectively. We have thus found solution enclosures and bounds for the objective values of both P and D.

7.4.5 Other projects

Proving convexity We never prove that problems are convex. Since our goal is to produce an equivalent (or reduced) problem that can be solved, strictly speaking, it is unnecessary. However, producing a proof using mathlib's definition of convexity would be interesting. This would happen at the DCP transformation step. We know that a problem that can be canonized is DCP convex and, therefore, convex. Every atom declaration would need to be extended with the appropriate proofs of curvature and monotonicity. For example, for logarithms, we could prove `ConcaveOn ℝ {x : ℝ | 0 < x} log`, using mathlib's definition of a concave function on a domain. For monotonicity, we would prove `MonotoneOn log {x : ℝ | 0 < x}` (recall that even though we used $\mathsf{monotonicity}_f$ in our presentation of atom declarations in Section 4.3, we do not actually prove it in the implementation). These proofs would then be combined using DCP composition and top-level rules, which have already been formalized in mathlib. We would, therefore, be able to establish, for DCP-compliant problems, that the feasible set is a convex set and that the objective function is convex.

Improved variable condition checking As argued in Section 4.5.2, the fact that mkAtomTrees (see Figure 4.3) and mkVcondElimMap (see Figure 4.5) are two separate passes is an issue. It disallows two atom declarations with the same expression but different variable conditions. We gave as an example the sq atom. It would be highly desirable to fix this issue in order to allow atom declarations in full generality. It would involve combining these two passes into one so that a different atom could be tried if its variable conditions could not be inferred.

Efficient problem data extraction The coefficient extraction mechanism discussed in Section 4.7.3 is computationally expensive. For example, suppose the

objective function is $c^T x$ where $x, c \in \mathbb{R}^{100}$, c is constant, and x is an optimization variable. Our current method would replace x with $\mathbf{e}_1, \ldots, \mathbf{e}_{100}$ (standard basis vectors) and compute the dot product 100 times to obtain the coefficients of the expression. It is clear, however, that the coefficients are just c. A much more refined approach is taken by `CVXPY`, and, in this scenario, they can immediately extract c. To handle large-scale optimization problems eventually, we would need to replicate their approach (perhaps by re-purposing their C++ backend, `cvxcore`) or find an alternative approach. One possibility is to use `SciLean`. All the expressions we need to extract coefficients from are affine. Hence, coefficient extraction (for the linear part) is the same as taking gradients with respect to the optimization variables, which `SciLean` supports. Another bottleneck is how array-like data structures are stored. In `CvxLean`, vectors are of type `Fin n → ℝ`, and they are usually built from successive applications of `Fin.cons`. Computing with these is much more expensive than using arrays. It would be interesting to explore ways to store both representations (`Fin n → ℝ` for reasoning, and `Array Float` for computing) as we build terms.

Other solvers A limitation of `CvxLean` is that it only supports `MOSEK` as a backend solver. Supporting other solvers, especially open-source ones, is an important next step that would allow us to reach a wider audience. We have two solvers in mind that would be suitable candidates: `ECOS` [DCB13], and `Clarabel`[3]. `ECOS` is a lightweight solver written in C, so we could use Lean's foreign function interface to call it. The main limitation is that it does not support semidefinite programs. `Clarabel` supports all the cones that we currently use in our `MOSEK` interface. It has a Rust interface, and since we already depend on Rust for the `pre_dcp` tactic, it would be easy to install. They each have their own input format (different from conic benchmark format shown in Section 4.7.3), so some work would be needed to adjust how the extracted data is used.

Maintainability improvements As discussed in Section 4.7.2.1, the meta-level code for the DCP transformation step can be brittle, and exploring alternatives to mitigate that would be interesting.

[3] `https://oxfordcontrol.github.io/ClarabelDocs/stable/` (accessed 2024-02-12).

7.5 Open questions

We use this section to report several interesting and open-ended questions arising from our research. In contrast to the previous section, we do not have concrete plans of action to tackle them. We hope that they can serve as a starting point for future research.

7.5.1 About pre-DCP transformations

Our e-graph-based system transforms optimization problems by either applying first-order rewrites to a component or mapping the objective function (by squaring it or by applying a logarithm). The first kind involves traditional arithmetic rewrites, but the second hints that other transformations might also be possible. For example, one can imagine using e-graphs to explore changes of variables. Note that these cannot be syntactic local rewrites, i.e., we cannot simply rewrite `(var "x")` into `(var (exp "u"))`, as it would break our curvature assumptions. Is it possible to exploit `egg`'s infrastructure for e-class analysis and dynamic rewrites to do this in a principled way?

Constraints are split into those used to determine the domain of the variables and those to be rewritten. Instead, we could consider a system with more interplay between constraints. All the constraints outside of the component being rewritten could be available in the context and usable for conditional rewrites. This could also allow for rewriting several constraints at once, which might open some opportunities for simplification.

It would also be interesting to explore other cost metrics. Currently, we are satisfied if we find a problem in DCP form. However, two equivalent problems in DCP form might result in different solving times. Therefore, searching for the problem with the best-expected solving time, in a similar spirit to program optimization in the compiler word, could be a fruitful research direction. Similarly, the expected accuracy of the solution could also be considered. How would this cost be estimated?

An even more ambitious project would be to also use e-graphs to find an equivalent problem in conic form. Is that realistic? This would be a drastically different approach to the systematic and efficient DCP algorithm. Nevertheless, the extra flexibility could potentially offer some advantages, such as finding a problem in conic form that can be solved faster or more accurately, as discussed

above.

7.5.2 About DCP transformations

How do we verify our DCP procedure? The soundness of our DCP procedure is established by type-checking the equivalence certificate it emits. However, there are no formal guarantees as to its completeness. Currently, it is built mostly of meta-level code that decomposes and combines Lean expressions as required. It would be possible, however, to define it as a mathematical function and prove statements about it. This would require extensive work formalizing the theory of DCP (for example, a formal definition of an atom would be needed, which we discussed in Section 7.4.1). Some parts of the procedure might still utilize metaprogramming facilities, such as storing and retrieving atoms from the library. That should not be a limitation as the statements could be phrased with some assumptions on the atom library. In other words, what we propose is making the DCP procedure reflective. The main advantage of this approach would be the possibility of avoiding type-checking the certificates, which is not a notable bottleneck at the moment, but it could be for large problems. This would likely be a highly challenging project because of the intricate canonization algorithm.

7.5.3 About solvers

Although this escapes the philosophy of our work, it would be interesting to develop numerical algorithms (e.g., interior-point methods) or parts of them within a proof assistant. This is different from the numerical verification method discussed in Section 7.4.4 and implemented in previous work [Har07, RVS18, HJL12]. These rely on a solver to output a certificate, which they then verify. Instead, we envision something closer to the verification of the ellipsoid method proposed by Cohen, Feron, and Garoche [CFG20], where they use Frama-C and Coq to verify their implementation. Why not write an interior-point conic solver algorithm directly in Lean? It would be interesting to study whether it is possible to do so efficiently and understand which properties can be verified.

7.6 Concluding thoughts

Proof assistants allow us to write code that we can *trust* to a high degree. Working with them still poses many challenges; mainly, there is a significant "expertise barrier". This barrier is most likely one of the main reasons why they are not used in many domains where they could be highly valuable. In this work, we have identified one such domain: convex optimization or, more precisely, problem transformations in convex optimization modeling software. `CvxLean` illustrates how complex yet widely used algorithms can be made fully rigorous by embedding them in a proof assistant. Our approach has been to develop a series of tactics and commands that ensure that every transformation step produces the appropriate correctness certificate. Along the way, we have kept in mind the importance of ergonomics, user-friendliness, and out-of-the-box automation, which are critical to ultimately reaching an audience beyond theorem-proving experts. We regard this thesis as an important step toward building formally verified and practical convex optimization modeling software.

Appendix A

Atom library overview

In this appendix, we list all the atoms that have currently been formalized in `CvxLean`. This serves several purposes. First, the reader can get a clearer picture of the range of problems that can be defined. It also shows how some unique features of our framework are reflected in the atom library. Finally, it is an excellent place to identify opportunities for further work. Many more atoms can be formalized[1], and some of the ones that have been formalized can be generalized.

For each atom, we give its mathematical expression (definition 4.3.7). We also show its domain (definition 4.3.3) together with the background and variable conditions (definitions 4.3.5 and 4.3.6) to simplify the presentation. For example, if the domain is $\mathbb{R}_+$, in reality it would be $\mathbb{R}$ with $\mathsf{vconds}(x) := x \geq 0$ (or $\mathsf{bconds}(x) := x \geq 0$ if the argument is a parameter). We ignore the "obvious" constant parameters, in particular n and m in atoms with vector or matrix input. The input kinds (definition 4.3.4) are also shown, with some new notation. Recall that the input kind tells us if an atom's formal argument is constant (C), increasing ($\nearrow$), decreasing ($\searrow$), or neither (?).

We have 10 classes of convex set atoms: 6 cone membership atoms shown in Table A.1; and 4 others shown in Table A.2, which, crucially, include "$=$" and "$\leq$". An interesting remark about cone membership atoms is that their input kind is always "?", which enforces the expression to be affine, as expected, following the curvature rules from Section 4.4.

Regarding function atoms, there are 39 classes in total. We list our 22 classes of affine atoms in Table A.3 and Table A.4. The former consists of elementary real

[1] See `https://www.cvxpy.org/tutorial/functions/index.html` for a list of atoms that one could target (accessed 2024-03-09).

operations, and the latter focuses specifically on vector and matrix operations. Table A.5 shows our 11 classes of convex function atoms, and Table A.6 our 6 classes of concave function atoms.

There are a couple of subtleties about convex atoms worth highlighting. First, note that there are instances of function with positive domain components (e.g., the domain for y in x^2/y) defined instead on $[10^{-6}, \infty)$. That is because of variable condition elimination (definition 4.3.16). The second-order cone constraint used to define this atom does not imply that y is positive. Our work-around is to require $y \geq 10^{-6}$ as a variable condition instead of $y > 0$ and add the constraint $(y - 10^{-6}) \in \mathbb{R}_+$ in the implementation. This, of course, makes variable condition elimination straightforward. The other observation is concerning the issue with condition checking discussed in Section 4.5.1. Looking at x^2 or $\|x\|_2$, for example, we see that nothing is claimed about their monotonicity. However, it is clear that x^2 is increasing on x when $x \geq 0$ and decreasing when $x \leq 0$. This is a limitation of our algorithm since making the atom tree does not take into account variable conditions, so we cannot have two atoms with the same expression but different variable conditions.

Expression	**Domain**	**Input kind**	**Comments**
$x \in 0$	$\mathbb{R}$	?	$\mathbb{R}^n$ and $\mathbb{R}^{n \times m}$ element-wise.
$x \in \mathbb{R}_+$	$\mathbb{R}$	?	$\mathbb{R}^n$ and $\mathbb{R}^{n \times m}$ element-wise.
$(t, s, r) \in \mathcal{K}_{\exp}$	$\mathbb{R} \times \mathbb{R} \times \mathbb{R}$	?	$\mathbb{R}^n \times \mathbb{R}^n \times \mathbb{R}^n$ element-wise.
$(t, x) \in \mathcal{Q}^{n+1}$	$\mathbb{R} \times \mathbb{R}^n$	? ?	$\mathbb{R}^m \times \mathbb{R}^{m \times n}$ element-wise.
$(u, v, x) \in \mathcal{Q}_r^{n+2}$	$\mathbb{R} \times \mathbb{R} \times \mathbb{R}^n$	? ? ?	$\mathbb{R}^m \times \mathbb{R}^m \times \mathbb{R}^{m \times n}$ element-wise.
$X \in \mathcal{S}_+^n$	$\mathbb{R}^{n \times n}$	?	

Table A.1: Cone membership atoms.

Expression	**Domain**	**Input kind**	**Comments**
$x = y$	$\mathbb{R} \times \mathbb{R}$	? ?	$\mathbb{R}^n$ and $\mathbb{R}^{n \times m}$ element-wise.
$x \leq y$	$\mathbb{R} \times \mathbb{R}$	$\searrow \nearrow$	$\mathbb{R}^n$ and $\mathbb{R}^{n \times m}$ element-wise.
$X \succeq 0$	$\mathbb{R}^{n \times n}$	?	
$\top$			For unconstrainted problems.

Table A.2: Other set atoms.

Expression	**Domain**	**Input kind**	**Comments**
$x + y$	$\mathbb{R} \times \mathbb{R}$	$\nearrow \nearrow$	$\mathbb{R}^n$ and $\mathbb{R}^{n \times m}$ element-wise.
$x - y$	$\mathbb{R} \times \mathbb{R}$	$\nearrow \searrow$	$\mathbb{R}^n$ and $\mathbb{R}^{n \times m}$ element-wise.
$-x$	$\mathbb{R}$	$\searrow$	$\mathbb{R}^n$ and $\mathbb{R}^{n \times m}$ element-wise.
$c \cdot x$	$\mathbb{R}_+ \times \mathbb{R}$	C $\nearrow$	Cases of multiplication by a constant.
$c \cdot x$	$\mathbb{R}_- \times \mathbb{R}$	C $\searrow$	
$x \cdot c$	$\mathbb{R} \times \mathbb{R}_+$	$\nearrow$ C	
$x \cdot c$	$\mathbb{R} \times \mathbb{R}_-$	$\searrow$ C	
x/c	$\mathbb{R} \times \mathbb{R}_+$	$\nearrow$ C	Similar idea.
x/c	$\mathbb{R} \times \mathbb{R}_-$	$\searrow$ C	
x^1	$\mathbb{R}$	$\nearrow$	Special case of power atom.

Table A.3: Real affine function atoms.

Expression	Domain	Input kind	Comments
$\left(\sum_{j=1}^{i} x_j\right)_{i=1,\ldots,n}$	$\mathbb{R}^n$	$\nearrow$	Cumulative sum.
$(X_{ii})_{i=1,\ldots,n}$	$\mathbb{R}^{n\times n}$	$\nearrow$	Diagonal of a matrix.
$\begin{pmatrix} x_1 & \cdots & 0 \\ 0 & \ddots & 0 \\ 0 & \cdots & x_n \end{pmatrix}$	$\mathbb{R}^n$	$\nearrow$	Make a diagonal matrix.
$\vec{x}\cdot\vec{c}$	$\mathbb{R}^n \times \mathbb{R}^n$	? C	Dot product.
$\vec{c}\cdot\vec{x}$	$\mathbb{R}^n \times \mathbb{R}^n$	C ?	
$\begin{pmatrix} X_{11} & X_{12} \\ X_{21} & X_{22} \end{pmatrix}$	$\mathbb{R}^{n\times n}(\times 4)$	$\nearrow\ \nearrow\ \nearrow\ \nearrow$	Block matrix.
$(c)_{i=1,\ldots,n}$	$\mathbb{N}\times\mathbb{R}$	C C	Constant vector.
$(c)_{i=1,\ldots,n;j=1,\ldots,m}$	$\mathbb{N}\times\mathbb{N}\times\mathbb{R}$	C C C	Constant matrix.
$X\cdot C$	$\mathbb{R}^{n\times n}\times\mathbb{R}^{n\times n}$	? C	Matrix multiplication.
$C\cdot X$	$\mathbb{R}^{n\times n}\times\mathbb{R}^{n\times n}$	C ?	
$C\vec{x}$	$\mathbb{R}^{n\times m}\times\mathbb{R}^n$	C ?	Matrix by vector.
$\vec{x}[i]$	$\mathbb{R}^n\times\mathbb{N}$	? C	Access with index.
$X[i]$	$\mathbb{R}^{n\times n}\times\mathbb{N}$	? C	
$X[i,j]$	$\mathbb{R}^{n\times n}\times\mathbb{N}\times\mathbb{N}$	? C C	
$c\vec{x}$	$\mathbb{R}_+\times\mathbb{R}^n$	C $\nearrow$	Scalar multiplication.
$c\vec{x}$	$\mathbb{R}_-\times\mathbb{R}^n$	C $\searrow$	
$\sum_{i=1}^{n} x_i$	$\mathbb{R}^n$	$\nearrow$	Summation.
$\sum_{i=1}^{n}\sum_{j=1}^{n} X_{ij}$	$\mathbb{R}^n$	$\nearrow$	
$\begin{pmatrix} X_{11} & \cdots & X_{1n} \\ 0 & \ddots & \vdots \\ 0 & \cdots & X_{nn} \end{pmatrix}$	$\mathbb{R}^{n\times n}$	$\nearrow$	Upper triangular matrix.
$\mathrm{Tr}(X)$	$\mathbb{R}^{n\times n}$	$\nearrow$	Matrix trace.
X^T	$\mathbb{R}^{n\times m}$	$\nearrow$	Matrix transpose.
$\begin{pmatrix} x \\ \vec{y} \end{pmatrix}$	$\mathbb{R}\times\mathbb{R}^n$	$\nearrow\ \nearrow$	Vector and matrix "cons".
$\begin{pmatrix} \vec{x} \\ Y \end{pmatrix}$	$\mathbb{R}^n\times\mathbb{R}^{m\times n}$	$\nearrow\ \nearrow$	
$(\vec{x})$	$\mathbb{R}^n$	?	Make an $n\times 1$ matrix.

Table A.4: Affine function atoms on vectors and matrices.

Expression	**Domain**	**Input kind**	**Comments**
$\lvert x\rvert$	$\mathbb{R}$	$\nearrow$	$\mathbb{R}^n$ and $\mathbb{R}^{n\times m}$ element-wise.
$\exp(x)$	$\mathbb{R}$	$\nearrow$	$\mathbb{R}^n$ and $\mathbb{R}^{n\times m}$ element-wise.
$h(x)$	$\mathbb{R}$	?	Huber function (see 4.6.3). $\mathbb{R}^n$ element-wise.
$1/x$	$\mathbb{R}_{++}$	$\searrow$	$\mathbb{R}^n$ element-wise.
$x\cdot\log(x/y)-x+y$	$\mathbb{R}_+\times\mathbb{R}_{\geq\varepsilon}$	? ?	$\mathbb{R}^n$ element-wise. $\varepsilon=10^{-6}$.
$\log\left(\sum_{i=0}^{n-1}\exp(x_i)\right)$	$\mathbb{R}^{n>0}$	$\nearrow$	Note that $n>0$.
$\max(x,y)$	$\mathbb{R}\times\mathbb{R}$	$\nearrow\ \nearrow$	
$\lVert\vec{x}\rVert_2$	$\mathbb{R}^n$	?	$\mathbb{R}^{n\times m}$ element-wise.
x^2	$\mathbb{R}$	?	$\mathbb{R}^n$ element-wise.
x^{-1}	$\mathbb{R}_{++}$	$\searrow$	
x^{-2}	$\mathbb{R}_{++}$	$\searrow$	
x^2/y	$\mathbb{R}\times\mathbb{R}_{\geq\varepsilon}$	? $\searrow$	$\mathbb{R}^n$ element-wise. $\varepsilon=10^{-6}$.
$x\cdot\exp(x)$	$\mathbb{R}_+$	?	

Table A.5: Convex function atoms.

Expression	**Domain**	**Input kind**	**Comments**
$-x\cdot\log(x)$	$\mathbb{R}_+$	?	$\mathbb{R}^n$ element-wise.
$\sqrt{x\cdot y}$	$\mathbb{R}_+\times\mathbb{R}_+$	$\nearrow\ \nearrow$	
$\log(x)$	$\mathbb{R}_{++}$	$\nearrow$	$\mathbb{R}^n$ element-wise.
$\log(\det(X))$	$\mathcal{S}^n_{++}$	?	
$\min(x,y)$	$\mathbb{R}\times\mathbb{R}$	$\nearrow\ \nearrow$	
$\sqrt{x}$	$\mathbb{R}_+$	$\nearrow$	

Table A.6: Concave function atoms.

Appendix B

Canonized expressions bound original expressions

Lemma B.1 (Canonized expressions bound original expressions)**.** Fix $\vec{x} : D$ and $\vec{v} : E$, and assume $ds(\vec{x}, \vec{v})$. Then, for a node n in an atom tree resulting from canonize, we have that

$$\mathsf{canonExpr}_n(\vec{x}, \vec{v}) \;\square\; \mathsf{ogExpr}_n(\vec{x}),$$

where

$$a \;\square\; b := \begin{cases} a = b & \text{if } \mathsf{compCurv}(n) \sqsubseteq \textsc{AffineFn} \\ a \geq b & \text{if } \mathsf{compCurv}(n) = \textsc{ConvexFn} \\ a \leq b & \text{if } \mathsf{compCurv}(n) = \textsc{ConcaveFn} \\ a \Leftrightarrow b & \text{if } \mathsf{compCurv}(n) = \textsc{AffineSet} \\ a \Rightarrow b & \text{if } \mathsf{compCurv}(n) = \textsc{ConvexSet}. \end{cases}$$

Recall that compCurv matches the curvature labels computed in mkAtomTree (see Figure 4.2). It is important to note that this does not always equal curv_f, in particular when an affine atom is at the root of a convex or concave expression.

Proof. We proceed by induction on an atom tree rooted at n, as usual. The base case is trivial, as leaves are not canonized. For the inductive case, let $c_1, \ldots, c_k$ be the children of n, with atom f. Define the vector of canonized expressions and the vector of original expressions as follows:

$$\vec{r} := (\mathsf{canonExpr}_{c_1}(\vec{x}, \vec{v}), \ldots, \mathsf{canonExpr}_{c_k}(\vec{x}, \vec{v}))$$
$$\vec{o} := (\mathsf{ogExpr}_{c_1}(\vec{x}), \ldots, \mathsf{ogExpr}_{c_k}(\vec{x}))$$

Both take values in D_f, the domain of atom f. A preliminary result is that since we have assumed $ds(\vec{x},\vec{v})$, we can argue as in Lemma 4.6.3 to see that $\mathsf{impConstrs}\left(\vec{r}, \pi_{\mathsf{impVars}_n^*}(\vec{v})\right)$ holds. We refer to this fact as $(\star)$.

For any $i \in \{1,\ldots,k\}$, we know $\mathsf{canonExpr}_{c_i}(\vec{x},\vec{v}) \sqsubset \mathsf{ogExpr}_{c_i}(\vec{x})$ by the inductive hypothesis, which we call (IH_i).

We need to consider different cases depending on the computed curvature at n and the input kinds of f. Depending on the curvature, we prove $\vec{r} \triangle \vec{o}$, $\vec{o} \triangle \vec{r}$ or $\vec{r} = \vec{o}$, where "$\triangle$" is as in definition 4.3.14. Then, we will use the monotonicity and bounding properties as needed. We proceed as follows.

Affine function case. If n is labeled with AffineFn, the goal is to show

$$\mathsf{canonExpr}_n(\vec{x},\vec{v}) = \mathsf{ogExpr}_n(\vec{x}).$$

What this says is that canonization does not alter affine expressions, as expected. It must be the case that f is an affine atom. We show $\vec{r} = \vec{o}$. By curvRule, we can see that all c_i are labeled with curvature $\sqsubseteq$ AffineFn. Therefore, $\vec{r} = \vec{o}$ follows from the inductive hypothesis.

By bounds_f (definition 4.3.15) together with $(\star)$, we have that

$$\mathsf{impObj}_f\left(\vec{r}, \pi_{\mathsf{impVars}_n^*}(\vec{v})\right) = \mathsf{expr}_f(\vec{o}).$$

The left-hand side is $\mathsf{canonExpr}_n(\vec{x},\vec{v})$ and the right-hand side is $\mathsf{ogExpr}_n(\vec{x})$ so we are done.

Convex function case. If n is labeled with ConvexFn, the goal is to show

$$\mathsf{canonExpr}_n(\vec{x},\vec{v}) \geq \mathsf{ogExpr}_n(\vec{x}).$$

In this case, f can be a convex atom, or an affine atom in the role of convex. We begin by showing $\vec{r} \triangle \vec{o}$. Pick any $i \in \{1,\ldots,k\}$. Consider the following cases:

- Suppose that $\mathsf{inputKind}_f(i) =$ Increasing. We must show that $r_i \geq o_i$. By curvRule, we know that c_i is labeled with curvature $\sqsubseteq$ ConvexFn. Hence, it follows from (IH_i) that

$$\mathsf{canonExpr}_{c_i}(\vec{x},\vec{v}) \geq \mathsf{ogExpr}_{c_i}(\vec{x}).$$

- Suppose that $\mathsf{inputKind}_f(i) = \textsc{Decreasing}$. We must show that $r_i \leq o_i$. By $\mathsf{curvRule}$, we know that c_i is labeled with curvature $\sqsubseteq \textsc{ConcaveFn}$. Hence, it follows from (IH_i) that

$$\mathsf{canonExpr}_{c_i}(\vec{x}, \vec{v}) \leq \mathsf{ogExpr}_{c_i}(\vec{x}).$$

- Otherwise, we need to show that $r_i = o_i$. Once again, $\mathsf{curvRule}$ will give us what we need. In this case, it tells us that c_i is labeled with curvature $\sqsubseteq \textsc{AffineFn}$. Thus, it follows from (IH_i) that

$$\mathsf{canonExpr}_{c_i}(\vec{x}, \vec{v}) = \mathsf{ogExpr}_{c_i}(\vec{x}).$$

The conditions hold for both $\vec{r}$ and $\vec{o}$ as established by Lemmas 4.6.3 and 4.6.2. Thus, we can apply $\mathsf{monotonicity}_f$ (definition 4.3.14) to to $\vec{r} \triangle \vec{o}$ and obtain

$$\mathsf{expr}_f(\vec{r}) \geq \mathsf{expr}_f(\vec{o}).$$

We use bounds_f with ($\star$) to deduce $\mathsf{boundsCond}_f\left(\vec{r}, \pi_{\mathsf{impVars}_n^*}(\vec{v})\right)$, which, since f is convex or affine, implies:

$$\mathsf{expr}_f(\vec{r}) \leq \mathsf{impObj}_f\left(\vec{r}, \pi_{\mathsf{impVars}_n^*}(\vec{v})\right).$$

By transitivity, we have:

$$\mathsf{impObj}_f\left(\vec{r}, \pi_{\mathsf{impVars}_n^*}(\vec{v})\right) \geq \mathsf{expr}_f(\vec{o}).$$

The left-hand side is $\mathsf{canonExpr}_n(\vec{x}, \vec{v})$ and the right-hand side is $\mathsf{ogExpr}_n(\vec{x})$ so we are done.

Concave function case. If n is labeled with $\textsc{ConcaveFn}$, the goal is to show

$$\mathsf{canonExpr}_n(\vec{x}, \vec{v}) \leq \mathsf{ogExpr}_n(\vec{x}).$$

In this case, f can be a concave atom, or an affine atom in the role of concave. We begin by showing $\vec{o} \triangle \vec{r}$. Pick any $i \in \{1, \ldots, k\}$. Consider the following cases:

- Suppose that $\mathsf{inputKind}_f(i) = \textsc{Increasing}$. We must show that $o_i \geq r_i$. By $\mathsf{curvRule}$, we know that c_i is labeled with curvature $\sqsubseteq \textsc{ConcaveFn}$. Hence, it follows from (IH_i) that

$$\mathsf{ogExpr}_{c_i}(\vec{x}) \geq \mathsf{canonExpr}_{c_i}(\vec{x}, \vec{v}).$$

- Suppose that $\mathsf{inputKind}_f(i) = \textsc{Decreasing}$. We must show that $o_i \leq r_i$. By $\mathsf{curvRule}$, we know that c_i is labeled with curvature $\sqsubseteq \textsc{ConvexFn}$. Hence, it follows from (IH_i) that

$$\mathsf{ogExpr}_{c_i}(\vec{x}) \leq \mathsf{canonExpr}_{c_i}(\vec{x}, \vec{v}).$$

- Otherwise, we need to show that $o_i = r_i$. Once again, $\mathsf{curvRule}$ will give us what we need. In this case, it tells us that c_i is labeled with curvature $\sqsubseteq \textsc{AffineFn}$. Thus, it follows from (IH_i) that

$$\mathsf{ogExpr}_{c_i}(\vec{x}) = \mathsf{canonExpr}_{c_i}(\vec{x}, \vec{v}).$$

The conditions hold for both $\vec{o}$ and $\vec{r}$ as established by Lemmas 4.6.2 and 4.6.3. Thus, we can apply $\mathsf{monotonicity}_f$ to to $\vec{o} \triangle \vec{r}$ and obtain

$$\mathsf{expr}_f(\vec{o}) \geq \mathsf{expr}_f(\vec{r}).$$

We use bounds_f with ($\star$) to deduce $\mathsf{boundsCond}_f\left(\vec{r}, \pi_{\mathsf{impVars}_n^*}(\vec{v})\right)$, which, since f is concave or affine, implies:

$$\mathsf{expr}_f(\vec{r}) \geq \mathsf{impObj}_f\left(\vec{r}, \pi_{\mathsf{impVars}_n^*}(\vec{v})\right).$$

By transitivity, we have:

$$\mathsf{impObj}_f\left(\vec{r}, \pi_{\mathsf{impVars}_n^*}(\vec{v})\right) \leq \mathsf{expr}_f(\vec{o}).$$

The left-hand side is $\mathsf{canonExpr}_n(\vec{x}, \vec{v})$ and the right-hand side is $\mathsf{ogExpr}_n(\vec{x})$ so we are done.

Affine set case. If n is labeled with $\textsc{AffineSet}$, the goal is to show

$$\mathsf{canonExpr}_n(\vec{x}, \vec{v}) \Leftrightarrow \mathsf{ogExpr}_n(\vec{x}).$$

It must be the case that f has $\mathsf{curv}_f = \textsc{AffineSet}$. Similarly to the affine function case, $\mathsf{curvRule}$ enforces all c_i to be labeled with curvature $\sqsubseteq \textsc{AffineFn}$. Therefore, $\vec{r} = \vec{o}$ follows by the inductive hypothesis.

By bounds_f together with ($\star$), we have that

$$\mathsf{impObj}_f\left(\vec{r}, \pi_{\mathsf{impVars}_n^*}(\vec{v})\right) \Leftrightarrow \mathsf{expr}_f(\vec{o}).$$

The left-hand side is $\mathsf{canonExpr}_n(\vec{x}, \vec{v})$ and the right-hand side is $\mathsf{ogExpr}_n(\vec{x})$ so we are done.

Convex set case. If n is labeled with ConvexSet, the goal is to show

$$\mathsf{canonExpr}_n(\vec{x}, \vec{v}) \Rightarrow \mathsf{ogExpr}_n(\vec{x}).$$

In this case, f is a convex set atom (note that affine set atoms are not analogous to affine function atoms; as seen from Section 4.4, they cannot take the role of convex set since we enforce both sides to be affine). We begin by showing $\vec{o} \triangle \vec{r}$. Pick any $i \in \{1, \ldots, k\}$. Consider the following cases:

- Suppose that $\mathsf{inputKind}_f(i) = \text{Increasing}$. We must show that $o_i \geq i_i$. By $\mathsf{curvRule}$, we know that c_i is labeled with curvature $\sqsubseteq$ ConcaveFn. Hence, it follows from (IH_i) that

$$\mathsf{ogExpr}_{c_i}(\vec{x}) \geq \mathsf{canonExpr}_{c_i}(\vec{x}, \vec{v}).$$

- Suppose that $\mathsf{inputKind}_f(i) = \text{Decreasing}$. We must show that $o_i \leq r_i$. By $\mathsf{curvRule}$, we know that c_i is labeled with curvature $\sqsubseteq$ ConvexFn. Hence, it follows from (IH_i) that

$$\mathsf{ogExpr}_{c_i}(\vec{x}) \leq \mathsf{canonExpr}_{c_i}(\vec{x}, \vec{v}).$$

- Otherwise, we need to show that $o_i = i_i$. Once again, $\mathsf{curvRule}$ will give us what we need. In this case, it tells us that c_i is labeled with curvature $\sqsubseteq$ AffineFn. Thus, it follows from (IH_i) that

$$\mathsf{ogExpr}_{c_i}(\vec{x}) = \mathsf{canonExpr}_{c_i}(\vec{x}, \vec{v}).$$

The conditions hold for both $\vec{o}$ and $\vec{r}$ as established by Lemmas 4.6.2 and 4.6.3. Thus, we can apply $\mathsf{monotonicity}_f$ to to $\vec{o} \triangle \vec{r}$ and obtain

$$\mathsf{expr}_f(\vec{r}) \Rightarrow \mathsf{expr}_f(\vec{o}).$$

We use bounds_f with ($\star$) to deduce $\mathsf{boundsCond}_f\left(\vec{r}, \pi_{\mathsf{impVars}_n^*}(\vec{v})\right)$, which, since f is a set atom, we have:

$$\mathsf{expr}_f(\vec{r}) \Leftrightarrow \mathsf{impObj}_f\left(\vec{r}, \pi_{\mathsf{impVars}_n^*}(\vec{v})\right).$$

Combining these two facts, we have:

$$\mathsf{impObj}_f\left(\vec{r}, \pi_{\mathsf{impVars}_n^*}(\vec{v})\right) \Rightarrow \mathsf{expr}_f(\vec{o}).$$

The left-hand side is $\mathsf{canonExpr}_n(\vec{x}, \vec{v})$ and the right-hand side is $\mathsf{ogExpr}_n(\vec{x})$ so we are done. □

Appendix C

An extension of interval arithmetic with open intervals

The standard treatment of interval arithmetic only considers closed intervals. However, this is insufficient to perform positivity checks in some relevant cases. Imagine that some expression real-valued r is positive, i.e., $r > 0$. The rewrite system described in Chapter 5 relies on annotating r with its domain (see Section 5.3.5). This information is then used to check whether, for instance, $1/r \leq 1$ can be rewritten into $1 \leq r$. We claim that, in order to handle many interesting examples in optimization, we cannot rely solely on closed intervals. To see that, let us consider some potential approaches and their shortcomings:

1. The simplest approach is to treat $r > 0$ as $r \in [0, \infty)$. We can rule out this approach as any rules requiring positivity will not apply. One might think that this situation will never occur as strict inequalities are not allowed in standard form. However, we may have $r := e^x$. Moreover, in `CvxLean`, strict inequalities are allowed (as long as they can be eliminated, as discussed in definition 4.3.16).

2. We may extend our annotations with a flag that tells us if zero is contained in the interval or not. Note that this already requires extending every operation. Moreover, we would not be able to detect that $\log(r + 1)$ is positive, as all we would have is $r + 1 \in [1, \infty)$ with the zero flag set to false. This example also shows that tagging the domain with sign information would not work as some functions, such as log, do not cross the x axis at the origin. It would also fail in cases where we know $r > 1$ and need to

detect that $r-1$ is positive.

3. Since all the computations are performed with floating-point numbers, we could use the machine epsilon ϵ for the given precision (the smallest representable number) and have $r \in [\epsilon, \infty)$. One problem with this approach is that the best approximation for the domain of r^2 would be $[0, \infty)$, so we would not be able to detect that it is indeed positive.

4. Instead of ϵ, one can choose a sufficiently small floating-point number $s > 0$ that does not vanish immediately when applying the operations we are interested in. The first observation is that we would still not be able to detect that r^n is positive for some large n. More importantly, this approach is unsound as $r - 3s/4$ would be tagged as positive, but r could theoretically be $s/2$.

We assume that r depends on free variables whose domain is given. Even though r is, theoretically, an expression in $\mathbb{R}$, in order to compute a concrete interval of its domain, we need to consider floating-point approximations of both the domains and the operations involved in the expression. In particular, we use multiple-precision floating-point numbers (see appendix C.1), often used in rigorous computing.

The theory developed in this appendix is designed so that already existing interval arithmetic libraries can be used directly, only extending the interval type with two extra bits indicating the openness of each endpoint. A similar approach is implemented and verified in Passmore's thesis [Pas11, Chapter 6.2] to decide the satisfiability of non-linear constraints. It builds on the arithmetic presented in Hickey, Ju, and van Enden [HJvE01], and we do the same here. We extend it mainly by defining the power operation. We also make some more straightforward additions, including the exponential, logarithm, absolute value, minimum and maximum functions. The notation is our own and refined from previous work to be more compact while maintaining the usual shape of conventional intervals.

C.1 IEEE-754 floating-point numbers

Let $\mathbb{F}$ be the set of all double-precision (64-bit) floating-point numbers represented following the IEEE-754 standard. In particular, although we will work with fixed precision, we will follow the GNU Multiple Precision Floating-Point Reliable

Library (GNU MPFR) library version of the IEEE-754 standard. The semantics are machine-independent and, in some ways, simpler as there are no subnormal numbers and there is a single NaN.

We can think of $\mathbb{F}$ as a finite subset of $\mathbb{R}_{ext} := \mathbb{R} \cup \{-0, -\infty, \infty, \mathrm{NaN}\}$, where $-0, -\infty, \infty, \mathrm{NaN}$ are symbols identifying special constants. Each element in $\mathbb{F}$ is represented by 1 sign bit $s \in \{0,1\}$, 11 exponent bits encoding $e \in \{0, \ldots, 2047\}$, and 52 fraction bits $b_1, \ldots, b_{52} \in \{0,1\}$. The fraction bits are interpreted as follows:

$$f := 1 + \sum_{i=1}^{52} b_{52-i} 2^{-i}.$$

The value in $\mathbb{F}$ of the bit representation follows the rules below:

$$\mathrm{val}(s,e,f) := \begin{cases} (-1)^s \cdot f \cdot 2^{e-1023} & \text{if } 0 < e < 2047 \\ 0 & \text{if } s = 0 \wedge e = 0 \\ -0 & \text{if } s = 1 \wedge e = 0 \\ \infty & \text{if } s = 0 \wedge e = 2047 \wedge \vec{b} = \vec{0} \\ -\infty & \text{if } s = 1 \wedge e = 2047 \wedge \vec{b} = \vec{0} \\ \mathrm{NaN} & \text{if } e = 2047 \wedge \vec{b} \neq \vec{0}. \end{cases}$$

The first case corresponds to the non-zero finite numbers. The remaining cases define the representation of signed zeros, infinities, and not-a-number (NaN).

The rules for comparing non-NaN elements in $\mathbb{F}$ correspond to the rules for $\mathbb{R}$ in finite cases, where the real value of -0 is 0. Infinities follow the expected rules with $-\infty < x$ for all $x \neq -\infty$ and $x < \infty$ for $x \neq \infty$. All comparisons involving a NaN return false, but these will be carefully avoided.

There are four rounding modes, but we only consider rounding towards ∞ and $-\infty$, as these are the only modes used in our implementation of interval arithmetic. A way to specify the semantics of the rounding modes is to consider their effect on $\mathbb{R}_{ext}$. To that end, we define $\mathrm{rnd}_{\downarrow}, \mathrm{rnd}_{\uparrow} : \mathbb{R}_{ext} \to \mathbb{F}$ as follows:

$$\mathrm{rnd}_{\downarrow}(x) := \begin{cases} \max\{y \in \mathbb{F} \mid y \leq x\} & \text{if } x \in \mathbb{R} \\ x & \text{otherwise,} \end{cases}$$

$$\mathrm{rnd}_{\uparrow}(x) := \begin{cases} \min\{y \in \mathbb{F} \mid x \leq y\} & \text{if } x \in \mathbb{R} \\ x & \text{otherwise.} \end{cases}$$

We take the minimum and maximum over finite linearly ordered subsets, so the operations are well-defined.

For a set of real-valued operations of interest, we have bounding functions in $\mathbb{F}$ that approximate the correct value. We illustrate it with an example. Consider the exponential function, appropriately extended to $\mathbb{R}_{ext}$, i.e. $\exp(-0) = 1$, $\exp(\infty) = \infty$, $\exp(-\infty) = 0$, and $\exp(\mathrm{NaN}) = \mathrm{NaN}$. There exist $\underline{\mathrm{fexp}}, \overline{\mathrm{fexp}} : \mathbb{F} \to \mathbb{F}$ such that $\underline{\mathrm{fexp}}(x) = \mathrm{rnd}_{\downarrow}(\exp(x))$ and $\overline{\mathrm{fexp}}(x) = \mathrm{rnd}_{\uparrow}(\exp(x))$. Note that the bounding functions may have undetermined values for functions where the domain is not all of $\mathbb{R}_{ext}$.

C.2 The extended interval set

We restrict our definitions to $\mathbb{F}$, as defined in the previous section.

Definition C.2.1 (Interval sets)**.** The following are sets of real intervals with every combination of openness, closeness, and unboundedness at their endpoints.

- Closed intervals:

$$\mathbb{IF}_{cc} := \{[a,b] \mid a,b \in \mathbb{F}, -\infty < a \leq b < \infty\},$$

  where $[a,b] := \{x \in \mathbb{F} \mid a \leq x \leq b\}$.

- Left-open right-closed intervals:

$$\mathbb{IF}_{oc} := \{(a,b] \mid a,b \in \mathbb{F}, a \leq b < \infty\},$$

  where $(a,b] := \{x \in \mathbb{F} \mid a < x \leq b\}$.

- Left-closed right-open intervals:

$$\mathbb{IF}_{co} := \{[a,b) \mid a,b \in \mathbb{F}, -\infty < a \leq b\},$$

  where $[a,b) := \{x \in \mathbb{F} \mid a \leq x < b\}$.

- Open intervals:

$$\mathbb{IF}_{oo} := \{(a,b) \mid a,b \in \mathbb{F}, a \leq b\},$$

  where $(a,b) := \{x \in \mathbb{F} \mid a < x < b\}$.

Note that left-open endpoints can be $-\infty$, and right-open endpoints can be ∞. It is also important to observe that some intervals can be empty, if the endpoints have the same value and at least one of them is open.

Definition C.2.2 (Extended interval set)**.** We define the set of all intervals as follows:

$$\mathbb{EIF} := \mathbb{IF}_{cc} \cup \mathbb{IF}_{oc} \cup \mathbb{IF}_{co} \cup \mathbb{IF}_{oo}$$

For $I \in \mathbb{EIF}$, it is easy to see that $-\infty, \infty, \mathrm{NaN} \notin I$. Furthermore, since -0 and 0 have the same numerical value, I can be seen as a subset of $\mathbb{R}$.

We use the notation $\lfloor a, b \rceil \in \mathbb{EIF}$ to indicate that each endpoint can be closed or open. We may also write $\lfloor a, b \rceil \in \mathbb{IF}_{lr}$ or ${}_l\lfloor a, b \rceil_r$ to indicate that

$$\lfloor a \text{ is } \begin{cases} [a & \text{if } l = c \\ (a & \text{if } l = o \end{cases} \quad \text{and} \quad b\rceil \text{ is } \begin{cases} b] & \text{if } r = c \\ b) & \text{if } r = o \end{cases}.$$

This will simplify our definitions and avoid cumbersome case-splitting.

Definition C.2.3 (Subset relation in $\mathbb{EIF}$)**.** Given two intervals $I_1 := {}_{l_1}\lfloor a_1, b_1 \rceil_{r_1}$ and $I_2 := {}_{l_2}\lfloor a_2, b_2 \rceil_{r_2}$, we have

$$I_1 \subseteq I_2 := \begin{cases} a_2 < a_1 & \text{if } l_1 = c \wedge l_2 = o \\ a_2 \leq a_1 & \text{otherwise} \end{cases} \wedge \begin{cases} b_1 < b_2 & \text{if } r_1 = o \wedge r_2 = c \\ b_1 \leq b_2 & \text{otherwise} \end{cases}.$$

In order to define many of the relevant operations, it is useful to define rules for openness. In particular, if an endpoint is computed from two input endpoints, its openness will depend on the openness of the openness of the input endpoints. This may also depend on the values themselves, but there are two common patterns that are captured by the notation below. For $p, q \in \{o, c\}$, define:

$$p \oplus q := \begin{cases} o & \text{if } p = o \vee q = o \\ c & \text{otherwise.} \end{cases} \qquad p \otimes q := \begin{cases} c & \text{if } p = c \vee q = c \\ o & \text{otherwise.} \end{cases}$$

The intuition for $\oplus$ is that open "beats" closed, and $\otimes$ says that closed "beats" open.

Definition C.2.4 (Union in $\mathbb{EIF}$)**.** Let I_1 and I_2 as in definition C.2.3.

$$I_1 \cup I_2 := {}_l\lfloor \min(a_1, a_2), \max(b_1, b_2) \rceil_r, \text{ where}$$

$$l := \begin{cases} l_1 & \text{if } a_1 < a_2 \\ l_2 & \text{if } a_2 < a_1 \\ l_1 \otimes l_2 & \text{otherwise,} \end{cases} \qquad r := \begin{cases} r_1 & \text{if } b_2 < b_1 \\ r_2 & \text{if } b_1 < b_2 \\ r_1 \otimes r_2 & \text{otherwise.} \end{cases}$$

Definition C.2.5 (Intersection in $\mathbb{EIF}$)**.** Let I_1 and I_2 as above.

$$I_1 \cap I_2 := {}_l\lfloor \max(a_1,a_2), \min(b_1,b_2) \rceil_r\,, \text{ where}$$

$$l := \begin{cases} l_1 & \text{if } a_2 < a_1 \\ l_2 & \text{if } a_1 < a_2 \\ l_1 \oplus l_2 & \text{otherwise,} \end{cases} \qquad r := \begin{cases} r_1 & \text{if } b_1 < b_2 \\ r_2 & \text{if } b_2 < b_1 \\ r_1 \oplus r_2 & \text{otherwise.} \end{cases}$$

C.3 Operations

In this section, we define how floating-point operations are lifted to $\mathbb{EIF}$. First, we define a map from $\mathbb{EIF}$ to subsets of $\mathbb{R}$, needed to state the requirements of the operations precisely, which we will call a *realization*. Let $\rho : \mathbb{EIF} \to \mathcal{P}(\mathbb{R})$ be defined as follows:

$$\rho({}_l\lfloor a,b \rceil_r) := \begin{cases} \{x \in \mathbb{R} \mid a \le x \le b\} & \text{if } l = c \wedge r = c \wedge a \neq -\infty \wedge b \neq \infty \\ \{x \in \mathbb{R} \mid a \le x < b\} & \text{if } l = c \wedge r = o \wedge a \neq -\infty \wedge b \neq \infty \\ \{x \in \mathbb{R} \mid a < x \le b\} & \text{if } l = o \wedge r = c \wedge a \neq -\infty \wedge b \neq \infty \\ \{x \in \mathbb{R} \mid a < x < b\} & \text{if } l = o \wedge r = o \wedge a \neq -\infty \wedge b \neq \infty \\ \{x \in \mathbb{R} \mid a \le x\} & \text{if } l = c \wedge r = o \wedge a \neq -\infty \wedge b = \infty \\ \{x \in \mathbb{R} \mid a < x\} & \text{if } l = o \wedge r = o \wedge a \neq -\infty \wedge b = \infty \\ \{x \in \mathbb{R} \mid x \le b\} & \text{if } l = o \wedge r = c \wedge a = -\infty \wedge b \neq \infty \\ \{x \in \mathbb{R} \mid x < b\} & \text{if } l = o \wedge r = o \wedge a = -\infty \wedge b \neq \infty \\ \mathbb{R} & \text{otherwise.} \end{cases}$$

In other words, $\rho(X)$ is the tightest real interval containing X. b We consider real-valued unary operations $u : X \to \mathbb{R}$ with $X \subseteq \mathbb{R}$. For each u, we define its lifted version $\widehat{u} : \mathbb{EIF} \to \mathbb{EIF}$ satisfying the property:[1]

$$\forall I \in \mathbb{EIF}.\ \rho(I) \subseteq X \Rightarrow \{u(x) \mid x \in \rho(I)\} \subseteq \rho(\widehat{u}(I))\,.$$

This essentially tells us that $\widehat{u}$ approximates u.

Similarly, we consider binary operations $b : X \times Y \to \mathbb{R}$ with $X, Y \subseteq \mathbb{R}$. In this case, the lifted $[\![b]\!] : \mathbb{EIF} \times \mathbb{EIF} \to \mathbb{EIF}$ must satisfy

$$\forall I_1, I_2 \in \mathbb{EIF}.\ \rho(I_1) \subseteq X \wedge \rho(I_2) \Rightarrow \{b(x,y) \mid x \in \rho(I_1), y \in \rho(I_2)\} \subseteq \rho\left(\widehat{b}(I_1,I_2)\right).$$

[1] Here, $\subseteq$ is defined on $\mathbb{R}$, not to be confused with Definition C.2.3.

Having laid out the requirements, we proceed by defining the lifted functions for each of the simple atoms in the language described in Section 5.3.2. We assume that none of the input intervals are empty, i.e., of the form (a,a), $(a,a]$, or $[a,a)$. For completeness, if that is the case, we can return $(-\infty,\infty)$.

Negation The floating-point operation fneg flips the sign bit and does not introduce any rounding error, so we do not need to split it into two bounding operations. The only extension to regular interval negation is that we need to flip the openness of the endpoints.

$$\widehat{\mathrm{neg}}\,({}_l\lfloor a,b\rceil_r) \;:=\; {}_r\lfloor \mathrm{fneg}(b),\mathrm{fneg}(a)\rceil_l\,.$$

Absolute value The floating-point operation fabs is also exact. We need to consider the following cases:

$$\widehat{\mathrm{abs}}\,({}_l\lfloor a,b\rceil_r) \;:=\; \begin{cases} {}_l\lfloor a,b\rceil_r & \text{if } a>0\wedge b>0\\ \widehat{\mathrm{neg}}\,({}_l\lfloor a,b\rceil_r) & \text{if } a<0\wedge b<0\\ [0,\mathrm{fabs}(b)\rceil_r & \text{if } \mathrm{fabs}(a)<\mathrm{fabs}(b)\\ [0,\mathrm{fabs}(a)\rceil_l & \text{if } \mathrm{fabs}(b)<\mathrm{fabs}(a)\\ [0,\mathrm{fabs}(a)\rceil_{l\otimes r} & \text{if } \mathrm{fabs}(a)=\mathrm{fabs}(b). \end{cases}$$

Square root The domain of the square root is the nonnegative interval. If the input is not nonnegative, we output the free domain. Note that the key property is still satisfied.

$$\widehat{\mathrm{sqrt}}\,({}_l\lfloor a,b\rceil_r) \;:=\; \begin{cases} {}_l\left\lfloor \underline{\mathrm{fsqrt}}(a),\overline{\mathrm{fsqrt}}(b)\right\rceil_r & \text{if } {}_l\lfloor a,b\rceil_r\subseteq[0,\infty)\\ (-\infty,\infty) & \text{otherwise.} \end{cases}$$

Logarithm Again, we need to take into account that logarithms are only defined in the positive domain.

$$\widehat{\log}\,({}_l\lfloor a,b\rceil_r) \;:=\; \begin{cases} {}_l\left\lfloor \underline{\mathrm{flog}}(a),\overline{\mathrm{flog}}(b)\right\rceil_r & \text{if } {}_l\lfloor a,b\rceil_r\subseteq(0,\infty)\\ (-\infty,\infty) & \text{otherwise.} \end{cases}$$

This is well-defined because $\underline{\mathrm{flog}}(0)=-\infty$.

Exponential In this case, we have a monotonically increasing function defined over all of $\mathbb{R}$, so the interval lift is straightforward.

$$\widehat{\exp}\,({}_l\lfloor a,b\rceil_r) \;:=\; {}_l\Big\lfloor \underline{\mathrm{fexp}}(a), \overline{\mathrm{fexp}}(b)\Big\rceil_r .$$

Note that the output is always strictly positive. To see that, consider the case when the left endpoint is unbounded. We have $\underline{\mathrm{fexp}}(-\infty) = 0$, and it is open as unbounded endpoints are open.

Addition We define the lift for addition as follows:

$$\widehat{\mathrm{add}}\left({}_{l_1}\lfloor a_1,b_1\rceil_{r_1}, {}_{l_2}\lfloor a_2,b_2\rceil_{r_2}\right) \;:=\; {}_{l_1\oplus l_1}\Big\lfloor \underline{\mathrm{fadd}}\,(a_1,a_2)\,, \overline{\mathrm{fadd}}\,(b_1,b_2)\Big\rceil_{r_1\oplus r_2} .$$

Subtraction Subtraction is simply a combination of addition and negation.

$$\widehat{\mathrm{sub}}\,(I_1,I_2) \;:=\; \widehat{\mathrm{add}}\,(I_1, \widehat{\mathrm{neg}}\,(I_2)) .$$

Multiplication Here, we need to consider different cases depending on the signs that appear in the interval. For $I \in \mathbb{EIF}$, define:

- $pos(I) := I \subseteq (0,\infty)$,
- $neg(I) := I \subseteq (-\infty,-0)$,
- $mix(I) := \neg(pos(I) \vee neg(I))$.

Note that the upper bound for negative numbers is -0. The rationale is that we want all negative numbers to be marked as negative in their floating-point representation. As we will see later, this is particularly important in division, as $\mathrm{fdiv}(1,-0) = -\infty$ whereas $\mathrm{fdiv}(1,0) = \infty$. Consider dividing the singleton interval containing 1 by the negative interval. The expected resulting interval is the negative interval. Following the rules of interval arithmetic, the left endpoint of the result is given by $\underline{\mathrm{fdiv}}(1,b)$, where b is the right endpoint of the negative interval. Thus, if we want the left endpoint of the result to be $-\infty$, we need $b = -0$.

Going back to multiplication, let $I_1 := {}_{l_1}\lfloor a_1, b_1 \rceil_{r_1}$ and $I_2 := {}_{l_2}\lfloor a_2, b_2 \rceil_{r_2}$.

$$
\widehat{\mathrm{mul}}\,(I_1, I_2) :=
$$

$$
\begin{cases}
{}_{l_1 \oplus l_2}\left\lfloor \underline{\mathrm{fmul}}\,(a_1, a_2), \overline{\mathrm{fmul}}\,(b_1, b_2) \right\rceil_{r_1 \oplus r_2} & \text{if } pos(I_1) \wedge pos(I_2) \\
{}_{r_1 \oplus l_2}\left\lfloor \underline{\mathrm{fmul}}\,(b_1, a_2), \overline{\mathrm{fmul}}\,(a_1, b_2) \right\rceil_{l_1 \oplus r_2} & \text{if } pos(I_1) \wedge neg(I_2) \\
{}_{r_1 \oplus r_2}\left\lfloor \underline{\mathrm{fmul}}\,(b_1, b_2), \overline{\mathrm{fmul}}\,(a_1, a_2) \right\rceil_{l_1 \oplus l_2} & \text{if } neg(I_1) \wedge neg(I_2) \\
{}_{l}\left\lfloor \underline{\mathrm{fmul}}\,(b_1, a_2), \overline{\mathrm{fmul}}\,(b_1, b_2) \right\rceil_{r}, \text{where} & \\
\quad l := \begin{cases} c & \text{if } a_2 = 0 \\ r_1 \oplus l_2 & \text{otherwise} \end{cases} & \text{if } pos(I_1) \wedge mix(I_2) \\
\quad r := \begin{cases} c & \text{if } b_2 = 0 \\ r_1 \oplus r_2 & \text{otherwise} \end{cases} & \\
{}_{l}\left\lfloor \underline{\mathrm{fmul}}\,(a_1, b_2), \overline{\mathrm{fmul}}\,(a_1, a_2) \right\rceil_{r}, \text{where} & \\
\quad l := \begin{cases} c & \text{if } b_2 = 0 \\ l_1 \oplus r_2 & \text{otherwise} \end{cases} & \text{if } neg(I_1) \wedge mix(I_2) \\
\quad r := \begin{cases} c & \text{if } a_2 = 0 \\ l_1 \oplus l_2 & \text{otherwise} \end{cases} & \\
U \cup V, \text{ where} & \\
\quad U := {}_{l_U}\left\lfloor \underline{\mathrm{fmul}}\,(b_1, a_2), \overline{\mathrm{fmul}}\,(b_1, b_2) \right\rceil_{r_U}, \text{where} & \\
\quad\quad l_U := \begin{cases} c & \text{if } a_2 = 0 \\ r_1 \oplus l_2 & \text{otherwise,} \end{cases} & \\
\quad\quad r_U := \begin{cases} c & \text{if } b_2 = 0 \\ r_1 \oplus r_2 & \text{otherwise,} \end{cases} & \text{if } mix(I_1) \wedge mix(I_2) \\
\quad V := {}_{l_V}\left\lfloor \underline{\mathrm{fmul}}\,(a_1, b_2), \overline{\mathrm{fmul}}\,(a_1, a_2) \right\rceil_{r_V}, \text{where} & \\
\quad\quad l_V := \begin{cases} c & \text{if } b_2 = 0 \\ l_1 \oplus r_2 & \text{otherwise} \end{cases} & \\
\quad\quad r_V := \begin{cases} c & \text{if } a_2 = 0 \\ l_1 \oplus l_2 & \text{otherwise} \end{cases} & \\
\widehat{\mathrm{mul}}\,(I_2, I_1) & \text{otherwise.}
\end{cases}
$$

To justify the correctness of the definition above, multiplication by zero requires special consideration, particularly multiplying zeros and infinities. Modulo

commutativity, the following equalities hold in our floating-point model:

- $\mathrm{fmul}(0, \infty) = 0$,
- $\mathrm{fmul}(0, -\infty) = -0$,
- $\mathrm{fmul}(-0, \infty) = -0$, and
- $\mathrm{fmul}(-0, -\infty) = 0$.

Division For I_1 and I_2 as above, division is well-defined when $pos(I_2)$ or $neg(I_2)$, which is exactly when $\{-0, 0\} \cap I_2 = \emptyset$.

$$
\widehat{\mathrm{div}}\,(I_1, I_2) :=
\begin{cases}
{\scriptstyle l_1 \oplus r_2}\left\lfloor \underline{\mathrm{fdiv}}\,(a_1, b_2), \overline{\mathrm{fdiv}}\,(b_1, a_2) \right\rceil_{\scriptstyle r_1 \oplus l_2} & \text{if } pos(I_1) \wedge pos(I_2) \\
{\scriptstyle r_1 \oplus r_2}\left\lfloor \underline{\mathrm{fdiv}}\,(b_1, b_2), \overline{\mathrm{fdiv}}\,(a_1, a_2) \right\rceil_{\scriptstyle l_1 \oplus l_2} & \text{if } pos(I_1) \wedge neg(I_2) \\
{\scriptstyle l_1 \oplus l_2}\left\lfloor \underline{\mathrm{fdiv}}\,(a_1, a_2), \overline{\mathrm{fdiv}}\,(b_1, b_2) \right\rceil_{\scriptstyle r_1 \oplus r_2} & \text{if } neg(I_1) \wedge pos(I_2) \\
{\scriptstyle r_1 \oplus l_2}\left\lfloor \underline{\mathrm{fdiv}}\,(b_1, a_2), \overline{\mathrm{fdiv}}\,(a_1, b_2) \right\rceil_{\scriptstyle l_1 \oplus r_2} & \text{if } neg(I_1) \wedge neg(I_2) \\
{\scriptstyle l}\left\lfloor \underline{\mathrm{fdiv}}\,(a_1, a_2), \overline{\mathrm{fdiv}}\,(b_1, a_2) \right\rceil_{\scriptstyle r}, \text{where} & \\
\quad l := \begin{cases} c & \text{if } a_1 = 0 \\ l_1 \oplus l_2 & \text{otherwise} \end{cases} & \text{if } mix(I_1) \wedge pos(I_2) \\
\quad r := \begin{cases} c & \text{if } b_1 = 0 \\ r_1 \oplus l_2 & \text{otherwise} \end{cases} & \\
{\scriptstyle l}\left\lfloor \underline{\mathrm{fdiv}}\,(b_1, b_2), \overline{\mathrm{fdiv}}\,(a_1, b_2) \right\rceil_{\scriptstyle r}, \text{where} & \\
\quad l := \begin{cases} c & \text{if } b_1 = 0 \\ r_1 \oplus r_2 & \text{otherwise} \end{cases} & \text{if } mix(I_1) \wedge neg(I_2) \\
\quad r := \begin{cases} c & \text{if } a_1 = 0 \\ l_1 \oplus r_2 & \text{otherwise} \end{cases} & \\
(-\infty, \infty) & \text{otherwise}
\end{cases}
$$

Similarly to multiplying by zero, special care is required when division involves infinities and zeros. Dividing a finite non-zero number by infinity is zero of the appropriate sign. Division by zero in fdiv is also possible. For finite numbers, when that happens, the result will be infinity with the appropriate sign. For infinities, we assume:

- $\mathrm{fdiv}(\infty, 0) = \infty$,
- $\mathrm{fdiv}(-\infty, 0) = -\infty$,
- $\mathrm{fdiv}(\infty, -0) = -\infty$, and
- $\mathrm{fdiv}(-\infty, -0) = \infty$.

Crucially, in the definition of lifted division, it is never the case that both arguments of fdiv are zero or that both arguments are infinities.

Inverse The inverse is easily defined from division as follows:

$$\widehat{\mathrm{inv}}\,({}_l\lfloor a, b\rceil_r) \;:=\; \widehat{\mathrm{div}}\,([1,1], {}_l\lfloor a, b\rceil_r)\,.$$

Minimum The operation fmin is exact, so we only need to adjust the openness of the endpoints.

$$\widehat{\mathrm{min}}\left({}_{l_1}\lfloor a_1, b_1\rceil_{r_1}, {}_{l_2}\lfloor a_2, b_2\rceil_{r_2}\right) \;:=\; {}_l\lfloor \mathrm{fmin}(a_1, a_2), \mathrm{fmin}(b_1, b_2)\rceil_r\,, \text{where}$$

$$l := \begin{cases} l_1 & \text{if } a_1 < a_2 \\ l_2 & \text{if } a_2 < a_1 \\ l_1 \otimes l_2 & \text{otherwise,} \end{cases} \qquad r := \begin{cases} r_1 & \text{if } b_1 < b_2 \\ r_2 & \text{if } b_2 < b_1 \\ r_1 \oplus r_2 & \text{otherwise.} \end{cases}$$

Maximum The logic here is dual to that of the operation above.

$$\widehat{\mathrm{max}}\left({}_{l_1}\lfloor a_1, b_1\rceil_{r_1}, {}_{l_2}\lfloor a_2, b_2\rceil_{r_2}\right) \;:=\; {}_l\lfloor \mathrm{fmax}(a_1, a_2), \mathrm{fmax}(b_1, b_2)\rceil_r\,, \text{where}$$

$$l := \begin{cases} l_2 & \text{if } a_1 < a_2 \\ l_1 & \text{if } a_2 < a_1 \\ l_1 \oplus l_2 & \text{otherwise,} \end{cases} \qquad r := \begin{cases} r_2 & \text{if } b_1 < b_2 \\ r_1 & \text{if } b_2 < b_1 \\ r_1 \otimes r_2 & \text{otherwise.} \end{cases}$$

Powers Finally, we define the lifted power operation. With I_1 and I_2 as earlier, the power operation is defined when $\neg neg(I_1)$ and for some special cases of I_2. If $I_1 \cap (-\infty, 0) \neq \emptyset$, we consider the following special cases:

- If $I_2 = [0,0]$ then $\widehat{\mathrm{pow}}\,(I_1, I_2) = [1,1]$.
- If $I_2 = [1,1]$ then $\widehat{\mathrm{pow}}\,(I_1, I_2) = I_1$.
- If $I_2 = [c,c]$ for $c \in \mathbb{N}$ and $2|c$ then $\widehat{\mathrm{pow}}\,(I_1, I_2) = {}_l\left\lfloor \underline{\mathrm{fpow}}\,(a', c), \overline{\mathrm{fpow}}\,(b', c)\right\rceil_r$, where ${}_l\lfloor a', b'\rceil_r = \widehat{\mathrm{abs}}\,(I_1)$.

Otherwise, a more refined classification of I_1 is required, as bases of powers smaller than 1 behave differently than bases larger than 1. For $I \in \mathbb{EIF}$ such that $I \subseteq [0,\infty)$ (or $\neg neg(I)$), define:

- $lrg(I) := (1,\infty) \subseteq I$.
- $sml(I) := (0,1) \subseteq I$.
- $unk(I) := \neg(lrg(I) \vee sml(I))$.

Just like with fmul and fdiv, we require some assumptions on fpow. In particular, we have that $\mathrm{fpow}(x,0) = \mathrm{fpow}(x,-0) = 1$ for all $x \in \mathbb{F} \cap \mathbb{R}$. Note that this includes the cases 0^0, $(-0)^0$, 0^{-0}, and $(-0)^{-0}$.

Firstly, as usual, we will default to $(-\infty,\infty)$ where the operation is not defined. We have chosen to not define $\widehat{\mathrm{pow}}(I_1,I_2)$ if $neg(I_1)$, so, if that is the case, we set $\widehat{\mathrm{pow}}(I_1,I_2) := (-\infty,\infty)$. Next, assume $\neg neg(I_1)$ and consider the following three cases.

If $lrg(I_1)$:

$$\widehat{\mathrm{pow}}\,(I_1,I_2) \;:=\; \begin{cases} {\scriptstyle l_1 \oplus l_2}\lfloor \underline{\mathrm{fpow}}\,(a_1,a_2), \overline{\mathrm{fpow}}\,(b_1,b_2) \rceil_{\scriptstyle r_1 \oplus r_2} & \text{if } pos(I_2) \\ {\scriptstyle r_1 \oplus l_2}\lfloor \underline{\mathrm{fpow}}\,(b_1,a_2), \overline{\mathrm{fpow}}\,(a_1,b_2) \rceil_{\scriptstyle l_1 \oplus r_2} & \text{if } neg(I_2) \\ {\scriptstyle l}\lfloor \underline{\mathrm{fpow}}\,(b_1,a_2), \overline{\mathrm{fpow}}\,(b_1,b_2) \rceil_{\scriptstyle r}, \text{where} & \\ \quad l := \begin{cases} c & \text{if } a_2 = 0 \\ r_1 \oplus l_2 & \text{otherwise,} \end{cases} \quad r := \begin{cases} c & \text{if } b_2 = 0 \\ r_1 \oplus r_2 & \text{otherwise} \end{cases} & \text{if } mix(I_2) \end{cases}$$

If $sml(I_1)$:

$$\widehat{\mathrm{pow}}\,(I_1,I_2) \;:=\; \begin{cases} {\scriptstyle l_1 \oplus r_2}\lfloor \underline{\mathrm{fpow}}\,(a_1,b_2), \overline{\mathrm{fpow}}\,(b_1,a_2) \rceil_{\scriptstyle r_1 \oplus l_2} & \text{if } pos(I_2) \\ {\scriptstyle r_1 \oplus r_2}\lfloor \underline{\mathrm{fpow}}\,(b_1,b_2), \overline{\mathrm{fpow}}\,(a_1,a_2) \rceil_{\scriptstyle l_1 \oplus l_2} & \text{if } neg(I_2) \\ {\scriptstyle l}\lfloor \underline{\mathrm{fpow}}\,(a_1,b_2), \overline{\mathrm{fpow}}\,(a_1,a_2) \rceil_{\scriptstyle r}, \text{where} & \\ \quad l := \begin{cases} c & \text{if } b_2 = 0 \\ l_1 \oplus r_2 & \text{otherwise,} \end{cases} \quad r := \begin{cases} c & \text{if } a_2 = 0 \\ l_1 \oplus l_2 & \text{otherwise} \end{cases} & \text{if } mix(I_2) \end{cases}$$

If $unk(I_1)$:

$$
\widehat{\mathrm{pow}}(I_1, I_2) :=
\begin{cases}
{}_{l}\lfloor \underline{\mathrm{fpow}}(a_1, b_2), \overline{\mathrm{fpow}}(b_1, b_2) \rceil_{r}, \text{where} \\
\quad l := \begin{cases} l_1 & \text{if } a_1 \in \{0,1\} \\ l_1 \oplus r_2 & \text{otherwise,} \end{cases} \\
\quad r := \begin{cases} r_1 & \text{if } b_1 \in \{0,1\} \\ r_1 \oplus r_2 & \text{otherwise} \end{cases}
& \text{if } pos(I_2) \\
{}_{l}\lfloor \underline{\mathrm{fpow}}(b_1, a_2), \overline{\mathrm{fpow}}(a_1, a_2) \rceil_{r}, \text{where} \\
\quad l := \begin{cases} r_1 & \text{if } b_1 \in \{0,1\} \\ r_1 \oplus l_2 & \text{otherwise,} \end{cases} \\
\quad r := \begin{cases} l_1 & \text{if } a_1 \in \{0,1\} \\ l_1 \oplus l_2 & \text{otherwise} \end{cases}
& \text{if } neg(I_2) \\
U \cup V, \text{ where} \\
\quad U := {}_{l_U}\lfloor \underline{\mathrm{fpow}}(b_1, a_2), \overline{\mathrm{fpow}}(a_1, a_2) \rceil_{r_U}, \text{where} \\
\qquad l_U := \begin{cases} c & \text{if } a_2 = 0 \\ r_1 & \text{if } b_1 \in \{0,1\} \\ r_1 \oplus l_2 & \text{otherwise,} \end{cases} \\
\qquad r_U := \begin{cases} c & \text{if } a_2 = 0 \\ l_1 & \text{if } a_1 \in \{0,1\} \\ l_1 \oplus l_2 & \text{otherwise,} \end{cases} \\
\quad V := {}_{l_V}\lfloor \underline{\mathrm{fpow}}(a_1, b_2), \overline{\mathrm{fpow}}(b_1, b_2) \rceil_{r_V}, \text{where} \\
\qquad l_V := \begin{cases} c & \text{if } b_2 = 0 \\ l_1 & \text{if } a_1 \in \{0,1\} \\ l_1 \oplus r_2 & \text{otherwise} \end{cases} \\
\qquad r_V := \begin{cases} c & \text{if } b_2 = 0 \\ r_1 & \text{if } b_1 \in \{0,1\} \\ r_1 \oplus r_2 & \text{otherwise} \end{cases}
& \text{if } mix(I_2)
\end{cases}
$$

Appendix D

A list of rewrite rules for optimization problems

This appendix gives the full list of the rewrite rules used by the algorithm presented in Chapter 5. These do not include congruence rules, trans, rw_obj, or $\mathsf{rw\_constr}_i$. They correspond to the rules declared in egg. While they are, for the most part, standard mathematical lemmas, the goal of this appendix is to show how the ground-term rewrites are lifted to logical rules. The rules are, by default, bidirectional. We indicate it with a "$*$" if they are not. We highlight the relations that the rewrite rules preserve.

Problem equivalence rewrites

These rules are unique because they are "final" in that they establish a full equivalence. Therefore, note that the variable and hypothesis contexts are empty in the conclusion.

$$\frac{\vec{x}; cs(\vec{x}) \vdash 0 < f(\vec{x})}{\vdash (f, cs) \;\equiv\; (\lambda\vec{x}.\log(f(\vec{x})), cs)}\ \mathsf{map\_objFun\_log}^*_{f,cs}$$

$$\frac{\vec{x}; cs(\vec{x}) \vdash 0 \leq f(\vec{x})}{\vdash (f, cs) \;\equiv\; (\lambda\vec{x}.(f(\vec{x}))^2), cs)}\ \mathsf{map\_objFun\_log}^*_{f,cs}$$

Hereafter, a, b, and c indicate real-valued expression metavariables. Each rule that follows is indeed a rule scheme parametrized by these. Another way to think about it is in terms of pattern-matching, where the left-hand side of the rule defines the pattern.

We also point out that all rules that follow are defined in terms of arbitrary contexts Γ and Σ. The rationale is that these rules need to work after

an application of $\mathsf{rw\_constr}_i$ or $\mathsf{rw\_obj}$; these are final and determine the context of subsequent rules depending on the component of the optimization problem where they are applied. There might be congruence rules in between, which do not change the contexts.

If-and-only-if rewrites

Note that "=" and "≤" at the left and right-hand sides of the rewrites come from constraints, so we think of them as propositional *terms*, whereas when these same symbols appear in the hypotheses, we think of them as *facts*.

$$\frac{\mathrm{fv}(a,b) \subseteq \Gamma \quad \Gamma;\Sigma \vdash 0 < a \quad \Gamma;\Sigma \vdash 0 < b}{\Gamma;\Sigma \vdash \log(a) = \log(b) \;\Leftrightarrow\; a = b}\ \mathsf{log\_eq\_log}_{a,b}$$

$$\frac{\mathrm{fv}(a,b,c) \subseteq \Gamma}{\Gamma;\Sigma \vdash a \le b - c \;\Leftrightarrow\; a + c \le b}\ \mathsf{le\_sub\_iff\_add\_le}_{\mathrm{a,b,c}}$$

$$\frac{\mathrm{fv}(a,b,c) \subseteq \Gamma}{\Gamma;\Sigma \vdash a - c \le b \;\Leftrightarrow\; a \le b + c}\ \mathsf{sub\_le\_iff\_le\_add}_{\mathrm{a,b,c}}$$

$$\frac{\mathrm{fv}(a,b,c) \subseteq \Gamma \quad \Gamma;\Sigma \vdash 0 < c}{\Gamma;\Sigma \vdash a/c \le b \;\Leftrightarrow\; a \le bc}\ \mathsf{div\_le\_iff}_{\mathrm{a,b,c}}$$

$$\frac{\mathrm{fv}(a,b) \subseteq \Gamma \quad \Gamma;\Sigma \vdash 0 < a \quad \Gamma;\Sigma \vdash 0 < b}{\Gamma;\Sigma \vdash \log(a) \le \log(a) \;\Leftrightarrow\; a \le b}\ \mathsf{log\_le\_log}_{\mathrm{a,b}}$$

$$\frac{\mathrm{fv}(a,b) \subseteq \Gamma \quad \Gamma;\Sigma \vdash 0 \le a \quad \Gamma;\Sigma \vdash 0 \le b}{\Gamma;\Sigma \vdash a^2 \le b^2 \;\Leftrightarrow\; a \le b}\ \mathsf{pow\_two\_le\_pow\_two}_{a,b}$$

Equality rewrites

Negation, addition, multiplication, and division

Commutativity and associativity rules deserve some attention. First, note that $\mathsf{add\_comm}_{a,b}$ and $\mathsf{mul\_comm}_{a,b}$ are implicitly bi-directional.

$$\frac{\mathrm{fv}(a,b) \subseteq \Gamma}{\Gamma;\Sigma \vdash a + b \;=\; b + a}\ \mathsf{add\_comm}^{*}_{\mathrm{a,b}} \qquad \frac{\mathrm{fv}(a,b) \subseteq \Gamma}{\Gamma;\Sigma \vdash a \cdot b \;=\; b \cdot a}\ \mathsf{mul\_comm}^{*}_{\mathrm{a,b}}$$

It is well-known that associativity rules can decrease the efficacy of equality saturation workloads by producing large numbers of often unnecessary new nodes. By default, egg uses a so-called "back off" scheduler designed to amortize the

cost of expansive rules such as associativity. It detects rules that dominate others and applies them less often. However, we found that associativity was still problematic. One solution is to limit the number of associativity rules, which we do by only allowing them to be applied in one direction. The other direction can be recovered in combination with commutativity so the rewrite system can still deduce it if necessary. In the future, a better solution would be to write a custom scheduler where we can control exactly when these rules are applied.

$$\frac{\mathrm{fv}(a,b,c) \subseteq \Gamma}{\Gamma;\Sigma \vdash (a+b)+c \;=\; a+(b+c)}\ \mathsf{add\_assoc}^{*}_{\mathrm{a,b,c}}$$

$$\frac{\mathrm{fv}(a,b,c) \subseteq \Gamma}{\Gamma;\Sigma \vdash (a\cdot b)\cdot c \;=\; a\cdot(b\cdot c)}\ \mathsf{mul\_assoc}^{*}_{\mathrm{a,b,c}}$$

The rest of the rules are straightforward. This set of rules is not minimal, and they do not follow the field axioms as one might expect. Some are derived rules that were added because they improved the overall runtime of the examples we targeted.

$$\frac{\mathrm{fv}(a) \subseteq \Gamma}{\Gamma;\Sigma \vdash -(-a) \;=\; a}\ \mathsf{neg\_neg}^{*}_{\mathrm{a}} \qquad \frac{\mathrm{fv}(a) \subseteq \Gamma}{\Gamma;\Sigma \vdash a+0 \;=\; a}\ \mathsf{add\_zero}^{*}_{\mathrm{a}}$$

$$\frac{\mathrm{fv}(a) \subseteq \Gamma}{\Gamma;\Sigma \vdash a-a \;=\; 0}\ \mathsf{sub\_self}^{*}_{\mathrm{a}} \qquad \frac{\mathrm{fv}(a) \subseteq \Gamma}{\Gamma;\Sigma \vdash 1\cdot a \;=\; a}\ \mathsf{one\_mul}_{\mathrm{a}}$$

$$\frac{\mathrm{fv}(a) \subseteq \Gamma}{\Gamma;\Sigma \vdash 0\cdot a \;=\; 0}\ \mathsf{mul\_zero}_{\mathrm{a}}{}^{*} \qquad \frac{\mathrm{fv}(a,b) \subseteq \Gamma}{\Gamma;\Sigma \vdash a-b \;=\; a+(-b)}\ \mathsf{sub\_eq\_add\_neg}_{\mathrm{a,b}}$$

$$\frac{\mathrm{fv}(a,b,c) \subseteq \Gamma}{\Gamma;\Sigma \vdash a+(b-c) \;=\; (a+b)-c}\ \mathsf{add\_sub}_{\mathrm{a,b,c}}$$

$$\frac{\mathrm{fv}(a,b,c) \subseteq \Gamma}{\Gamma;\Sigma \vdash (a+b)\cdot c \;=\; (a\cdot c)+(b\cdot c)}\ \mathsf{add\_mul}_{\mathrm{a,b,c}}$$

$$\frac{\mathrm{fv}(a,b,c) \subseteq \Gamma}{\Gamma;\Sigma \vdash a\cdot(b+c) \;=\; (a\cdot b)+(a\cdot c)}\ \mathsf{mul\_add}_{\mathrm{a,b,c}}$$

$$\frac{\mathrm{fv}(a,b,c) \subseteq \Gamma}{\Gamma;\Sigma \vdash a\cdot(b-c) \;=\; (a\cdot b)-(a\cdot c)}\ \mathsf{mul\_sub}_{\mathrm{a,b,c}} \qquad \frac{\mathrm{fv}(a) \subseteq \Gamma}{\Gamma;\Sigma \vdash a/1 \;=\; a}\ \mathsf{div\_one}^{*}_{\mathrm{a}}$$

$$\frac{\mathrm{fv}(a,b,c) \subseteq \Gamma \quad \Gamma;\Sigma \vdash c \neq 0}{\Gamma;\Sigma \vdash (a+b)/c \;=\; (a/c)+(b/c)}\ \mathsf{add\_div}_{\mathrm{a,b,c}}$$

$$\frac{\mathrm{fv}(a,b,c) \subseteq \Gamma \quad \Gamma;\Sigma \vdash c \neq 0}{\Gamma;\Sigma \vdash a \cdot (b/c) \;=\; (a \cdot b)/c}\ \mathsf{mul\_div}_{\mathrm{a,b,c}}$$

$$\frac{\mathrm{fv}(a,b,c) \subseteq \Gamma \quad \Gamma;\Sigma \vdash b \neq 0 \quad \Gamma;\Sigma \vdash c \neq 0}{\Gamma;\Sigma \vdash (a/b) \cdot c \;=\; a/(b/c)}\ \mathsf{div\_mul}_{\mathrm{a,b,c}}$$

$$\frac{\mathrm{fv}(a,b,c) \subseteq \Gamma \quad \Gamma;\Sigma \vdash c \neq 0}{\Gamma;\Sigma \vdash (c \cdot a)/(c \cdot b) \;=\; a/b}\ \mathsf{mul\_div\_mul\_left}^{*}_{\mathrm{a,b,c}}$$

$$\frac{\mathrm{fv}(a,b,c) \subseteq \Gamma \quad \Gamma;\Sigma \vdash b \neq 0 \quad \Gamma;\Sigma \vdash c \neq 0}{\Gamma;\Sigma \vdash (a/b)/c \;=\; a/(b \cdot c)}\ \mathsf{div\_div}_{\mathrm{a,b,c}}$$

$$\frac{\mathrm{fv}(a) \subseteq \Gamma \quad \Gamma;\Sigma \vdash a \neq 0}{\Gamma;\Sigma \vdash a/a \;=\; 1}\ \mathsf{div\_self}^{*}_{\mathrm{a}}$$

Powers and square root

$$\frac{\mathrm{fv}(a) \subseteq \Gamma}{\Gamma;\Sigma \vdash 1^a \;=\; 1}\ \mathsf{one\_pow}^{*}_{\mathrm{a}} \qquad \frac{\mathrm{fv}(a) \subseteq \Gamma}{\Gamma;\Sigma \vdash a^1 \;=\; a}\ \mathsf{pow\_one}^{*}_{\mathrm{a}}$$

$$\frac{\mathrm{fv}(a,b,c) \subseteq \Gamma \quad \Gamma;\Sigma \vdash a > 0}{\Gamma;\Sigma \vdash a^{b+c} \;=\; a^b \cdot a^c}\ \mathsf{pow\_add}_{\mathrm{a,b,c}}$$

$$\frac{\mathrm{fv}(a,b,c) \subseteq \Gamma \quad \Gamma;\Sigma \vdash a \geq 0 \quad \Gamma;\Sigma \vdash b \geq 0}{\Gamma;\Sigma \vdash (a \cdot b)^c \;=\; a^c \cdot b^c}\ \mathsf{mul\_pow}_{\mathrm{a,b,c}}$$

$$\frac{\mathrm{fv}(a,b,c) \subseteq \Gamma \quad \Gamma;\Sigma \vdash a \geq 0}{\Gamma;\Sigma \vdash a^{b \cdot c} \;=\; (a^b)^c}\ \mathsf{pow\_mul}_{\mathrm{a,b,c}}$$

$$\frac{\mathrm{fv}(a,b,c) \subseteq \Gamma \quad \Gamma;\Sigma \vdash a \geq 0 \quad \Gamma;\Sigma \vdash b > 0}{\Gamma;\Sigma \vdash (a/b)^c \;=\; a^c/b^c}\ \mathsf{div\_pow}_{\mathrm{a,b,c}}$$

$$\frac{\mathrm{fv}(a,b,c) \subseteq \Gamma \quad \Gamma;\Sigma \vdash a > 0}{\Gamma;\Sigma \vdash a^{b-c} \;=\; a^b/a^c}\ \mathsf{pow\_sub}_{\mathrm{a,b,c}}$$

$$\frac{\mathrm{fv}(a,b,c) \subseteq \Gamma \quad \Gamma;\Sigma \vdash b > 0}{\Gamma;\Sigma \vdash a/b^n \;=\; a \cdot b^{-n}}\ \mathsf{div\_pow\_eq\_mul\_pow\_neg}_{\mathrm{a,b,c}}$$

$$\frac{\mathrm{fv}(a) \subseteq \Gamma \quad \Gamma;\Sigma \vdash a > 0}{\Gamma;\Sigma \vdash 1/a \;=\; a^{-1}}\ \mathsf{one\_div\_eq\_pow\_neg\_one}_{\mathrm{a}}$$

$$\frac{\mathrm{fv}(a) \subseteq \Gamma}{\Gamma;\Sigma \vdash \sqrt{a} \;=\; a^{0.5}}\ \mathsf{sqrt\_eq\_rpow}_{\mathrm{a}}$$

$$\frac{\mathrm{fv}(a,b) \subseteq \Gamma \quad \Gamma;\Sigma \vdash a \geq 0 \quad \Gamma;\Sigma \vdash b \geq 0}{\Gamma;\Sigma \vdash \sqrt{a \cdot b} \;=\; \sqrt{a} \cdot \sqrt{b}}\ \mathsf{sqrt\_mul}_{\mathrm{a,b}}$$

$$\frac{\mathrm{fv}(a) \subseteq \Gamma}{\Gamma;\Sigma \vdash a^2 \;=\; a \cdot a}\ \mathsf{pow\_two_a} \qquad \frac{\mathrm{fv}(a) \subseteq \Gamma \quad \Gamma;\Sigma \vdash a \geq 0}{\Gamma;\Sigma \vdash (a^{0.5})^2 \;=\; a}\ \mathsf{pow\_half\_two_a}$$

$$\frac{\mathrm{fv}(a,b) \subseteq \Gamma}{\Gamma;\Sigma \vdash (a+b)^2 \;=\; a^2+2ab+b^2}\ \mathsf{binomial\_two^*_{a,b}}$$

Inverse

$$\frac{\mathrm{fv}(a) \subseteq \Gamma \quad \Gamma;\Sigma \vdash a \neq 0}{\Gamma;\Sigma \vdash \mathrm{inv}(a) \;=\; 1/a}\ \mathsf{inv\_eq\_one\_div^*_a} \qquad \frac{\mathrm{fv}(a) \subseteq \Gamma \quad \Gamma;\Sigma \vdash a \neq 0}{\Gamma;\Sigma \vdash \mathrm{inv}(\mathrm{inv}(a)) \;=\; a}\ \mathsf{inv\_inv^*_a}$$

$$\frac{\mathrm{fv}(a) \subseteq \Gamma \quad \Gamma;\Sigma \vdash a \neq 0}{\Gamma;\Sigma \vdash \mathrm{inv}(a) \;=\; a^{-1}}\ \mathsf{inv\_eq\_pow\_neg\_one^*_a}$$

Logarithm and exponential

$$\frac{\mathrm{fv}(a,b) \subseteq \Gamma}{\Gamma;\Sigma \vdash \exp(a+b) \;=\; \exp(a)\cdot\exp(b)}\ \mathsf{exp\_add_{a,b}}$$

$$\frac{\mathrm{fv}(a,b) \subseteq \Gamma}{\Gamma;\Sigma \vdash \exp(a-b) \;=\; \exp(a)/\exp(b)}\ \mathsf{exp\_sub_{a,b}}$$

$$\frac{\mathrm{fv}(a,b) \subseteq \Gamma}{\Gamma;\Sigma \vdash \exp(a\cdot b) \;=\; \exp(a)^b}\ \mathsf{exp\_mul_{a,b}}$$

$$\frac{\mathrm{fv}(a) \subseteq \Gamma}{\Gamma;\Sigma \vdash \exp(-a) \;=\; 1/\exp(a)}\ \mathsf{exp\_neg\_eq\_one\_div_a}$$

$$\frac{\mathrm{fv}(a,b) \subseteq \Gamma \quad \Gamma;\Sigma \vdash a > 0 \quad \Gamma;\Sigma \vdash b > 0}{\Gamma;\Sigma \vdash \log(a\cdot b) \;=\; \log(a)+\log(b)}\ \mathsf{log\_mul_{a,b}}$$

$$\frac{\mathrm{fv}(a,b) \subseteq \Gamma \quad \Gamma;\Sigma \vdash a > 0 \quad \Gamma;\Sigma \vdash b > 0}{\Gamma;\Sigma \vdash \log(a/b) \;=\; \log(a)-\log(b)}\ \mathsf{log\_div_{a,b}}$$

$$\frac{\mathrm{fv}(a,b) \subseteq \Gamma \quad \Gamma;\Sigma \vdash a > 0}{\Gamma;\Sigma \vdash \log(a^b) \;=\; b\cdot\log(a)}\ \mathsf{log\_pow_{a,b}}$$

$$\frac{\mathrm{fv}(a,b) \subseteq \Gamma \quad \Gamma;\Sigma \vdash b \in \mathbb{N}}{\Gamma;\Sigma \vdash \log(a^b) \;=\; b\cdot\log(a)}\ \mathsf{log\_pow\_nat_{a,b}}$$

$$\frac{\mathrm{fv}(a) \subseteq \Gamma \quad \Gamma;\Sigma \vdash a > 0}{\Gamma;\Sigma \vdash \exp(\log(a)) \;=\; a}\ \mathsf{exp\_log^*_a} \qquad \frac{\mathrm{fv}(a) \subseteq \Gamma}{\Gamma;\Sigma \vdash \log(\exp(a)) \;=\; a}\ \mathsf{log\_exp^*_a}$$

Absolute value

$$\frac{\mathrm{fv}(a) \subseteq \Gamma \quad \Gamma;\Sigma \vdash a \geq 0}{\Gamma;\Sigma \vdash \mathrm{abs}(a) \;=\; a}\ \mathsf{abs\_nonneg}^*_{\mathrm{a}} \qquad \frac{\mathrm{fv}(a) \subseteq \Gamma \quad \Gamma;\Sigma \vdash a \leq 0}{\Gamma;\Sigma \vdash \mathrm{abs}(a) \;=\; -a}\ \mathsf{abs\_nonpos}^*_{\mathrm{a}}$$

Maximum and minimum

$$\frac{\mathrm{fv}(a,b) \subseteq \Gamma \quad \Gamma;\Sigma \vdash a \leq b}{\Gamma;\Sigma \vdash \min(a,b) \;=\; a}\ \mathsf{min\_eq\_left}^*_{\mathrm{a,b}} \qquad \frac{\mathrm{fv}(a,b) \subseteq \Gamma \quad \Gamma;\Sigma \vdash b \leq a}{\Gamma;\Sigma \vdash \min(a,b) \;=\; b}\ \mathsf{min\_eq\_right}^*_{\mathrm{a,b}}$$

$$\frac{\mathrm{fv}(a,b) \subseteq \Gamma \quad \Gamma;\Sigma \vdash a \geq b}{\Gamma;\Sigma \vdash \max(a,b) \;=\; a}\ \mathsf{max\_eq\_left}^*_{\mathrm{a,b}} \qquad \frac{\mathrm{fv}(a,b) \subseteq \Gamma \quad \Gamma;\Sigma \vdash b \geq a}{\Gamma;\Sigma \vdash \max(a,b) \;=\; b}\ \mathsf{max\_eq\_right}^*_{\mathrm{a,b}}$$

Atom folding and unfolding

$$\frac{\mathrm{fv}(a) \subseteq \Gamma \quad \Gamma;\Sigma \vdash a \geq 0}{\Gamma;\Sigma \vdash a \cdot \exp(a) \;=\; \mathrm{xexp}(a)}\ \mathsf{xexp\_fold}_{\mathrm{a}}$$

$$\frac{\mathrm{fv}(a) \subseteq \Gamma \quad \Gamma;\Sigma \vdash a > 0}{\Gamma;\Sigma \vdash -(a \cdot \log(a)) \;=\; \mathrm{entr}(a)}\ \mathsf{entr\_fold}_{\mathrm{a}}$$

$$\frac{\mathrm{fv}(a,b) \subseteq \Gamma \quad \Gamma;\Sigma \vdash b > 0}{\Gamma;\Sigma \vdash a^2/b \;=\; \mathrm{qol}(a,b)}\ \mathsf{qol\_fold}_{\mathrm{a,b}}$$

$$\frac{\mathrm{fv}(a,b) \subseteq \Gamma \quad \Gamma;\Sigma \vdash a > 0 \quad \Gamma;\Sigma \vdash b > 0}{\Gamma;\Sigma \vdash \sqrt{a \cdot b} \;=\; \mathrm{geo}(a,b)}\ \mathsf{geo\_fold}_{\mathrm{a,b}}$$

$$\frac{\mathrm{fv}(a,b) \subseteq \Gamma}{\Gamma;\Sigma \vdash \log(\exp(a) + \exp(b)) \;=\; \mathrm{lse}(a,b)}\ \mathsf{lse\_fold}_{\mathrm{a,b}}$$

$$\frac{\mathrm{fv}(a,b) \subseteq \Gamma}{\Gamma;\Sigma \vdash \sqrt{a^2 + b^2} \;=\; \mathrm{norm2}(a,b)}\ \mathsf{norm2\_fold}_{\mathrm{a,b}}$$

Appendix E

Supplementary egg-pre-dcp results

The results in this appendix extend Table 5.3, where we compared the two extraction mechanisms for the e-graph-based pre-DCP rewriting system. Recall that the problems used for evaluation are a subset of the total benchmark consisting of problems coming from geometric programming, quasiconvex programming, the Stanford convex optimization course, the DCP quiz, and the motivating example in Chapter 5.

We extend the results in several ways. First, we split the runtime into building time (**Build**) and explanation time (**Explain**). Building time is the time elapsed between creating the initial e-graph and finding the best term. Explanation time is the time it takes to generate a sequence of rewrites once the best term is found. Second, we add the number of e-nodes in the final e-graph before extraction (**Nodes**).

We provide two tables. Table E.1 shows the results for the stop-on-success approach, and Table E.2 shows the results for the iterative approach.

Note that in the iterative approach, the node limits are never reached exactly; the final number of nodes is always below one of the limits. That is because every iteration adds several nodes, and `egg` will stop if it detects that an iteration will go over the limit.

Problem	Time			Nodes	Steps	Term size
	Total	Build	Explain			
gp1	5 ms	5 ms	< 1 ms	54	1	13
gp2	2 ms	2 ms	< 1 ms	60	1	13
gp3	5 ms	4 ms	< 1 ms	58	4	16
gp4	5766 ms	5372 ms	394 ms	23879	47	43
gp5	5474 ms	5092 ms	381 ms	24452	38	45
gp6	603 ms	516 ms	86 ms	8235	53	43
gp7	193 ms	166 ms	26 ms	3613	31	61
gp8	4939 ms	4629 ms	310 ms	19327	79	72
gp9	31158 ms	30225 ms	933 ms	41012	123	104
agp1	2 ms	2 ms	< 1 ms	58	2	15
agp2	2 ms	1 ms	< 1 ms	64	2	15
agp3	4638 ms	4179 ms	459 ms	19860	29	46
agp4	9 ms	8 ms	< 1 ms	245	3	20
qcp1	6 ms	5 ms	< 1 ms	137	6	12
qcp2	3 ms	3 ms	< 1 ms	107	6	35
qcp3	1 ms	1 ms	< 1 ms	49	1	15
qcp4	1128 ms	988 ms	139 ms	10847	17	19
stan1	2 ms	2 ms	< 1 ms	60	1	17
stan2	32 ms	26 ms	6 ms	563	9	16
stan3	201 ms	178 ms	23 ms	2572	20	31
stan4	746 ms	697 ms	48 ms	7754	38	48
quiz1	1 ms	1 ms	< 1 ms	19	1	4
quiz2	2 ms	2 ms	< 1 ms	82	3	11
quiz3	1 ms	1 ms	< 1 ms	17	2	4
quiz4	1 ms	1 ms	< 1 ms	17	2	6
quiz5	1 ms	1 ms	< 1 ms	22	1	6
quiz6	1 ms	1 ms	< 1 ms	35	1	10
quiz7	33 ms	26 ms	6 ms	619	11	17
quiz8	1 ms	1 ms	< 1 ms	14	1	6
quiz9	227 ms	148 ms	78 ms	1868	12	9
quiz10	1 ms	1 ms	< 1 ms	20	1	3
main	5 ms	5 ms	< 1 ms	185	3	15

Table E.1: Stop-on-success full results on selected problems.

Problem	Time			Nodes	Steps	Term size
	Total	Build	Explain			
gp1	81 ms	63 ms	18 ms	2143	7	11
gp2	57 ms	44 ms	12 ms	2255	6	12
gp3	92 ms	56 ms	36 ms	2033	16	12
gp4	5426 ms	4988 ms	437 ms	26980	46	43
gp5	5511 ms	5070 ms	441 ms	26388	37	45
gp6	474 ms	393 ms	80 ms	8022	51	43
gp7	237 ms	201 ms	36 ms	4819	35	61
gp8	7255 ms	6717 ms	538 ms	28614	180	72
gp9	51033 ms	37311 ms	13722 ms	57229	175	104
agp1	69 ms	47 ms	21 ms	2368	10	11
agp2	68 ms	48 ms	20 ms	2389	9	12
agp3	5648 ms	5108 ms	539 ms	24878	35	46
agp4	93 ms	52 ms	40 ms	2049	10	19
qcp1	85 ms	47 ms	37 ms	1730	9	10
qcp2	92 ms	67 ms	24 ms	1848	16	28
qcp3	59 ms	44 ms	14 ms	2330	2	15
qcp4	699 ms	607 ms	92 ms	8220	20	19
stan1	75 ms	58 ms	17 ms	2362	1	17
stan2	104 ms	87 ms	17 ms	2416	8	16
stan3	262 ms	231 ms	30 ms	4037	28	29
stan4	813 ms	765 ms	47 ms	8658	35	48
quiz1	100 ms	44 ms	56 ms	1835	1	4
quiz2	74 ms	53 ms	21 ms	1783	3	11
quiz3	61 ms	47 ms	13 ms	2036	2	4
quiz4	91 ms	77 ms	13 ms	2448	2	6
quiz5	82 ms	58 ms	24 ms	2246	1	6
quiz6	88 ms	71 ms	17 ms	2155	1	10
quiz7	121 ms	62 ms	58 ms	2482	10	10
quiz8	92 ms	76 ms	16 ms	2241	1	6
quiz9	289 ms	194 ms	94 ms	3387	12	9
quiz10	83 ms	66 ms	16 ms	1991	1	3
main	65 ms	48 ms	16 ms	2234	4	15

Table E.2: Iterative node limits ($N = 2500, 5000, 10000, 20000, 40000, 80000$) full results on selected problems.

Appendix F

Obtaining and navigating CvxLean

This appendix contains complete installation instructions[1] for CvxLean 2.0 and an overview of its structure.

F.1 Requirements

The prerequisites are Lean, Rust, and MOSEK. We also recommend using Lean's VSCode interface.

Lean

The most convenient way to install Lean is through `elan`, the Lean version manager. Detailed installation instructions can be found on its GitHub repository.

`https://github.com/leanprover/elan`

We use v4.6.0-rc1, which we explain how to install in appendix F.2.

Rust

The pre-DCP procedure requires a Rust installation, including Cargo. The instructions are given on the official website.

`https://www.rust-lang.org/tools/install`

We have tested it with `rustc` and `cargo` versions 1.78.0-nightly.

[1] These rely on several external links, all of which were accessed on 2024-02-10.

MOSEK

Install MOSEK 10.2.12 from the link below and add the location of the binary, e.g., `$HOME/mosek/10.0/tools/platform/linux64x86/bin`, to PATH.

`https://www.mosek.com/downloads/10.0.12/`

Further instructions can be found on the official website.

`https://docs.mosek.com/latest/install/installation.html`

MOSEK requires a license. A personal academic license can be obtained for free using an e-mail address from an academic institution.

`https://www.mosek.com/license/request/`

Users that do not have this option can skip this step, and they will be able to install CvxLean, although all calls to `solve` will fail.

It is recommended to set the MOSEKLM_LICENSE_FILE environment variable to the path of the `mosek.lic` file, e.g., `$HOME/mosek/mosek.lic`.

VSCode extension

The extension can be found on the VSCode marketplace by searching "Lean 4". The link below gives further information.

`https://github.com/leanprover/vscode-lean4`

F.2 Installation

First, download the sources for CvxLean 2.0.

`https://github.com/verified-optimization/CvxLean/releases/tag/v2.0`

Go to the CvxLean directory or open it in VSCode as a workspace. On a terminal, make sure you have the correct Lean version, found on `lean-toolchain`, or simply run the command below.

```
elan override set leanprover/lean4:4.6.0-rc1
```

At this point, we can build CvxLean by executing the following command.

```
./build.sh
```

It uses Lake to download all dependencies, build the `egg-pre-dcp` Rust library, and build `CvxLean`.

In some cases, there might be issues finding the `MOSEK` binary, even after adding it to the environment. Users can specify the path on the file below to avoid these issues completely.

`CvxLean/Tactic/Solver/Mosek/Path.lean`

To ensure that the installation was successful, open any file in the `Test` or `Examples` directories. Clicking on the source file should show the state on the infoview.

F.3 Tests

Alternatively, you can run all tests and examples running the following command.

```
lake build CvxLeanTest
```

On the Rust side, the tests are found in `/egg-pre-dcp/tests` that can be run from the `/egg-pre-dcp` directory in the usual way (`cargo test`).

F.4 Directory and file structure

This section gives an overview of the project's contents, highlighting the main files and directories. For each, we provide a short summary.

The core is found in the `/CvxLean` directory and structured as follows:

- `/Lib`: definitions of the main building blocks and results about them, mostly covered in Chapter 3.
  - `Minimization.lean`: main definitions (minimization problems and solutions).
  - `Equivalence.lean`: definition of an equivalence, a strong equivalence, and some equivalence-preserving transformations.
  - `Reduction.lean`: definition of a reduction and some reduction-preserving transformations.
  - `Relaxation.lean`: definition of a relaxation and some relaxation-preserving transformations.

  - `/Cones`: definitions of all the cones that we consider.
  - `/Math`: some definitions and results that are not currently in mathlib.

- `/Syntax`: custom syntax to define optimization problems.
- `/Meta`: common meta-level code used by tactics and commands.
- `/Tactic`: custom tactics.
  - `/Arith`: extensions of mathlib arithmetic tactics.
  - `/Basic`: basic tactics covered in Chapter 3 including changes of variables, rewriting, converting, renaming, etc.
  - `/DCP`: DCP transformation presented in Chapter 4.
    * `DCP*.lean`: canonicalization procedure and the `dcp` tactic.
    * `AtomCmd*.lean`: `declare_atom` command.
    * `/AtomLibrary`: concrete atom declarations.
  - `/PreDCP`: pre-DCP transformation presented in Chapter 5.
    * `PreDCP.lean`: `pre_dcp` tactic.
    * `/Egg`: interface with egg.
    * `RuleToTacticLibrary.lean`: translations from egg rewrite rules to Lean tactics.
- `/Command`: custom commands.
  - `Equivalence.lean`: `equivalence` command.
  - `Reduction.lean`: `reduction` command.
  - `Relaxation.lean`: `relaxation` command.
  - `Solve.lean` and `/Solve`: `solve` command, including data extraction, and the MOSEK interface.
- `/Examples`: examples of applications, including the case studies covered in Chapter 6.
- `/Test`: unit tests and small examples.

There is also automatically generated documentation for the files above and the rest of the Lean files.

https://verified-optimization.github.io/CvxLean/

The Rust code needed for the pre-DCP procedure is in `/egg-pre-dcp/src` and includes the following files:

- `domain.rs`: implementation of extended interval arithmetic presented in Appendix C.
- `optimization.rs`: language definition and e-class analysis based on the domain.
- `rules.rs`: `egg` rewrite rules corresponding to the rules in Appendix D.
- `curvature.rs`: DCP curvature rules.
- `cost.rs`: cost calculation based on the curvature.
- `extract.rs`: extraction mechanism of best term and sequence on rewrites based on the cost and e-class analysis, including the equality saturation setup and configuration (limits).

F.5 Project size

We conclude by showing the lines of code in `CvxLean` 2.0 to give the reader an idea of the size of the project. The Lean code is found in `/CvxLean`, and the Rust code is found in `/egg-pre-dcp`, as explained in the previous section. Other code includes Python and bash scripts, mainly to compare to `CVXPY` and for evaluation. We split it into core code and code used for tests and examples.

	Total	**Code**	**Docs**	**Blank**
Lean core	16702	11540	2294	2868
Rust core	3455	2846	166	443
Lean test	4250	2765	519	966
Rust test	1398	1025	169	204
Other	1448	1102	46	300
Total core	20157	14386	2460	3311
Total	27253	19278	3194	4781

Bibliography

[AA00] Erling D Andersen and Knud D Andersen. The MOSEK interior point optimizer for linear programming: an implementation of the homogeneous algorithm. In *High performance optimization*, pages 197–232. Springer, 2000.

[AB20] Akshay Agrawal and Stephen P. Boyd. Disciplined quasiconvex programming. *Optim. Lett.*, 14(7):1643–1657, 2020. doi:10.1007/S11590-020-01561-8.

[ADB19] Akshay Agrawal, Steven Diamond, and Stephen P. Boyd. Disciplined geometric programming. *Optim. Lett.*, 13(5):961–976, 2019. doi:10.1007/S11590-019-01422-Z.

[ADL+11] Martin Andersen, Joachim Dahl, Zhang Liu, Lieven Vandenberghe, S Sra, S Nowozin, and S Wright. Interior-point methods for large-scale cone programming. *Optimization for machine learning*, 5583, 2011.

[AdMKU24] Jeremy Avigad, Leonardo de Moura, Soonho Kong, and Sebastian Ullrich. *Theorem Proving in Lean 4*, 2024. https://lean-lang.org/theorem_proving_in_lean4/ (accessed 2024-03-23).

[AGL+22] Jeremy Avigad, Lior Goldberg, David Levit, Yoav Seginer, and Alon Titelman. A verified algebraic representation of cairo program execution. In Andrei Popescu and Steve Zdancewic, editors, *CPP '22: 11th ACM SIGPLAN International Conference on Certified Programs and Proofs, Philadelphia, PA, USA, January 17 - 18, 2022*, pages 153–165. ACM, 2022. doi:10.1145/3497775.3503675.

[AK19] Xavier Allamigeon and Ricardo D. Katz. A formalization of convex polyhedra based on the simplex method. *J. Autom. Reason.*, 63(2):323–345, 2019. doi:10.1007/S10817-018-9477-1.

[AM24] Jeremy Avigad and Patrick Massot. *Mathematics in Lean*, 2024. https://leanprover-community.github.io/mathematics_in_lean/ (accessed 2024-03-14).

[ART03] Erling D. Andersen, Cornelis Roos, and Tamás Terlaky. On implementing a primal-dual interior-point method for conic quadratic optimization. *Math. Program.*, 95(2):249–277, 2003. doi:10.1007/S10107-002-0349-3.

[AVDB18] Akshay Agrawal, Robin Verschueren, Steven Diamond, and Stephen Boyd. A rewriting system for convex optimization problems. *Journal of Control and Decision*, 5(1):42–60, 2018.

[Aye21] Edward William Ayers. *A Tool for Producing Verified, Explainable Proofs*. PhD thesis, Corpus Christi College, University of Cambridge, 2021.

[BA21] Alexander Bentkamp and Jeremy Avigad. Verified optimization (work in progress). In Jasmin Blanchette, James H. Davenport, Peter Koepke, Michael Kohlhase, Andrea Kohlhase, Adam Naumowicz, Dennis Müller, Yasmine Sharoda, and Claudio Sacerdoti Coen, editors, *Joint Proceedings of the FMM, FVPS, MathUI,NatFoM, and OpenMath Workshops, Doctoral Program, and Work in Progress at the Conference on Intelligent Computer Mathematics 2021 co-located with the 14th Conference on Intelligent Computer Mathematics (CICM 2021), Virtual Event, Timisoara, Romania, July 26 - 31, 2021*, volume 3377 of *CEUR Workshop Proceedings*. CEUR-WS.org, 2021. URL: https://ceur-ws.org/Vol-3377/fmm7.pdf.

[Baa22] Anne Baanen. Use and abuse of instance parameters in the Lean mathematical library. In June Andronick and Leonardo de Moura, editors, *13th International Conference on Interactive Theorem Proving, ITP 2022, August 7-10, 2022, Haifa, Israel*, volume 237 of

LIPIcs, pages 4:1–4:20. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2022. doi:10.4230/LIPICS.ITP.2022.4.

[BBCD23] Anne Baanen, Alex J. Best, Nirvana Coppola, and Sander R. Dahmen. Formalized class group computations and integral points on Mordell elliptic curves. In Robbert Krebbers, Dmitriy Traytel, Brigitte Pientka, and Steve Zdancewic, editors, *Proceedings of the 12th ACM SIGPLAN International Conference on Certified Programs and Proofs, CPP 2023, Boston, MA, USA, January 16-17, 2023*, pages 47–62. ACM, 2023. doi:10.1145/3573105.3575682.

[BBHI05] Alan Bundy, David A. Basin, Dieter Hutter, and Andrew Ireland. *Rippling: meta-level guidance for mathematical reasoning*, volume 56 of *Cambridge Tracts in Theoretical Computer Science*. Cambridge University Press, 2005.

[BCM20] Kevin Buzzard, Johan Commelin, and Patrick Massot. Formalising perfectoid spaces. In Jasmin Blanchette and Catalin Hritcu, editors, *Proceedings of the 9th ACM SIGPLAN International Conference on Certified Programs and Proofs, CPP 2020, New Orleans, LA, USA, January 20-21, 2020*, pages 299–312. ACM, 2020. doi:10.1145/3372885.3373830.

[BFMA23] Alexander Bentkamp, Ramon Fernández Mir, and Jeremy Avigad. Verified reductions for optimization. In Sriram Sankaranarayanan and Natasha Sharygina, editors, *Tools and Algorithms for the Construction and Analysis of Systems - 29th International Conference, TACAS 2023, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2022, Paris, France, April 22-27, 2023, Proceedings, Part II*, volume 13994 of *Lecture Notes in Computer Science*, pages 74–92. Springer, 2023. doi:10.1007/978-3-031-30820-8_8.

[BHC95] Clemens Ballarin, Karsten Homann, and Jacques Calmet. Theorems and algorithms: An interface between Isabelle and Maple. In A. H. M. Levelt, editor, *Proceedings of the 1995 International Symposium on Symbolic and Algebraic Computation, ISSAC '95,*

Montreal, Canada, July 10-12, 1995, pages 150–157. ACM, 1995. doi:10.1145/220346.220366.

[BHL+22] Kevin Buzzard, Chris Hughes, Kenny Lau, Amelia Livingston, Ramon Fernández Mir, and Scott Morrison. Schemes in Lean. *Exp. Math.*, 31(2):355–363, 2022. doi:10.1080/10586458.2021.1983489.

[BKPU16] Jasmin Christian Blanchette, Cezary Kaliszyk, Lawrence C. Paulson, and Josef Urban. Hammering towards QED. *J. Formaliz. Reason.*, 9(1):101–148, 2016. doi:10.6092/ISSN.1972-5787/4593.

[BKVH07] Stephen Boyd, Seung-Jean Kim, Lieven Vandenberghe, and Arash Hassibi. A tutorial on geometric programming. *Optimization and engineering*, 8(1):67–127, 2007.

[BM11] Sylvie Boldo and Guillaume Melquiond. Flocq: A unified library for proving floating-point algorithms in Coq. In Elisardo Antelo, David Hough, and Paolo Ienne, editors, *20th IEEE Symposium on Computer Arithmetic, ARITH 2011, Tübingen, Germany, 25-27 July 2011*, pages 243–252. IEEE Computer Society, 2011. doi:10.1109/ARITH.2011.40.

[BP99] Clemens Ballarin and Lawrence C. Paulson. A pragmatic approach to extending provers by computer algebra - with applications to coding theory. *Fundam. Informaticae*, 39(1-2):1–20, 1999. doi:10.3233/FI-1999-391201.

[BPT12] Grigoriy Blekherman, Pablo A Parrilo, and Rekha R Thomas. *Semidefinite optimization and convex algebraic geometry*. SIAM, 2012.

[BV04] Stephen P. Boyd and Lieven Vandenberghe. *Convex Optimization*. Cambridge University Press, 2004. doi:10.1017/CBO9780511804441.

[Car19] Mario Carneiro. The type theory of Lean. Master's thesis, Carnegie Mellon University, 2019.

[CDT24] The Coq Development Team. *The Coq Reference Manual, version 8.19.0*, January 2024. Available electronically at `https://coq.inria.fr/doc/v8.19/refman/` (accessed 2024-03-20).

[CFG20] Raphael Cohen, Eric Feron, and Pierre-Loïc Garoche. Verification and validation of convex optimization algorithms for model predictive control. *Journal of Aerospace Information Systems*, 17(5):257–270, 2020.

[CH88] Thierry Coquand and Gérard P. Huet. The calculus of constructions. *Inf. Comput.*, 76(2/3):95–120, 1988. `doi:10.1016/0890-5401(88)90005-3`.

[CL21] Johan Commelin and Robert Y. Lewis. Formalizing the ring of Witt vectors. In *Proceedings of the 10th ACM SIGPLAN International Conference on Certified Programs and Proofs*, CPP 2021, page 264–277, New York, NY, USA, 2021. Association for Computing Machinery. `doi:10.1145/3437992.3439919`.

[CP88] Thierry Coquand and Christine Paulin. Inductively defined types. In Per Martin-Löf and Grigori Mints, editors, *COLOG-88, International Conference on Computer Logic, Tallinn, USSR, December 1988, Proceedings*, volume 417 of *Lecture Notes in Computer Science*, pages 50–66. Springer, 1988. `doi:10.1007/3-540-52335-9\_47`.

[Dav23] James Harold Davenport. Proving an execution of an algorithm correct? In Catherine Dubois and Manfred Kerber, editors, *Intelligent Computer Mathematics - 16th International Conference, CICM 2023, Cambridge, UK, September 5-8, 2023, Proceedings*, volume 14101 of *Lecture Notes in Computer Science*, pages 255–269. Springer, 2023. `doi:10.1007/978-3-031-42753-4\_17`.

[DB16] Steven Diamond and Stephen P. Boyd. CVXPY: A python-embedded modeling language for convex optimization. *J. Mach. Learn. Res.*, 17:83:1–83:5, 2016. URL: `http://jmlr.org/papers/v17/15-408.html`.

[DCB13] Alexander Domahidi, Eric Chu, and Stephen Boyd. ECOS: An SOCP solver for embedded systems. In *2013 European Control Conference (ECC)*, pages 3071–3076. IEEE, 2013.

[Del00] David Delahaye. A tactic language for the system Coq. In Michel Parigot and Andrei Voronkov, editors, *Logic for Programming and Automated Reasoning, 7th International Conference, LPAR 2000, Reunion Island, France, November 11-12, 2000, Proceedings*, volume 1955 of *Lecture Notes in Computer Science*, pages 85–95. Springer, 2000. doi:10.1007/3-540-44404-1_7.

[dF22] María Inés de Frutos-Fernández. Formalizing the ring of adèles of a global field. In June Andronick and Leonardo de Moura, editors, *13th International Conference on Interactive Theorem Proving, ITP 2022, August 7-10, 2022, Haifa, Israel*, volume 237 of *LIPIcs*, pages 14:1–14:18. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2022. doi:10.4230/LIPICS.ITP.2022.14.

[DM05] David Delahaye and Micaela Mayero. Dealing with algebraic expressions over a field in Coq using Maple. *J. Symb. Comput.*, 39(5):569–592, 2005. doi:10.1016/J.JSC.2004.12.004.

[DM22] Yaël Dillies and Bhavik Mehta. Formalising Szemerédi's regularity lemma in Lean. In June Andronick and Leonardo de Moura, editors, *13th International Conference on Interactive Theorem Proving, ITP 2022, August 7-10, 2022, Haifa, Israel*, volume 237 of *LIPIcs*, pages 9:1–9:19. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2022. doi:10.4230/LIPICS.ITP.2022.9.

[dMB07] Leonardo de Moura and Nikolaj S. Bjørner. Efficient e-matching for SMT solvers. In Frank Pfenning, editor, *Automated Deduction - CADE-21, 21st International Conference on Automated Deduction, Bremen, Germany, July 17-20, 2007, Proceedings*, volume 4603 of *Lecture Notes in Computer Science*, pages 183–198. Springer, 2007. doi:10.1007/978-3-540-73595-3_13.

[dMKA+15] Leonardo de Moura, Soonho Kong, Jeremy Avigad, Floris van Doorn, and Jakob von Raumer. The Lean theorem prover (sys-

tem description). In Amy P. Felty and Aart Middeldorp, editors, *Automated Deduction - CADE-25 - 25th International Conference on Automated Deduction, Berlin, Germany, August 1-7, 2015, Proceedings*, volume 9195 of *Lecture Notes in Computer Science*, pages 378–388. Springer, 2015. doi:10.1007/978-3-319-21401-6_26.

[dMU21] Leonardo de Moura and Sebastian Ullrich. The Lean 4 theorem prover and programming language. In André Platzer and Geoff Sutcliffe, editors, *Automated Deduction - CADE 28 - 28th International Conference on Automated Deduction, Virtual Event, July 12-15, 2021, Proceedings*, volume 12699 of *Lecture Notes in Computer Science*, pages 625–635. Springer, 2021. doi:10.1007/978-3-030-79876-5_37.

[DNS05] David Detlefs, Greg Nelson, and James B. Saxe. Simplify: a theorem prover for program checking. *J. ACM*, 52(3):365–473, 2005. doi:10.1145/1066100.1066102.

[Dow97] Mark Dowson. The ariane 5 software failure. *ACM SIGSOFT Softw. Eng. Notes*, 22(2):84, 1997. doi:10.1145/251880.251992.

[Ede97] Alan Edelman. The mathematics of the Pentium division bug. *SIAM Rev.*, 39(1):54–67, 1997. doi:10.1137/S0036144595293959.

[EMT+17] Burak Ekici, Alain Mebsout, Cesare Tinelli, Chantal Keller, Guy Katz, Andrew Reynolds, and Clark W. Barrett. SMTCoq: A plug-in for integrating SMT solvers into Coq. In Rupak Majumdar and Viktor Kuncak, editors, *Computer Aided Verification - 29th International Conference, CAV 2017, Heidelberg, Germany, July 24-28, 2017, Proceedings, Part II*, volume 10427 of *Lecture Notes in Computer Science*, pages 126–133. Springer, 2017. doi:10.1007/978-3-319-63390-9_7.

[EUR+17] Gabriel Ebner, Sebastian Ullrich, Jared Roesch, Jeremy Avigad, and Leonardo de Moura. A metaprogramming framework for formal verification. *Proc. ACM Program. Lang.*, 1(ICFP):34:1–34:29, 2017. doi:10.1145/3110278.

[FCW+22] Oliver Flatt, Samuel Coward, Max Willsey, Zachary Tatlock, and Pavel Panchekha. Small proofs from congruence closure. In Alberto Griggio and Neha Rungta, editors, *22nd Formal Methods in Computer-Aided Design, FMCAD 2022, Trento, Italy, October 17-21, 2022*, pages 75–83. IEEE, 2022. doi:10.34727/2022/ISBN.978-3-85448-053-2_13.

[FNB20] Anqi Fu, Balasubramanian Narasimhan, and Stephen Boyd. CVXR: An R package for disciplined convex optimization. *Journal of Statistical Software*, 94(14):1–34, 2020. doi:10.18637/jss.v094.i14.

[Fos19] Simon Foster. Hybrid relations in Isabelle/UTP. In Pedro Ribeiro and Augusto Sampaio, editors, *Unifying Theories of Programming - 7th International Symposium, UTP 2019, Dedicated to Tony Hoare on the Occasion of His 85th Birthday, Porto, Portugal, October 8, 2019, Proceedings*, volume 11885 of *Lecture Notes in Computer Science*, pages 130–153. Springer, 2019. doi:10.1007/978-3-030-31038-7_7.

[Fri14] Henrik A Friberg. The conic benchmark format: version 4 - technical reference manual, 2014. Available electronically at https://cblib.zib.de/doc/format4.pdf (accessed 2024-03-20).

[FyMS20] Simon Foster, Jonathan Julián Huerta y Munive, and Georg Struth. Differential Hoare logics and refinement calculi for hybrid systems with Isabelle/HOL. In Uli Fahrenberg, Peter Jipsen, and Michael Winter, editors, *Relational and Algebraic Methods in Computer Science - 18th International Conference, RAMiCS 2020, Palaiseau, France, April 8-11, 2020, Proceedings [postponed]*, volume 12062 of *Lecture Notes in Computer Science*, pages 169–186. Springer, 2020. doi:10.1007/978-3-030-43520-2_11.

[GAA+13] Georges Gonthier, Andrea Asperti, Jeremy Avigad, Yves Bertot, Cyril Cohen, François Garillot, Stéphane Le Roux, Assia Mahboubi, Russell O'Connor, Sidi Ould Biha, Ioana Pasca, Laurence Rideau, Alexey Solovyev, Enrico Tassi, and Laurent Théry. A machine-checked proof of the odd order theorem. In Sandrine Blazy,

Christine Paulin-Mohring, and David Pichardie, editors, *Interactive Theorem Proving - 4th International Conference, ITP 2013, Rennes, France, July 22-26, 2013. Proceedings*, volume 7998 of *Lecture Notes in Computer Science*, pages 163–179. Springer, 2013. doi:10.1007/978-3-642-39634-2_14.

[GB] Michael Grant and Stephen Boyd. CVX: MATLAB software for disciplined convex programming, version 2.2. http://cvxr.com/cvx (accessed 2024-03-20).

[GB08] Michael C. Grant and Stephen P. Boyd. Graph implementations for nonsmooth convex programs. In Vincent D. Blondel, Stephen P. Boyd, and Hidenori Kimura, editors, *Recent Advances in Learning and Control*, volume 371 of *Lecture Notes in Control and Information Sciences*, pages 95–110. Springer, 2008. doi:10.1007/978-1-84800-155-8_7.

[GB20] Michael C. Grant and Stephen P. Boyd. The CVX users' guide, release 2.2. https://cvxr.com/cvx/doc/CVX.pdf (accessed 2024-03-15), 2020.

[GBY06] Michael Grant, Stephen Boyd, and Yinyu Ye. Disciplined convex programming. *Global optimization: From theory to implementation*, pages 155–210, 2006.

[GKN15] Adam Grabowski, Artur Korniłowicz, and Adam Naumowicz. Four decades of Mizar - foreword. *J. Autom. Reason.*, 55(3):191–198, 2015. doi:10.1007/S10817-015-9345-1.

[GM05] Benjamin Grégoire and Assia Mahboubi. Proving equalities in a commutative ring done right in Coq. In Joe Hurd and Thomas F. Melham, editors, *Theorem Proving in Higher Order Logics, 18th International Conference, TPHOLs 2005, Oxford, UK, August 22-25, 2005, Proceedings*, volume 3603 of *Lecture Notes in Computer Science*, pages 98–113. Springer, 2005. doi:10.1007/11541868_7.

[GMM21] Alena Gusakov, Bhavik Mehta, and Kyle A. Miller. Formalizing Hall's marriage theorem in Lean. *CoRR*, abs/2101.00127, 2021. URL: https://arxiv.org/abs/2101.00127, arXiv:2101.00127.

[Gon07] Georges Gonthier. The four colour theorem: Engineering of a formal proof. In Deepak Kapur, editor, *Computer Mathematics, 8th Asian Symposium, ASCM 2007, Singapore, December 15-17, 2007. Revised and Invited Papers*, volume 5081 of *Lecture Notes in Computer Science*, page 333. Springer, 2007. doi:10.1007/978-3-540-87827-8_28.

[Gou22] Sébastien Gouëzel. A formalization of the change of variables formula for integrals in mathlib. In Kevin Buzzard and Temur Kutsia, editors, *Intelligent Computer Mathematics - 15th International Conference, CICM 2022, Tbilisi, Georgia, September 19-23, 2022, Proceedings*, volume 13467 of *Lecture Notes in Computer Science*, pages 3–18. Springer, 2022. doi:10.1007/978-3-031-16681-5_1.

[Gra05] Michael Charles Grant. *Disciplined convex programming*. PhD thesis, Stanford University, 2005.

[Gre11] Bogdan Grechuk. Lower semicontinuous functions. *Arch. Formal Proofs*, 2011, 2011. URL: https://www.isa-afp.org/entries/Lower_Semicontinuous.shtml.

[GS20] Emil Holm Gjørup and Bas Spitters. Congruence closure in cubical type theory. In *Workshop on Homotopy Type Theory/Univalent Foundations*. https://www.cs.au.dk/~spitters/Emil.pdf (accessed 2024-03-20), 2020.

[Har97] John Harrison. Floating point verification in HOL light: The exponential function. In Michael Johnson, editor, *Algebraic Methodology and Software Technology, 6th International Conference, AMAST '97, Sydney, Australia, December 13-17, 1997, Proceedings*, volume 1349 of *Lecture Notes in Computer Science*, pages 246–260. Springer, 1997. doi:10.1007/BFB0000475.

[Har99] John Harrison. A machine-checked theory of floating point arithmetic. In Yves Bertot, Gilles Dowek, André Hirschowitz, Christine Paulin-Mohring, and Laurent Théry, editors, *Theorem Proving in Higher Order Logics, 12th International Conference, TPHOLs'99,*

Nice, France, September, 1999, Proceedings, volume 1690 of *Lecture Notes in Computer Science*, pages 113–130. Springer, 1999. doi:10.1007/3-540-48256-3_9.

[Har07] John Harrison. Verifying nonlinear real formulas via sums of squares. In Klaus Schneider and Jens Brandt, editors, *Theorem Proving in Higher Order Logics, 20th International Conference, TPHOLs 2007, Kaiserslautern, Germany, September 10-13, 2007, Proceedings*, volume 4732 of *Lecture Notes in Computer Science*, pages 102–118. Springer, 2007. doi:10.1007/978-3-540-74591-4_9.

[Har09] John Harrison. HOL light: An overview. In Stefan Berghofer, Tobias Nipkow, Christian Urban, and Makarius Wenzel, editors, *Theorem Proving in Higher Order Logics, 22nd International Conference, TPHOLs 2009, Munich, Germany, August 17-20, 2009. Proceedings*, volume 5674 of *Lecture Notes in Computer Science*, pages 60–66. Springer, 2009. doi:10.1007/978-3-642-03359-9_4.

[HHM+10] Thomas C. Hales, John Harrison, Sean McLaughlin, Tobias Nipkow, Steven Obua, and Roland Zumkeller. A revision of the proof of the Kepler conjecture. *Discret. Comput. Geom.*, 44(1):1–34, 2010. doi:10.1007/S00454-009-9148-4.

[HJL12] Viktor Härter, Christian Jansson, and Marko Lange. VSDP: A MATLAB toolbox for verified semidefinite-quadratic-linear programming, 2012. Available electronically at https://www.tuhh.de/ti3/jansson/vsdp/VSDP2012Guide.pdf (accessed 2024-03-20).

[HJvE01] Timothy J. Hickey, Qun Ju, and Maarten H. van Emden. Interval arithmetic: From principles to implementation. *J. ACM*, 48(5):1038–1068, 2001. doi:10.1145/502102.502106.

[HL78] Gérard P. Huet and Bernard Lang. Proving and applying program transformations expressed with second-order patterns. *Acta Informatica*, 11:31–55, 1978. doi:10.1007/BF00264598.

[HT98] John Harrison and Laurent Théry. A skeptic's approach to combining HOL and Maple. *Journal of Automated Reasoning*, 21:279–294, 1998.

[HUW14] John Harrison, Josef Urban, and Freek Wiedijk. History of interactive theorem proving. In Jörg H. Siekmann, editor, *Computational Logic*, volume 9 of *Handbook of the History of Logic*, pages 135–214. Elsevier, 2014. doi:10.1016/B978-0-444-51624-4.50004-6.

[HvD20] Jesse Michael Han and Floris van Doorn. A formal proof of the independence of the continuum hypothesis. In Jasmin Blanchette and Catalin Hritcu, editors, *Proceedings of the 9th ACM SIGPLAN International Conference on Certified Programs and Proofs, CPP 2020, New Orleans, LA, USA, January 20-21, 2020*, pages 353–366. ACM, 2020. doi:10.1145/3372885.3373826.

[Imm15] Fabian Immler. Verified reachability analysis of continuous systems. In Christel Baier and Cesare Tinelli, editors, *Tools and Algorithms for the Construction and Analysis of Systems - 21st International Conference, TACAS 2015, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2015, London, UK, April 11-18, 2015. Proceedings*, volume 9035 of *Lecture Notes in Computer Science*, pages 37–51. Springer, 2015. doi:10.1007/978-3-662-46681-0_3.

[IT16] Fabian Immler and Christoph Traut. The flow of ODEs. In Jasmin Christian Blanchette and Stephan Merz, editors, *Interactive Theorem Proving - 7th International Conference, ITP 2016, Nancy, France, August 22-25, 2016, Proceedings*, volume 9807 of *Lecture Notes in Computer Science*, pages 184–199. Springer, 2016. doi:10.1007/978-3-319-43144-4_12.

[Jac95] Paul Bernard Jackson. *Enhancing the Nuprl proof development system and applying it to computational abstract algebra*. PhD thesis, Cornell University, 1995.

[Jan06] Christian Jansson. VSDP: A MATLAB software package for verified semidefinite programming. *NOLTA*, pages 327–330, 2006.

[JGK+17] Jean-Baptiste Jeannin, Khalil Ghorbal, Yanni Kouskoulas, Aurora C. Schmidt, Ryan W. Gardner, Stefan Mitsch, and André Platzer. A formally verified hybrid system for safe advisories in

the next-generation airborne collision avoidance system. *Int. J. Softw. Tools Technol. Transf.*, 19(6):717–741, 2017. doi:10.1007/S10009-016-0434-1.

[KGSJ17] Yanni Kouskoulas, Daniel Genin, Aurora C. Schmidt, and Jean-Baptiste Jeannin. Formally verified safe vertical maneuvers for non-deterministic, accelerating aircraft dynamics. In Mauricio Ayala-Rincón and César A. Muñoz, editors, *Interactive Theorem Proving - 8th International Conference, ITP 2017, Brasília, Brazil, September 26-29, 2017, Proceedings*, volume 10499 of *Lecture Notes in Computer Science*, pages 336–353. Springer, 2017. doi:10.1007/978-3-319-66107-0_22.

[KMM00] Matt Kaufmann, J. Strother Moore, and Panagiotis Manolios. *Computer-Aided Reasoning: An Approach*. Kluwer Academic Publishers, USA, 2000.

[Kud22] Yury Kudryashov. Formalizing the divergence theorem and the Cauchy integral formula in Lean. In June Andronick and Leonardo de Moura, editors, *13th International Conference on Interactive Theorem Proving, ITP 2022, August 7-10, 2022, Haifa, Israel*, volume 237 of *LIPIcs*, pages 23:1–23:19. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2022. doi:10.4230/LIPICS.ITP.2022.23.

[KY09] Phil N Klein and Neal E Young. Approximation algorithms for NP-hard optimization problems. In *Algorithms and Theory of Computation Handbook, Volume 1*, pages 953–972. Chapman and Hall/CRC, 2009.

[Lan20] Marko Lange. Verification methods for conic linear programming problems. *Nonlinear Theory and Its Applications, IEICE*, 11(3):327–358, 2020.

[Lew18] Robert Y Lewis. *Two Tools for Formalizing Mathematical Proofs*. PhD thesis, Carnegie Mellon University, 2018.

[Lew19] Robert Y. Lewis. A formal proof of Hensel's lemma over the p-adic integers. In Assia Mahboubi and Magnus O. Myreen, editors, *Proceedings of the 8th ACM SIGPLAN International Con-*

ference on Certified Programs and Proofs, CPP 2019, Cascais, Portugal, January 14-15, 2019, pages 15–26. ACM, 2019. doi:10.1145/3293880.3294089.

[LF23] Jannis Limperg and Asta Halkjær From. Aesop: White-box best-first proof search for Lean. In Robbert Krebbers, Dmitriy Traytel, Brigitte Pientka, and Steve Zdancewic, editors, *Proceedings of the 12th ACM SIGPLAN International Conference on Certified Programs and Proofs, CPP 2023, Boston, MA, USA, January 16-17, 2023*, pages 253–266. ACM, 2023. doi:10.1145/3573105.3575671.

[LLP17] Xinfu Liu, Ping Lu, and Binfeng Pan. Survey of convex optimization for aerospace applications. *Astrodynamics*, 1:23–40, 2017.

[LMS+10] Zhi-Quan Luo, Wing-Kin Ma, Anthony Man-Cho So, Yinyu Ye, and Shuzhong Zhang. Semidefinite relaxation of quadratic optimization problems. *IEEE Signal Process. Mag.*, 27(3):20–34, 2010. doi:10.1109/MSP.2010.936019.

[Lof04] Johan Lofberg. YALMIP: A toolbox for modeling and optimization in MATLAB. In *2004 IEEE international conference on robotics and automation (IEEE Cat. No. 04CH37508)*, pages 284–289. IEEE, 2004.

[LW22] Robert Y. Lewis and Minchao Wu. A bi-directional extensible interface between Lean and Mathematica. *J. Autom. Reason.*, 66(2):215–238, 2022. doi:10.1007/S10817-021-09611-1.

[McC92] William McCune. Experiments with discrimination-tree indexing and path indexing for term retrieval. *J. Autom. Reason.*, 9(2):147–167, 1992. doi:10.1007/BF00245458.

[MGVP17] Stefan Mitsch, Khalil Ghorbal, David Vogelbacher, and André Platzer. Formal verification of obstacle avoidance and navigation of ground robots. *Int. J. Robotics Res.*, 36(12):1312–1340, 2017. doi:10.1177/0278364917733549.

[Mil72] Robin Milner. *Logic for computable functions description of a machine implementation.* Computer Science Department, Stanford University, 1972.

[NAE23] Wojciech Nawrocki, Edward W. Ayers, and Gabriel Ebner. An extensible user interface for Lean 4. In Adam Naumowicz and René Thiemann, editors, *14th International Conference on Interactive Theorem Proving, ITP 2023, July 31 to August 4, 2023, Białystok, Poland*, volume 268 of *LIPIcs*, pages 24:1–24:20. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2023. doi:10.4230/LIPICS.ITP.2023.24.

[Nec97] George C. Necula. Proof-carrying code. In Peter Lee, Fritz Henglein, and Neil D. Jones, editors, *Conference Record of POPL'97: The 24th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, Papers Presented at the Symposium, Paris, France, 15-17 January 1997*, pages 106–119. ACM Press, 1997. doi:10.1145/263699.263712.

[NN94] Yurii E. Nesterov and Arkadii Nemirovskii. *Interior-point polynomial algorithms in convex programming*, volume 13 of *Siam studies in applied mathematics*. SIAM, 1994. doi:10.1137/1.9781611970791.

[NO80] Greg Nelson and Derek C. Oppen. Fast decision procedures based on congruence closure. *J. ACM*, 27(2):356–364, 1980. doi:10.1145/322186.322198.

[NO05] Robert Nieuwenhuis and Albert Oliveras. Proof-producing congruence closure. In Jürgen Giesl, editor, *Term Rewriting and Applications, 16th International Conference, RTA 2005, Nara, Japan, April 19-21, 2005, Proceedings*, volume 3467 of *Lecture Notes in Computer Science*, pages 453–468. Springer, 2005. doi:10.1007/978-3-540-32033-3_33.

[Nor09] Ulf Norell. Dependently typed programming in Agda. In Andrew Kennedy and Amal Ahmed, editors, *Proceedings of TLDI'09: 2009 ACM SIGPLAN International Workshop on Types in Languages Design and Implementation, Savannah, GA, USA, January 24, 2009*, pages 1–2. ACM, 2009. doi:10.1145/1481861.1481862.

[NPW02] Tobias Nipkow, Lawrence C. Paulson, and Markus Wenzel. *Isabelle/HOL - A Proof Assistant for Higher-Order Logic*, volume 2283 of *Lecture Notes in Computer Science*. Springer, 2002. doi:10.1007/3-540-45949-9.

[Opt20] Gurobi Optimization. Gurobi optimizer reference manual, version 9.0. https://www.gurobi.com/wp-content/plugins/hd_documentations/documentation/9.0/refman.pdf (accessed 2024-03-20), 2020.

[OSRSC99] Sam Owre, Natarajan Shankar, John M Rushby, and David WJ Stringer-Calvert. Pvs language reference. *Computer Science Laboratory, SRI International, Menlo Park, CA*, 1999.

[Pap07] Christos H. Papadimitriou. *Computational complexity*. Academic Internet Publ., 2007.

[Pas11] Grant Olney Passmore. *Combined Decision Procedures for Nonlinear Arithmetics, Real and Complex*. PhD thesis, University of Edinburgh, 2011.

[Pau83] Lawrence Paulson. Tactics and tacticals in Cambridge LCF. Technical Report UCAM-CL-TR-39, University of Cambridge, Computer Laboratory, July 1983. URL: https://www.cl.cam.ac.uk/techreports/UCAM-CL-TR-39.pdf, doi:10.48456/tr-39.

[Pau89] Lawrence C. Paulson. The foundation of a generic theorem prover. *J. Autom. Reason.*, 5(3):363–397, 1989. doi:10.1007/BF00248324.

[PJP07] Stephen Prajna, Ali Jadbabaie, and George J. Pappas. A framework for worst-case and stochastic safety verification using barrier certificates. *IEEE Trans. Autom. Control.*, 52(8):1415–1428, 2007. doi:10.1109/TAC.2007.902736.

[PQ08] André Platzer and Jan-David Quesel. Keymaera: A hybrid theorem prover for hybrid systems (system description). In Alessandro Armando, Peter Baumgartner, and Gilles Dowek, editors, *Automated Reasoning, 4th International Joint Conference, IJCAR 2008, Sydney, Australia, August 12-15, 2008, Proceedings*, volume 5195 of

Lecture Notes in Computer Science, pages 171–178. Springer, 2008. doi:10.1007/978-3-540-71070-7_15.

[PSWT15] Pavel Panchekha, Alex Sanchez-Stern, James R. Wilcox, and Zachary Tatlock. Automatically improving accuracy for floating point expressions. In David Grove and Stephen M. Blackburn, editors, *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation, Portland, OR, USA, June 15-17, 2015*, pages 1–11. ACM, 2015. doi:10.1145/2737924.2737959.

[PTA+24] Arthur Paulino, Damiano Testa, Edward Ayers, Evgenia Karunus, Henrik Böving, Jannis Limperg, Siddhartha Gadgil, and Siddharth Bhat. *Metaprogramming in Lean 4*, 2024. https://github.com/leanprover-community/lean4-metaprogramming-book (accessed 2024-02-27).

[Roc70] R. Tyrrell Rockafellar. *Convex Analysis*. Princeton Landmarks in Mathematics and Physics. Princeton University Press, 1970.

[Rum98] Siegfried M. Rump. INTLAB - interval laboratory. In Tibor Csendes, editor, *Developments in Reliable Computing, International Symposium on Scientific Computing, Computer Arithmetic, and Validated Numerics, SCAN 1998, Szeged, Hungary, September 22-25, 1998*, pages 77–104. Springer, 1998. doi:10.1007/978-94-017-1247-7_7.

[Rum06] Siegfried M Rump. Verification of positive definiteness. *BIT Numerical Mathematics*, 46(2):433–452, 2006.

[RVS18] Pierre Roux, Yuen-Lam Voronin, and Sriram Sankaranarayanan. Validating numerical semidefinite programming solvers for polynomial invariants. *Formal Methods Syst. Des.*, 53(2):286–312, 2018. doi:10.1007/S10703-017-0302-Y.

[SBB+89] Leon Sterling, Alan Bundy, Lawrence Byrd, Richard A. O'Keefe, and Bernard Silver. Solving symbolic equations with PRESS. *J. Symb. Comput.*, 7(1):71–84, 1989. doi:10.1016/S0747-7171(89)80007-0.

[SDAT15] Ons Seddiki, Cvetan Dunchev, Sanaz Khan Afshar, and Sofiène Tahar. Enabling symbolic and numerical computations in HOL light. In Manfred Kerber, Jacques Carette, Cezary Kaliszyk, Florian Rabe, and Volker Sorge, editors, *Intelligent Computer Mathematics - International Conference, CICM 2015, Washington, DC, USA, July 13-17, 2015, Proceedings*, volume 9150 of *Lecture Notes in Computer Science*, pages 353–358. Springer, 2015. doi:10.1007/978-3-319-20615-8_27.

[SdM16] Daniel Selsam and Leonardo de Moura. Congruence closure in intensional type theory. In Nicola Olivetti and Ashish Tiwari, editors, *Automated Reasoning - 8th International Joint Conference, IJCAR 2016, Coimbra, Portugal, June 27 - July 2, 2016, Proceedings*, volume 9706 of *Lecture Notes in Computer Science*, pages 99–115. Springer, 2016. doi:10.1007/978-3-319-40229-1_8.

[SL91] J.J.E. Slotine and W. Li. *Applied Nonlinear Control*. Prentice Hall, 1991.

[SLD17] Daniel Selsam, Percy Liang, and David L. Dill. Developing bug-free machine learning systems with formal mathematics. In Doina Precup and Yee Whye Teh, editors, *Proceedings of the 34th International Conference on Machine Learning, ICML 2017, Sydney, NSW, Australia, 6-11 August 2017*, volume 70 of *Proceedings of Machine Learning Research*, pages 3047–3056. PMLR, 2017. URL: http://proceedings.mlr.press/v70/selsam17a.html.

[SN08] Konrad Slind and Michael Norrish. A brief overview of HOL4. In Otmane Aït Mohamed, César A. Muñoz, and Sofiène Tahar, editors, *Theorem Proving in Higher Order Logics, 21st International Conference, TPHOLs 2008, Montreal, Canada, August 18-21, 2008. Proceedings*, volume 5170 of *Lecture Notes in Computer Science*, pages 28–32. Springer, 2008. doi:10.1007/978-3-540-71067-7_6.

[STL11] Michael Stepp, Ross Tate, and Sorin Lerner. Equality-based translation validator for LLVM. In Ganesh Gopalakrishnan and Shaz Qadeer, editors, *Computer Aided Verification - 23rd International*

Conference, CAV 2011, Snowbird, UT, USA, July 14-20, 2011. Proceedings, volume 6806 of *Lecture Notes in Computer Science*, pages 737–742. Springer, 2011. doi:10.1007/978-3-642-22110-1_59.

[Stu99] Jos F Sturm. Using SeDuMi 1.02, a MATLAB toolbox for optimization over symmetric cones. *Optimization methods and software*, 11(1-4):625–653, 1999.

[SUdM20] Daniel Selsam, Sebastian Ullrich, and Leonardo de Moura. Tabled typeclass resolution. *CoRR*, abs/2001.04301, 2020. URL: https://arxiv.org/abs/2001.04301, arXiv:2001.04301.

[Tar75] Robert Endre Tarjan. Efficiency of a good but not linear set union algorithm. *J. ACM*, 22(2):215–225, 1975. doi:10.1145/321879.321884.

[The20] The mathlib community. The Lean mathematical library. In *Proceedings of the 9th ACM SIGPLAN International Conference on Certified Programs and Proofs, CPP 2020, New Orleans, LA, USA, January 20-21, 2020*, pages 367–381, 2020. doi:10.1145/3372885.3373824.

[TSTL09] Ross Tate, Michael Stepp, Zachary Tatlock, and Sorin Lerner. Equality saturation: a new approach to optimization. In Zhong Shao and Benjamin C. Pierce, editors, *Proceedings of the 36th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2009, Savannah, GA, USA, January 21-23, 2009*, pages 264–276. ACM, 2009. doi:10.1145/1480881.1480915.

[UdM19] Sebastian Ullrich and Leonardo de Moura. Counting immutable beans: reference counting optimized for purely functional programming. In Jurriën Stutterheim and Wei-Ngan Chin, editors, *IFL '19: Implementation and Application of Functional Languages, Singapore, September 25-27, 2019*, pages 3:1–3:12. ACM, 2019. doi:10.1145/3412932.3412935.

[UdM20] Sebastian Ullrich and Leonardo de Moura. Beyond notations: Hygienic macro expansion for theorem proving languages. In Nicolas

Peltier and Viorica Sofronie-Stokkermans, editors, *Automated Reasoning - 10th International Joint Conference, IJCAR 2020, Paris, France, July 1-4, 2020, Proceedings, Part II*, volume 12167 of *Lecture Notes in Computer Science*, pages 167–182. Springer, 2020. doi:10.1007/978-3-030-51054-1_10.

[UdM22] Sebastian Ullrich and Leonardo de Moura. 'do' unchained: embracing local imperativity in a purely functional language (functional pearl). *Proc. ACM Program. Lang.*, 6(ICFP):512–539, 2022. doi:10.1145/3547640.

[Ull23] Sebastian Andreas Ullrich. *An Extensible Theorem Proving Frontend*. PhD thesis, Karlsruher Instituts für Technologie (KIT), 2023.

[UMZ+14] Madeleine Udell, Karanveer Mohan, David Zeng, Jenny Hong, Steven Diamond, and Stephen P. Boyd. Convex optimization in Julia. In Jiahao Chen, Alan Edelman, Wade Shen, and Andy Terrel, editors, *Proceedings of the 1st First Workshop for High Performance Technical Computing in Dynamic Languages, HPTCDL '14, New Orleans, Louisiana, USA, November 16-21, 2014*, pages 18–28. IEEE Computer Society, 2014. doi:10.1109/HPTCDL.2014.5.

[VNL+21] Alexa VanHattum, Rachit Nigam, Vincent T. Lee, James Bornholt, and Adrian Sampson. Vectorization for digital signal processors via equality saturation. In Tim Sherwood, Emery D. Berger, and Christos Kozyrakis, editors, *ASPLOS '21: 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Virtual Event, USA, April 19-23, 2021*, pages 874–886. ACM, 2021. doi:10.1145/3445814.3446707.

[WHS+20] Yisu Remy Wang, Shana Hutchison, Dan Suciu, Bill Howe, and Jonathan Leang. SPORES: sum-product optimization via relational equality saturation for large scale linear algebra. *Proc. VLDB Endow.*, 13(11):1919–1932, 2020. URL: http://www.vldb.org/pvldb/vol13/p1919-wang.pdf.

[WNW+21] Max Willsey, Chandrakana Nandi, Yisu Remy Wang, Oliver Flatt, Zachary Tatlock, and Pavel Panchekha. egg: Fast and extensible

equality saturation. *Proc. ACM Program. Lang.*, 5(POPL):1–29, 2021. doi:10.1145/3434304.

[YFN+10] Makoto Yamashita, Katsuki Fujisawa, Kazuhide Nakata, Maho Nakata, Mituhiro Fukuda, Kazuhiro Kobayashi, and Kazushige Goto. A high-performance software package for semidefinite programs: SDPA 7, 2010.

[YPW+21] Yichen Yang, Phitchaya Mangpo Phothilimthana, Yisu Remy Wang, Max Willsey, Sudip Roy, and Jacques Pienaar. Equality saturation for tensor graph superoptimization. In Alex Smola, Alex Dimakis, and Ion Stoica, editors, *Proceedings of Machine Learning and Systems 2021, MLSys 2021, virtual, April 5-9, 2021*. mlsys.org, 2021.

[YSG+23] Kaiyu Yang, Aidan M. Swope, Alex Gu, Rahul Chalamala, Peiyang Song, Shixing Yu, Saad Godil, Ryan J. Prenger, and Animashree Anandkumar. LeanDojo: Theorem proving with retrieval-augmented language models, 2023. URL: http://papers.nips.cc/paper_files/paper/2023/hash/4441469427094f8873d0fecb0c4e1cee-Abstract-Datasets_and_Benchmarks.html.

[ZWF+23] Yihong Zhang, Yisu Remy Wang, Oliver Flatt, David Cao, Philip Zucker, Eli Rosenthal, Zachary Tatlock, and Max Willsey. Better together: Unifying Datalog and equality saturation. *Proc. ACM Program. Lang.*, 7(PLDI):468–492, 2023. doi:10.1145/3591239.