Programming Language Tools and Techniques for Computational Fabrication

Chandrakana Nandi

A dissertation
submitted in partial fulfillment of the
requirements for the degree of

Doctor of Philosophy

University of Washington

2021

Reading Committee:

Dan Grossman, Chair

Zachary Tatlock, Chair

Adriana Schulz

Eva Darulova

Program Authorized to Offer Degree:
Computer Science and Engineering

University of Washington

Abstract

Programming Language Tools and Techniques for Computational Fabrication

Chandrakana Nandi

Co-Chairs of the Supervisory Committee:
Dan Grossman

Zachary Tatlock

Recent democratization of fabrication techniques has revolutionized manufacturing. Desktop versions of devices like 3D printers, laser cutters, and CNC mills are now available at affordable prices making them increasingly common among end-users, hobbyists, and non-experts. While the reduced hardware cost of manufacturing devices has created new opportunity for casual makers to innovate and explore, the corresponding software pipeline has not advanced as dramatically. Design tools, mesh editing software, and CAD frameworks are developed primarily for trained experts and thus pose a steep learning curve for beginners. Even with crowd-sourced solutions, working with pre-existing models requires substantial expertise. Together with the fact that the fabrication process (typically multi-stage) itself is slow and hard to debug, these design level challenges frequently lead to delayed results, failed prints, wasted resources, and frustration.

This thesis focuses on some of these challenges by viewing the entire computational fabrication pipeline as a compiler. First, it provide formal semantics to commonly used representations for 3D modeling. This formal foundation aids the development of a decompiler that automatically reverse engineers high-level computer-aided designs from low-level representations by recovering latent structure. An equality saturation based second decompilation phase discovers even high-level representations using *dynamic rewrites* and *inverse transformations*. A second application of dynamic rewrites in the domain of carpentry shows wider applicability of this approach for developing optimizing compilers for different kinds of fabrication techniques. Both these equality saturation based approaches require manually writing the rewrites which itself is tedious and error-prone. A novel framework for automatically inferring rewrites shows how the entire pipeline can be automated, and be applied to more traditional compilers.

TABLE OF CONTENTS

Page

Chapter 1: Introduction . . . 1
- 1.1 Background . . . 2
- 1.2 My Work . . . 4
- 1.3 Beyond Computational Fabrication . . . 5
- 1.4 Delta Between Published Papers and this Thesis . . . 5

Chapter 2: Compilers for Geometric Languages . . . 7
- 2.1 Formalizing CAD and Mesh . . . 8
- 2.2 λCAD Compiler . . . 16
- 2.3 Compiler Validity . . . 24
- 2.4 Compiler Correctness . . . 28
- 2.5 Implementation and Challenges . . . 31
- 2.6 Related work . . . 35
- 2.7 Conclusions . . . 38

Chapter 3: Decompiling Mesh to CSG . . . 39
- 3.1 Synthesis Example . . . 40
- 3.2 Specifying Reverse Compilation . . . 41
- 3.3 Decompilation Algorithm . . . 45
- 3.4 Evaluation . . . 51
- 3.5 Related work . . . 57
- 3.6 Conclusions . . . 60

Chapter 4: CSG to Structured CAD with Equality Saturation . . . 61
- 4.1 Second Stage Decompilation . . . 63
- 4.2 Shrinking by Rerolling Loops . . . 65
- 4.3 E-graphs and CAD Equality Saturation . . . 69

4.4 Inverse Transformations 74
4.5 Evaluation 81
4.6 Related Work 86
4.7 Conclusions 87

Chapter 5: Carpentry Compiler and Equality Saturation 89
5.1 Overview 91
5.2 Language and Compiler 95
5.3 Fabrication Optimization 99
5.4 Evaluation 106
5.5 Related Work 112
5.6 Conclusions 114

Chapter 6: Rules Inference for and by Equality Saturation 116
6.1 Implementing Rewrite Systems 117
6.2 Ruler's Algorithm 118
6.3 Evaluation 127
6.4 Limitations and Future Work 140
6.5 Related Work 141
6.6 Conclusion 144

Chapter 7: Future Work and Conclusions 145
7.1 Limitations 146
7.2 Future Work 146

ॐ त्र्यम्बकं यजामहे सुगन्धिं पुष्टिवर्धनम् |
उर्वारुकमिव बन्धनान्मृत्योर्मुक्षीय माऽमृतात् ||

I must not fear.
Fear is the mind-killer.
Fear is the little-death that brings total obliteration.
I will face my fear.
I will permit it to pass over me and through me.
And when it has gone past, I will turn the inner eye to see its path.
Where the fear has gone there will be nothing.
Only I will remain.

— Frank Herbert, Dune

ACKNOWLEDGMENTS

I have been incredibly fortunate in grad school; the work in this thesis is the result of collaborating with many incredible mentors and friends.

First of all, I would like to thank my advisors, Dan Grossman and Zachary Tatlock. Being advised by the two of you is one of the best things that has happened to me. The funny thing is that when I applied to grad school, you were the only two core PLSE faculty at UW I had not mentioned in my research statement. The fact that I ended up being advised by you is therefore the best miracle of my life! You are absolutely the best advisors ever and I hope to be like you when I grow up.

Dan, I cannot thank you enough for taking me in as a student back in 2016. Having your support that year meant a lot and I hope you know that. Thank you for allowing me to pursue this research direction. There have been countless occasions when your advice and support has been critical. I will share three of them here. First, back when I was just starting Reincarnate, you insisted that I write the entire CAD compiler for it from scratch in OCaml. I was not sure if that was a good idea back then and even recall pushing back, but in hindsight, I think my research career would not have been the same if I hadn't. Thank you for being assertive yet patient with me! Second, during the pandemic when I decided to start Ruler and pause Gayatri, I was worried you would not be happy about me starting a new project so late. Your responded by saying that no matter what I chose to do, you will support my decision as long as I made reasonable progress because you want me to be happy and excited about the project. That was extremely important to me and I will always be grateful to you for it. Third, you insisted that I put in the effort to make this thesis as good as I can and helped me put together the missing pieces. Thank you for doing that — because of your advice I am proud of this document. In short, your wisdom, infinite patience and support, and of course countless dad jokes were critical to the work presented in this thesis! Thank you for so genuinely caring for your students and always prioritizing their needs.

Zach, I don't have words to express how grateful I am to you, so let me first just thank you for *everything*. I am so glad I stopped by your office back in November 2016 for a "10 minute" chat about 3D printing. My entire PhD work started from that 10 min chat! You have the most amazing and unique approach towards picking and executing research projects which I hope to carry forward and emulate in the rest of my career. Your incredible research mentorship, deep insights, contagious enthusiasm, and endless support has been invaluable to me. Not only have I learned a *ton* about PL research from you, but you have also helped me develop several — what I like to call —"auxiliary" grad school skills: making great talks, doing great evaluations, mentoring junior students, organizing research notes and ideas, effectively collaborating with others, and most importantly being fearless when it comes to pursuing new ideas. You have made every paper deadline in grad school memorable — I don't recall a single one where you were not there working with us. You have this amazing ability to quickly identify every student's "super power" and help them use it to accomplish their goals to their best ability. I don't know if you remember, but just when we started Reincarnate you told me that my super power was my persistence (you literally used the term "super power")! You have no idea how much I needed to hear that, especially during that year. Thank you for believing in me, sometimes more than I did! One of the hardest things about research is to be OK with uncertainty and keep going no matter what. Your enthusiasm and optimism always made this much easier that it would otherwise be. Thank you also for making PLSE the most inclusive, fun, and friendly lab I could have imagined possible. PLSE has been like a family to me and it's because of all the effort you have put into establishing the culture, not to mention all the hand-picked lab snacks, deadline food, PLSE hangouts, and so much more!

I am grateful to the rest of my committee that includes Adriana Schulz, Nadya Peek, and Eva Darulova for their time and feedback. Thank you Adriana for introducing me to the carpentry compiler team which led to multiple awesome collaborations. Thank you Nadya for your feedback on the story of my PhD work. I am glad I got to know the wonderful team at Machine Agency. Eva, thank you for being such an awesome collaborator and friend. You are one of the first PL people I met back in our EPFL days and I am not only grateful to you for your collaboration on Szalinski and inviting me to visit your awesome group at MPI-SWS, but also for being a great friend and mentor. You were there for my MS thesis defense! You helped me navigate the first PL/FM conference (FMCAD 2014!) I ever attended. You even gave me advice on picking grad school back in 2014 over a cup of coffee in EPFL's BC building! I won't forget any of it :)

Huge thanks to all my collaborators on projects that are covered in this thesis or are related to my "PL for Fabrication" research agenda: Max Willsey, Amy Zhu, Brett Saiki, Remy Wang, Oliver Flatt, Pavel Panchkekha, Jasper Tran O'Leary, James Wilcox, Molly Carton, Adam Anderson, Taylor Blau, Haisen Zhao, Chenming Wu, Anat Caspi, and Jeff Lipton. As I have always said, paper deadlines are some of my favorite times and you all are big reason for that. I have enjoyed staying up late in the lab with you, hanging out and generating results in the last minute, and brain storming about various aspects of research for hours together.

Thank you also to Adrian Sampson, Kathryn McKinley, Todd Mytkowicz, and Mike Ernst for your collaboration on earlier projects in grad school.

I am deeply grateful to all my other friends in PLSE. A very special thanks to Amanda Swearngin for numerous movie nights, dinners, and hangouts throughout grad school. I am lucky to have you as one of my closest friends. Thank you Talia for giving me my first tour of Seattle downtown, and for sending me Edible Arrangements when I was very sick in my first year! It really meant a lot to me. Thank you Martin Kellogg for introducing me to D&D and helping me discover and accept my chaotic evil personality ;)

Thank you Max for introducing me to Rust and for organizing and leading various fun things like PoCSci and crossbot! Thank you James Wilcox for being my accountability buddy! I actually managed to finish my thesis thanks to accountability-coop and all its participants! Thank you Pavel for being a great grad student mentor and for teaching me so many things about the US! Thank you Steven Lyubomirsky for writing and directing some of the best Holiday skits and giving me roles in them. Also thanks to you and Bill Zorn for very generously allowing me to use the servers at short notice during deadlines. Thank you Doug for your friendship and I guess also for the pranks (foot fingers, aardvarks, Steve Wonders...). Thank you Gus Smith and Joe Redmon for inviting me to awesome parties at your houses! Thank you John Toman for helping with random PL things at super short notice and for being such a good friend. Thank you Jared Roesch for your willingness to talk about basically any topic, and for cooking some of the most delicious food ever.

Thank you Sarah Chasins for numerous dinners and fun conversations. I am so glad you and Mangpo moved to UW and am very happy that we are friends. Thank you Eunice Jun for teaching me about fashion :) I will always cherish our hangouts and of course your enthusiastic Jai Shri Krishna! Thank you Amy Zhu for your unique and awesome sense of humor and for teaching me some of the most obscure things ever. Thank you Sami Davies for being one of the most thoughtful people I know. Thank you for bringing freshly baked goods before deadlines and for inviting me over for so many memorable parties. Thank you James Bornholt for grad school / career advice on various occasions.

Thanks to Stuart Pernsteiner and Chenglong Wang for fun days back in our old office. Many thanks to Konne Weitz, Eric Mullen, Mangpo Phothilimthana, Gus Smith, Maaz Ahmad, Rashmi Mudduluru, Julie Newcomb, Sam Kaufman, Krzystof Drewniak, Yu Feng, Ali Sinan Koksal, Sam Elliot, Alex Sanchez Stern, Ben Kushigian, Sorawee Porncharoenwase, Jacob van Geffen, and the Sad+ (Josh, Logan, Marisa, Altan) for fun times over the years. Thanks to every current and past member of PLSE for contributing to "lab culture", making the lab a happy place for me, and being incredibly supportive.

Thank to all my friends outside of PLSE! Special shout-out to Srini, one of the first people I met at UW. Srini, thanks for being so generous with your time and for the countless rides home in the middle of the night. You and your BMW will always be remembered :) Thank you Swati for always being my laughing companion — I can never forget those two dinners at Rancho Bravo and Chilis. I am laughing now just thinking about them ;) Thank you Bindita, Deepali, Liang, Koosha, Dhruv, Venkatesh, Pratyush, Ben Jones, Kiana Ehsani for your friendship and help over the years.

Thank you to the Allen School staff: Elise Dorough, Anna Wehowsky, Amanda Robles, Sophie, Alex Lefort, Mel Kadenko, Sandy Kaplan, and everyone else for their help on numerous occasions. Thank you Franklin for making us feel safe late at night and always saying "Hi", and thank you Eunhee for keeping the lab clean—will miss seeing you early in the mornings!

This acknowledgement would be incomplete without mentioning the Race Condition Running (RCR) Club. I am incredibly grateful to my advisor, Zach, for establishing RCR, inviting me to join and helping me (and so many others!) become runners. Some of my closest friends in grad school are from RCR. Thanks to my marathon training buddies, Max, Ellis, and Zach, with whom I have probably ran thousands of miles at this point. The conversations we have during RCR runs are some of my favorites — including countless "research meetings" with Max and Zach, arguments about how egraphs are more than just union-find (to the annoyance of the rest). Our endless banters, Ellis's carefree singing sessions post-mile-15, sharing delicious desserts with Nick after runs, are some of my favorite memories. Special thanks to Ellis for organizing Ragnar together and for leading many other running events. Also thanks to Ellis for starting RCR's brunch reviews! Thanks to Remy for always keeping things exciting when it comes to races, and to Trang for being a spectacular running companion. Thanks to our Ragnar 2019 team (Max, Sami, Ellis, Zach, Trang, Willie, Remy, Talia, Yu, Maureen, John Leo) and our Ragnar 2021 team (Max, Sami, Ellis, Zach, Gus, Eric, James Wilcox, Aishwarya, Daniel, Yuxuan, Nick). These Ragnars are truly two of my favorite grad school adventures. You all were such a joy to run and hangout with. Specifically, Olive Garden with Van 2 (both years!) will always have a special place in my heart, as will the rest of the insane adventures we had. Thanks to Yuxuan and Nick for organizing short runs to help recruit newer members to RCR. Thanks to Amanda Robles and Zach for supporting me during my first "accidental" 17 mile run, and to everyone else who I have ever run with. Thanks to RCR, I also got to know Aishwarya and Ethan better and will always remember the dinners, desserts, and boba you guys have so generously shared with me!

I would like to thank several mentors early in my career. Thank you Prof. Joseph Sifakis for allowing me to join the RiSD lab at EPFL. You introduced me to formal methods and gave me the opportunity to learn so much. Your group was simply amazing and I was fortunate to be part of it. Thank you for encouraging me to pursue a PhD. Thank you Prof. Viktor Kuncak for being my MS thesis advisor and allowing me to attend my first ever PL/FM conference (FMCAD 2014)! Thank you Manuel Oriol and Aurelien Monot, my mentors at ABB Research, for helping me publish my first paper ever! Thank you Prof. Auke Ijspeert for one of my earliest research experiences and for making me feel so welcome at BIOROB. Thank you all my friends at EPFL, ETH, and ABB — Farzaneh, Hamedeh, Janine, Ursula, Fabian, Artyom, Greg, Camille Frejaville, Samuel Humeau, Xavier, Andrea Zanelli and Andrea Stefani, Anna, and everyone else I had a chance to hangout with. Thank you all my professors at BHU (especially Prof. S. K. Basu, Prof. Umesh, and Prof. S. K. Singh), where it all started. I will always remember Vishwanath Temple (VT) where I first learned the power of meditation. Thank you also to Swati Gupta — you are still my favorite roommate and studying companion.

I am incredibly fortunate to have a supportive family. Thank you Ma, Bapi for teaching me the value of hard work and for teaching me to do my best in whatever I choose to do. Ma, you have always supported all my decisions and always cheered for me. Thank you for being the best mom, an incredible human, and one of my closest friends! Bapi, I am grateful that you guided me early on in my career and encouraged me to pursue a PhD. You were right, this was one of the best decisions ever! Many many thanks to Joy for being the most awesome brother in the world. I can never thank you enough for all the help you have offered over the years whether it be for moving, or finding a new place to live in, or literally anything. You are just amazing. Thank you Yue for being a wonderful older sister to me! Having you and Joy together with Lily, Harry, and Hui in Seattle has been a true blessing. Thank you Luna for being the best sister and one of my best friends. Your playlists, no matter how terrible they are, always make me smile :) Thank you to my grandparents for your love and support. A special thank you to Thamma for being a fierce critic and pushing me to be better from a very early age. I wish you all were still around today. Thank you to my entire family (including Pisai and Pisimoni!) for all your patience, love, and understanding over the years.

And finally, thank you dearest Puma. You came to my family during a rough time and magically made everything better. As superstitious as it sounds, I always thought all good things came from having you in my life. I miss you every day and wish you did not leave so soon and so suddenly. Thank you for teaching me the power of humility and patience, and for your unconditional love. You have made me a better person.

DEDICATION

to Puma

Chapter 1

INTRODUCTION

Desktop-class 3D printers, laser cutters, and Computer Numerical Control (CNC) mills are now available to millions of people. [1] The potential social benefits of broad access to these technologies have been much hyped, but the current reality is that desktop-class hardware and tools are often substantially less accurate, fast, and reliable than their industrial counterparts where these processes are managed by trained professionals and require substantial expertise. Without significant improvements, democratized manufacturing practice is bound to fall short of its ambitious promise. Many compiler and programming language (PL) techniques can address challenges in manufacturing; developing Computer Aided Design (CAD) models and editing existing objects are analogous to synthesis; whereas generating accurate, efficient tool paths or instruction sets from a CAD model is analogous to optimizing compilation.

The work presented in this thesis is based on the fundamental insight that

viewing the computational fabrication pipeline as a compiler enables the use of (1) modern PL theory to guide the systematic development and formal reasoning of this pipeline, and (2) reverse compilation and compiler optimization techniques to build novel program synthesizers and reliable compilers.

To validate this claim, this thesis presents several new tools that I have built with my collaborators. Additionally, we have found that these ideas are applicable beyond computational fabrication and can be used to build better program synthesis tools and optimizing compilers for other domains as well.

The rest of this chapter is organized as follows: Section 1.1 provides an overview of how computational fabrication pipelines work, Section 1.2 describes how we applied PL and compiler techniques to various stages of the pipeline, and Section 1.3 discusses how the work of this thesis generalizes to domains beyond computational fabrication. Section 1.4 lists new content that appears only in this thesis and not in any published / under-review manuscripts.

[1] https://www.gartner.com/en/documents/3132417

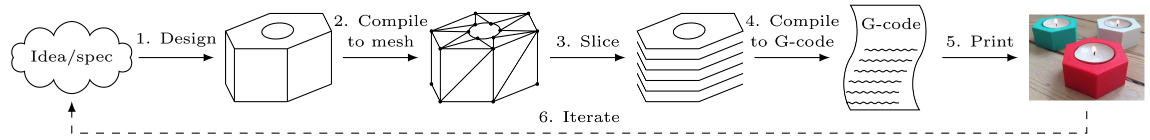


Figure 1.1: *The 3D Printing Development Cycle.* To 3D print a device: (1) An engineer first designs a 3D model using standard CAD tools (e.g., Openscad OpenScad [2019], Rhino Rhinoceros [2018], SolidWorks Solidworks [2018], or SketchUp SketchUp [2018]). (2) This model is compiled into a low-level triangle mesh representation, (3) The mesh is then *sliced* into horizontal 2D layers, (4) The layers are compiled into a sequence of low-level *G-code* commands that corresponds to basic actions the printer can take (move the print head, start/stop extrusion, lower the build plate, etc.). (5) The printer then directly executes the G-code, producing a physical device.

1.1 Background

This section first provides necessary background on how typical computational fabrication pipelines work, and then provides a brief overview of E-graphs Nelson [1980] and equality saturation Tate et al. [2009], Willsey et al. [2021] which several later chapters rely on.

1.1.1 Computational Fabrication

To understand how PL and compiler techniques can be developed for computational fabrication, this section gives an overview of a typical fabrication pipeline by using additive manufacturing as an example. Concretely, it describes how "cartesian fused filament fabrication" (FFF) printers, one of the most common and affordable models, work. Later chapters in the thesis will show how the techniques developed in this thesis apply to not just additive, but also subtractive processes (e.g., carpentry). Figure 1.1 depicts the typical workflow for a FFF 3D printer.

Just as programmers rarely write assembly directly, users of 3D printers do not write direct instructions for the motors. They instead produce them via compilation from a high-level design based on a specification or idea, created in a computer-aided design (CAD) software such as Openscad OpenScad [2019], Rhino Rhinoceros [2018], SolidWorks Solidworks [2018], or SketchUp SketchUp [2018]. This compilation process is complex and, similar to classical compilers, typically proceeds through a sequence of intermediate languages. After designing a model, the next step is to compile it to a surface mesh representation. A surface mesh is a triangulation of the surface of the 3D object represented by the design. The third step *slices* the mesh into 2D layers that are stacked on top of each other during the printing phase. The next step generates *G-code*, which is similar to assembly-level instructions for manufacturing devices Smid [2003]. The G-code is then interpreted by printer firmware to control the print (much as Postscript can be sent directly to many 2D printers).

In most desktop-class 3D printers, a spool feeds filament (typically a plastic) into an *extruder*, which heats and melts the filament before extruding it through a nozzle onto a *print bed*. It is this extruder that is controlled by the G-code, via low-level commands that actuate stepper motors to move the print head in any of three dimensions.

After completing the printing step, the user compares the output with the original specification and decides to either iterate over the above steps or terminate the process. Designing a CAD model is analogous to writing a program in a high level programming language. Converting it to a 3D mesh is similar to an intermediate representation. Slicing it and generating G-code is analogous to generating assembly. Viewing this workflow as a compiler has multiple advantages. It brings the formalisms of programming languages theory to bear, which in turn helps in reasoning about the correctness of the pipeline. It also supports the implementation of additional tools that can make these systems more accessible to end users.

1.1.2 E-graphs and Equality Saturation

Several chapters rely on equality saturation, so this section provides a high-level overview of the technique. Later chapters may have additional details on equality saturation and e-graphs as required. In all the uses of E-graphs in this thesis, the goal is always to either (1) optimize an input program given a set of cost metrics (Chapter 5), (2) synthesize a smaller program from a potentially large input program (Chapter 4), [2] or (3) find equivalences between large sets of programs thereby shrinking the space of candidates to explore during program synthesis (Chapter 6).

An E-graph is a set of *eclasses*, and each eclass is a set of equivalent *enodes*. An enode is an operator ($+$, $\times$, literal, etc.) applied to zero or more child eclasses. An eclass c *represents* expression e if c contains an enode n with the same operator as e and the children of n represent the children of e. Each eclass represents an exponential number of equivalent expressions (w.r.t. the number of enodes), since each of its enodes point to eclasses themselves.

Adding an expression to an E-graph works bottom up: first add the leaves as enodes each in their own eclasses, then recursively add operators as enodes pointing to the eclasses of their operands as children. Hashconsing (also known as memoization) ensures enodes are never duplicated in an E-graph Willsey et al. [2021]. This sharing compactly represents many equivalent expressions.

[2]This can also be viewed as a form of optimization. On the other hand, from a synthesis point of view, the input program can be viewed as a specification — the synthesized program must be equivalent to the input.

E-graphs also provide a *unify* operation that combines two eclasses and maintains their congruence closure. For example, if eclasses c_1 and c_2 represent (+ x y) and (+ x z) respectively, then unifying the eclasses representing y and z would cause c_1 and c_2 to be unified as well since they both contain "+" enodes with equivalent children.

E-graphs can easily be extended with syntactic rewrites $a \rightsquigarrow b$: whenever an eclass c represents an expression that matches pattern a under substitution ϕ, the eclass representing $\phi(b)$ is found (or constructed) and unified with c; the resulting eclass will represent both expressions $\phi(a)$ and $\phi(b)$. An advantage of E-graph-based optimizers over sequential rewrite engines is that they avoid the *phase ordering problem*. Phase ordering occurs when optimizations are applied destructively in sequential order, thereby causing the quality of the resulting code to depend on the order of application of the optimizations Bansal and Aiken [2008], Whitfield and Soffa [1990, 1997b]. In E-graph-based rewrite systems, the rewrites only expand the E-graph and therefore all previous expressions are still represented.

The process of repeatedly applying rewrites to grow an E-graph is called *equality saturation* Tate et al. [2009], Willsey et al. [2021]. In most applications of equality saturation, a final program must be extracted from the E-graph after saturation (i.e., when further application of rewrites does not change it), or when a timeout is reached. Typically, a cost function Tate et al. [2009], Willsey et al. [2021] is used to extract the optimal expressions from each eclass which are then composed to return the best program from the E-graph. Later chapters provide further details on the cost functions this thesis has explored.

1.2 My Work

This section describes new PL and compiler techniques we have developed for improving different stages of the fabrication pipeline described above.

λCAD is a new functional domain specific language (DSL) we developed for representing CAD models. We provided denotational semantics for λCAD using regular open sets. We also developed a language for representing surface meshes that closely resembles those used in commercial mesh editing and rendering tools, and provided semantics using ray intersection. We then developed a compiler from λCAD to mesh and proved that it is semantics preserving (Chapter 2). These results were published at ICFP 2018 Nandi et al. [2018].

Reincarnate Using the compiler and semantics of λCAD, we developed a decompiler / synthesizer called Reincarnate that automatically infers a high-level CAD program in λCAD from a low-level triangle mesh (Chapter 3). Reincarnate is useful for exposing high-level structure from low-level representations commonly shared in online 3D design forums (Thingiverse [2019], GrabCAD [2019]), thereby making the designs more readable and editable. These results were also published at ICFP 2018 Nandi et al. [2018] together with λCAD.

Szalinski Even with Reincarnate, the resulting programs in λCAD can still be too long to be useful, especially for large, repetitive models. To mitigate this problem, we developed a new tool, called Szalinski, that uses equality saturation Tate et al. [2009] to automatically infer maps and folds from flat, loop-free programs (Chapter 4). Szalinski was published at PLDI 2020 Nandi et al. [2020].

HELM A key insight in Szalinski was the use of *dynamic rewrites* to discover new equivalences. We used the same insight to develop an optimizing compiler for carpentry. This work was published at SIGGRAPH Asia 2019 Wu et al. [2019b].

Ruler Based on the two previous projects that use equality saturation for program synthesis and optimizing compilation, we observed that a key challenge in equality saturation based systems is the effort required to manually write the rewrite rules. To mitigate this problem, we developed Ruler, a framework for automatically inferring rewrite rules for any domain, given a grammar and an interpreter (Chapter 6). Ruler will be published at OOPSLA 2021 Nandi et al. [2021].

1.3 Beyond Computational Fabrication

Several ideas in this thesis are are applicable in domains beyond computational fabrication. This section highlights them.

- Beyond Syntactic Rewrites: Both Szalinski and HELM rely on term-rewriting — specifically equality saturation. They both use dynamic rewrites and eclass-analyses to find non-syntactic equivalences (Chapter 4). These ideas have since been applied in other equality saturation based tools Smith et al. [2021], Wang et al. [2020], VanHattum et al. [2021] and also inspired the egg Willsey et al. [2021] equality saturation engine to support dynamic rewrites and eclass-analyses as basic features. Further details are in Chapter 4.

- Rewrite Inference: Ruler has proved to be useful for a variety of domains like bitvectors, rationals, floats, integers, and strings (Chapter 6).

1.4 Delta Between Published Papers and this Thesis

Most of the work in this thesis is already published or under-review. This section summarizes any new additions the rest of the chapters may have.

- Chapter 2 has additional details about λCAD and the compiler's correctness proof (e.g., reasoning about mesh splitting) that did not appear in the published manuscript.

- Even though the core synthesis algorithm in Reincarnate has not changed, Chapter 3 shows additional results and a more elaborate explanation of the use of evaluation context that did not appear in the ICFP 2018 paper.

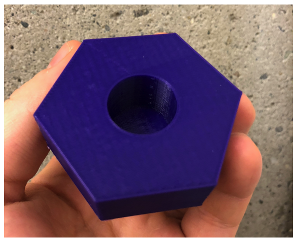
- The CAD language in the PLDI 2020 paper on Szalinski was called *Caddy* — Szalinski inferred high-level Caddy programs from a subset called *Core Caddy* that only supported Constructive Solid Geometry (CSG) primitives. Chapter 4 unifies the input language to Szalinski with the output language of Reincarnate (λCAD) because these two tools are intended to be used together.

- The content in Chapter 5 is identical to the SIGGRAPH Asia 2019 paper. If you have read the paper, you may want to skip this chapter.

- The content in Chapter 6 is identical to the OOPSLA 2021 paper which is soon to be published. If you have read the paper, you may want to skip this chapter.

Chapter 2
COMPILERS FOR GEOMETRIC LANGUAGES

```
⟦ Diff
    (Scale [2.5, 2.5, 1]
      HexPrism[1, 1]
    )
    (Scale [1, 1, 0.9]
      (Translate [0, 0, 0.5]
        Cylinder[1, 1]
      )
    )                         ⟧  =
```



A computer-aided design program denotes a geometric object.

Democratized computer-aided manufacturing has made available—at modest cost—design and fabrication capabilities that were previously reserved for large-scale commercial applications. Desktop-class 3D printers, laser cutters, and computer numerical control mills affordably enable educators, hobbyists, and researchers to rapidly prototype designs, manufacture tool parts, and even create custom prostheses The Future [2018]. 3D-printers in particular are now standard tools in maker communities and may some day replace the need for small-scale manufacturing much as conventional printers fundamentally changed the role of commercial printing shops.

However, despite the wide availability of hardware components at much lower costs than ever before, the corresponding software pipeline does not sufficiently support even tech-savvy early adopters. For democratized manufacturing techniques to reach their full potential, makers must be able to design and manufacture a wide variety of objects on demand.

The current state of tools in this space expects users to compose idiosyncratic CAD packages that are incompatible and whose interfaces are not clearly specified. Together with the fact that most of these tools are also proprietary, it makes the design experience for novice users and hobbyists unnecessarily awkward. The lack of specification also hampers efforts to build other tools that can make CAD programming more accessible to amateur enthusiasts. For example, it would benefit users to have tools for debugging their designs before printing to avoid waste of time and material, optimizing them to find an equivalent but simpler program, doing program analysis to detect violations of various geometric and physical properties, or synthesizing CAD programs for them so that they do not have to program from scratch.

This chapter addresses some of these challenges from a programming-languages perspective. Our first contribution is based on the insight that the desktop manufacturing pipeline is inherently compositional and functional in nature. We view this pipeline as a compilation task by modeling 3D solid geometry as a purely functional programming language equipped with a natural and tangible denotation to 3D solids. To relate this high-level CAD language to intermediate mesh representations, we formalize the popular STL mesh format Grimm [2004] as a low-level language and define a meaning-preserving compiler from CAD programs to meshes. We have designed and implemented a prototype of our declarative CAD language called λCAD that supports 3D primitives such as cubes, spheres, and cylinders; affine transformations such as translation, scaling, and rotation; and binary or constructive solid geometry (CSG) operations such as difference, intersection, and union. λCAD also supports standard functional features such as let bindings, functions, recursion, and conditionals.

Having developed this foundation, we define compiler correctness in terms of solid geometry, and provide a proof that our compiler is correct under this definition. Our approach toward formalizing CAD and STL then leads to the other main contribution — the first synthesis algorithm to our knowledge that converts meshes back into CAD programs, which we view as a reverse compilation task (Chapter 3).

Ultimately, my vision is to use tools and techniques from functional programming to build a new generation of tools that can enable non-expert end users to effectively work with desktop manufacturing devices. This chapter presents a first step in that direction by laying the programming languages foundation necessary to approach the problem in a rigorous and principled way.

2.1 Formalizing CAD and Mesh

CAD and mesh can be viewed as two fundamentally different ways of representing an object in 3D space. While CAD representations are based on solid geometry, mesh representations are based on surface geometry. Any translation between these two conceptually different approaches requires finding a way to map the concepts from one to the other. To that end, this section presents the syntax and denotational semantics for two languages for 3D modeling, λCAD, a high-level functional programming language, and *Mesh*, an intermediate surface representation based on industry-standard formats.

2.1.1 λCAD Language

We designed and implemented λCAD, a functional programming language with primitives for representing and manipulating geometric objects. Since the other features are standard, this section focuses on the syntax and semantics of the geometric fragment of the language.

Figure 2.1 shows the syntax of the geometric core of λCAD. It supports (1) 3D primitives such as *Cuboid*, *HexPrism*, *Cylinder*, etc., (2) affine transformations such as translation (*Translate*), rotation about X, Y, and Z axes (*Rotate*), uniform and non-uniform scaling (*Scale*) shown in Figure 2.2, and, (3) set-theoretic operations (*binops*): *Union*, *Difference*, and *Intersection*. The primitives are parametrized by their dimensions — whenever we skip them in code snippets and figures, it indicates unit dimensional primitives. For example, *Cuboid* has all sides of unit length and the bottom left corner at the origin, $(0, 0, 0)$ and is written as *Cube* in the rest of the thesis. *Cylinder* is a cylinder with unit radius and height whose base is centered at the origin. Similarly, *HexPrism* is a cylinder with 6 vertical sides along with a top and a base which are hexagonal. In general, both *Cylinder* and *HexPrism* take a 2D vector, $v2$, as parameter, where the first element represents radius and the other, height. *Cuboid* takes a 3D vector, $v3$, as parameter to represent length, breadth, and height. Note that as presented, all primitive objects are piecewise linear, thus requiring curves (e.g., spheres, and cylinders) to be approximated. Truly curved primitives are interesting and possible, but complicate the semantics, compilation, and synthesis approaches discussed in this chapter. The possibility of developing a compositional notion of equality between piecewise-linear approximations to curves in a way that supports correctness proofs for compilation and synthesis is a significant challenge left for future work. This work represents curves using approximations—for example, to represent a cylinder we set $n = 50$ as a default for *Cylinder*. All the affine transformations are represented using an invertible 3×3 matrix and a 3D vector Brannan et al. [1999] in the core CAD syntax in Figure 2.1. λCAD also supports user-provided raw meshes using the *Mesh* construct. Figure 2.1 also describes a denotational semantics based on regular open sets Ronse [1990] for CAD that maps each object to the set of 3D points inside it. The primitive *Empty* maps to the empty set, while *Cuboid*$[X, Y, Z]$ maps to the set of all points: $\{(x, y, z) \mid 0 < x < X, 0 < y < Y, 0 < z < Z\}$. Other primitives are similarly straightforward. Affine transformations are denoted by applying the transformation to every point in the denotation of e. The denotation of *Union* $e_1 e_2$ is the union of the denotations of e_1 and e_2. *Inter* (intersection) and *Diff* (difference) are similar.

2.1.2 Surface Mesh

A surface polygon mesh is a geometric representation of the surface of an object in 3D space using *vertices*, *edges*, and *faces*. The faces of a mesh are typically convex polygons. This work formalizes triangular meshes as shown in Figure 2.3a. A mesh is a list of faces, each of

$$op \ ::= \ + \mid \text{-} \mid \times \mid / \qquad \text{n} \ ::= \ \mathbb{R} \mid var \mid n \ op \ n \qquad v2 ::= [n,\ n] \qquad v3 ::= [n,\ n,\ n]$$

$$\begin{aligned} C \ ::= \ & Mesh\ M \mid Empty \ \mid Cuboid\ v3 \mid HexPrism\ v2 \mid Cylinder\ v2 \mid \ldots \\ \mid \ & Affine\ \mathbb{R}^{3\times 3}\ \mathbb{R}^3\ C \mid binop\ C\ C \end{aligned}$$

$$m \ ::= \ (\mathbb{R}^3, \mathbb{R}^3, \mathbb{R}^3)* \qquad binop ::= Union \mid Inter \mid Diff$$

$$\begin{aligned} [\![\, Mesh\ m \,]\!] &= [\![\, m \,]\!] \\ [\![\, Empty \,]\!] &= \{\} \\ [\![\, Cuboid\ [X, Y, Z] \,]\!] &= \{(x, y, z) \mid 0 < x < X,\ 0 < y < Y,\ 0 < z < Z\} \\ [\![\, HexPrism\ [r, h] \,]\!] &= \{(x, y, z) \mid -r < x < r,\ -\tfrac{r\sqrt{3}}{2} < y < \tfrac{r\sqrt{3}}{2},\ 0 < z < h\} \\ [\![\, Cylinder\ [r, h] \,]\!] &= \{(x, y, z) \mid -r < x < r,\ -r < y < r,\ 0 < z < h\} \\ [\![\, Affine\ p\ q\ c \,]\!] &= \{pv + q \mid v \in [\![\, c \,]\!]\} \\ [\![\, binop\ c_1\ c_2 \,]\!] &= [\![\, c_1 \,]\!]\, [\![\, binop \,]\!]\, [\![\, c_2 \,]\!] \end{aligned}$$

$$[\![\, Union \,]\!] = \cup \qquad [\![\, Inter \,]\!] = \cap \qquad [\![\, Diff \,]\!] = \setminus$$

Figure 2.1: Core λCAD syntax and semantics. λCAD programs denote to regular open sets Ronse [1990] in $\mathbb{R}^3$. Affine transformations are given by an *invertible* 3×3 matrix and translation vector. *Translate*, *Rotate*, and *Scale* are syntactic sugar for affine transformations depending on the first two arguments (Figure 2.2). Binary operators denote to set operations. Mesh denotation ($[\![\, m \,]\!]$) is detailed later in the chapter.

$$Translate[x,y,z] = Affine \begin{bmatrix}1&0&0\\0&1&0\\0&0&1\end{bmatrix} \begin{bmatrix}x\\y\\z\end{bmatrix}$$

$$Rotate[x,y,z] = Affine \left(\begin{bmatrix}cos(z)&-sin(z)&0\\sin(z)&cos(z)&0\\0&0&1\end{bmatrix} \begin{bmatrix}cos(y)&0&sin(y)\\0&1&0\\-sin(y)&0&cos(y)\end{bmatrix} \begin{bmatrix}1&0&0\\0&cos(x)&-sin(x)\\0&sin(x)&cos(x)\end{bmatrix} \right) \begin{bmatrix}0\\0\\0\end{bmatrix}$$

$$Scale[x,y,z] = Affine \begin{bmatrix}x&0&0\\0&y&0\\0&0&z\end{bmatrix} \begin{bmatrix}0\\0\\0\end{bmatrix}$$

Figure 2.2: Translation, Rotation, and Scale are syntactic sugar for affine transformations, depending on the parameters. *Translate* only requires a translation vector and the 3×3 identity matrix. *Rotate* requires a 3×3 rotation matrix, whose entries determine the axes of rotation. Rotation is first about x-axis, then y-axis, and then z-axis and is defined by the angles. *Scale* is determined by a diagonal matrix wholes entries determine the scaling factors along each axis.

which is a triangle represented by its three vertices. Note that for brevity, we use the term "triangle" or simply "face" to refer to triangular faces. This simple and flat representation is as expressive as other representations Grimm [2004] yet serves as a high-level executable specification, which could be used, for example, to differentially test against more sophisticated implementations such as STL and OFF Grimm [2004], OFF [2018]. Section 2.1.2 describes how we use *normals* to determine which side of a triangular face is inside/outside a 3D object (Norm in Figure 2.3b).

A mesh partitions the 3D space into (1) a space that is in the interior of the 3D object represented by the mesh (*inside* of the mesh), (2) a space that is in the exterior of the 3D object represented by the mesh (*outside* of the mesh), and (3) the surface of the mesh itself.

Valid Mesh

In order for a 3D CAD model to be printable, the mesh should be *valid* or well-formed. Invalid meshes can have a variety of problems such as zero volume, holes, and dangling faces, which make them unfit for printing. A *valid* 3D mesh is one that satisfies the following invariants:

- no overlapping or intersecting faces

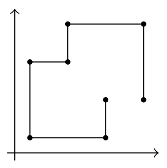
$$
\begin{aligned}
x,\ y,\ z &\in \mathbb{R} \\[1em]
pt &\in \mathsf{point} \\
&::= (x,\ y,\ z) \\[1em]
f &\in \mathit{Face} \\
&::= (pt,\ pt,\ pt) \\[1em]
m &\in \mathit{Mesh} \\
&::= f^*
\end{aligned}
$$

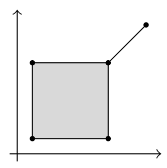
(a) Syntax of Mesh.

$$
\begin{array}{rcl}
r & \in & \mathbb{R} \\
d & \in & \mathit{Direction} \\
d & ::= & (r, r, r) \\
h & \in & \mathit{HalfLine} \\
h & ::= & (pt, d) \\
\mathsf{Midpoint} & : & \mathit{Face} \to \mathsf{point} \\
\mathsf{Norm} & : & \mathit{Mesh} \times \mathit{Face} \to \{L, R\} \\
\mathsf{On} & : & \mathit{Mesh} \times \mathit{Face} \to \mathsf{bool} \\
\mathsf{On}_{\mathsf{pf}} & : & \mathsf{point} \times \mathit{Face} \to \mathsf{bool} \\
\mathsf{isect\_tri} & : & \mathit{Face} \to \mathit{Face} \to \mathsf{point}^* \\
\mathsf{isect\_tris} & : & \mathit{Face}^* \to \mathit{Face} \to \mathsf{point}^* \\
\mathsf{triangulate} & : & \mathit{Face} \to \mathsf{point}^* \to \mathit{Face}^* \\
\mathsf{equiv} & : & \mathit{Mesh} \to \mathit{Mesh} \to \mathsf{bool} \\
\mathsf{is\_vertex\_of} & : & \mathsf{point} \times \mathit{Face} \to \mathsf{bool}
\end{array}
$$

(b) Mesh functions used in the compiler (Figure 2.6).

Figure 2.3: Syntax and auxiliary definitions for mesh. In Figure 2.3b, $\mathsf{Midpoint}$ is the centroid of a face. A point is $\mathsf{On}_{\mathsf{pf}}$ a face if it is coplanar with the face, and is in the interior of the face or on one of its edges (including vertices). On indicates whether syntactically, a face $f \in m$, i.e., whether f appears in the list of faces of m.


(a) 2D model with missing face.


(b) 2D model with extra face.

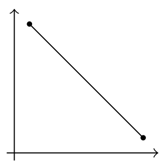

(c) Flat 2D model with zero area

Figure 2.4: Analogues of ill-formed meshes in 2D (used for simpler visualization). A 2D face is a line segment whereas a 3D face is a triangular plane. Thus, a missing face in 3D would be a missing triangle, an extra face would be an extra triangle, and a mesh with zero volume would be a plane.

- no edges that occur in an odd number of faces
- must be a valid two-manifold i.e., should not have holes (or missing faces). This happens if an edge is on an odd number of faces.

Figure 2.4 shows 2D analogues of some invalid meshes. We use 2D in the figure for simpler visualization. In 2D, the *faces* are segments instead of triangular planes. The analogue of *edges* in 2D are the vertices. The first figure in Figure 2.4 shows a 2D mesh that is open. This can happen when a vertex appears in an odd number of segments. The second figure is another example of an invalid mesh with a lone face. The 3D analogue of this is a mesh with an extra triangular face. The third figure is an example of a mesh with zero area. An example of this in 3D would be a mesh with just one triangular face, which would have zero volume.

Point With Respect To Face

Relative to a triangular face, f, a point pt may be positioned in the following ways:

- pt is a vertex of f
- pt is on an edge of f
- pt is strictly in the *interior* of f where interior does not contain the edges or the vertices.
- pt is outside f

$$\begin{array}{rcl}
\mathsf{intersect} & : & \mathit{Face} \times \mathit{HalfLine} \rightarrow \{\mathsf{None}, \mathsf{InteriorPt}, \mathsf{Other}\} \\
\\
\mathsf{InsideVia}(m, pt, d) & : & \mathit{Mesh} \times \mathsf{point} \times \mathit{Direction} \rightarrow \mathsf{bool} \\
\mathsf{InsideVia}(m, pt, d) & = & \mathsf{let}\ h = (pt, d)\ \mathsf{in} \\
 & & \{f \mid f \in m \wedge \mathsf{intersect}(f, h) = \mathsf{Other}\} = \emptyset \\
 & & \wedge\ |\{f \mid f \in m \wedge \mathsf{intersect}(f, h) = \mathsf{InteriorPt}\}| \bmod 2 = 1 \\
\\
[\![\cdot]\!] & : & \mathit{Mesh} \rightarrow \mathcal{P}(\mathsf{point}) \\
[\![m]\!] & = & \{pt \mid \exists d.\ \mathsf{InsideVia}(m, pt, d)\}
\end{array}$$

Figure 2.5: Semantics of Mesh using intersection of faces with halflines (rays).

$\mathsf{On}_{\mathsf{pf}}(pt, f)$ (Figure 2.3b) is therefore `true` for the first three cases and `false` for the last case.

Point With Respect To Mesh

Relative to a valid mesh, m, a point pt may be positioned in the following ways:

- pt is outside m
- pt is inside m
- $\mathsf{On}_{\mathsf{pf}}(pt, f) = \mathit{true}$ for some triangle face, $f \in m$.

Sides of a Mesh

Knowing the vertices of the faces of a mesh is not sufficient to determine the *inside* and *outside* of the shape. This information is given by *normal vectors* for each face of a mesh, which are unit vectors orthogonal to the face that point toward the outside of the shape (Norm in Figure 2.3b). We use L (left) and R (right) to indicate the two possible directions for normals (Figure 2.3b), depending on whether the left-hand or right-hand rule should be used on the given face. Typical industrial formats store normal vectors in the representation of each face Grimm [2004], but for conceptual parsimony, we instead compute normals as required using global properties of the mesh.

Specifically, in a valid mesh, the normal vectors can be computed once we have a way of determining the position of a point w.r.t. the mesh. For this, we use the well-known method of casting rays de Berg [1997]. A ray, or halfline, h is represented by a starting point, pt and a direction, d, as shown in Figure 2.3b. A point is inside a 3D mesh if there is a *good* halfline starting at the point that crosses an *odd* number of faces of the mesh (Figure 2.5).

A good halfline is one that does not intersect the mesh at its vertices or edges. An important result is that many good halflines exist for any point not on the boundary of the mesh (Theorem 1).

Theorem 1. For any valid mesh m and point pt not on the boundary of m (i.e., $\mathsf{On}_{\mathsf{pf}}(pt, t) = \mathit{false}\ \forall\ t \in m$), *almost all* directions d result in good halflines (that is, all directions outside a set of measure 0 result in good halflines).

Proof. The edges and vertices of m, when projected onto a unit sphere around pt, form a set of measure 0. Any direction d on the sphere outside of this set forms a good halfline for pt. □

It is also essential that the choice of halfline does not matter in a valid mesh (Theorem 2).

Theorem 2. For any valid mesh m, point pt not on any face of m (i.e., $\mathsf{On}_{\mathsf{pf}}(pt, f) = \mathit{false}\ \forall\ f \in m$), and good halflines h_1 and h_2 each starting at pt, the halfline h_1 intersects m an odd number of times if and only if h_2 intersects m an odd number of times.

Proof. First, note that there exists a plane containing h_1 and h_2. The intersection of this plane and the mesh is a simple 2D polygon m_2 (the mesh is valid so faces do not intersect) containing pt (since both h_1 and h_2 contain pt). We must show that h_1 and h_2 intersect m_2 with equal parity. As shown by, for example, Hormann and Agathos Hormann and Agathos [2001], this parity is equal to a formula over the angles between m_2's edges, and must thus be the same for h_1 and h_2. □

We can now compute normals for a face using any test point pt in the interior of f (we use $\mathsf{Midpoint}(f)$ in our implementation). Consider any good halfline h for pt. If h crosses m an odd number of times, then h lies on the same side of f as the outward-facing normal. Otherwise, it is on the opposite side.

Mesh semantics The above technique also determines a denotational semantics for meshes that denotes a mesh to the set of points inside it, thus enabling easy comparison with the denotation of CAD objects. Figure 2.5 defines this semantics based on face and good halfline intersection. The intersection of a face and a halfline can have three outcomes: (1) `None` indicates that the face and the halfline do not intersect at any point, (2) `InteriorPt` indicates that the halfline goes through the face at exactly one point in its *interior*, and (3) `Other` indicates all other possible interactions of a face and a halfline: they are coplanar and the halfline goes through an edge or a vertex of the face (e.g., it is not a good halfline).

`InsideVia(m, pt, d)` is a predicate that defines when `pt` is inside the mesh `m`: if there exists a direction `d`, such that for the halfline `h = (pt, d)`, (1) there is no face in `m` that results in an `Other` intersection with `h`, and (2) `h` crosses the mesh at an odd number of faces, then `pt` is inside the mesh, `m`. Notably, even though the choice of the halfline does not have to be unique (in the sense that any good halfline would suffice), this technique provides a deterministic semantics for meshes.

2.2 λCAD Compiler

This section presents a meaning-preserving compiler that generates a triangular mesh from a subset of λCAD. The compiler's specification is given in terms of the geometric denotational semantics of the source and target languages. The straightforward compilation algorithm described here is used in some form or another in all industrial CAD tools. Our contributions are (1) to formalize it in terms of structural recursion and denotational semantics, which enables (2) a proof of correctness and validity. Figure 2.6 defines the compiler as a recursive function on the syntax of the CAD program.

Compiling Primitives and Affine Transformations. The output of compiling a *Mesh* m construct is the underlying mesh, m. Compiling an *Empty* CAD model simply generates an empty mesh. The mesh for *Cube* is as defined in Figure 2.6. Since we use pre-defined meshes to approximate curves in this chapter, the output of compiling them is simply the corresponding pre-defined mesh. For affine transformations, the compiler generates the mesh by applying the transformation to the result of the recursive call, i.e., to the vertices of the faces of the mesh returned by the recursive call.

Compiling Binops. Similar to primitives and affine transformations, the set-theoretic binary operations (binops) are also compiled by first obtaining the left and right meshes by recursive calls on the children of the binop and then using the corresponding mesh-level functions: $\mathit{mBop}(\mathit{Union})(m_1, m_2)$, $\mathit{mBop}(\mathit{Difference})(m_1, m_2)$, and $\mathit{mBop}(\mathit{Intersection})(m_1, m_2)$, shown in Figure 2.6. These operations are, however, non-trivial for *overlapping* meshes, since the faces of the resulting mesh are a complex subset of a refinement of both input meshes. If two input faces overlap, then some parts of each face may be discarded in the output, while other parts remain. Since faces can overlap in arbitrary ways, preserving the mesh invariants defined in Section 2.1.2 requires first *splitting* the two meshes to ensure that the overlapping faces are correctly re-triangulated and then applying the mesh-level binop.

Specifically, for the λCAD compiler to correctly compile the set-theoretic binops over two meshes m_1 and m_2 (definition in Figure 2.6) and produce a valid output mesh, the conditions in Property 3 must be satisfied. These properties are guaranteed by the relation *split* which,

$$
\begin{aligned}
m_{cube} = \; [\; & ((0,0,0),(1,0,0),(1,1,0)) \\
, & ((0,0,0),(1,1,0),(0,1,0)) \\
, & ((0,0,0),(1,0,0),(1,0,1)) \\
, & ((0,0,0),(1,0,1),(0,0,1)) \\
, & ((0,0,0),(0,0,1),(0,1,1)) \\
, & ((0,0,0),(0,1,1),(0,1,0)) \\
, & ((0,0,1),(1,0,1),(1,1,1)) \\
, & ((0,0,1),(1,1,1),(0,1,1)) \\
, & ((1,0,0),(1,1,0),(1,1,1)) \\
, & ((1,0,0),(1,1,1),(1,0,1)) \\
, & ((0,1,0),(1,1,1),(1,1,0)) \\
, & ((0,1,0),(0,1,1),(1,1,1)) \\
]
\end{aligned}
$$

$$
\begin{aligned}
\mathit{compile}(\mathit{Mesh}\ m) &= m \\
\mathit{compile}(\mathit{Empty}) &= [] \\
\mathit{compile}(\mathit{Cube}) &= m_{cube} \\
\mathit{compile}(\mathit{Affine}\ p\ q\ c) &= \mathsf{map}_{\mathsf{vertex}}\ (\lambda v.\ pv + q)\ (\mathit{compile}(c)) \\
\mathit{compile}(\mathit{Binop}\ c_1\ c_2) &= \mathsf{let}\ m'_1, m'_2 \text{ s.t., } \mathit{split}\ (\mathit{compile}(c_1), \mathit{compile}(c_2), m'_1, m'_2)\ \mathsf{in} \\
&\ \mathit{mBop}(\mathit{Binop})(m'_1, m'_2) \\
\mathit{mBop}(\mathit{Union})(m_1, m_2) &= [f \in m_1 \mid \nexists d.\ \mathsf{InsideVia}(m_2, \mathsf{Midpoint}(f), d)] \mathbin{++} \\
&\ [f \in m_2 \mid \nexists d.\ \mathsf{InsideVia}(m_1, \mathsf{Midpoint}(f), d)] \mathbin{++} \\
&\ [f \in m_1 \mid \mathsf{On}(m_2, f) \wedge \mathsf{Norm}(m_1, f) = \mathsf{Norm}(m_2, f)] \\
\mathit{mBop}(\mathit{Difference})(m_1, m_2) &= [f \in m_1 \mid \nexists d.\ \mathsf{InsideVia}(m_2, \mathsf{Midpoint}(f), d)] \mathbin{++} \\
&\ [f \in m_2 \mid \exists d.\ \mathsf{InsideVia}(m_1, \mathsf{Midpoint}(f), d)] \mathbin{++} \\
&\ [f \in m_1 \mid \mathsf{On}(m_2, f) \wedge \mathsf{Norm}(m_1, f) \neq \mathsf{Norm}(m_2, f)] \\
\mathit{mBop}(\mathit{Intersection})(m_1, m_2) &= [f \in m_1 \mid \exists d.\ \mathsf{InsideVia}(m_2, \mathsf{Midpoint}(f), d)] \mathbin{++} \\
&\ [f \in m_2 \mid \exists d.\ \mathsf{InsideVia}(m_1, \mathsf{Midpoint}(f), d)] \mathbin{++} \\
&\ [f \in m_1 \mid \mathsf{On}(m_2, f) \wedge \mathsf{Norm}(m_1, f) = \mathsf{Norm}(m_2, f)]
\end{aligned}
$$

Figure 2.6: Representative cases of CAD-to-Mesh compiler. Midpoint, On and Norm are as defined in Figure 2.3b. *split* ensures that the meshes have no overlapping faces. *mBop*(*Union*), *mBop*(*Intersection*), and *mBop*(*Difference*) are mesh level binops that operate on the valid meshes generated by *split*.

as Figure 2.6 shows, generates the split meshes m_1' and m_2' for the mesh-level binops to operate over.

Property 3. Pre-conditions of *mBop* / Post-conditions of *split*.

- m_1 and m_2 are valid meshes
- $\forall\ f_i\ \in\ m_1$, f_i is either inside m_2, outside m_2, or On m_2
- $\forall\ g_i\ \in\ m_2$, g_i is either inside m_1, outside m_1, or On m_1
- if $f_i \in m_1 \wedge \mathsf{On}(m_2, f_i)$ then $f_i \in m_2$ and if $g_i \in m_2 \wedge \mathsf{On}(m_1, g_i)$ then $g_i \in m_1$
- $\forall\ pt,\ if\ \exists\ f\ \in\ m_1 \mid \mathsf{On}_{\mathsf{pf}}(pt, f)\ =\ true\ \wedge\ \exists\ f'\ \in\ m_2 \mid \mathsf{On}_{\mathsf{pf}}(pt, f')\ =\ true\ then\ pt$ is either a vertex of f' or on an edge of f', or f and f' coincide.

where, a face, f is outside a mesh, m, if all *pt* in the interior of f are outside m, and a face, f is inside a mesh, m, if all *pt* in the interior of f are inside m.

2.2.1 Splitting Meshes

Since meshes are composed of triangular faces, mesh splitting ultimately operates on intersecting triangles (faces) to produce new faces that do not intersect.

Triangle-Triangle Intersections. In order to understand *split*, let us first look at various way in which two triangles (faces) t_1, t_2 can overlap or intersect.

Coplanar. If t_1 and t_2 are coplanar:

- edges of t_1 may intersect with edges of t_2. In this case, we are interested in the points of intersection.
- one or more vertices of t_1 may be $\mathsf{On}_{\mathsf{pf}}$ t_2 (and vice versa) due to edge intersection or t_1 being contained inside t_2. In this case, we are interested in the vertices that are $\mathsf{On}_{\mathsf{pf}}$ t_2 and the points of intersection of the edges if any.
- they may share an edge.
- they may coincide.

Note that in the first two scenarios, unless the points coincide with vertices of the triangles, they lead to a shared area between the two faces whose perimeter is determined by the segments that connect the intersection points.

Non-coplanar. If t_2 and t_2 are not on the same plane:

- edges of t_1 may intersect with edges of t_2. In this case, we are interested in the points of intersection.
- edges t_1 may go through or touch t_2 at some points. In this case, we are interested in those points. This also covers the case where a vertex of t_1 touches t_2.
- they may share an edge.

We define a function, isect_tri (Figure 2.3b) that returns the above points of interest.

split. There are many ways to split two meshes. Figure 2.7 shows the splitting algorithm in the λCAD compiler. `split` takes two valid meshes as input. `equiv` performs a face-wise comparison of two meshes, i.e., two meshes m_1 and m_2 are `equiv` iff

$$\forall\ f_1\ \in\ m_1,\ \exists\ f_2\ \in\ m_2 \mid f_1 = f_2 \wedge \forall\ f_2\ \in\ m_2,\ \exists\ f_1\ \in\ m_1 \mid f_1 = f_2$$

where $f1 = f2$ iff their vertices coincide. `isect_tris` finds all the "interesting" points, *pts* of intersection (see above) of a triangle, t, with a list of triangles, ts by using isect_tri. `triangulate` takes a triangle, t and a list of points, *pts*, and re-triangulates t to generate new triangles, ts such that

Property 4. Triangulate Specification.

$$\bigcup ts = t \wedge \forall\ t'\ \in\ ts,\ \textit{if}\ \exists\ p\ \in\ \textit{pts} \mid \mathsf{On}_{\mathsf{pf}}(p, t') = \textit{true}\ \textit{then}\ \mathsf{is\_vertex\_of}(p, t') = \textit{true}$$

where is_vertex_of is defined in Figure 2.3b. Figure 2.7 shows a simple implementation of `triangulate` in the λCAD compiler. In this implementation, `connect_to_closest` finds the closest triangular face, t, such that p is $\mathsf{On}_{\mathsf{pf}}$ t and connects p to all three vertices of t to make `new_tris`. However, we have found that this implementation of `triangulate` can lead to needle-like triangles which cause rounding errors. The λCAD compiler implements two solutions to mitigate that: (1) it sets an upper bound on the number of calls to `loop` in `split` (Figure 2.7) to ensure termination so that the algorithm does not get stuck splitting very tiny triangles, (2) it supports different implementations of triangulate that avoid needle-like triangles like Delaunay triangulation de Berg [1997]. The latter is a more efficient approach but we have found that the former works reliably in practice as well.

Property 5. *split* must ensure that after splitting f_i in to $f_{i1}, f_{i2}, ..., f_{in}$,

$$\bigcup_{j=1}^{n} f_{ij} = f_i \quad \text{and} \quad \forall\ f_{ij}\ f_{ik}\ .\ j \neq k \implies f_{ij} \cap f_{ik} = \emptyset.$$

```
let split m1 m2 =
  let rec loop m1 m2 =
     let (m1', m2') = split' m1 m2 in
     if equiv m1 m1' && equiv m2 m2'
     then
       (m1, m2)
     else
       loop m1' m2'
  in
  loop m1 m2


let triangulate t pts =
  let rec loop acc pts =
    match pts with
    | [] -> acc
    | p :: ps ->
       let new_tris =
         connect_to_closest p acc
       in
       loop new_tris ps
  in
  loop [t] pts
```

```
let split' m1 m2 =
  (* tris_of_mesh is a helper function to
   * get the triangles from the mesh *)
  let m1_tris = tris_of_mesh m1 in
  let m2_tris = tris_of_mesh m2 in
  let m1_tris' =
    List.map
      (fun t ->
        isect_tris m2_tris t
          |> triangulate t)
      m1_tris
  in
  let m2_tris' =
    List.map
      (fun t ->
        isect_tris m1_tris' t
          |> triangulate t)
      m2_tris
  in
  ( tris_to_mesh m1_tris',
    tris_to_mesh m2_tris')
```

Figure 2.7: Mesh Splitting Algorithm in the λCAD compiler. Signatures of `isect_tris` and `triangulate`, are as defined in Figure 2.3b. We show a naive implementation of `triangulate` here but in our current implementation, we use Delaunay triangulation de Berg [1997] which is more efficient and does not produce needle-like triangles.

In the case of no overlap between f_i with faces from another mesh, this is trivially true since *split* will simply return f_i.

Axiom 6. Any implementation of *split*(m_1, m_2, m'_1, m'_2) must satisfy Properties 3 and 5.

2.2.2 Split's Termination

As stated previously, two meshes can be split and re-triangulated in various ways, as long as the resulting meshes satisfy the properties in Property 3. We provided example triangulation strategies that the λCAD compiler implements in Section 2.2.1 that are fast and produce better triangles. In order to demonstrate that mesh splitting can be done in a way that succeeds and terminates, we present yet another splitting strategy (Figure 2.8) which is simpler to reason about. The termination guarantee is required for proving total correctness and validity of the λCAD compiler in later sections.

This approach first finds all the intersections between m_1 and m_2 and re-triangulates only those faces that intersect, leaving the rest of the meshes untouched. In Figure 2.8, we define `isect_cat` to represent the nature of intersection between two triangles. As section 2.2.1 described, two triangle faces, t_1, t_2 can intersect in multiple ways: `Point` captures intersections where a vertex from t_2 is $\mathsf{On}_{\mathsf{pf}}$ t_1. `Edge` captures intersections resulting in two points that are $\mathsf{On}_{\mathsf{pf}}$ t_1. This leads to an edge. `Area` captures intersections resulting in three or more points that are $\mathsf{On}_{\mathsf{pf}}$ t_1. This leads to an area shared between the two faces. Figure 2.9 shows some examples of these types of intersections.

`isect_tris'` takes the triangles from two meshes and splits the ones from the latter w.r.t. the former. It generates a tuple `(P, E, A)` that classifies the intersection based on `isect_cat`. The points from all of these intersections are accumulated in `all_isects` and used to re-triangulate (`triangulate'`) the two meshes while satisfying the following constraints together with the constraint previously stated for `triangulate` in Property 4. While we do not show an implementation of `triangulate'`, we rely on the fact that given an area to triangulate and a set of points to be used, such a triangulation can be done.

- the vertices and edges of the original triangles must be retained. This is essential for ensuring that more intersections between the new faces and the old faces are not introduced by `triangulate'`.
- the shared areas (`A`) must be triangulated the same way for all the involved triangles from both meshes.
- no new triangles must intersect with the edges in `E`.

Theorem 7. The splitting algorithm in Figure 2.8 terminates.

```
type isect_cat =
  | Point
  | Edge
  | Area

type isect' =
{ curr: Tri
; others: Tri list
; pts: Point list
; cat: isect_cat
}

let split m1 m2 =
  let m1_tris = tris_of_mesh m1 in
  let m2_tris = tris_of_mesh m2 in
  let (P1, E1, A1) : (isect' list * isect' list * isect' list) =
    isect_tris' m2_tris m1_tris
  in
  let (P2, E2, A2) : (isect' list * isect' list * isect' list) =
    isect_tris' m1_tris m2_tris
  in
  let all_isects =
    P1.pts ++ P2.pts ++ E1.pts ++ E2.pts ++ A1.pts ++ A2.pts
  in
  triangulate' all_isects (P1, E1, A1) (P2, E2 A2)
```

Figure 2.8: A simpler splitting algorithm.

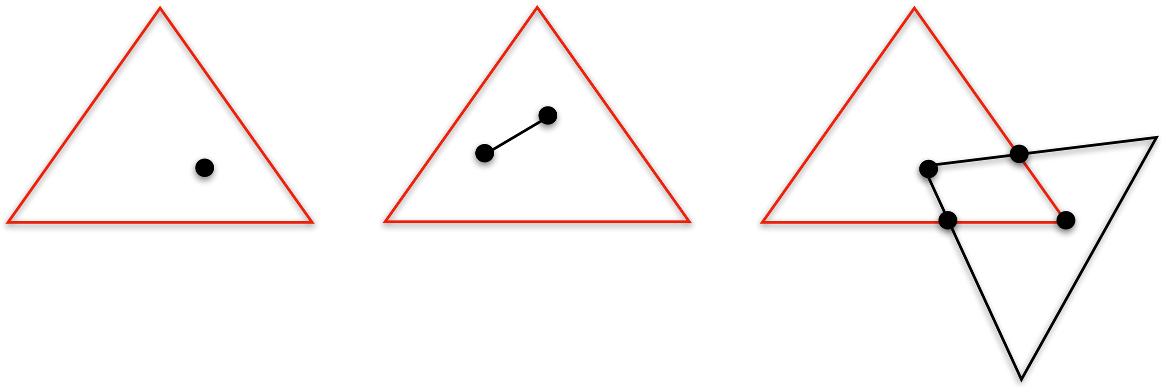


Figure 2.9: Examples of `P`, `E`, and `A` respectively for the simpler splitting algorithm in Figure 2.8. The first is an example of another non-coplanar triangle's vertex touching this triangle at the point. The second is an example of another non-coplanar triangle going through this triangle at the two points which form a segment. The third is an example of two coplanar triangles intersecting.

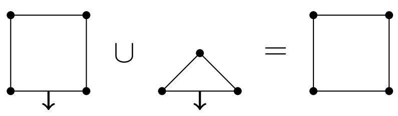
Proof. Algorithm 2.8 does not recursively split using newly created triangles. `all_isects` is a finite set of points which are then used in `triangulate'` to generate a finite set of non-intersecting triangles. Since no new intersections are introduced, this *split* terminates once `triangulate'` is done. □

2.2.3 Implementing Binops over Split Meshes

Mesh splitting gets rid of any partial triangles or overlaps that lead to non-triangular surfaces after which the mesh functions for the binary operations: $\mathit{mBop}(\mathit{Union})(m_1, m_2)$, $\mathit{mBop}(\mathit{Difference})(m_1, m_2)$, and $\mathit{mBop}(\mathit{Intersection})(m_1, m_2)$ determine which faces from the split meshes m_1' and m_2' should be *kept* in the final mesh and which ones should be discarded.

- $\mathit{mBop}(\mathit{Union})(m_1', m_2')$ keeps faces from m_1' that are outside m_2' as well as faces from m_2' that are outside m_1'. For faces from m_1' that are also `On` m_2' (`On` is defined in Figure 2.3b), if the face has the same *normal*, then it is kept, otherwise it is discarded. This is illustrated (in 2D) in Figure 2.10.

- $\mathit{mBop}(\mathit{Difference})(m_1', m_2')$ keeps faces from m_1' that are outside m_2', faces from m_2' that are inside m_1'. For faces from m_1' that are also `On` m_2', if the face has the same *normal* (L or R), then it is discarded, otherwise it is kept.


(a) Keeping a common edge during mesh union.

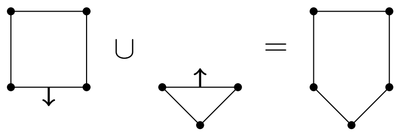

(b) Removing a common edge during mesh union.

Figure 2.10: Examples (in 2D) demonstrating when common faces are retained or removed by *mBop*(*Union*). The rules for *mBop*(*Intersection*) and *mBop*(*Difference*) are in Section 2.2.1.

- *mBop*(*Intersection*)(m'_1, m'_2) keeps faces from m'_1 that are inside m'_2, faces from m'_2 that are inside m'_1. For faces from m'_1 that are also On m'_2, if the face has the same *normal* (L or R), then it is kept, otherwise it is discarded.

2.3 Compiler Validity

We prove that the λCAD compiler produces a valid mesh.

Theorem 8 (Compiler Validity)**.** For all CAD expressions e, *compile*(e) is a valid mesh.

Proof. By induction on e.

Case *Empty***:** *compile*(*Empty*) $= \emptyset$ which is valid.

Case *Cube***:** *compile*(*Cube*) is the predefined valid mesh of *Cube* shown in Figure 2.6.

Case *Affine* p q e'**:** For affine transformed expressions, since the transformations only move the position of the mesh but do not introduce new faces, if the underlying mesh is valid, the affine transformed mesh is also valid. The invertibility of the matrix p ensures that applying the transformation on a triangle face preserves the triangular shape of the face, i.e., the lines connecting the vertices are preserved Brannan et al. [1999].

Case *binop* o c_1c_2**:** For binary operations, we must show that validity is preserved even when the underlying meshes have overlapping faces, i.e., we show that the resulting mesh does not have (1) overlapping faces (2) edges that occur in an odd number of faces. Axiom 6 which is strictly stronger than mesh validity, Theorem 7, and the face dropping/keeping strategy of binops (Section 2.2.3) ensure that (1) holds after mesh splitting.

Since Axiom 6 holds for *split* and it is strictly stronger than mesh validity, (2) is also satisfied by *split*. For clarity, here we provide further intuition for how *split* guarantees

(2). After splitting, every edge in the new meshes is either (a) a new edge added in the interior of an original face, or (b) a subedge of an existing edge in the original face. Let us first look at (a): Since Property 5 ensures that $\bigcup_{j=1}^{n} f_{ij} = f_i$, where f_{ij} are the new faces f_i is split into, there cannot be any odd-faced edge because the only way it can happen is if there were fewer triangles whose $\bigcup$ led to f_i. In other words, since no space inside f_i is left un-triangulated, each new pair of triangles must share an edge. Otherwise, there will be space left in side f_i that is not a triangle. Now let us look at (b): Every edge e in (b) will be an edge for exactly one sub-triangle of each original triangle that used the edge that e is a subedge of. The number of such faces e is part of is still an even number; in fact it is the same number as the edge e is a subedge of had.

Now we need to show that the mesh-level binops operating on the split meshes also guarantees (2) which we prove in Lemma 9.

□

Lemma 9. Given two meshes, m_1, m_2 that satisfy Property 3, *mBop(Union)*(m_1, m_2), *mBop(Intersection)*(m_1, m_2), *mBop(Difference)*(m_1, m_2), *mBop(Difference)*(m_2, m_1) are valid meshes.

Proof. For faces of m_1 and m_2 that are completely inside / outside each other, all three mesh binops keep or discard them *and* all other faces that have a common edge. Therefore the edges of these faces are either discarded or still have an even number of faces that share them. Let's focus on (1) faces that are *on both meshes* and (2) faces from m_1 and m_2 that share a common edge. Mesh binops must guarantee that they generate even-faced edges in these two scenarios in order for the final mesh to be valid.

Consider the first case where two valid meshes share an entire face f, i.e., the face is On both m_1 and m_2. Since both meshes are valid, all its edges must appear on an even number of faces. If the common face is to be discarded from the final mesh by the binop, then it must be removed twice — once from m_1 and once from m_2. Since this leads to the removal of an even number of faces, the edges will continue to appear on an even number of faces. If the common face is to be kept, it will still only appear once in the final mesh and therefore is removed once from one of the meshes together with any other faces of that mesh sharing an edge with this face. Since both meshes were originally valid, this still ensures validity.

Next, consider the second case where two valid meshes have faces that only have an edge in common. Since each edge in both valid meshes m_1 and m_2 are on an even number of faces, the faces must form pairs that lead to *wedges*. A wedge is the angle formed by two triangle faces, f_1 and f_2, and their common edge e such that this angle is inside the mesh and no other face that shares e lies in the space between f_1 and f_2 inside the mesh. If the faces of m_1 and m_2 do not form these pair-wise wedges, then they are not valid meshes to begin with. Now let us take an arbitrary wedge made by faces f_1 and f_2 in m_1 and call the shared edge in the wedge e. We proceed by considering the number of faces of m_2 inside the wedge. Case 0 and Case 1 below are special cases of the latter ones, but we state them separately to give better intuition before stating the general cases.

Case 0. If no face of m_2 is inside the wedge then for

(1) $\mathit{mBop}(\mathit{Union})(m_1, m_2)$: both f_1 and f_2 will appear in the final mesh,

(2) $\mathit{mBop}(\mathit{Intersection})(m_1, m_2)$: neither f_1 nor f_2 will appear in the final mesh,

(3) $\mathit{mBop}(\mathit{Difference})(m_1, m_2)$: both f_1 and f_2 will appear in the final mesh and,

(4) $\mathit{mBop}(\mathit{Difference})(m_2, m_1)$: neither f_1 nor f_2 will appear in the final mesh.

Therefore, in all cases, either both faces (precisely 2 faces) are kept or discarded, which results in the shared edge appearing in an even number of faces after the *mBop*.

Case 1. If one face, g_1 of m_2 is inside the wedge, then g_1 must also be paired with some g_2 to form a wedge since m_2 is a valid mesh. This also means that g_2 is outside the wedge formed by f_1, f_2, and the common edge e. For

(1) $\mathit{mBop}(\mathit{Union})(m_1, m_2)$: g_2 and only one of f_1 and f_2 are in the final mesh (depending on whether f_1 or f_2 is outside the wedge made by g_1, g_2, and e),

(2) $\mathit{mBop}(\mathit{Intersection})(m_1, m_2)$: g_1 and only one of f_1 and f_2 are in the final mesh (depending on whether f_1 or f_2 is inside the wedge made by g_1, g_2, and e),

(3) $\mathit{mBop}(\mathit{Difference})(m_1, m_2)$: g_1 and only one of f_1 and f_2 are in the final mesh (depending on whether f_1 or f_2 is outside the wedge made by g_1, g_2, and e),

(4) $\mathit{mBop}(\mathit{Difference})(m_2, m_1)$: g_2 and only one of f_1 and f_2 are in the final mesh (depending on whether f_1 or f_2 is inside the wedge made by g_1, g_2, and e).

In all cases, only two faces are kept in the final mesh, which again ensures that the edge, e, will appear in an even number of faces.

Case $n > 1 \mid n \;\%\; 2 = 0$. There are two sub-cases. First, let's consider the case where 0 of these n faces of m_2 have a wedge parter outside the wedge formed by f_1 and f_2. We end up with all n faces of m_2 inside the wedge. In this case, for

(1) $\mathit{mBop}(\mathit{Union})(m_1, m_2)$: only f_1 and f_2 are in the final mesh,

(2) $mBop(Intersection)(m_1, m_2)$: only the n faces of m_2 are in the final mesh,

(3) $mBop(Difference)(m_1, m_2)$: f_1, f_2, and all the n faces are in the final mesh,

(4) $mBop(Difference)(m_2, m_1)$: none of the faces are in the final mesh.

In all cases, an even number of faces sharing e are in the final mesh.

Next, let us consider the case where two more faces, g_1 and g_2 from m_2 are outside the wedge, one on either side of f_1 and f_2, and the rest of the n faces are inside the wedge, i.e., f_1, f_2, and the n faces of m_2 are inside a wedge made by g_1, g_2, and e. In this case, for

(1) $mBop(Union)(m_1, m_2)$: only g_1 and g_2 are in the final mesh,

(2) $mBop(Intersection)(m_1, m_2)$: f_1, f_2, and the n faces are in the final mesh,

(3) $mBop(Difference)(m_1, m_2)$: only the n faces are in the final mesh,

(4) $mBop(Difference)(m_2, m_1)$: g_1, g_2, f_1, f_2 are in the final mesh.

Again, in all cases, an even number of faces sharing e are in the final mesh.

Case $n > 1 \mid n \ \% \ 2 \neq 0$**.** If an odd number ($n$) of faces of m_2 are inside the wedge, it means that one additional face, g_1 of m_2 must be outside the wedge. In this case, for

(1) $mBop(Union)(m_1, m_2)$: g_1 and only one of f_1 and f_2 are in the final mesh (depending on whether f_1 or f_2 is outside the wedge made by g_1, one of the other n faces, and e),

(2) $mBop(Intersection)(m_1, m_2)$: the n faces, and only one of f_1 and f_2 are in the final mesh (depending on whether f_1 or f_2 is inside the wedge made by g_1, one of the other n faces, and e),

(3) $mBop(Difference)(m_1, m_2)$: all the n faces of m_2 and only one of f_1 and f_2 are in the final mesh (depending on whether f_1 or f_2 is outside the wedge made by g_1, one of the other n faces, and e),

(4) $mBop(Difference)(m_2, m_1)$: g_1 and only one of f_1 and f_2 are in the final mesh (depending on whether f_1 or f_2 is inside the wedge made by g_1, one of the other n faces, and e).

In all cases, an even number of faces sharing e are kept in the final mesh.

$\square$

2.4 Compiler Correctness

Our definition of compiler correctness is based on the denotational semantics we described in Section 2.1. Specifically, we prove that the compiler returns a *valid* mesh with the same denotation as the input CAD program. For binary operations, we provide the proof for union. The cases for intersection and difference are similar and hence omitted. We prove compiler correctness for a subset of λCAD that only has *Cube* and *Cuboid* as 3D primitives [1]. The λCAD compiler uses approximations for cylinders and spheres since it currently does not support true curves.

Theorem 10 (Compiler correctness)**.** For all CAD expressions e, $[\![\, compile(e) \,]\!] = [\![\, e \,]\!]$.

Proof. By induction on e. We show a few representative cases.

Case *Empty***:**

$$\begin{array}{rcll} [\![\, compile(Empty) \,]\!] & = & [\![\, [\,] \,]\!] & \text{By definition of } compile(). \\ & = & \emptyset & \text{By definition of mesh } [\![\ ]\!]. \\ & = & [\![\, Empty \,]\!] & \text{By definition of CAD } [\![\ ]\!]. \end{array}$$

Case *Cube***:**

$$\begin{array}{rcll} [\![\, compile(Cube) \,]\!] & = & [\![\, m_{cube} \,]\!] & \text{By definition of } compile(). \\ & = & [\![\, Cube \,]\!] & \text{By Lemma 11.} \end{array}$$

Case *Affine p q e′***:** Let m' represent $compile(e')$.

$$\begin{array}{rcll} [\![\, compile(Affine\ p\ q\ e') \,]\!] & = & [\![\, \mathsf{map}_{\mathsf{vertex}}\ (\lambda v.\ pv + q)\ (m') \,]\!] & \text{By definition of } compile(). \\ & = & \{pv + q \mid v \in [\![\, m' \,]\!]\} & \text{By Lemma 12.} \\ & = & \{pv + q \mid v \in [\![\, e' \,]\!]\} & \text{By induction hypothesis.} \\ & = & [\![\, Affine\ p\ q\ e' \,]\!] & \text{By definition of CAD } [\![\ ]\!]. \end{array}$$

Case *Union* e_1 e_2**:** Let m_1, m_2 represent $compile(e_1)$ and $compile(e_2)$ respectively. $mBop(Union)$ first splits m_1 and m_2 to generate two meshes m_1' and m_2' which satisfy Property 3. Lemma 13 ensures that the split meshes are semantically equivalent to the original meshes.

$$\begin{array}{rcll} [\![\, compile(Union\ e_1\ e_2) \,]\!] & = & [\![\, mBop(Union)(m_1, m_2) \,]\!] & \text{By definition of } compile(). \\ & = & [\![\, m_1 \,]\!] \cup [\![\, m_2 \,]\!] & \text{By Lemma 14.} \\ & = & [\![\, e_1 \,]\!] \cup [\![\, e_2 \,]\!] & \text{By induction hypothesis.} \\ & = & [\![\, Union\ e_1\ e_2 \,]\!] & \text{By definition of CAD } [\![\ ]\!]. \end{array}$$

[1] Cuboid can be obtained from Cube by applying non-uniform Scaling.

□

Lemma 11. $[\![\, m_{cube} \,]\!] = [\![\, Cube \,]\!]$

Proof. ($\subseteq$) Suppose $pt \in [\![\, m_{cube} \,]\!]$. Then there exists d such that $\mathsf{InsideVia}(m, pt, d)$. Let h be the half-line from pt in direction d. Since m_{cube} is convex, there is exactly one face through which h passes. Let f_0 be this unique face and consider the intersection of f_0 and h. Since h intersects m_{cube} exactly once, h must leave the cube at f_0. So just before leaving the cube at f_0, h is inside the cube. Unless $pt \in Cube$, h would leave the cube again.

($\supseteq$) Suppose $pt \in [\![\, Cube \,]\!]$. If pt is on the boundary of *Cube*, say on face f, then let d be the outward-facing normal of f, so that $h = (pt, d)$ intersects m_{cube} exactly once. On the other hand, suppose pt is in the interior of *Cube*. Then choose $d = (0, 0, 1)$, somewhat arbitrarily. Again, $h = (pt, d)$ intersects m_{cube} exactly once.

□

Lemma 12. For all meshes m and invertible affine transformations given by p and q,

$$\{pv + q \mid v \in [\![\, m \,]\!]\} = [\![\, \mathsf{map}_{\mathsf{vertex}}\ (\lambda v.\ pv + q)\ m \,]\!].$$

Proof. Let pt be arbitrary.

$$\begin{aligned} pt \in \{pv + q \mid v \in [\![\, m \,]\!]\} &\iff p^{-1}(pt - q) \in [\![\, m \,]\!] \\ &\iff \exists d.\ \mathsf{InsideVia}(m, p^{-1}(pt - q), d) \\ &\iff \exists d'.\ \mathsf{InsideVia}(\mathsf{map}_{\mathsf{vertex}}\ (\lambda v.\ pv + q)\ m, pt, d') \\ &\iff pt \in [\![\, \mathsf{map}_{\mathsf{vertex}}\ (\lambda v.\ pv + q)\ m \,]\!] \end{aligned}$$

□

Lemma 13 (Mesh splitting correctness)**.** Given two valid meshes, m_1 and m_2, $split(m_1, m_2, m'_1, m'_2)$ generates two meshes, m'_1 and m'_2 such that:

$$[\![\, m_1 \,]\!] = [\![\, m'_1 \,]\!] \quad \text{and} \quad [\![\, m_2 \,]\!] = [\![\, m'_2 \,]\!]$$

Proof. We prove $[\![\, m_1 \,]\!] = [\![\, m'_1 \,]\!]$; the proof for the second part is similar. Let pt be an arbitrary point. We show $pt \in [\![\, m_1 \,]\!] \iff pt \in [\![\, m'_1 \,]\!]$. Consider any halfline h that is good for pt and m'_1 (such a halfline exists by Theorem 1), and consider the points of intersection between h and the two meshes m_1 and m'_1. Since *split* ensures that the intersection of split faces are disjoint (Property 5), each point of intersection between h and m_1 lies on exactly one face of m'_1. Conversely, split also ensures that the union of split faces give the original face (Property 5), so each intersection point between h and m'_1 also lies on a face of m_1. Thus the intersection points along h are exactly the same for m_1 and m'_1. □

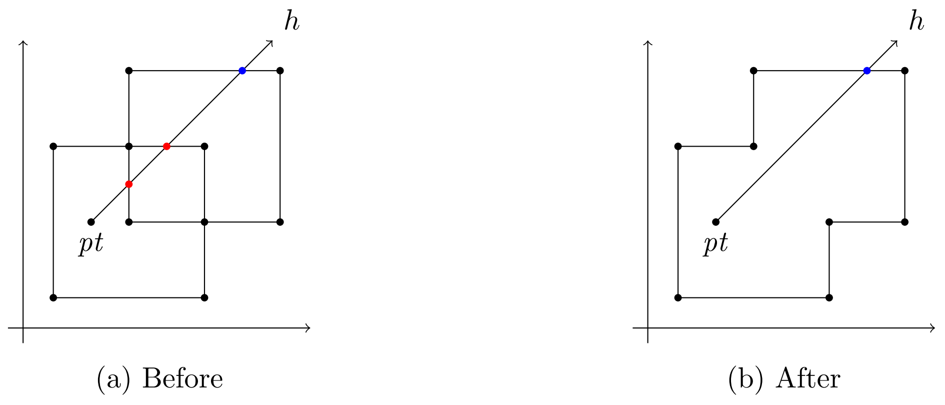


Figure 2.11: Examples demonstrating (in 2D) several cases in the proof of Lemma 14.

Lemma 14. For all meshes m_1 and m_2,

$$[\![\, mBop(Union)(m_1, m_2) \,]\!] = [\![\, m_1 \,]\!] \cup [\![\, m_2 \,]\!].$$

Proof. Let $m_3 = mBop(Union)(m_1, m_2)$ and let pt be an arbitrary point. We show $pt \in m_3 \iff pt \in [\![\, m_1 \,]\!] \vee pt \in [\![\, m_2 \,]\!]$. Consider a ray h that intersects only interior points of the faces of m_1 and m_2 (and thus also of m_3). It suffices to show that h crosses an odd number of faces in m_3 iff it crosses an odd number of faces of m_1 or an odd number of faces of m_2. Subdivide h into n contiguous regions h_i, separated by h's intersections with m_1 and m_2, which we call *crossing points.* The first region, h_0, starts at infinity and proceeds to the first crossing point. Each subsequent pair of regions is divided by a crossing point on the face of one or several of m_1, m_2, and m_3. These crossing points are not considered to be included in any regions. Finally, the region h_{n-1} ends at pt, which *is* considered a part of that region, since it is not itself a crossing point.

Since m_1 and m_2 are split, each h_i is entirely inside or outside of m_1 and m_2. Since each face of m_3 is a face of either m_1 or m_2, each region h_i is also entirely inside or outside of m_3. We now show that h_i is inside m_3 iff it is inside m_1 or inside m_2.
We proceed by induction on i:

Case $i = 0$: The statement follows since h_0 is the infinitely long region, which lies outside all three meshes.

Case $i + 1$: Consider the crossing that happens between h_i and h_{i+1}. There are 16 cases in total, depending on whether h_i and h_{i+1} lie inside or outside of m_1 and m_2. We illustrate four typical cases. A 2D example is described in Figure 2.11.

Case h_i outside m_1 and m_2; h_{i+1} inside m_1 but outside m_2**:** Consider the face f that divides h_i and h_{i+1}. The case hypothesis implies that f is a face of m_1 but not m_2. This further means that f is entirely outside of m_2, and so f is also a face of m_3 by definition of $mBop(Union)$. Thus h also crosses m_3 at f, and so h_{i+1} is inside m_3.

Case h_i inside m_1 but outside m_2; h_{i+1} inside both m_1 and m_2**:** The crossing face f is a face of m_2, which is entirely inside m_1, and thus not included in m_3. Inductively, h_i is inside m_3, and since f is not in m_3, h_{i+1} is also inside m_3.

Case h_i inside both m_1 and m_2; h_{i+1} outside both m_1 and m_2**:** The crossing face f is a face of both m_1 and m_2, and f's normals with respect to each mesh point in the same direction. Further, these normals are on the same side of f as pt. Thus f is a face of m_3. Inductively, h_i is in m_3, and so it crosses out of m_3 for h_{i+1}.

Case h_i inside m_1 but outside m_2; h_{i+1} inside m_2 but outside m_1**:** The crossing face f is a face of both m_1 and m_2, but the normals point in opposite directions. In m_1, the outward normal is on the same side as pt, while for m_2 it is on the opposite side. Thus, no copy of the face appears in m_3. Inductively, h_i is in m_3, and since the crossing face is not in m_3, h_{i+1} is as well.

The remaining cases are similar.

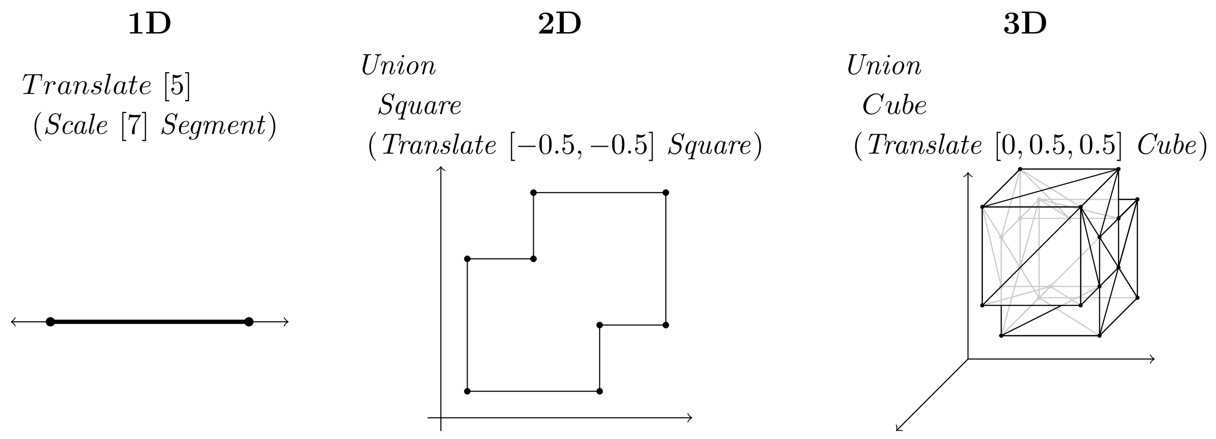
□

2.5 Implementation and Challenges

λCAD and its compiler are implemented in OCaml. Implementing the CAD compiler required several nontrivial computational geometry routines, which involved issues from 3D geometry as well as numerical computing. This section describes some design decisions targeted at reducing the burden of implementing the compiler.

1D → 3D

Problems that arise in 3D geometry often have analogous problems in lower dimension. Understanding which parts of the problem cut across all dimensions versus those that arise only in 3D helped us develop clean solutions that are as dimension-agnostic as possible. To that end, we first implemented a 1D CAD compiler, then moved on to 2D and finally to 3D. λCAD supports all three dimensions. An example of a dimension agnostic concept in our compiler is the technique for compiling CSG operations in Section 2.2.1. On the other hand, the technique for finding the intersection of a face and a halfline in Section 2.1.2 is


1D

Translate [5]
  (*Scale* [7] *Segment*)

2D

Union
  Square
  (*Translate* $[-0.5, -0.5]$ *Square*)

3D

Union
  Cube
  (*Translate* $[0, 0.5, 0.5]$ *Cube*)

Figure 2.12: Examples of 1D, 2D, 3D CADs and meshes. In order to keep the figures simple, the axes are not shown to intersect at the origin.

a geometric operation, which is more complex in 3D where faces are triangular planes and halflines are 3D rays than in 2D where faces are 2D segments and halflines are 2D rays.

1-dimensional CAD : 1D CAD objects are simply line segments represented by 1D end points. The only affine transformations in 1D are translation and scaling. The binary set-theoretic operations are analogous in all dimensions. A 1D CAD compiler compiles a 1D CAD program to generate a 1D *mesh*. Figure 2.12 shows a 1D CAD program and the corresponding mesh. *Segment* represents a unit segment starting at 0 and ending at 1. A face of a 1D mesh is merely a 1D point. As explained in section 2.1.2, a *valid* 1D mesh should not have repeating faces or odd number of faces.

2-dimensional CAD : 2D CAD objects include rectangles, squares, circles, triangles etc. In 2D, affine transformations include those from 1D (i.e. translation and scaling, but with 2D vectors) together with rotations about the origin. A 2D mesh consists of faces that are line segments ending in vertices. Figure 2.12 shows a 2D CAD program and the mesh generated by our compiler.

3-dimensional CAD : In 3D, rotations about many different axes are possible. In our implementation, we provide convenient syntax for rotating about the coordinate axes, X, Y, Z. Translation and scaling use 3D vectors. Figure 2.12 shows a 3D CAD program and the corresponding triangular mesh.

2.5.1 Fully Functorial Design

We designed our compiler infrastructure in a hierarchical manner using a *fully functorial* approach which allows us to swap out components of the compiler with other implementations. OCaml's module system facilitated this design decision. This is particularly useful for differential testing our compiler against other solid geometry based tools such as OpenSCAD OpenScad [2019], swapping our geometry module with another computational geometry library for comparison, and in tackling numerical issues. The geometric functionalities in our compiler and synthesis implementation are conceptually designed to execute using real numbers. Since reals are only approximated by floating point numbers, running these routines using floating point often leads to rounding errors due to semantic mismatch between floats and reals, and undecidable branching. We implemented several number systems with varying levels of accuracy and were able to use them interchangeably as and when required. All modules are functorized over a *number system*, whose signature contains basic arithmetic, square root, and trigonometric operations.

2.5.2 Limitations and Future Work

This section briefly surveys some opportunities for future work to build upon our programming-languages foundation for 3D printing tools. We focus on numeric and computational geometry challenges to improving mesh-to-CAD synthesis in particular and discuss some directions for exploring further stages of the 3D printing software pipeline.

Exact Arithmetic

One challenge to implementing the semantics described in Sections 2.1.1 and 2.1.2 is the need to implement mathematical operations such as square roots and trigonometric functions. Standard floating-point arithmetic and its inherent rounding error is unattractive for reasoning about numerical equivalence Goldberg [1991], Panchekha et al. [2015]. However, standard exact approaches such as rational arithmetic lack support for trigonometric functions, which are essential in geometry. We have started to investigate the problem of accurate mathematical computation based on the insight that most *angles* in CAD programs are rational multiples of π. Such values are algebraic, so can be represented in a *splitting field* Artin [2011] of the rational numbers with exact operations and decidable equality/inequality. We can choose a representation of the splitting field where any number is represented by the field size n, integer coefficients a_i and denominator d, representing the value

$$\frac{1}{d}\sum_{i=0}^{n-1} a_i \cos\frac{\pi i}{2n}$$

We implemented a prototype of arithmetic operations over these values, including decidable ordering and equality functions, and symbolic square root and arctangent functions, all free from any rounding error. With an overhead of roughly 600×, these exact operations are substantially slower than floating-point operations, but competitive with arbitrary-precision packages. Thanks to our fully-functorial design, users can choose whether to use floating-point or exact arithmetic for their CAD programs, depending on whether speed or high assurance is more important to them. Independently from its benefit to users, this design also allowed us to easily test and debug floating-point code. As a result, though our CAD compiler carries weaker guarantees when run in floating-point mode, we have fairly high confidence that the code is correct. In the future, we would like to pursue this direction further and investigate ways to make the number system more complete and performant.

Hull

Computing the *convex hull* of an object can be added as a built-in unary operator in CAD, and it is a useful one provided by various other tools. The λCAD implementation already has support for convex hull, but it causes a number of semantic complications that we have not yet fully investigated. Notably, the denotation of the hull operation is not compositional — we need to "inspect" the object whose hull is being computed for things like minimal and maximal points in various dimensions. Semantically this is no problem since we can specify such points with existential quantifiers, but the connection to how hull is compiled is much more subtle.

Challenges in Computational Geometry

Implementing computational geometry involves numerous challenges relating to robustness and performance Demmel and Hida [2004]. Many of these challenges are due to numerical precision problems Shewchuk [1997] and as mentioned in Section 2.5.2, we have started some preliminary investigation in this direction. However, this work's main focus has been on using programming languages to address orthogonal issues of formal specification, correctness and synthesis for CAD. These ideas generalize beyond the details of specific computational geometry techniques and can serve as a foundation for future research.

Compiling down to G-code

Section 1.1 described that after compiling a CAD program to a mesh, there are two more main compilation steps: *slicing* and generation of G-code. We have implemented prototypes of both but have not yet proven correctness in terms of semantics. Slicing inherently introduces approximation via discretization since each slice must have a small but nonzero height. Also, not all approaches to slicing produce achievable print strategies due to issues like gravity and the size of the printer.

2.6 Related work

This section discusses related projects and tools on semantics and sound compilation of computational geometry.

2.6.1 Soundness and formal semantics for 3D modeling

Recently, Sherman et al. Sherman et al. [2019] proposed a technique for sound and robust solid modeling using exact arithmetic and continuity in a library called StoneWorks. Here, continuity implies that to compute the output of a function's application to any finite level of precision, it is sufficient to represent the input to the function to a finite level of precision. They introduce a new representation for solids, compact-representation (*k-rep*), that can express complex properties of 3D solids such as volume, nonemptiness, and Hausdorff distance. They rely on Marshall Bauer [2008], CGAL [2019], which is a functional programming language supporting exact real arithmetic. To support computation of volume and other properties, they extend Marshall with Booleans and integrals. This variant of Marshall is called MarshallB. The semantics of MarshallB are based on category theory and topology. StoneWorks supports two different representations for solids: k-rep and open representation (o-rep) Edalat and Lieutier [2002]. The o-rep of a shape is a partial map such that $o(x) = tt$ if x is in the interior of the shape, $o(x) = \mathit{ff}$ if x is in the exterior of the shape. o diverges for boundary points. A k-rep is defined as the following mapping: $k\colon (E \to B) \to B$. For a predicate, $p\colon E \to B$, $k(p) = tt$ if the predicate is true for all points in a shape, $k(p) = \mathit{ff}$ if the predicate is false for some point in the shape. To evaluate the language's implementation, they show the example of a ray tracing algorithm and verify a cam-piston system. Since the library is implemented in exact arithmetic, there are no rounding errors due to floating point, which makes their implementations sound. On the other hand, exact arithmetic makes the system slow—it took more than two hours to create a 2D depth map from a 3D model whereas for commercial systems it takes seconds to generate a depth map.

Edalat et al. Edalat and Lieutier [2002] use a domain theoretic approach to reason about the soundness of a solid modeling framework. They introduced O-reps as a language for solids. Both Sherman et al. and Edalat et al. have demonstrated that their frameworks can soundly compute complex features such as Minkowski sums. Unlike Sherman et al. the latter authors do not present an implementation of their framework. However, O-reps Edalat and Lieutier [2002] have been implemented by Sherman et al. in MarshallB.

Hoffman et al. Hoffmann et al. [1988] proposed approximate primitives that are ϵ-correct.

In this definition, a polygon is ϵ-correct if the vertices are within ϵ of a model polygon. The authors present sound and robust implementations of several geometric algorithm such as line segment intersection, and polygon intersection. However, they were not able to prove the robustness of polyhedral intersections which makes it unclear whether their approach can be easily applied to 3D geometry. They were also not able to guarantee that their algorithms would have good numerical behavior.

Boundary-representations (B-reps) and Function-representations (F-reps) Pasko et al. [1995] are two other representations used in solid modeling and computer graphics. B-reps are the standard in many state-of-the-art CAD tools due to their rich expressivity. They can represent many complex functions such as chamfers and fillets. F-reps define objects as continuous real-valued functions over points. Sharma et al. Yap and Sharma [2008] proposed a system that can provide similar soundness guarantee for geometric computations as Sherman et al, but on F-reps. Even thought B-reps are a popularly used data structure, there are no frameworks that have been able to reason about soundness of algorithms that use B-reps due to their complex representation.

Sherman et al.'s StoneWorks is closely related to λCAD and its semantics. λCAD has been the first attempt towards viewing the CAD/CAM pipeline as a compiler and applying PL reasoning to this domain. While StoneWorks introduces a new representation for 3D solids (k-reps), λCAD formalizes an existing widely used representation, CSG. The StoneWorks semantics relies heavily on a combination of topology and category theoretic reasoning of various geometric operations. In λCAD, the semantics of CAD are based on regular open sets and the semantics of meshes are based on ray-intersections. Stonework's semantics are more complex than λCAD's, but on the other hand, while λCAD's semantics can be applied to reason about simple geometric operations such as affine transformations and boolean operations, StoneWorks can reason about more advanced concepts like Minkowski sums.

2.6.2 Compilers for 3D Printing

Sutherland's Sketchpad Sutherland [1964], invented in the 1960s, is one of the first computer-aided design tools. It revolutionized the field of graphical user interfaces and computer-aided simulations. Since then, numerous CAD tools have been developed Solidworks [2018], Rhinoceros [2018], SketchUp [2018]. Unfortunately, many of these are proprietary and do not provide clear semantics, so it is difficult to reason about them formally.

The core CAD components of λCAD are similar to OpenSCAD OpenScad [2019], which is a popular programmatic CAD tool in the 3D design sharing community Thingiverse [2019]. OpenSCAD builds on the CGAL CGAL [2018] computational geometry library. There are other programmatic CAD languages such as ImplicitCAD ImplicitCAD [2019] which is implemented in Haskell. ImplicitCAD is similar to OpenSCAD but provides more functionality. Our language and formalism inherits certain restrictions that are also present in OpenSCAD such as lack of direct support for fillets. Unlike our tool, both OpenSCAD and ImplicitCAD lack formal semantics for reasoning about CAD programs. We show that functional programming techniques can be extended to provide a rigorous foundation for reasoning about the implementation and composition of CAD tools. Several projects have investigated 3D-printing performance. WirePrint Mueller et al. [2014a] and faBrickator Mueller et al. [2014b] show how non-uniform height slicing and hybrid build approaches can expedite rapid prototyping. OpenFab Vidimče et al. [2013b] is a framework for specifying material and texture properties for 3D printing with the help of a domain-specific language.

Several projects have developed CAD compilers for unconventional tasks like automated knitting McCann et al. [2016]. There are also design tools to use 3D printing for modifying existing objects Chen et al. [2016, 2015] and tools that allow users to correct for measurement errors in CAD models Kim et al. [2017]. Dumas et al. Dumas et al. [2015] proposed a texture-synthesis technique that can be used to synthesize texture based on input patterns. Schulz et al. Schulz et al. [2014] have designed a system that lets casual users design 3D models by example. They first create a database of design templates based on designs by experts, and then let users choose a template and change the parameters. We have previously proposed using PL techniques for 3D-printing, but presented only a preliminary vision without results Nandi et al. [2017].

2.6.3 Analysis of CAD Models

CAD models can be analyzed before printing to check for structural defects using properties related to materials and geometry Stava et al. [2012], Zhou et al. [2013]. There are interactive interfaces McCrae et al. [2014] that let user specify functional parts and provide real-time simulations visualizing stress. Print orientation is a well studied area that focuses on statically analyzing CAD models for maximizing mechanical strength Umetani and Schmidt [2013]. Other constraints to optimize for could be minimal material usage. Patching existing prints Teibrich et al. [2015] and analyzing strength properties at the CAD level Galjaard et al. [2015] are two techniques to accomplish this. Smooth surface finish is another interesting requirement. Delfs et al. Delfs et al. [2016] developed a tool that achieves smooth surfaces by optimizing the orientation of the part during printing. Krishnamurthy et al. Krishnamurthy and Levoy [1996] introduced a technique that uses b-splines to smooth models at the mesh level. This work has witnessed tremendous application in the graphics community for rendering 3D characters.

2.7 Conclusions

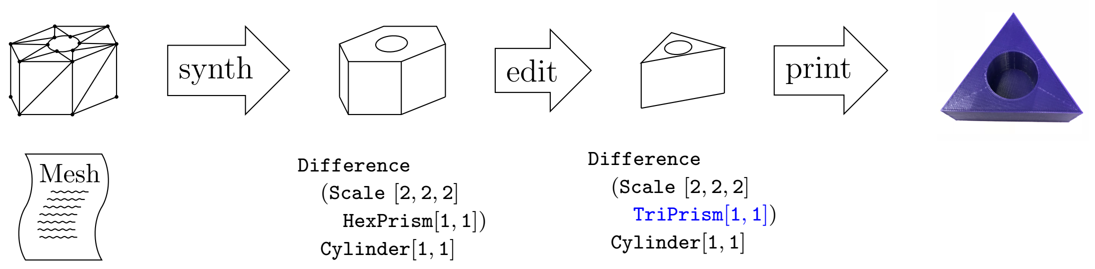
This chapter presented a functional-programming approach to designing and implementing computer-aided design tools. It introduced λCAD, a functional programming language that supports CAD features, and formalized both λCAD and surface mesh using denotational semantics. This chapter then presented a compiler from λCAD to mesh and provided a correctness proof using the concept of regular-open sets.

Chapter 3

DECOMPILING MESH TO CSG

To demonstrate how the foundations established in Chapter 2 can help develop better tools for desktop-manufacturing users, this chapter describes a novel algorithm for "reverse compiling" meshes to CAD programs that recaptures the high-level structure of a design; our CAD and mesh formalizations suggest a natural search strategy for synthesis. It turns out that reverse compilation may have the potential to solve a key problem for the current state of the 3D-printer enthusiast community: Many hobbyists and makers lack the requisite expertise to translate their ideas into CAD programs from scratch. To overcome this barrier, they often download and print existing designs from online communities Thingiverse [2019], GrabCAD [2019] where experts share their work freely. These repositories distribute designs as *polygon meshes* instead of CAD programs because CAD does not have standardized representations, so meshes, in the standard STL format, are the cross-platform distribution language. However, users are rarely able to *customize designs* shared as meshes Alcock et al. [2016], Hudson et al. [2016]. Mesh modification tools Meshmixer [2018] are useful for only some types of low-level modifications, and even then are difficult to use because they can easily break the model, thus making it invalid and unprintable. In large part, this is because mesh models have had all high-level design information "compiled away," analogous to how program binaries have had high-level operations compiled down to primitive machine operations.

Reverse compilation extracts high-level structural information from the design that enables rich edits to the design, which would otherwise require tedious low-level edits directly on the surface mesh. The algorithm in this chapter combines basic computational geometry with program synthesis to elegantly search the space of possible CAD programs. It repurposes the traditional PL machinery of evaluation contexts to guide the search of the synthesis algorithm toward the lowest-cost (ideally, the most "human-editable") CAD program. We built the algorithm in the form of a tool, Reincarnate. Figure 3.1 shows Reincarnate's workflow using an example of creating a triangle candle holder by starting with a hexagonal one in mesh form, synthesizing CAD, then changing the primitive before printing the desired object.

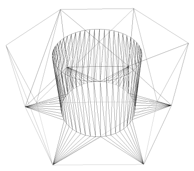


```
Difference
  (Scale [2,2,2]
    HexPrism[1,1])
  Cylinder[1,1]
```

```
Difference
  (Scale [2,2,2]
    TriPrism[1,1])
  Cylinder[1,1]
```

Figure 3.1: Reincarnate's Synthesis workflow: starting with a mesh for a 3D model, Reincarnate reverse engineers a CAD program in λCAD, which can be easily edited to get a different 3D model.

(a) Candle holder.


(b) Rendered mesh.

```
facet normal 0.866025 0.5 0
  outer loop
    vertex 1.0 0.0 1
    vertex 0.5 0.866025 0
    vertex 0.5 0.866025 1
  endloop
endfacet
```

(c) Snippet from the STL mesh.

Figure 3.2: Candle holder from Thingiverse Thingiverse [2018a] after printing, a 3D rendering of its STL mesh, and a snippet from the STL mesh showing one triangular face. Each face is represented by three vertices and a normal vector that points outward from the 3D object.

3.1 *Synthesis Example*

Consider the model of a hexagonal candle holder from Thingiverse Thingiverse [2018a] shown in Figure 3.2a. Like most models shared in online repositories, it is shared as a mesh. (Figure 3.2b shows the rendering of the mesh.) Figure 3.2c shows a very small snippet from the mesh showing just one face and its normal direction vector. The full mesh is made of 548 triangular faces and is about 4000 lines long in the STL format. This vast sea of triangle vertices does not explicitly convey structural information about the object's shape.

A user may want to make different modifications to the model. For example, they may wish to (1) change the depth/width of the candle hole, (2) tilt the hole (to make a holder for other items), (3) change the shape of the hole from a cylinder to a star-like prism or a cuboid, or (4) make a larger holder for two candles by combining two copies of the object. Making these edits to the mesh is non-trivial because the user must maintain certain geometric well-formedness constraints in order to ensure that the model is printable. On the other hand, having access to a higher-level representation of the model that contains more structural information such as any CAD representation would make these tasks quite easy. Figure 3.3a shows the code for this model in our language, λCAD. It shows that to make this model, one can create a 6-sided prism primitive (`HexPrism`) and subtract a (high-degree approximation of a) cylindrical hole (`Cylinder`) [1] from its center. With access to this program, editing it is straightforward. Figure 3.3b shows a modification to the model that changes the dimensions of the hole; an example of a modification requested by a user on the Thingiverse website Thingiverse [2018a].

To summarize, (1) designing 3D models in CAD from scratch is difficult but editing an existing CAD program is relatively easy, and, (2) sharing models in a standardized mesh format makes them more accessible to users but editing them is difficult and even impossible in some cases. Based on these two observations, we came up with an alternate strategy that has the best of both worlds: We describe the first synthesis algorithm (Section 3) that automatically *finds a CAD program from a surface mesh*. Our tool gives the users a high level CAD program to get started with and prevents them from having to make tedious mesh modifications while still allowing them to download mesh models from the internet.

3.2 Specifying Reverse Compilation

Just as in traditional compilation, translating a CAD program to a mesh loses source-level information. For example, consider the intersection of two cylinders placed side-by-side to form a rounded lozenge in Figure 3.5a. The resulting mesh only contains fragments of the cylinder primitives the programmer originally specified, yet intuitively we expect synthesis to “figure it out”.

To develop the synthesis algorithm, we first rephrase our CAD compiler as an equivalent small-step relation using evaluation contexts and then “flip the arrows” to formalize possible reverse compilations. The resulting *synthesis relation* captures the fact that many distinct CAD designs may compile down to the same mesh, leading us to introduce a notion of *geometric oracles* which model the mesh-level heuristics necessary to guide synthesis toward more-editable CAD programs. Following the synthesis relation, our algorithm provides a principled approach to reverse compiling meshes to CAD and enables proving properties of the algorithm as well as clearly delineating the role of heuristics. We assign specifications for the oracles and prove that our synthesis algorithm is correct, i.e., that it preserves semantics.

[1]Our approximation of a cylinder uses 50 sides.

```
Difference
  (Scale [4.0, 4.0, 2.0]
    HexPrism [1, 1]
  )
  (Scale [2.0, 2.0, 3.0]
    (Translate [0, 0, 0.3]
      Cylinder [1, 1]
    )
  )
```

(a) λCAD program for candle holder.

```
Difference
  (Scale [4.0, 4.0, 2.0]
    HexPrism [1, 1]
  )
  (Scale [2.5, 2.5, 3.5]
    (Translate [0, 0, 0.3]
      Cylinder [1, 1]
    )
  )
```

(b) Edited λCAD program in blue.

Figure 3.3: λCAD program for the hexagonal candle holder in Figure 3.2, and example of a modification that changes the dimension of the hole (in blue). The hexagonal outer part is represented by `HexPrism` (a cylindrical prism with 6 sides), and the hole is represented by `Cylinder`. As shown, in λCAD this edit is done by changing the scaling factor for the cylindrical hole.

The small step relation $\rightarrow_c$ for the CAD compiler from Chapter 2 is in Figure 3.4. It satisfies the property

$$c \rightarrow_c^* \mathit{Mesh}\ m \iff \mathit{compile}(c) = m$$

At this point, we could specify the target of synthesis as the inverse of $\rightarrow_c{}^*$. However, a key component of any mesh-to-CAD synthesis algorithm will be heuristics which infer information lost during compilation. To support reasoning about heuristics' role in synthesis, we instead define the synthesis relation $\rightarrow_s$ on the right of Figure 3.4. The geometric oracle Ω_{prim} provides the base case for synthesis by directly recognizing meshes that correspond to a sequence of affine transformations applied to a primitive, while Ω_{add} and Ω_{sub} indicate when a mesh can be generated by unioning or differencing two "simpler" meshes:

$$\begin{aligned} c \in \Omega_{\mathsf{prim}}(m) &\implies [\![m]\!] = [\![c]\!] \\ (m_1, m_2) \in \Omega_{\mathsf{add}}(m) &\implies [\![m]\!] = [\![\mathit{Union}\ (\mathit{Mesh}\ m_1)\ (\mathit{Mesh}\ m_2)]\!] \\ (m_1, m_2) \in \Omega_{\mathsf{sub}}(m) &\implies [\![m]\!] = [\![\mathit{Diff}\ (\mathit{Mesh}\ m_1)\ (\mathit{Mesh}\ m_2)]\!] \end{aligned}$$

Assuming these oracle specifications, $\mathit{Mesh}\ m \rightarrow_s^* c$ implies $\mathit{compile}(c) = m$. Also, in principle, there exist oracles such that $\mathit{compile}(c) = m$ implies $\mathit{Mesh}\ m \rightarrow_s^* c$. However, synthesis does *not* assume that its input was generated by our compiler; in fact, we intend synthesis to work for meshes obtained from arbitrary sources like online repositories and 3D

$$op ::= + \mid \text{-} \mid \times \mid / \qquad \text{num} ::= \mathbb{R} \mid var \mid num\ op\ num$$

$$vec2 ::= [num,\ num] \qquad vec3 ::= [num,\ num,\ num]$$

$$E ::= [\cdot] \mid affine\ \text{vec3}\ E \mid binop\ E\ C \mid binop\ (Mesh\ \text{m})\ E$$

$$S ::= A \mid binop\ S\ S \qquad A ::= P \mid affine\ \text{vec3}\ A$$

$$affine ::= \text{Translate} \mid \text{Scale} \mid \text{Rotate} \qquad binop ::= \text{Union} \mid \text{Diff}$$

$$P ::= Cuboid\ vec3 \mid Cylinder\ vec2 \mid TriPrism\ vec2 \mid HexPrism\ vec2 \mid Sphere\ num$$

$$\frac{c \rightarrow_p c'}{E[c] \rightarrow_c E[c']} \qquad \frac{}{Cube \rightarrow_p Mesh\ m_{cube}} \qquad \frac{m \rightarrow_\Omega c}{E[Mesh\ m] \rightarrow_s E[c]} \qquad \frac{p \in \Omega_{\mathsf{prim}}(m)}{Mesh\ m \rightarrow_\Omega p}$$

$$\frac{\mathsf{map}_{\mathsf{vertex}}\ (\lambda v.\ pv + q)\ m = m', vec = p \times q}{affine\ vec\ (Mesh\ m) \rightarrow_p Mesh\ m'} \qquad \frac{(m_1, m_2) \in \Omega_{\mathsf{add}}(m)}{Mesh\ m \rightarrow_\Omega Union\ (Mesh\ m_1)\ (Mesh\ m_2)}$$

$$\frac{mBop(binop)(m_1,\ m_2) = m'}{binop\ (Mesh\ m_1)(Mesh\ m_2) \rightarrow_p Mesh\ m'} \qquad \frac{(m_1, m_2) \in \Omega_{\mathsf{sub}}(m)}{Mesh\ m \rightarrow_\Omega Diff\ (Mesh\ m_1)\ (Mesh\ m_2)}$$

Figure 3.4: Representative cases of small step CAD compilation ($\rightarrow_c$, left) and synthesis ($\rightarrow_s$, right) with evaluation contexts and synthesis target language (E, S). *Cube* is syntactic sugar for *Cuboid*[1, 1, 1].

scanners. In such cases, it is impossible to know what CAD operations (if any) were used to generate the input model.

Synthesis is inherently under-constrained since there is never a unique CAD program that compiles to a given mesh, e.g., for all c, $[\![c]\!] = [\![\mathit{Union}\ c\ c]\!]$. Furthermore, for any mesh m, there is a trivial "complete" synthesis strategy: simply map each face of m to the base of an appropriate inward-facing tetrahedron and take the intersection of the resulting set of tetrahedrons. Such approaches are clearly undesirable as they fail to recover any of the higher-level structure of the original design. To address this, synthesis depends on a ranking function $c_1 \leq_{\mathsf{edit}} c_2$ to capture the notion that c_2 is "more editable" than c_1. In general, the right choice for $\leq_{\mathsf{edit}}$ will depend on how a user wants to customize a given design, but we have found that program size serves as a good default proxy.

Another challenge is the branching factor in the search space of CAD programs. To help mitigate this issue we restrict the target language of synthesis to the subset S in Figure 3.4. S only has two binary operators: *Union* and *Diff* and they are both positioned above all affine transformations which in turn are positioned above all primitives in the program's AST. Intuitively, these restrictions are mild since intersections $A \cap B$ can be equivalently expressed as differences $A - (A - B)$ and affine transformations distribute through CSG operations. Two key benefits of this approach are that it focuses the search space by eliminating many equivalent candidates and also suggests a high level strategy composing the primitive, additive, and subtractive oracles. One downside is that CAD programs where affine operations have all been distributed down below CSG operations can be substantially larger. S treats the sugared versions of the affine transformations as primitives — synthesized programs therefore do not contain expressions with *Affine* $\mathbb{R}^{3\times3}\mathbb{R}^3$, instead they contain expressions with *Translate*, *Rotate*, *Scale* (Figure 2.2). While *Affine* $\mathbb{R}^{3\times3}\mathbb{R}^3$ would be a much more concise representation as it can combine multiple transformations into one, the choice of synthesizing the sugared tranformations makes the programs more readable, even if potentially longer. Chapter 4 shows a novel technique based on equality saturation that further improves editability and optimizes the programs once a mesh has been synthesized up to the CAD level.

Given these definitions and design considerations, we can prioritize some general guidelines for mesh-to-CAD synthesis algorithms with oracles Ω:

Correct: Synthesis must preserve semantics, $[\![\mathit{synth}_\Omega(m)]\!] = [\![m]\!]$.

Useful: Synthesis should strive to generate editable CAD models, i.e., maximize $\leq_{\mathsf{edit}}$.

Predictable: Synthesis should be deterministic in that $\mathit{synth}_\Omega(\mathit{compile}(\mathit{synth}_\Omega(m))) = \mathit{synth}_\Omega(m)$.

Complete: $\leq_{\mathsf{edit}}$ should prefer CAD models without embedded meshes (e.g., in S).

(a) Information lost in compilation. (b) Composition depends on context.

Figure 3.5: Figure 3.5a shows how compiling a CAD to a mesh leads to loss of high level structural information (the fact that the lozenge shape is obtained by intersecting two cylinders). Figure 3.5b shows how evaluation context can be used to synthesize the union of two spheres.

3.3 Decompilation Algorithm

Algorithm 3.6 shows our synthesis strategy $\mathit{synth}_{\Omega}()$. The core search_{Ω} function maintains a worklist of candidate CAD programs reachable from the input mesh by the $\rightarrow_s$ relation. In each iteration, it pops the most promising candidate c from the front of the worklist, *focuses* on a particular mesh m within c, applies oracles in Ω to m to generate new candidates, and *schedules* those candidates in the worklist. The algorithm is bounded by a *fuel* parameter to ensure termination and once it runs out or no candidates remain, search_{Ω} returns the most editable result according to $\leq_{\mathsf{edit}}$.

Our synthesis algorithm is designed to be modular: it is straightforward to implement and add new oracles to synthesize a greater variety of CAD programs and control the search by modifying the *fuel*, *focus*, *schedule*, and $\leq_{\mathsf{edit}}$ parameters. Below we describe strategies for effectively implementing geometric oracles and setting these parameters. Since this the goal of this section is to demonstrate the utility of our programming language foundation for CAD, we describe geometric heuristics at a high level.

Ω_{prim} This oracle recognizes meshes that can be generated by CAD programs in language A from Figure 3.4, i.e., a sequence of affine transformations applied to a basic primitive. This is straightforward when the mesh corresponds to a primitive in language P, but is more challenging for meshes which correspond to rotated, translated, scaled, or skewed versions of a primitive. In such cases, the oracle implementation *canonicalizes* the input mesh and compares it to canonicalized versions of primitives. If a match is found, it *reorients* the

```
$synth_\Omega(m) \;=\; search_\Omega([Mesh\ m],\ [],\ fuel)$
$search_\Omega(cs,\ fs,\ 0) \;=\; \mathsf{max}_{\leq_{\mathsf{edit}}}\ (fs \mathbin{++} cs)$
$search_\Omega([],\ fs,\ fuel) \;=\; \mathsf{max}_{\leq_{\mathsf{edit}}}\ fs$
$search_\Omega(c :: cs,\ fs,\ fuel) \;=$
    $\mathsf{let}\ E[Mesh\ m] = focus(c)\ \mathsf{in}$
    $\mathsf{let}\ ps = \mathsf{map}\ (\lambda c.\ E[c])\ \ \Omega_{\mathsf{prim}}(m)\ \mathsf{in}$
    $\mathsf{let}\ as = \mathsf{map}\ (\lambda(m_1, m_2).\ E[Union\ \ (Mesh\ m_1)\ \ (Mesh\ m_2)])\ \ \Omega_{\mathsf{add}}(m)\ \mathsf{in}$
    $\mathsf{let}\ ss = \mathsf{map}\ (\lambda(m_1, m_2).\ E[Diff\ \ (Mesh\ m_1)\ \ (Mesh\ m_2)])\ \ \Omega_{\mathsf{sub}}(m)\ \mathsf{in}$
    $\mathsf{let}\ (fs', cs') = \mathsf{partition}\ (\lambda c.\ c \in \mathrm{S})\ \ (ps \mathbin{++} as \mathbin{++} ss)\ \mathsf{in}$
    $\mathsf{let}\ cs'' = \mathsf{fold}\ schedule\ cs\ cs'\ \mathsf{in}$
    $search_\Omega(cs'',\ c :: fs' \mathbin{++} fs,\ fuel - 1)$
```

Figure 3.6: Core synthesis algorithm.

mesh using the inferred canonicalization parameters and returns the result. We describe canonicalization in more detail in Section 3.3.4.

To implement mesh matching, we designed *recognizers* for the basic primitives in P. These recognizers use geometric properties of the corresponding primitive 3D solids. For example, in order to recognize a cuboid, we check that the mesh is composed of 6 *face groups* (sets of adjacent faces with equivalent normals), and use the normals of each group to invert any affine transformations which may have been applied to the underlying *Cube* primitive[2]. To recognize spheroids, we similarly check for (potentially multiple) centroids that have equivalent distances to the faces of the mesh. Similarly, for cylinder and hexagon, we partition the mesh into face groups and use normals to heuristically invert affine transformations.

Ω_{add} We experimented with several mesh splitting strategies for this oracle, and ultimately settled on three high level strategies. *Disjoint* split partitions the mesh by connected components. *Convex split* identifies a splitting based on rings of coplanar gradient changes. *Group split* identifies common features between face groups, e.g., being parallel/orthogonal, and separates the mesh along those boundaries.

[2] *Cube* is syntactic sugar for *Cuboid* [1, 1, 1]

Ω_{sub} Given a mesh m, this oracle searches for a *bounding mesh* that snugly contains m and returns the bound and its difference with m. In our current implementation, we limit bounding meshes to those corresponding to CAD primitives. This can be relaxed, and we have observed examples where it would be useful to recursively synthesize more complex bounds, e.g., using convex hull.

Scheduling To effectively navigate the exponential synthesis search space, the function $search_{\Omega}$ prioritizes meshes in its worklist deemed more likely to lead to editable CAD programs. As one example from our current implementation of the *schedule* parameter, we detect when newly generated candidates have meshes of higher overall complexity than where they started and insert such candidates later in the worklist.

focus In general, a CAD program can match many evaluation contexts. For compilation, only one of these will leave a redex in the context's hole, but for synthesis we may choose any context that places a mesh in the hole. *focus* examines the full CAD program c and decomposes it to select the most promising mesh. In our current implementation, meshes are selected based on number of face groups and their height in the CAD syntax tree. Focusing has a significant impact on synthesis performance, and more work is needed to characterize the tradeoff between spending more time to accurately select the most promising mesh and quickly exploring many candidates. We speculate that the tradeoff is actually dynamic in the sense that it often seems to depend on the depth of a mesh within the CAD program.

3.3.1 Context and Sharing

The context of a mesh plays a critical role in synthesis. To understand, let's walk through the case of a union of two overlapping spheres[3] in Figure 3.5b. Ω_{add} will use convex splitting to break the model into two meshes, each of which corresponds to a *truncated sphere*. These two truncated sphere meshes appear under a *Union*. No affine transformed primitive can match these new meshes, i.e., Ω_{prim} cannot successfully infer a λCAD from them. Further splitting using different splitting oracles will only make the truncated sphere meshes smaller but not easier to infer a λCAD for. Similarly, using Ω_{sub} also does not lead to simpler meshes, it only moves the position of the complex mesh in the AST. Without evaluation context, synthesis would at best be able to return the union of the two meshes of the truncated spheres. However, in context, using full spheres to match the truncated spheres works correctly. This is because when we union the complete spheres, the parts of the spheres that are "lost" inside each other are correctly compiled away, as we explained in Section 2.2.

[3] This work approximates curves. To represent the sphere, we use a predefined mesh.

Another important implementation technique is sharing common structure among evaluation contexts to limit memory usage. While the worklist may grow exponentially, we can efficiently represent its contents by reusing common prefixes of each generated evaluation context across the explored programs.

3.3.2 Synthesis Correctness

We briefly sketch the correctness of our synthesis algorithm below and defer evaluating other criteria to the case studies in the following section. One property we rely on is that for any CAD programs c_1 and c_2 and evaluation context E, $[\![c1]\!] = [\![c2]\!]$ implies $[\![E[c1]]\!] = [\![E[c2]]\!]$ by the compositional definition of $[\![\ ]\!]$.

$[\![synth_\Omega(m)]\!] = [\![m]\!]$. We first show that $Mesh\ m \rightarrow_s^* c$ implies $[\![m]\!] = [\![c]\!]$. To get a strong enough induction hypothesis we generalize to prove that $c \rightarrow_s^* c'$ implies $[\![c]\!] = [\![c']\!]$, and proceed by induction on the derivation.

Base Case: For 0 steps, $c = c'$ and the goal trivially holds.

Inductive case: $c \rightarrow_s c'' \rightarrow_s^* c'$.
By inversion, $c = E\ [\ Mesh\ m\ ]$ and $c'' = E\ [\ c^*\ ]$ for some CAD c^*.
By the induction hypothesis, $[\![c'']\!] = [\![c']\!]$.
To show $[\![c]\!] = [\![c']\!]$, it is sufficient to establish $[\![c]\!] = [\![c'']\!]$. This follows from case analysis on the synthesis step, the oracle specifications, and the compositional definition of $[\![\ ]\!]$.

Now $[\![synth_\Omega(m)]\!] = [\![m]\!]$ follows from the invariant that all CAD programs in the worklist during search are reachable under the synthesis relation and the fact that the result of synthesis is drawn from the worklist.

□

3.3.3 Implementation and Challenges

We implemented the synthesis algorithm in the form of a tool, Reincarnate, to work with the compiler tools in Chapter 2. Due to the compiler's full functorial design strategy (Chapter 2), we were able to implement synthesis as an extension to the existing system by making `Synthesis` a functor over `NumSys`, `Geometry`, `Mesh` and `CAD`.

3.3.4 Canonicalization and Re-orientation

In order to match a mesh to a primitive (from a list of predefined primitives) in an arbitrary location and orientation in 3D space, $\Omega_{\textsf{prim}}$ performs canonicalization, which is a series of affine transformations applied at the mesh level. This normalizes a mesh with respect to affine transformations—for a mesh, m, and an arbitrary sequence of affine transformations, given by a matrix p and translation vector q, `canonicalize m` $=$ `canonicalize` $(pm+q)$. The order in which the series of affine transformations in canonicalization are applied is important due to the non-commutativity of affine transformations. The first step in canonicalization is to identify three mutually perpendicular axes of m. We do this by identifying three orthogonal directions along which the sum of the areas of the faces is the largest. We call these three axes, x_o, y_o, and z_o the object coordinate system. We already know the orthogonal coordinates of the Cartesian coordinate system: $x = (1,0,0)$, $y = (0,1,0)$, and $z = (0,0,1)$ (we call this the world coordinate system). `canonicalize` solves a linear system of equations using Euler angles Kim [2013] to find three rotations, about x, y, and z that can align the world coordinate system to the object coordinate system. Note that this is the opposite of our goal—we want to align the object coordinate system to the world coordinate system. `canonicalize` does this by using the angles [4] obtained from Euler equations, but applying them in the reverse order, and negating the value. If the Euler angles are r_x, r_y, and r_z, then after canonicalizing with respect to rotation, the new mesh is:

$$m_r = \texttt{Rotate}[-r_x, 0, 0]\ (\texttt{Rotate}[0, -r_y, 0]\ (\texttt{Rotate}[0, 0, -r_z]\ m))$$

After the axes are aligned, the next step is to `Scale` the mesh to unit dimensions. `canonicalize` does this by first computing the dimensions of the bounding box of m_r, (d_x, d_y, d_z) and scaling it by the reciprocal of the dimensions:

$$m_s = \texttt{Scale}[1/d_x, 1/d_y, 1/d_z]\ m_r$$

The final step of canonicalization is to place the center of m_s at the origin by `translation` by finding the bounding box of m_s and translating the mid point along each dimension to $(0,0,0)$. If c_x, c_y, and c_z are the centers along x, y, and z respectively, then

$$m_{canonicalized} = m_t = \texttt{Translate}[-c_x, -c_y, -c_z]\ m_s$$

Canonicalization is used for primitive matching. Once a primitive p is matched, to synthesize the correct CAD, p has to be *re-oriented* to the original location in 3D space. For this, the algorithm applies the above affine transformations to p in the order:

[4]Angles are represented in degrees in this Chapter.

$$p' = \texttt{Rotate}[0, 0, r_z](\texttt{Rotate}[0, r_y, 0](\texttt{Rotate}[r_x, 0, 0](\texttt{Scale}[d_x, d_y, d_z]\ p)))$$

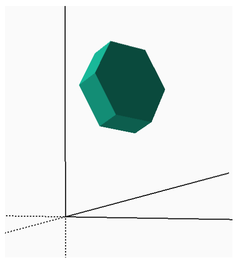
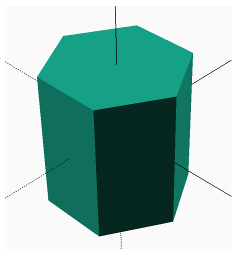
The last step is to translate the scaled and rotate primitive, p' to the right coordinates. The distance to be translated is the distance between the center of *compile*(p') and the center of the original mesh m: $(c_x^{p'}, c_y^{p'}, c_z^{p'}) - (c_x^m, c_y^m, c_z^m)$.

$$p_{oriented} = \texttt{Translate}[(c_x^{p'}, c_y^{p'}, c_z^{p'}) - (c_x^m, c_y^m, c_z^m)]\ p'$$

The elegance of canonicalization and reorientation for primitives is that it pushes the affine transformations to the leaves of the AST. This makes the rest of synthesis simpler because it saves us from finding canonical orientations of arbitrary binary combinations of CAD programs. This design decision was based on the key insight that while the order of application of affine transformations cannot be changed within themselves, when affine transformations appear with binary transformations, they ***can*** be pushed inside the binary operations.

3.3.5 A Concrete Instance of the Synthesis Algorithm

Following is a concrete example of how the Reincarnate algorithm in Figure 3.6 works. Consider the mesh m of the model in Figure 3.7 showing a hexagonal prism in arbitrary location in 3D space. Initially, *fuel* is greater than 0 and the worklist, cs has one candidate, *Mesh m*. Consequently, m is the mesh in *focus*. From Figure 3.6, we can see that the algorithm will attempt to apply all three oracles to m. Ω_{prim} will canonicalize the mesh and apply the primitive *recognizers*. Figure 3.7 (second figure) shows the canonicalized mesh. Since this already matches with an affine transformed primitive (`HexPrism [1, 1]`), ps will be a list containing the corresponding λCAD program. This program is shown in Figure 3.7. Next, the other two oracles, Ω_{add} and Ω_{sub} will also be applied. Ω_{add} will return the original candidate *Mesh m* since none of the splitting strategies will generate two sub-meshes from m. Ω_{sub} will return the same result as Ω_{prim} since in this case, the snuggest fitting bounding primitive is the affine transformed hexagonal prism. Hence, as will contain *Mesh m* and ss will contain the same λCAD program as ps. The algorithm will concatenate ps, as and ss and remove duplicates before applying the `partition` function. This will split the list into two parts: fs' will contain the λCAD program shown in Figure 3.7, and cs' will contain the remaining program, i.e. the result of applying Ω_{add}, which is the original program, *Mesh m*. Before *scheduling* the next mesh to focus on, the algorithm checks whether any of the current candidates were already explored in the previous step. In this case, *Mesh m* has already been explored, so it will be removed from the list of candidates. At this point, cs'' thus is an empty list. Therefore, according to the third line in Figure 3.6, it will now apply

```
Translate[2.4, 0.8, 5.2]
  (Rotate[0, 0, 80.5]
    (Rotate[0, −40.5, 0]
      (Rotate[−124.7, 0, 0]
        (Scale[2.0, 1.7, 1.0]
          HexPrism[1, 1]))))
```

Figure 3.7: Canonicalization and λCAD synthesis of arbitrary object in 3D space.

$\mathsf{max}_{\leq_{\mathsf{edit}}}$ to *fs*. Since *fs* contains only one λCAD program (shown in Figure 3.7), this will be the output of synthesis.

3.4 Evaluation

Table 3.1 shows the result of running Reincarnate on 10 meshes downloaded from Thingiverse Thingiverse [2019]. Reincarnate reduces the size of the programs by 22% on average and by up to 38.6%. Notice that for the `TackleBox`, there was no reduction in the size of the program compared to the number of triangles in the input mesh, exposing a limitation of Reincarnate— it produces only a flat program which can still be very long for repetitive models. Chapter 4 will show a unique solution to this problem using equality saturation.

This section also demonstrates three case studies on which we ran Reincarnate. Two of the case studies are downloaded from Thingiverse and one is our own design (Table 3.2). They are representative of three of the most common tasks end users of 3D printers typically tend to design for: tools parts, household items, and hobbyist designs Alcock et al. [2016]. In order to evaluate Reincarnate's output, we define six tasks:

- *scale* components of a model, for example a hole inside a bigger part.
- *translate* components of a model with respect to each other.
- *rotate* a model as a whole or part of it about one or more axes.
- *combine* two models or add a new component to an existing model.
- *remove* a component from a model.

Table 3.1: Results of running Reincarnate on 10 meshes downloaded from Thingiverse Thingiverse [2019]. # Tri shows the number of triangles in the mesh and c_{out} the output costs, i.e, AST size of programs generated by Reincarnate.

Id	# Tri	c_{out}	% Shrink
TackleBox	280	280	0
SDCardRack	236	206	12.7
CardFramer	200	172	14.0
CassetteStorage	172	141	18.0
CNCBitCase	268	219	18.3
RaspberryPiCover	332	271	18.4
ChargingStation	192	141	26.6
CircleCell	124	79	36.3
SingleRowHolder	320	198	38.1
HexWrenchHolder	516	317	38.6
Average	264.0	202.4	22.1

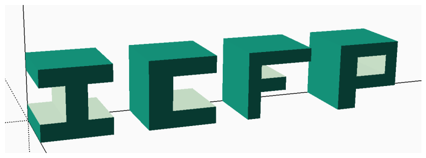
- *change # sides* in a regular polygon primitive. This could be for example changing a hexagonal prism to a cylinder or a pentagonal prism.

We give examples of editing tasks from the above categories for each case study to discuss the relative difficulty at both λCAD and mesh levels. Our overall conclusion is that editing a model after generating λCAD using Reincarnate is always easier or the same level of difficulty as editing the corresponding mesh using mesh editing tools. In several cases, editing the mesh model is as difficult as manually editing the triangular faces which is usually not recommended.

Table 3.2: Summary of case studies.

Benchmark	Source	Category
ICFP	original	hobby
Candle holder	Thingiverse	household
Hex wrench holder	Thingiverse	tool

```
Difference
  (Translate[1.5, -1.9, 0.5]
    (Scale[3.0, 6.0, 1.0]
      (Translate[-0.5, -0.5, -0.5]
        Cube
      )))
  (Union
    (Translate[2.5, -2.0, 0.5]
      (Scale[1.0, 4.0, 1.0]
        (Translate[-0.5, -0.5, -0.5]
          Cube
        )))
    (Translate[0.5, -2.0, 0.5]
      (Scale[1.0, 4.0, 1.0]
        (Translate[-0.5, -0.5, -0.5]
          Cube
        ))))
```



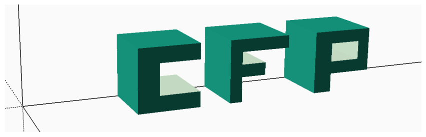


Figure 3.8: λCAD program for `I` synthesized by Reincarnate. Rendered λCAD programs for ICFP and CFP.

3.4.1 ICFP

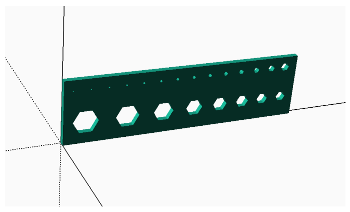
This model (shown in Figure 3.8) was entirely generated by our tools. We designed it in λCAD, compiled it to STL using our compiler (the mesh has approx. 150 faces), and then synthesized a λCAD program using Reincarnate. The synthesized programs for the individual letters `I, C, F, P` are 25, 16, 23, and 23 LOC respectively. The synthesized CAD program for the model `ICFP` has 89 LOC. Figure 3.8 shows the λCAD program for `I` synthesized by Reincarnate.

- Remove: Consider the task of removing a letter from the model. For example, one can remove the `I` to model the acronym for *Call For Papers* shown in Figure 3.8. Figure 3.10 shows the λCAD program for `CFP` that we obtained by editing the program Reincarnate synthesized for `ICFP`. The program is a union of the three letters. Since all the letters are separated, this task should also be relatively easy to perform at the mesh level.

- Translate: Consider an edit where the user wants to increase the spacing between all the letters uniformly. After running Reincarnate on the mesh, this task is easy: one needs to simply change the translation vector that is used to separate the letters. At the mesh level, this seemingly simple task can get confusing and tedious because the user has to drag the letters around to make the spacing uniform.

```
Difference
  (Translate[0, 0, 0.5]
    (Scale[2, 1.732, 1]
      (Translate[-0, -0, -0.5]
        (Scale[0.5, 0.57, 1.0]
          HexPrism[1, 1]
        ))))
  (Translate[0.0, 0.0, 0.55]
    (Scale[1.0, 0.998, 0.9]
      (Translate[-0.0, -0.0, -0.5]
        (Scale[0.5, 0.5, 1.0]
          Cylinder[1, 1]
        ))))
```

(a) Synthesized λCAD for the candle holder.



(b) Rendered hex holder.

Figure 3.9: Figure 3.9a shows the λCAD program that Reincarnate synthesized for the candle holder Thingiverse [2018a]. Figure 3.9b shows the hex wrench holder from Thingiverse Thingiverse [2018c].

3.4.2 Candle Holder

Our next example is the candle holder (shown in Figure 3.2 in Section 1.1) that we downloaded from Thingiverse Thingiverse [2018a]. This model is available in STL format, and also in a specific CAD format Rhinoceros [2018], which is only useful for users who have that CAD package. From a mesh with hundreds of triangular faces, Reincarnate produced a 20 line CAD program shown in Figure 3.9a.

- Comments from users Thingiverse [2018a] indicate that they were unsuccessful in modifying the mesh model to scale it only along z-axis using mesh editing tools—it would also change the x and y dimensions. From the user's comment, it seems like even though in theory this task is possible using mesh editing tools Meshmixer [2018], Blender [2018], it is much more tedious than editing the λCAD model where it is as simple as adding a *Scale* affine transformation with the right vector (*Scale*[1, 1, δ] for example, would suffice as Figure 3.3 shows).

- Rotating part of the mesh such as the cylindrical hole (about any axis perpendicular to the length of the cylinder) is as difficult as editing the mesh manually because it causes the triangular faces of the hole and the base to intersect with one another, thereby breaking the mesh. This task can be easily done at the λCAD level using

$Rotate(x, 0, 0)$, $Rotate(0, y, 0)$.

- Due to the same reason as above, combining the outer polygons to make a bigger base for more than one candle is nearly impossible at the mesh level but very easy at the λCAD level (using the *Union* operation).

- Changing the number of sides on the outer polygon is trivial at the λCAD level (it only requires replacing the primitive at a single location in the program as shown in Figure 3.1) but as difficult as manually editing the triangular faces at the mesh level.

3.4.3 Hex Wrench Holder

We were inspired to synthesize the CAD program for a hex wrench holder Thingiverse [2018c] by a hobbyist maker who downloaded a hex wrench holder mesh and 3D printed it only to find that his hex wrenches did not fit right due to the holes being oriented differently from the shape of his wrenches. The hobbyist tried to used a mesh editing tool to rotate the holes but it was impossible to do this edit because the triangulation of the mesh would break. We synthesized λCAD for the holder using Reincarnate. The entire λCAD program has 196 LOC (the mesh has over 500 faces). Figure 3.10 shows a part of the λCAD program.

- One can use $Rotate(0, 0, \theta)$ as shown in Figure 3.10 to rotate the holes easily in λCAD, but at the mesh level, it is very difficult.

- Scaling the holder holes is yet another task that is very tedious at the mesh level, especially if it is non-uniform. It has the same problem as rotation in that it causes the mesh to break due to intersections between faces.

- Adding or removing a hole are both very easy at the λCAD level because it requires one to simply scale the cuboidal base of the holder and either subtract another hexagonal prism (adding a hole) or union the model with a hexagonal prism of the right dimensions (removing a hole). These tasks while not impossible at the mesh level, are extremely tedious.

Notice that for repetitive models like this, the λCAD program Reincarnate synthesizes is still long. For this particular model (Figure 3.9b) the holes in the hex holder are all made using the same λCAD primitive, but they are separated by some distance and are scaled differently. The next chapter (Chapter 4) shows how we can further shrink such repetitive λCAD programs by inferring maps and folds automatically.

Synthesized λCAD for hex holder:

```
1  Difference
2    (Translate[65.0, 15.0, 2.5]
3      (Scale[130.0, 30.0, 5.0]
4        (Translate[-0.5, -0.5, -0.5]
5          Cube
6        )))
7    (Translate[5.0, 23.0, 2.5]
8      (Scale[2.0, 1.7, 0.5]
9        HexPrism[1, 1]
10     ))
11   ...)
```

Adding rotation about Z on line 9:

```
1  Difference
2    (Translate[65.0, 15.0, 2.5]
3      (Scale[130.0, 30.0, 5.0]
4        (Translate[-0.5, -0.5, -0.5]
5          Cube
6        )))
7    (Rotate[0, 0, 15.0]
8      (Translate[5.0, 23.0, 2.5]
9        (Scale[2.0, 1.7, 5.0]
10         HexPrism[1, 1]
11       )))
12   ...)
```

```
1  Union
2    (Union
3      (* C *)
4      (Difference
5        (Translate[2.0, 0.5, 0.5]
6          (Translate[-0.5, -0.5, -0.5]
7            Cube ))
8        (Translate[2.175, 0.5, 0.5]
9          (Scale[0.65, 0.5, 1.0]
10           (Translate[-0.5, -0.5, -0.5]
11             Cube ))))
12     (* F *)
13     (Difference
14       (Translate[3.5, 0.5, 0.5]
15         (Translate[-0.5, -0.5, -0.5]
16           Cube ))
17       (Difference
18         (Translate[3.6, 0.4, 0.5]
19           (Scale[0.8, 0.8, 1.0]
20             (Translate[-0.5, -0.5, -0.5]
21               Cube )))
22         (Translate[3.4, 0.5, 0.5]
23           (Scale[0.4, 0.2, 1.0]
24             (Translate[-0.5, -0.5, -0.5]
25               Cube ))))))
26   (* P *)
27   (Difference
28     (Translate[5.0, 0.5, 0.5]
29       (Translate[-0.5, -0.5, -0.5]
30         Cube ))
31     (Union
32       (Difference
33         (Translate[5.15, 0.15, 0.5]
34           (Scale[0.7, 0.3, 1.0]
35             (Translate[-0.5, -0.5, -0.5]
36               Cube )))
37         Empty i)
38       (Difference
39         (Translate[5.05, 0.65, 0.5]
40           (Scale[0.5, 0.3, 1.0]
41             (Translate[-0.5, -0.5, -0.5]
42               Cube )))
43         Empty )))
```

Figure 3.10: The top left code fragment is part of the λCAD program for the hex holder synthesized by Reincarnate. The bottom left code fragment shows an edit to a hole in the hex wrench holder by rotating the hole by 15 degrees about the z-axis on **line 9**. The λCAD program on the right shows the model for CFP that can be obtained from the λCAD program of ICFP synthesized by Reincarnate by deleting the **I**.

3.4.4 Limitations

Reincarnate's implementation currently works on geometric shapes that do not have rounded or sloped corners and edges. While this is not a limitation of the core synthesis algorithm, our primitive matching oracle (Ω_{prim}) is not capable of synthesizing features like chamfers and fillets. Using approximate techniques (e.g., RANSAC Du et al. [2018]) can allow Reincarnate to infer the shape underlying these features but further research is required for allowing Reincarnate to automatically infer a chamfer or fillet from a mesh. Reincarnate also does not use Hull. Hull is a powerful primitive; determining when its usage helps or hinders producing editable CAD models is an interesting problem for future work.

3.5 Related work

There are numerous examples from other fields such as human computer interaction, computational geometry, mechanical engineering, computer vision, and design, that have explored 3D models, mesh generation, slicing, and user interfaces to help mitigate current limitations in 3D printing. Below we highlight examples from other communities working on desktop manufacturing. We also provide an overview of state-of-the-art in program synthesis research and how it has been used for CAD in the recent years.

3.5.1 Recreating CAD Models

There are several tools for reverse engineering CAD from 3D scans Geomagic Design X [2018], Powershape [2018], SpaceClaim [2018]. The goal of these tools is to help experts manually (re-)create a CAD design. These tools enhance the traditional CAD workflow primarily by enabling an engineer to "snap" features and dimensions to points from a scan or mesh. Some of these tools also attempt to detect some features and suggest possible primitives (which is similar to the role of Ω_{prim} in our synthesis algorithm) or detect coplanar features. Since these tools are proprietary, few details about their implementations are available. These tools are designed to be interactively driven by an expert CAD engineer and do not produce full CAD programs from meshes.

Thingiverse Customizer Thingiverse [2018b] is a tool that allows one to modify 3D models uploaded on Thingiverse. It is however only useful for models that include the underlying CAD file. The majority of Thingiverse models do not have an accompanying CAD file, and consist only of mesh-level information in the form of STL files. Customizer cannot reverse engineer CAD programs from the STL meshes, which is the novelty of Reincarnate.

3.5.2 Applications

Desktop-class 3D printing has started to reach mainstream adoption. Its applications are not only confined to rapid prototyping, and printing tool parts and aesthetic models. The accessibility community has started to use democratized manufacturing to make society more inclusive for people with disabilities Guo et al. [2017], Baldwin et al. [2017], Banovic et al. [2013], Hofmann [2015], Hofmann et al. [2016b]. The Enable community The Future [2018] uses 3D printing to print custom prosthesis. This has a huge impact in the developing world where doctors and medical facilities are not available in abundance Hofmann et al. [2016a].

3.5.3 Program Synthesis

Program synthesis is applied to a wide range of applications such as super optimizations for low power spatial architectures Phothilimthana et al. [2016, 2014], education Alur et al. [2013] and end-user programming Wang et al. [2017]. Program synthesis can be inductive or deductive. Inductive syntax-guided program synthesis techniques Solar-Lezama [2008a] fall into the following categories: (1) enumerative search Udupa et al. [2013], (2) stochastic search Schkufza et al. [2013], (3) symbolic Jha et al. [2010]. The main components of these techniques are: a specification that is used to guide the synthesis, a search algorithm to find a candidate program that satisfies the specification, and a feedback mechanism to efficiently prune the search space. In deductive synthesis Joshi et al. [2002], the specification is a reference implementation and the synthesis algorithm finds an optimal program that is equivalent to the specification on all inputs. Unlike both traditional deductive and inductive synthesis, neither meshes nor CAD programs take inputs or produce outputs.

Program synthesis for 2D and 3D designs

Du et al. Du et al. [2018] have developed a tool, InverseCSG, that can decompile low-level polygon meshes for 3D models to CSGs. The tool uses program synthesis together with domain specific computational geometric algorithms to discover structures in the meshes. The synthesis technique used is CEGIS, i.e. Counter Example Guided Inductive Synthesis. Specifically, the authors use Sketch Solar-Lezama [2008b]. A CEGIS system starts with a specification for a program, uses a synthesiser to produce a candidate that may satisfy the specification, and then verifies that the candidate actually satisfies the specification. If it does not, then the system generates a feedback to guide the synthesiser to produce better candidates. This feedback is typically a counterexample, i.e. an input where the candidate does not satisfy the specification Bornholt [2019]. One of the crucial steps in any tool for reverse engineering CSGs from meshes is primitive detection. In InverseCSG, this is done using a modification of the RANSAC algorithm that makes the detection more robust. RANSAC gives surface primitives from which the solid are constructed using geometric heuristics. InverseCSG can handle noisy input meshes, performs various simplifications using equivalence rules, and supports meta-parametrization, i.e. extracting common parameters to share among shapes.

Tian et al. Tian et al. [2019] used machine learning to infer "3D shape programs" from 3D voxel representations in a tool called Shape2Prog. They refer to CSG based CAD programs as 3D shapes and propose a DSL for representing 3D shapes. Their language supports `for` loops to capture repetitive structure in the programs. They train their model on synthetic data that they produced. They use two LSTMs for learning the programs and 3D convolution for detecting the shapes. One interesting feature of Shape2Prog is their use of "self-supervised" learning. Shape2Prog has a program generator and a neural program executor. The program generator generates 3D shape programs from 3D shapes represented as voxels. The neural program executor takes the generated program and "executes" it to generate a new 3D shape. The difference between the new 3D shape is back-propagated to fine tune the learning process. This allows Shape2Prog to work on unlabeled data.

CSGNet Sharma et al. [2017] uses machine learning to generate flat CSG programs for 2D and 3D shapes. The input to CSGNet is an image of a 2D shape represented as pixels, or a 3D shape represented by a voxel occupancy grid. The output is a CSG program. There are three components of CSGNet. First a CNN takes an image as input and generates a feature vector. The feature vector is then converted into a sequence of modeling instructions (a visual program). Finally, a rendering engine renders the generated visual program as a CSG tree. Their CNN model is trained on a synthetic dataset the authors generated. CSGNet uses supervised learning. The input dataset is a tuple of images and their corresponding visual programs. The difference between CSGNet and Shape2Prog is that the latter supports generating loops in programs and they use different neural network architectures.

To the best of our knowledge, Reincarnate is the first tool that automatically decompiles full triangular meshes to synthesize CSG-based CAD programs. Reincarnate's core algorithm is search-based, similar to InverseCSG Du et al. [2018]. The main difference between Reincarnate and InvereCSG is the use of evaluation context to guide synthesis in the former. The use of evaluation context leads to an efficient navigation of the vast search space of CAD programs. In InverseCSG, the search is guided by a CEGIS strategy. Similar to InverseCSG, Reincarnate also concludes with a "simplification" step where a set of purely syntactic rewrites are applied to the CSG tree to get rid of redundant CAD operations. InverseCSG's primitive detection is more robust than Reincarnate's—it uses RANSAC for estimating the surface, which makes it resilient to more noisy inputs than Reincarnate.

Compared to the machine learning based approaches like Shape2Prog and CSGNet, Reincarnate provides a stronger correctness guarantee that the synthesized CAD program is semantically equivalent to the input mesh, modulo the oracles that are used for primitive detection. Shape2Prog and CSGNet cannot provide any guarantee because they rely on black box neural networks which are difficult to reason about.

3.6 Conclusions

This chapter presented a synthesis algorithm and tool, Reincarnate, that uses reverse engineering and geometric oracles to automatically infer high-level CAD programs in a language, λCAD, from low-level triangular surface meshes. It showed that Reincarnate can successfully infer λCAD programs from real surface meshes downloaded from one of the most popular online repositories for 3D models (Thingiverse), and significantly reduce program size. We are optimistic that programming-language semantics can continue to provide clarity and functionality in this space, positively affecting an emerging area of computing with potential for mass adoption.

Chapter 4

CSG TO STRUCTURED CAD WITH EQUALITY SATURATION

Around the same time that our work Reincarnate (Chapter 3) was published (Nandi et al. [2018]), the programming languages and machine learning communities developed several other techniques to decompile Computer-Aided Design (CAD) models from low-level numerical representations to Constructive Solid Geometry (CSG) expressions Du et al. [2018], Ellis et al. [2018], Tian et al. [2019], Sharma et al. [2017], Sherman et al. [2019], Friedrich et al. [2019]. These techniques aim to help users modify designs shared in online repositories Alcock et al. [2016], Hudson et al. [2016], Thingiverse [2019].

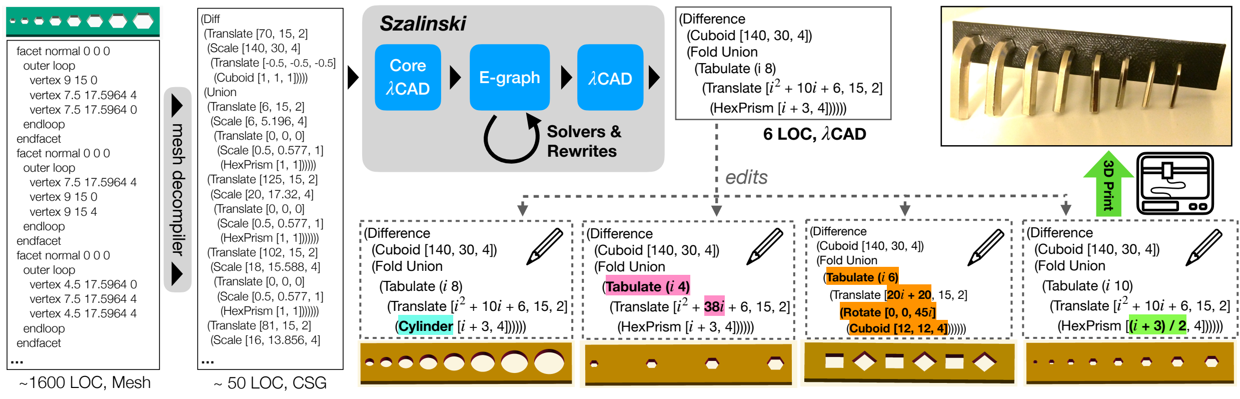
The program synthesis based tools Du et al. [2018], Nandi et al. [2018] decompile *meshes*, i.e., sets of triangles defining an object's surface, into equivalent CSG expressions. CSG includes geometric primitives like cylinders, affine transformations like translate, and set theoretic operators like union. Crucially, these existing mesh decompilers synthesize *flat* output: CSG has no loops or functions (Figure 4.1, left). Therefore, CSG synthesized from large meshes with repetitive features also tends to be large and repetitive. As in traditional programming, repetition makes otherwise intuitive edits tedious and error-prone.

Mesh decompilation is under-constrained Du et al. [2018], Nandi et al. [2018], so past tools rely on heuristics which cause them to exhibit two challenging features: **(C1)** they synthesize equivalent but dissimilar CSG expressions for the same feature repeated under different transformations, and **(C2)** they arbitrarily order CSG subexpressions. These two features, **(C1)** and **(C2)** obfuscate high-level structure latent in synthesized CSG.

This chapter proposes a second decompilation stage that composes with prior work: given a flat CSG expression, produce an equivalent, smaller, and more editable program with map and fold operators for expressing repetition. we built *Szalinski* [1] (Figure 4.1), a tool which combines semantics-preserving rewrites with simple solvers to synthesize structured CAD programs in λCAD.

Szalinski is designed to robustly handle the noisy and unstructured outputs of existing mesh decompilers. In many of these outputs, high-level structure is only apparent after a set of CAD-specific rewrites have been judiciously applied **(C1)**. Past work on Equality

[1] The protagonist in the hit movie Honey I Shrunk the Kids was named Dr.Szalinski. Our work shrinks CADs rather than kids. Thanks to Dan Grossman for this name :)



```
facet normal 0 0 0
  outer loop
    vertex 9 15 0
    vertex 7.5 17.5964 4
    vertex 7.5 17.5964 0
  endloop
endfacet
facet normal 0 0 0
  outer loop
    vertex 7.5 17.5964 4
    vertex 9 15 0
    vertex 9 15 4
  endloop
endfacet
facet normal 0 0 0
  outer loop
    vertex 4.5 17.5964 0
    vertex 7.5 17.5964 4
    vertex 4.5 17.5964 4
  endloop
endfacet
...
```

~1600 LOC, Mesh

```
(Diff
 (Translate [70, 15, 2]
  (Scale [140, 30, 4]
   (Translate [-0.5, -0.5, -0.5]
    (Cuboid [1, 1, 1]))))
 (Union
  (Translate [6, 15, 2]
   (Scale [6, 5.196, 4]
    (Translate [0, 0, 0]
     (Scale [0.5, 0.577, 1]
      (HexPrism [1, 1])))))
  (Translate [125, 15, 2]
   (Scale [20, 17.32, 4]
    (Translate [0, 0, 0]
     (Scale [0.5, 0.577, 1]
      (HexPrism [1, 1]))))))
  (Translate [102, 15, 2]
   (Scale [18, 15.588, 4]
    (Translate [0, 0, 0]
     (Scale [0.5, 0.577, 1]
      (HexPrism [1, 1]))))))
  (Translate [81, 15, 2]
   (Scale [16, 13.856, 4]
...
```

~ 50 LOC, CSG

```
(Difference
  (Cuboid [140, 30, 4])
  (Fold Union
    (Tabulate (i 8)
      (Translate [i^2 + 10i + 6, 15, 2]
        (HexPrism [i + 3, 4])))))
```

6 LOC, λCAD

```
(Difference
  (Cuboid [140, 30, 4])
  (Fold Union
   (Tabulate (i 8)
    (Translate [i^2 + 10i + 6, 15, 2]
     (Cylinder [i + 3, 4])))))
```

```
(Difference
  (Cuboid [140, 30, 4])
  (Fold Union
   (Tabulate (i 4)
    (Translate [i^2 + 38i + 6, 15, 2]
     (HexPrism [i + 3, 4])))))
```

```
(Difference
  (Cuboid [140, 30, 4])
  (Fold Union
   (Tabulate (i 6)
    (Translate [20i + 20, 15, 2]
     (Rotate [0, 0, 45i]
      (Cuboid [12, 12, 4]))))))
```

```
(Difference
  (Cuboid [140, 30, 4])
  (Fold Union
   (Tabulate (i 10)
    (Translate [i^2 + 10i + 6, 15, 2]
     (HexPrism [(i + 3) / 2, 4])))))
```

Figure 4.1: Existing mesh decompilers turn triangle meshes into CSG expressions. Szalinski robustly synthesizes smaller, structured λCAD programs from CSG expressions. This can ease customization by simplifying edits: small, mostly local changes yield usefully different models. The photo shows the 3D printed hex wrench holder after customizing hole sizes.

Saturation Tate et al. [2009] suggests that Equality Graphs (E-graphs) Nelson [1980]—an efficient data structure underlying SMT solvers De Moura and Bjørner [2008], Detlefs et al. [2005] and program optimizers Joshi et al. [2002], Tate et al. [2009], Stepp et al. [2011], Wu et al. [2019a]—would make a good fit for Szalinski because E-graphs can compactly encode many of the equivalent ways to express a program with respect to a set of rewrites. Unfortunately, reordering with associative and commutative rewrites can cause E-graphs to blow up exponentially. This is known as the *AC-matching problem* Belkhir and Giorgetti [2012], Kirchner and Moreau [2001], Clavel et al. [2007a]. It presents a significant challenge for Szalinski because existing mesh decompilers typically output CSG features ordered by heuristics (e.g., geometric proximity) rather than high-level structure **(C2)**. To address the AC-matching problem Szalinski uses *inverse transformations*, a novel way for solvers to *speculatively* unify expressions in an E-graph which would be equivalent modulo reordering or partitioning. Before unifying a result R with its input I, a solver can annotate R with an inverse transformation which encodes how it manipulated I to find the more-profitable R. Szalinski then uses syntactic rewrites to *propagate* and *eliminate* inverse transformations when opportunities to use such results arise.

This chapter presents a library of 65 CAD rewrites and a prototype implementation of Szalinski in 3,000 lines of Rust (Section 4.4.4). Section 4.5 shows how composing Szalinski with Reincarnate (Chapter 3) qualitatively improves editability (sketched in Figure 4.1) and describes an evaluation of Szalinski's performance and correctness on real-world CAD models downloaded from Thingiverse. Section 4.6 briefly surveys the most relevant related work and

Section 4.7 concludes.

4.1 Second Stage Decompilation

The λCAD language (Figure 4.2) provides map- and fold-like functional list operators to express repetitive structure in CAD models, as well as a *Core* λCAD fragment that corresponds directly to CSG (similar to Figure 3.4). The λCAD semantics fully unroll a program's functional list operators to produce a Core λCAD (CSG) expression. Szalinski "goes the other way," decompiling a Core λCAD expression to a λCAD program that aims to expose latent repetitive structure. This section introduces a running example that subsequent sections extend to illustrate challenges that arise when shrinking noisy, unstructured outputs from existing mesh decompilers.

4.1.1 Core λCAD, λCAD, Equivalence

Core λCAD includes various primitives parametrized by dimensions— cuboids parametrized by side length, spheres by radius, cylinders and hexagonal prisms by height and radius, etc. λCAD also provides binary[2] set theoretic operators Union, Difference, and Intersection, and affine[3] transformations like Translate, Rotate, and Scale that are parameterized by 3D vectors. For example, (Translate [1,0,0] (Sphere 2)) shifts a sphere with radius 2 a single unit of distance along the x-axis. TranslateSpherical (not present in Core λCAD or CSG) captures a common pattern in models relying on translations in spherical rather than Cartesian coordinates.

Figure 4.3 gives semantics for the functional list operators λCAD provides on top of Core λCAD. Tabulate takes pairs of variables and positive integers $(x_1\, b_1) \ldots (x_n\, b_n)$ as well as a λCAD expression e, and returns the list of length Πb_i generated by n nested loops evaluating e over the variables $x_1 \ldots x_n$ up to the bounds $b_1 \ldots b_n$:

$$(\textsf{List}\ \ e[0/x_1]...[0/x_n]\ \ \ldots\ \ e[b_1 - 1/x_1]...[b_n - 1/x_n])$$

where $e[i/x]$ denotes substituting all free occurrences (not bound by nested Tabulates) of x in e with i. For example,

```
(Tabulate (i 2) (j 3) (Cuboid [2 × i + 2, 7, j + 1])) ⇒
    (List (Cuboid [2, 7, 1]) (Cuboid [2, 7, 2]) (Cuboid [2, 7, 3])
          (Cuboid [4, 7, 1]) (Cuboid [4, 7, 2]) (Cuboid [4, 7, 3]))
```

[2] Note the use of syntactic sugar to present binary nested operators as left-associative over multiple arguments, e.g., (Union a b c) means (Union (Union a b) c).

[3] Here affine means that parallel lines remain parallel after transformation.

$$\mathit{op} ::= + \mid \text{-} \mid \times \mid / \qquad \mathsf{num} ::= \mathbb{R} \mid \mathit{var} \mid \mathit{num}\ \mathit{op}\ \mathit{num}$$

$$\mathit{vec2} ::= [\mathit{num},\ \mathit{num}] \qquad \mathsf{vec3} ::= [\mathit{num},\ \mathit{num},\ \mathit{num}]$$

$$\mathit{affine} ::= \mathsf{Translate} \mid \mathsf{Rotate} \mid \mathsf{Scale} \mid \textbf{TranslateSpherical}$$

$$\mathit{binop} ::= \mathsf{Union} \mid \mathsf{Difference} \mid \mathsf{Intersection}$$

$$\begin{array}{rl} \mathit{cad} ::= & (\mathsf{Cuboid}\ \mathit{vec3}) \mid (\mathsf{Sphere}\ \mathit{num}) \\ \mid & (\mathsf{Cylinder}\ \mathit{vec2}) \mid (\mathsf{HexPrism}\ \mathit{vec2}) \mid \ldots \\ \mid & (\mathit{affine}\ \mathit{vec3}\ \mathit{cad}) \\ \mid & (\mathit{binop}\ \mathit{cad}\ \mathit{cad}) \\ \mid & (\textbf{Fold}\ \mathit{binop}\ \mathit{cad\text{-}list}) \end{array}$$

$$\begin{array}{rl} \mathit{cad\text{-}list} ::= & (\mathsf{List}\ \mathit{cad}+) \mid (\mathsf{Concat}\ \mathit{cad\text{-}list}+) \mid (\mathsf{Tabulate}\ (\mathit{var}\ \ \mathbb{Z}^+)+\ \mathit{cad}) \\ \mid & (\mathsf{Map2}\ \mathit{affine}\ \mathit{vec3\text{-}list}\ \mathit{cad\text{-}list}) \end{array}$$

$$\mathit{vec3\text{-}list} ::= (\mathsf{List}\ \mathit{vec3}+) \mid (\mathsf{Concat}\ \mathit{vec3\text{-}list}+) \mid (\mathsf{Tabulate}\ (\mathit{var}\ \ \mathbb{Z}^+)+\ \mathit{vec3})$$

Figure 4.2: λCAD syntax. The Core λCAD (CSG) subset omits variables, list forms (those using **Fold**), and **TranslateSpherical**.

For the frequent special case of (Tabulate (x n) e) when x is not free in e, we write (Repeat n e) as syntactic sugar.

Map2 produces a list of Core λCAD expressions by applying an affine operator to a list of transformation parameters and a list of CAD arguments. For example,

```
(Map2 Scale (List  [2,2,2]  [3,3,3])  (Repeat 2 (Sphere 1)))  ⇒
    ( List  (Scale  [2,2,2]  (Sphere 1)) (Scale  [3,3,3]  (Sphere 1)))
```

λCAD programs are equivalent iff they evaluate to equivalent Core λCAD programs. By design, Core λCAD directly corresponds to CSG, whose semantics is given in Chapter 2. Section 4.5 describes practically testing λCAD equivalence by evaluating programs to Core λCAD, compiling them to meshes, and comparing Hausdorff distances.[4]

4.1.2 A Running Example for Shrinking λCAD

Figure 4.4a shows a simple CAD model of a ship's wheel and Figure 4.4b shows the corresponding desired λCAD output from Szalinski. Figure 4.4b reifies repetitive structure: making a change to all the spokes only requires a single edit instead of six coordinated modifications in different locations.

[4]Informally, the Hausdorff distance between two meshes is small if every point on each mesh is near some point on the other.

$$\frac{e \Rightarrow (\textsf{List}\ v_1\ ...\ v_n) \quad f_1 = v_1 \quad f_i = (binop\, f_{i-1}\, v_i)}{(\textsf{Fold}\ binop\ e) \Rightarrow f_n}$$

$$\frac{e \Rightarrow\ (\textsf{List}\ (\textsf{List}\ v_{1,1}\ v_{1,2}\ ...)\ (\textsf{List}\ v_{2,1}\ v_{2,2}\ ...)\ ...)}{(\textsf{Concat}\ e) \Rightarrow (\textsf{List}\ v_{1,1}\ v_{1,2}\ ...\ v_{2,1}\ v_{2,2}\ ...)}$$

$$\frac{e[i_1/x_1]...[i_n/x_n] \Rightarrow v_{(i_1,\ldots,i_n)}}{(\textsf{Tabulate}\ (x_1\, b_1)\ ...\ (x_n\, b_n)\ e) \Rightarrow (\textsf{List}\ v_{(0,\ldots,0)}\ \ ...\ \ v_{(b_1-1,\ldots,b_n-1)}\ )}$$

$$\frac{ps \Rightarrow (\textsf{List}\ [a_1,b_1,c_1]\ [a_2,b_2,c_2]\ ...) \qquad es \Rightarrow (\textsf{List}\ v_1\ v_2\ ...)}{(\textsf{Map2}\ affine\ ps\ es) \Rightarrow (\textsf{List}\ \ (\textsf{\textit{affine}}[a_1,b_1,c_1]\, v_1)\ \ (\textsf{\textit{affine}}[a_2,b_2,c_2]\, v_2)\ ...)}$$

$$\frac{e \Rightarrow v \qquad \textsf{to\_cartesian}(r,\phi,\theta) = (x,y,z)}{(\textsf{TranslateSpherical}\ [r,\phi,\theta]\ e) \Rightarrow (\textsf{Translate}\ [x,y,z]\ v)}$$

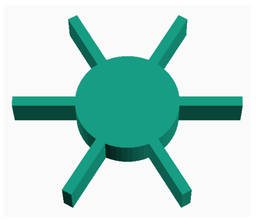
Figure 4.3: Big step semantics reducing well-formed λCAD programs to Core λCAD expressions. $e[i/x]$ denotes substituting all free occurrences of x in e with i. Additional rules (not shown) also evaluate under List, affines, and binops.

When repetitive structure is easily exposed, as in the ideal Core λCAD of Figure 4.4c, solvers can infer the arithmetic function relating instances of repeated design components. Section 4.2 describes Szalinski's rewrite-driven approach to infer such functions and shrink programs by rerolling loops.

In practice, given a mesh representing Figure 4.4a, mesh decompilers can generate CSG expressions equivalent to Figure 4.4c, but which obfuscate repetitive structure. Affine transformations may be different or missing and, from a solver's perspective, lists may be inconveniently ordered or partitioned. Comparing Figure 4.4c to 4.4d, Rotate [0,0,180] has been replaced with an equivalent Scale [-1,-1,1], identity transformations have been omitted, the Union has been reordered, and Scales and Translates have been inconsistently swapped. Sections 4.3 and 4.4 walk through progressively more challenging variants of Core λCAD inputs for the ship's wheel to illustrate how Szalinski uses E-graphs and inverse transformations to robustly handle such variation.

4.2 Shrinking by Rerolling Loops

Szalinski shrinks repetitive λCAD programs by "rerolling loops". First, rewrites *find structure* by separating affine operators from their parameters and CAD arguments under Map2s.



(a) CAD model of ship's wheel

```
(Union
 (Cylinder [1, 5, 5])
 (Fold Union
  (Tabulate (i 6)
   (Rotate [0,0,60i]
    (Translate [1,−0.5,0]
     (Cuboid [10,1,1]))))))
```

(b) λCAD program

```
(Union
 (Cylinder [1,5])
 (Union
  (Rotate [0,0,0] (Translate [1,−0.5,0] (Cuboid [10,1,1])))
  (Rotate [0,0,60] (Translate [1,−0.5,0] (Cuboid [10,1,1])))
  (Rotate [0,0,120] (Translate [1,−0.5,0] (Cuboid [10,1,1])))
  (Rotate [0,0,180] (Translate [1,−0.5,0] (Cuboid [10,1,1])))
  (Rotate [0,0,240] (Translate [1,−0.5,0] (Cuboid [10,1,1])))
  (Rotate [0,0,300] (Translate [1,−0.5,0] (Cuboid [10,1,1])))))
```

(c) Ideal Core λCAD expression

```
(Union
  (Rotate [0,0,120] (Translate [1,−0.5,0] (Cuboid [10,1,1])))
  (Scale  [10,1,1] (Translate [0.1,−0.5,1] (Cuboid [1,1,1])))
  (Rotate [0,0,300] (Translate [1,−0.5,0] (Cuboid [10,1,1])))
  (Scale  [5,5,1] (Cylinder [1,1]))
  (Translate [−1,0.5,0] (Scale [−1,−1,1] Cuboid [10,1,1]))
  (Rotate [0,0,240] (Translate [1,−0.5,0] (Cuboid [10,1,1])))
  (Rotate [0,0,60] (Translate [1,−0.5,0] (Cuboid [10,1,1]))))
```

(d) Equivalent, structure obfuscating Core λCAD

Figure 4.4: (a) CAD model for a ship's wheel. (b) λCAD features like Tabulate express repeated design components. Such repetition can be obvious and exposed in an ideal Core λCAD program (c), but existing mesh decompilers obfuscate structure (d).

Binop Fold

$(binop\ c_1\ c_2\ ...) \rightsquigarrow (\mathsf{Fold}\ binop\ (\mathsf{List}\ c_1\ c_2\ ...))$

Structure Finding

$(\mathsf{List}\ (aff\ p_1\ c_1)\ (aff\ p_2\ c_2)\ ...) \rightsquigarrow (\mathsf{Map2}\ aff\ (\mathsf{List}\ p_1\ p_2\ ...)\ (\mathsf{List}\ c_1\ c_2\ ...))$

Repeat

$(\mathsf{List}\ a\ a\ a\ ...\ n \text{ times}) \rightsquigarrow (\mathsf{Repeat}\ n\ a)$

List Solve (single loop)

$(\mathsf{List}\ [f_x(0), f_y(0), f_z(0)]\ \ ...\ \ [f_x(n-1), f_y(n-1), f_z(n-1)])$
$\rightsquigarrow (\mathsf{Tabulate}\ (i\ n)\ [f_x(i), f_y(i), f_z(i)])$

Repeat over Map2

$(\mathsf{Map2}\ aff\ (\mathsf{Repeat}\ n\ p)\ (\mathsf{Repeat}\ n\ c) \rightsquigarrow (\mathsf{Repeat}\ n\ (aff\ p\ c))$

Tabulate over Map2 where $b = \Pi b_i$

$(\mathsf{Map2}\ aff\ \ (\mathsf{Tabulate}\ (x_1\ b_1)\ ...\ p)\ \ (\mathsf{Tabulate}\ (x_1\ b_1)\ ...\ c))$
$\rightsquigarrow (\mathsf{Tabulate}\ (x_1\ b_1)\ ...\ (aff\ p\ c))$

$(\mathsf{Map2}\ aff\ (\mathsf{Tabulate}\ (x_1\ b_1)\ ...\ p)\ (\mathsf{Repeat}\ b\ c))$
$\rightsquigarrow (\mathsf{Tabulate}\ (x_1\ b_1)\ ...\ (aff\ p\ c))$

$(\mathsf{Map2}\ aff\ (\mathsf{Repeat}\ b\ p)\ (\mathsf{Tabulate}\ (x_1\ b_1)\ ...\ c))$
$\rightsquigarrow (\mathsf{Tabulate}\ (x_1\ b_1)\ ...\ (aff\ p\ c))$

Figure 4.5: Rewrite rules for loop rerolling

This can expose program repetition as repetitive Lists. Next, arithmetic solvers find equivalent closed form Tabulates for repetitive lists. These Tabulates generalize the program and provide parameters that simplify future edits. Finally, rewrites *restore structure* by recombining the (generalized) affine parameters and CAD arguments from Map2s into a single Tabulate. Figure 4.5 shows this strategy's key rewrites.

Because Szalinski uses an E-graph, these rewrites can actually be repeatedly applied in any order and still efficiently yield the same final result. For simplicity, this section steps through the ship's wheel example assuming a particular fortuitous order of rewrites that just so happens to nicely shrink the ideal Core λCAD input from Figure 4.4c.

4.2.1 Finding Structure: A Bird's-Eye View

Applying **Binop Fold** to the inner Union in Figure 4.4c produces:

```
(Union (Union (Union ...
 (Rotate [0,0,0] cad_1)
 (Rotate [0,0,60] cad_2)
 (Rotate [0,0,120] cad_3) ...)
```

►

```
(Fold Union (List
 (Rotate [0,0,0] cad_1)
 (Rotate [0,0,60] cad_2)
 (Rotate [0,0,120] cad_3) ...))
```

A structure finder (detailed in Section 4.3) searches for a list of affine transformations all using the same operator *aff*. **Structure Finding** separates the affine parameters and CAD arguments out into two Lists under a Map2 with *aff*:

```
(Fold Union (List
 (Rotate [0,0,0] cad_1)
 (Rotate [0,0,60] cad_2)
 (Rotate [0,0,120] cad_3) ...))
```

►

```
(Fold Union
 (Map2 Rotate
  (List [0,0,0] [0,0,60] [0,0,120] ...)
  (List cad_1 cad_2 cad_3 ...)))
```

The structure finder is applied repeatedly. Here it exposes lists of identical elements, letting the **Repeat** rewrite produce:

```
(Fold Union
 (Map2 Rotate
  (List [0,0,0] [0,0,60] [0,0,120] ...)
  (Map2 Translate
   (List [1,−0.5,0] ...)
   (List (Cube [10,1,1]) ...))))
```

►

```
(Fold Union
 (Map2 Rotate
  (List [0,0,0] [0,0,60] [0,0,120] ...)
  (Map2 Translate
   (Repeat 6 [1,−0.5,0])
   (Repeat 6 (Cube [10,1,1])))))
```

4.2.2 Introducing Tabulate by Solving Lists

Once structure finding has isolated a List of vectors ℓ, arithmetic solvers attempt to find equivalent Tabulates. The current Szalinski prototype provides simple solvers for first- and second-degree polynomials in both Cartesian and spherical coordinates.

Given ℓ = (List $[x_1, y_1, z_1]$ ... $[x_n, y_n, z_n]$), these solvers infer independent functions f_x, f_y, f_z for the x, y, z components of ℓ respectively. In practice, running arithmetic solvers on floating point numbers output by existing mesh decompilers requires accepting Tabulates within some ϵ of ℓ, especially for tools that rely on randomized algorithms Du et al. [2018] like RANSAC Schnabel et al. [2007].

For the Rotate parameters (List $[0,0,0]$ $[0,0,60]$ ... $[0,0,300]$), solvers find (Tabulate (i 6) $[0,0,60i]$). **List Solve** then produces:

```
(Fold Union                                        (Fold Union
 (Map2 Rotate                                       (Map2 Rotate
  ( List [0,0,0] [0,0,60] [0,0,120] ...)      ►      (Tabulate (i 6)  [0,0,60i])
  (Map2 Translate                                    (Map2 Translate
   (Repeat 6 [1,−0.5,0])                              (Repeat 6 [1,−0.5,0])
   (Repeat 6 (Cube [10,1,1])))))                      (Repeat 6 (Cube [10,1,1])))))
```

In this example, the solvers relied on their input arriving in just the right order. Section 4.4 shows how inverse transformations allow solvers to *reorder* their input to infer better Tabulates while preserving equivalence.

4.2.3 The Final Squeeze: Recombining Map2s

Finally, since both the Repeats and Tabulate have matching bounds, **Repeat over Map2** and **Tabulate over Map2** recombine the separated affine parameters and CAD arguments to produce the desired output from the inner Union of Figure 4.4c:

```
(Fold Union                                   (Fold Union
 (Map2 Rotate                                  (Tabulate (i 6)
  (Tabulate (i 6) [0,0,60i])             ►      (Rotate [0,0,60i]
   (Map2 Translate                               ( Translate [1,−0.5,0]
    (Repeat 6 [1,−0.5,0])                          (Cube [10,1,1])))))
    (Repeat 6 (Cube [10,1,1])))))
```

This section illustrated Szalinski's core strategy: shrinking λCAD by rerolling loops. However, the example relied on a specific rewrite order and Figure 4.4c as an unrealistically ideal input. Subsequent sections show how E-graphs and inverse transformations enable Szalinski to robustly shrink noisy and unstructured CSGs.

4.3 E-graphs and CAD Equality Saturation

Rewrites to shrink λCAD by rerolling loops must be applied in just the right order to programs that already make structure apparent as in Figure 4.4c. Simply interleaving additional CAD rewrites to expose repetitive structure initially seems infeasible because the necessary rewrites are not confluent and the space of possible orderings explodes exponentially. However, past work on Equality Saturation Tate et al. [2009] demonstrates how E-graphs Nelson

[1980] can make this strategy efficient for many rewrite rules. This section shows how Szalinski applies Equality Saturation in the CAD domain to robustly handle CSG variations when shrinking λCAD programs. Chapter 1 (Section 1.1.2) provides an introduction to E-graphs and equality saturation.

4.3.1 Rewrite Phase Ordering: What, When, Where

A slightly perturbed λCAD example for the spokes of the ship's wheel omits Rotate $[0, 0, 0]$ and replaces Rotate $[0, 0, 180]$ by the equivalent Scale $[-1, -1, 1]$:

```
(Fold Union  (List
  (Translate  [1, -0.5, 0] (Cube [10, 1, 1]))
  (Rotate [0, 0, 60] (Translate [1, -0.5, 0] (Cube [10, 1, 1])))
  (Rotate [0, 0, 120] (Translate [1, -0.5, 0] (Cube [10, 1, 1])))
  (Scale   [-1, -1, 1] (Translate [1, -0.5, 0] (Cube [10, 1, 1])))
  (Rotate [0, 0, 240] (Translate [1, -0.5, 0] (Cube [10, 1, 1])))
  (Rotate [0, 0, 300] (Translate [1, -0.5, 0] (Cube [10, 1, 1])))))
```

The three-phase loop rerolling strategy from Section 4.2 now breaks: Szalinski must interleave its search with additional CAD rewrites (Figure 4.6) to expose the repeated affine transformations as in Figure 4.4c. This *phase ordering problem* Touati and Barthou [2006], Tate et al. [2009] makes it difficult to determine when to apply which rewrites and where. Poor choices will only further obfuscate repetitive structure and no single strategy is best in general.

Equality Saturation Tate et al. [2009] is a technique to mitigate phase ordering that uses E-graphs to compactly represent equivalence relations over large sets of expressions. Instead of destructively modifying a particular concrete term, rewrites extend the E-graph by adding and unifying classes of expressions. This eliminates the need to choose any particular rewrite ordering. By repeatedly applying the rules in Figures 4.5 and 4.6 to an E-graph and using a structure finding heuristic (Section 4.3.4), Szalinski's loop rerolling strategy can robustly handle variations in how mesh decompilers synthesize affine operators.

4.3.2 E-graphs in Szalinski

Figure 4.7 shows how an E-graph can compactly represent equivalent expressions generated by rewrites, in this case, one of the CAD rewrites needed to expose repetitive structure for the ship's wheel example.

We slightly generalize rewrites from two patterns to a pattern L and a *function* R that, given a substitution ϕ, returns an expression to be added to the E-graph and unified with the eclass that matched L. This generalization allows rewrites to implement rules which are

Affine Identities

(Rotate $[0, 0, 180]$ cad)	$\leftrightsquigarrow$	(Scale $[-1, -1, 1]$ cad)
(Rotate $[0, 0, 0]$ cad)	$\leftrightsquigarrow$	cad
(Translate $[0, 0, 0]$ cad)	$\leftrightsquigarrow$	cad
(Scale $[1, 1, 1]$ cad)	$\leftrightsquigarrow$	cad

Affine Interchanging

(Scale $[a, b, c]$ (Translate $[d, e, f]$ cad))	$\leftrightsquigarrow$	(Translate $[ad, be, cf]$ (Scale $[a, b, c]$ cad))

Affine Combination

(Scale $[a, b, c]$ (Scale $[d, e, f]$ cad))	$\rightsquigarrow$	(Scale $[ad, be, cf]$ cad)
(Translate $[a, b, c]$ (Translate $[d, e, f]$ cad))	$\rightsquigarrow$	(Translate $[a + d, b + e, c + f]$ cad)

Primitive-Affine Conversion

(Cuboid $[x, y, z]$)	$\leftrightsquigarrow$	(Scale $[x, y, z]$ (Cuboid $[1, 1, 1]$))
(Sphere r)	$\leftrightsquigarrow$	(Scale $[r, r, r]$ (Sphere 1))
(Cylinder $[h, r]$)	$\leftrightsquigarrow$	(Scale $[r, r, h]$ (Cylinder $[1, 1]$))
(Hexprism $[h, r]$)	$\leftrightsquigarrow$	(Scale $[r, r, h]$ (Hexprism $[1, 1]$))

Figure 4.6: Selected CAD identities. Bidirectional arrows indicates Szalinski has a rule for each direction.

```
(Scale   [−1, −1, 1]                        (Rotate [0, 0, 180]
 (Translate  [1, −0.5, 0]          ▶         (Translate  [1, −0.5, 0]
  (Cube [10, 1, 1])))                         (Cube [10, 1, 1])))
```

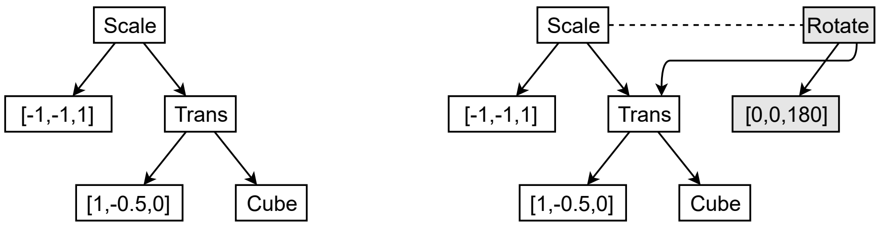


Figure 4.7: E-graph before and after a CAD rewrite. Boxes represent enodes and dashed edges indicate equivalence (membership in the same eclass). Directed solid edges connect enodes to their child eclasses. Both the original and transformed programs are represented in the resulting E-graph.

not purely syntactic, like constant folding (ex: rewriting $2 + 3$ to 5). Many of Szalinski's list-manipulating rewrites are implemented this way, which is convenient for rules like **Repeat** which need to extract the length of a matched list pattern. This generalization also allows Szalinski to integrate arithmetic solvers with the E-graph—Tabulate expressions returned by solvers are unified with the eclass that matched the **List Solve** rule's list pattern.

4.3.3 Equality Saturation in Szalinski

Szalinski implements Equality Saturation Tate et al. [2009] for λCAD (Figure 4.8). First, an E-graph is created from the input Core λCAD expression. Then Szalinski expands the E-graph by repeatedly applying rewrites. Searching the E-graph for a rewrite's left-hand side pattern results in a list of (eclass, substitution) pairs that indicate where and how a pattern was matched. For each pair (c, ϕ), Szalinski generates an expression e by applying the rewrite's right-hand side function to ϕ, adding e to the E-graph yielding eclass c', and unifying c and c'. Szalinski continues applying rewrites until the E-graph *saturates* (reaches a fixpoint where no rewrites further expand the E-graph), or a timeout is reached. In the case of saturation, Szalinski has discovered *all* equivalences derivable from its rewrites.

Finally, Szalinski extracts the smallest λCAD program represented by the initial Core λCAD input's eclass in a simple bottom-up traversal of the E-graph. Szalinski uses program size as a proxy for editability. Past work provides extraction strategies for various kinds of cost functions Tate et al. [2009], Panchekha et al. [2015], but we leave further exploration of CAD cost functions in Szalinski to future work. Chapter 5 discusses the use of genetic

```
def Szalinski(csg : core_lambdaCAD):
    egraph, root = make_egraph(csg)
    while egraph.changed()
        for (lhs, rhs) in SZALINSKI_REWRITES:
            matches = egraph.search(lhs)
            for (eclass, subst) in matches:
                c = egraph.add(apply(rhs, subst))
                egraph.unify(eclass, c)
    return egraph.extract(root, min_size)
```

Figure 4.8: Equality Saturation for λCAD in Szalinski.

algorithms for extraction in the context of carpentry.

4.3.4 Structure Finding in E-graphs

Since Szalinski's rewrites contain CAD identities that can fire in every iteration, the structure finding procedure as presented in Section 4.2.1 must be enhanced. It must consider that multiple affine transformations may be introduced in the same eclass by the CAD identities. Given a list of eclasses $e_1, e_2, ..., e_n$, the structure finder aims to extract Map2s that remove one level of structure. However, due to rules like Affine Combination from Figure 4.6, each eclass may contain multiple equivalent enodes with the same affine operation. If eclass e_i has 2 enodes with the Rotate operator, for example, the structure finder can choose from 2 different Rotates at each of the n eclasses in the list. Each of these 2^n Map2s has distinct children, and will therefore be a distinct enode in the E-graph, all unified in the same eclass as the list itself. Szalinski must operate on large lists of Core λCAD programs, but such an exponential number of enodes would blow up the E-graph.

Szalinski instead capitalizes on the observation that it is not useful to pick different affine enodes within similar-looking eclasses. Consider again the ship's wheel example presented in Section 4.3.1. After applying the two Rotate identities from Figure 4.6, the eclasses for the top-level affines in the list contain the following enodes (one eclass per row, enodes shown with their parameters for clarity):

a:	(Translate [1,-0.5,0] x_1)	(Rotate [0,0,0] a)	
b:	(Rotate [0,0,60] x_2)	(Rotate [0,0,0] b)	
c:	(Rotate [0,0,120] x_3)	(Rotate [0,0,0] c)	
d:	(Scale [-1,-1,1] x_4)	(Rotate [0,0,0] d)	(Rotate [0,0,180] x_4)
e:	(Rotate [0,0,240] x_5)	(Rotate [0,0,0] e)	
f:	(Rotate [0,0,300] x_6)	(Rotate [0,0,0] f)	

The structure finder calculates the *affine signature* of each eclass as the multiset of the kinds of affine operators in the eclass. In the above example, eclass a's affine signature is $\{$Translate, Rotate$\}$, d's is $\{$Scale, Rotate, Rotate$\}$, and the others all share the same signature: $\{$Rotate, Rotate$\}$. A *group* is a set of eclasses that share the same affine signature. When trying to extract a Rotate, the structure finder will *not* take the Cartesian product of the Rotates in each eclass—doing so would lead to 2^5 possible ways to combine Rotate. Instead, it takes the Cartesian product of affine choices for each group, and extends the *same* choice of affine over all eclasses within the group (using the order of affines in the eclasses). In this example, the only affine that can be extracted is Rotate, since the other affines do not appear in the affine signature of all groups. For the Rotate affine, group a has one choice, group d has 2 choices, and group b, c, e, f also has 2 choices. This reduces the number of (Map2 Rotate ...) expressions introduced from $2^5 = 32$ to 4.

4.4 Inverse Transformations

E-graphs and CAD rewrites allow Szalinski to expose repetitive structure and reroll loops even when a Core λCAD input exhibits obfuscating variations (e.g., Scale [-1,-1,1] instead of Rotate [0,0,180]). However, existing mesh decompilers tend to also order and group CAD subexpressions by geometric proximity or other heuristics that, from Szalinski's perspective, make recovering high-level structure challenging. Unless the right reordering and regrouping of subexpressions can be found, list solvers will fail to infer Tabulates and Szalinski will be unable to reroll loops and shrink λCAD programs.

To address this challenge, we introduce *inverse transformations*, a novel way for solvers to optimistically unify expressions in an E-graph that would be equivalent modulo reordering or regrouping.

Figure 4.9 shows how far CAD rewrites combined with techniques from previous sections get for the Figure 4.4d example. Unfortunately, Cylinder is still Unioned with Cuboids, preventing the structure finder from pulling out the Rotate. Even if the Cylinder were removed, the list order would prevent solvers from inferring a Tabulate for the Rotate parameters.

Unlike the previous section, adding more rewrites does not help.[5] E-graphs do *not* compactly represent equivalences due to reordering associative and commutative operators like Union. This is known as the AC-matching problem Belkhir and Giorgetti [2012] (A stands for associativity, and C for commutativity) and it prevents efficiently exploring all possible reorderings and regroupings.

Szalinski addresses this with a new technique, *inverse transformations*, that allows solvers to speculatively transform their inputs to allow for more profitable rewriting. A solver that

[5] AC-matching is a problem both in theory and practice.

```
(Fold Union (List
  (Rotate [0,0,120] (Translate [1,−0.5,0] (Cuboid [10,1,1])))
  (Rotate [0,0,0]   (Translate [1,−0.5,0] (Cuboid [10,1,1])))
  (Rotate [0,0,300] (Translate [1,−0.5,0] (Cuboid [10,1,1])))
  (Cylinder [1,5])
  (Rotate [0,0,180] (Translate [1,−0.5,0] (Cuboid [10,1,1])))
  (Rotate [0,0,240] (Translate [1,−0.5,0] (Cuboid [10,1,1])))
  (Rotate [0,0,60]  (Translate [1,−0.5,0] (Cuboid [10,1,1])))))
```

Figure 4.9: Section 4.2 and 4.3 techniques find the "Rotate then Translate" structure from the realistic Figure 4.4d. Without inverse transformations, loop rerolling now gets stuck.

cannot simplify input A may, for some transformation F, be able simplify $F(A)$ to B. Inverse transformations simply allows the solver to "wrap" B with F^{-1} before unifying it with A, even though A and B are not equivalent.

Inverse transformations enable locally-reasoning solvers to register potentially profitable regroupings and reorderings in an E-graph. Simple syntactic rewrites then propagate these "hints" globally through the E-graph, allowing other solvers to try them, and contextually eliminate inverse transformations when possible (e.g., under order-insensitive operations like Fold Union).

4.4.1 Extended λCAD

Extended λCAD (Figure 4.10 and 4.11) adds inverse transformations that allow solvers to record how they manipulated their input. These extended forms are only introduced in the E-graph; Szalinski's cost function ensures extraction produces regular λCAD programs. Semantically, these constructs either undo the transformation performed by the solver to recover the input, or perform the transformation on some other part of the program. Sort and Unsort take a permutation p and a list ℓ, imposing (respectively, undoing) p on ℓ. Part takes a partitioning P (a list of lengths) and a list ℓ, breaking down ℓ into a list of sublists according to P. Unpart takes a partitioning and a list of lists and flattens the latter; the partitioning is only used to propagate information. TranslateSpherical and Unspherical take a 3D vector c and a list of 3D vectors in spherical coordinates about c, returning a list of the vectors in Cartesian coordinates (and vice versa).

$permutation$ ::= $\langle$ n, n, ... $\rangle$ $partitioning$ ::= $\langle$ n, n, ... $\rangle$

```
inv ::= (Sort permutation *-list)
      | (Unsort permutation *-list)
      | (Part partitioning *-list)
      | (Unpart partitioning *-list)
      | (Spherical vec3 vec3-list)
      | (Unspherical vec3 vec3-list)
```

Figure 4.10: Syntax of Extended λCAD.

$$\frac{e \Rightarrow (\text{List } v_1\ v_2\ \ldots\ v_n)}{(\text{Sort } \langle i_1, i_2, \ldots, i_n\rangle \text{ e}) \Rightarrow (\text{List } v_{i_1}\ v_{i_2}\ \ldots\ v_{i_n})}$$

$$\frac{e \Rightarrow (\text{List } v_{i_1}\ v_{i_2}\ \ldots\ v_{i_n})}{(\text{Unsort } \langle i_1, i_2, \ldots, i_n\rangle \text{ e}) \Rightarrow (\text{List } v_1\ v_2\ \ldots\ v_n)}$$

$$\frac{\begin{array}{ll} sum_0 = 0 & sublist_i = (\text{List } v_{sum_{i-1}}\ \ldots\ v_{sum_i}) \\ sum_i = sum_{i-1} + l_i & e \Rightarrow (\text{List } v_1\ v_2\ \ldots\ v_{sum_n}) \end{array}}{(\text{Part } \langle l_1, l_2, \ldots, l_n\rangle \text{ e}) \Rightarrow (\text{List } sublist_1\ \ldots\ sublist_n)}$$

$$\frac{\begin{array}{ll} sum_0 = 0 & sublist_i = (\text{List } v_{sum_{i-1}}\ \ldots\ v_{sum_i}) \\ sum_i = sum_{i-1} + l_i & e \Rightarrow (\text{List } sublist_1\ \ldots\ sublist_n) \end{array}}{(\text{Unpart } \langle l_1, l_2, \ldots, l_n\rangle \text{ e}) \Rightarrow (\text{List } v_1\ v_2\ \ldots\ v_{sum_n})}$$

$$\frac{e \Rightarrow (\text{List } v'_1\ v'_2\ \ldots\ v'_n) \qquad v_i = \text{to\_spherical } (center,\ v'_i)}{(\text{Spherical n } center \text{ e}) \Rightarrow (\text{List } v_1\ v_2\ \ldots\ v_n)}$$

$$\frac{e \Rightarrow (\text{List } v'_1\ v'_2\ \ldots\ v'_n) \qquad v_i = \text{to\_cartesian } (center,\ v'_i)}{(\text{Unspherical n } center \text{ e}) \Rightarrow (\text{List } v_1\ v_2\ \ldots\ v_n)}$$

Figure 4.11: Big step semantics for Extended λCAD.

4.4.2 Restructuring with Unpart *and* Unsort

Using inverse transformations, Szalinski can finally get the desired output given the realistic input for the ship's wheel (Figure 4.4d). Starting from Figure 4.9, Szalinski separates the Cylinder from the Cuboids with partitioning and sorts the list of Cuboids on their Rotate parameters, revealing repetitive structure similar to the ideal input (Figure 4.4c).

Partitioning Szalinski includes a *partitioning solver* that uses inverse transformations and a set of heuristics to restructure lists in ways that group similar list elements together (e.g., by kind of geometric primitive). The partitioner can split up elements of a list by equivalence class, individual components of 3D vectors, and kinds of affine transformations. In Figure 4.9, the partitioner will split the list into:

```
(Fold Union
  (Unpart ⟨1, 6⟩
    (List (Cylinder [1, 5]))
    (List (Rotate [0, 0, 120] (Translate [1, −0.5, 0] (Cuboid [10, 1, 1])))
          (Rotate [0, 0, 0]   (Translate [1, −0.5, 0] (Cuboid [10, 1, 1])))
          (Rotate [0, 0, 300] (Translate [1, −0.5, 0] (Cuboid [10, 1, 1])))
          (Rotate [0, 0, 180] (Translate [1, −0.5, 0] (Cuboid [10, 1, 1])))
          (Rotate [0, 0, 240] (Translate [1, −0.5, 0] (Cuboid [10, 1, 1])))
          (Rotate [0, 0, 60]  (Translate [1, −0.5, 0] (Cuboid [10, 1, 1]))))))
```

The introduced Unpart is equivalent to Concat, but additionally stores partitioning hints. Now that the Rotates are gathered uniformly in a list, the structure finder will rewrite the list to:

```
(Map2 Rotate
    (List [0, 0, 120] [0, 0, 0] [0, 0, 300] [0, 0, 180] [0, 0, 240] [0, 0, 60])
    (Repeat 6 (Translate [1, −0.5, 0] (Cuboid [10, 1, 1]))))
```

The arithmetic solver from Section 4.2.2 cannot find a closed form for this list of Rotate parameters. The solver could, however, find a closed form if it were free to sort the list (by z-coordinate, in this case). The sorted list is **not** equivalent to the original. Since the solver only rewrites locally, it does not know if the list appears under a Fold Union (which is AC) or a Fold Diff (which is *not* AC). In the E-graph, *both* situations could actually hold due to sharing. The solver *cannot* soundly rewrite the original list to the closed form Tabulate, but it *can* soundly rewrite the list to:

```
(Unsort ⟨1, 5, 0, 3, 4, 2⟩ (Tabulate (i 6) [0, 0, 60i]))
```

The Unsort inverse transformation allows the solver to introduce the closed form Tabulate in the E-graph, but Szalinski will never extract it or any other program using the inverse transformation forms from Extended λCAD. Instead, rewrites propagate inverse transformations between invocations of locally-reasoning solvers, and additional rules eliminate inverse transformations in contexts invariant to the relevant transformation; these rules are shown in Figure 4.12. The **Map2 Unsort Params** rewrite applies to the running example, producing:

```
(Unsort ⟨1,5,0,3,4,2⟩ (Sort ⟨1,5,0,3,4,2⟩
  (Map2 Rotate
    (Unsort ⟨1,5,0,3,4,2⟩ (Tabulate (i 6) [0, 0, 60i]))
    (Repeat 6 (Translate [1,−0.5,0] (Cuboid [10,1,1]))))))
```

Semantically, this is no different, as (Unsort p (Sort p x)) $= x$, but since the Map2 is in the same eclass as the original list of Rotates, the **Sort Application** rule can fire, communicating the profitable ordering of the Rotate parameters to the outer list. Now, the structure finder and arithmetic solver apply to the sorted list of Rotates, bringing the whole program to:

```
(Fold Union
  (Unpart ⟨1,6⟩
    (List (Cylinder [1,5]))
    (Unsort ⟨1,5,0,3,4,2⟩
      (Tabulate (i 6)
        (Rotate [0, 0, 60i]
          (Translate [1,−0.5,0]
            (Cuboid [10,1,1])))))))
```

From here an additional rewrite (elided from Figure 4.12) can lift the Unsort over the Unpart:

```
(Fold Union
  (Unsort ⟨0,2,6,1,4,5,3⟩
    (Unpart ⟨1,6⟩
      (List (Cylinder [1,5]))
      (Tabulate (i 6)
        (Rotate [0, 0, 60i]
          (Translate [1,−0.5,0]
            (Cuboid [10,1,1])))))))
```

Next, the **Unsort Elimination** rule removes the Unsort, since Fold Union is invariant to order. Finally one additional rule that transforms a Union of an Unpart into a Union of Unions (not shown), produces the desired λCAD output (Figure 4.4b).

4.4.3 Solving for Spherical Coordinates

Inverse transformations are not restricted to list manipulations. In addition to sorting, Szalinski's arithmetic solvers can convert lists to spherical coordinates Moon and Spencer

Map2 Unsort Params - *cads* rule analogous

```
                                      (Unsort perm (Sort perm
(Map2 affine                            (Map2 affine
  (Unsort perm params)          ⇝         (Unsort perm params)
    cads)                                   cads)))
```

Sort Application

```
(Sort ⟨$i_1, ...i_n$⟩ (List $x_1$ ... $x_n$))     ⇝   ( List $x_{i_1}$ ... $x_{i_n}$)
```

Unsort Elimination

```
(Fold Union (Unsort perm l))    ⇝   (Fold Union l)
(Unsort perm (Repeat n x))      ⇝   (Repeat n x)
```

Map2 Unpart Cads - *params* rule analogous

```
                                      (Unpart part (Part part
(Map2 affine                            (Map2 affine
  params                        ⇝         params
  (Unpart part cads))                     (Unpart part cads))))
```

Unpart to Concat

```
(Unpart part lists)             ⇝   (Concat lists)
```

Unspherical Trans

```
                                      (Map2 Trans
(Map2 Trans                             (Repeat n center)
  (Unspherical n center params) ⇝       (Map2 TranslateSpherical
  cads)                                   params cads))
```

Figure 4.12: Representative set of rewrite rules for propagation and elimination of inverse transformations.

[1988]. The resulting list may be easier to find a closed form Tabulate for, but it is not equivalent to the input. Therefore, the solver wraps the Tabulate in an inverse transformation, Unspherical, before passing it to the E-graph for unification. If the Unspherical propagates under a Translate, then the Unspherical Trans rule can replace it with TranslateSpherical form. This approach allows Szalinski to solve for closed forms of lists in spherical coordinates without the solver knowing whether or not it is solving for a list of Translate parameters.

4.4.4 Inverse Transformations, Broadly

This section and the evaluation show that inverse transformations are effective for shrinking λCAD programs, but the technique could be applied more broadly to other uses of Equality Saturation. The key insight is that solvers can remain simple because they only have to reason locally. They are given the flexibility to speculate on potentially profitable ways to transform their inputs. Rewrites can then propagate this information and contextually eliminate the transformations. As in traditional Equality Saturation, these rewrites (and now simple solvers) compose in emergent ways, leading to unexpectedly powerful outcomes, that would have otherwise required more complicated solvers with deep, contextual reasoning ability.

Implementation

Szalinski is implemented in 3000 lines of Rust and uses egg Willsey et al. [2021], an open source E-graph library. Table 4.1 provides a break down of the LOC for each of Szalinski's components. Szalinski uses only simple, custom solvers for arithmetic and list partitioning. Most of Szalinski's 65 rewrites are syntactic and compactly expressed, and the remainder either call out to the solvers or manipulate lists.

Table 4.1: Approximate LOC breakdown of Szalinski

λCAD	Rewrites	Solvers	Main loop	Validation
300	900	400	300	1100

Correctness To validate Szalinski's correctness, we test that the initial and final λCAD programs compile to similar meshes (Figure 4.13). Szalinski first evaluates a λCAD program back to a flat Core λCAD program which is then pretty printed to a CSG program. This evaluation uses the open source OpenSCAD OpenScad [2019] tool to compile the CSGs to triangular meshes and then uses the CGAL CGAL [2018] library to compute the Hausdorff

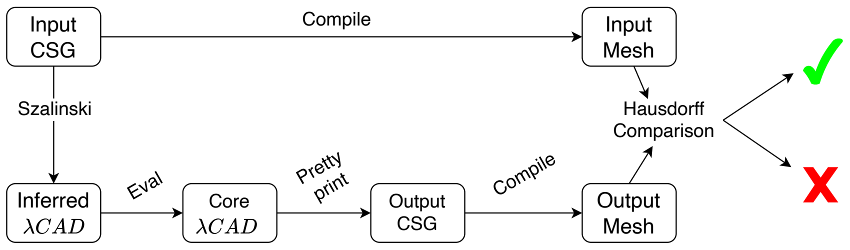


Figure 4.13: The Szalinski tool. The simplification process outputs a parameterized program in the λCAD language. The validation step evaluates Szalinski's output to Core λCAD, pretty prints it to CSG and uses an open source CAD compiler to generate a mesh. The input to Szalinski is also compiled to a mesh. The two meshes are then compared using Hausdorff distance.

distance Munkers [2000], Du et al. [2018] between the two meshes. A Hausdorff distance less than a small ϵ indicates equivalence (ideally it should be zero, but due to rounding errors, it is sufficient to check against ϵ).

4.5 Evaluation

In evaluating Szalinski, we were interested in the following research questions:

- *End-to-End.* (Section 4.5.1) Does Szalinski compose with prior mesh decompilation tools and find parametrizable programs from the flat CSG expressions generated by the latter?

- *Scalability.* (Section 4.5.2) Does Szalinski scale to large flat CSGs? How fast can it find equivalent smaller λCAD programs?

- *Sensitivity analysis.* How do the different components of Szalinski, in particular CAD rewrites and inverse transformations, affect its results?

We ran the evaluation on a 6 core Intel i7-8700K processor with 32 GB of RAM.

4.5.1 End-to-End Experiments

To evaluate the composability of Szalinski with mesh decompilation tools, we ran Szalinski on flat CSGs generated by the Reincarnate (Chapter 3) mesh decompiler. This required

Table 4.2: End-to-end evaluation of Szalinski on the results of Reincarnate Nandi et al. [2018]. SCAD show LOC in original parametrized OpenSCAD implementations, # Tri shows the number of triangles in the mesh, c_{in} and c_{out} are the input and output costs. The last two columns indicate the cost of the output λCAD program when Szalinski does not apply any CAD identities, and when inverse transformations are turned off, respectively.

Id	SCAD	# Tri	c_{in}	c_{out}	No CAD	No Inv
TackleBox	48	280	280	26	60	41
SDCardRack	13	236	206	26	57	49
SingleRowHolder	10	320	198	16	31	38
CircleCell	14	124	79	16	31	16
CNCBitCase	59	268	219	15	27	27
CassetteStorage	13	172	141	15	27	25
RaspberryPiCover	34	332	271	12	27	32
ChargingStation	45	192	141	18	27	29
CardFramer	11	200	172	42	83	42
HexWrenchHolder	13	516	317	16	31	52
Average	26.0	264.0	202.4	20.2	40.1	35.1

investigating what kinds of models Reincarnate supports; we found that it worked best on models that do not contain round edges. We found 10 such models from Thingiverse Thingiverse [2019] and ran Reincarnate on their mesh files to get flat CSGs and converted those to Core λCAD.

Given the Core λCAD inputs, Szalinski synthesizes λCAD programs (Figure 4.13). We compared the parametrized programs synthesized by Szalinski from Reincarnate's output with manually written parametrized programs in OpenSCAD (column 1 in Table 4.2). For four of the 10 models, we found a parametrized OpenSCAD implementation on Thingiverse. For the other six, we manually wrote a parametrized implementation in OpenSCAD. Table 4.2 shows the comparison of the lines of code at every stage of the end-to-end synthesis process, and the cost of the flat input Core λCAD and the output λCAD. Szalinski was able to reduce the cost of the programs by 86% on average. The last two columns report a sensitivity analysis of Szalinski on Reincarnate's output. It shows that both CAD identities and inverse transformations contribute significantly to shrinking λCAD programs.

Compiling the λCAD programs to mesh resulted in meshes equivalent to the source meshes (Hausdorff distance < 0.001). We also manually validated that all 10 inferred λCAD programs are structurally similar to the parameterized input OpenSCAD programs.

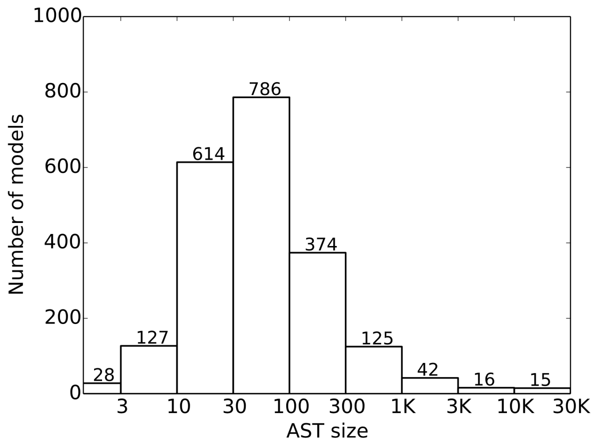


Figure 4.14: Summary of input AST size for Szalinski's large scale evaluation.

4.5.2 Large Scale Evaluation on Thingiverse Models

Mesh decompilation tools have limitations. Reincarnate for example, works mainly on shapes without rounded corners and edges (Chapter 3). Therefore, in order to evaluate Szalinski further, we performed a larger scale evaluation on models from Thingiverse Thingiverse [2019], a popular online model sharing website.

The goals for this part of the evaluation are: (1) to simulate the behavior of mesh decompilation tools by flattening parametrized programs and perturbing them to reproduce the challenges **(C1)** and **(C2)** (introduced earlier), and run Szalinski on these flat CSGs, (2) to analyze the scalability, correctness and efficiency of Szalinski on large-scale real world programs.

Data Collection We built a scraper that downloaded customizable models from Thingiverse. While most models in Thingiverse are shared as triangular meshes which are hard to customize, models under the "Customizable" category Customizable [2019] are intended to be editable, and are therefore more likely to be accompanied with higher-level programmatic representation. The scraper found 12,939 OpenSCAD files from the "Customizable". 912 of these files were invalid, i.e. they were empty, could not be compiled, or used debug features. We filtered out files using features we do not support (like linear extrusion), leaving 2,127 models. Similar to λCAD, the OpenSCAD language supports CSG and also has features like `for` loops that can be used to write more parametrizable CAD programs. OpenSCAD can compile these programs to flat CSG, which Szalinski then accepts as input. Figure 4.14 summarizes the AST sizes of these inputs.

OpenSCAD primitives like spheres and cylinders are parameterized by their geometric precision. The geometric precision indicates the quality of the mesh obtained when the CSG is compiled. For example, a sphere with resolution 100 has a more fine-grained mesh than a sphere with resolution 10. There were several examples where the precision of the primitives was as high as 100. However, OpenSCAD's compiler is slower when generating finer resolution meshes. Since the verifier (Section 4.4.4) uses the OpenSCAD compiler, we capped the precision of all primitives to 25.

Results Figure 4.15 shows the results with a 60 second timeout. The baseline result (leftmost) is slightly perturbed, as OpenSCAD represents affine transformations in an ambiguous way in its CSG format (ex: the representation of `Scale [-1,-1,1]` and `Rotate [0,0,180]` are identical). The second result shows that Szalinski is fast; limiting it to 1 second has very little effect on the result. The third result shows Szalinski is robust to reordering of the inputs. The final two results show CAD rewrites or inverse transformations significantly contribute to Szalinski's performance. We validated all results by comparing the meshes of the original programs against Szalinski's output. All Hausdorff distances were under 0.01, except for 148 cases where CGAL failed to compute the distance and we visually compared the meshes.

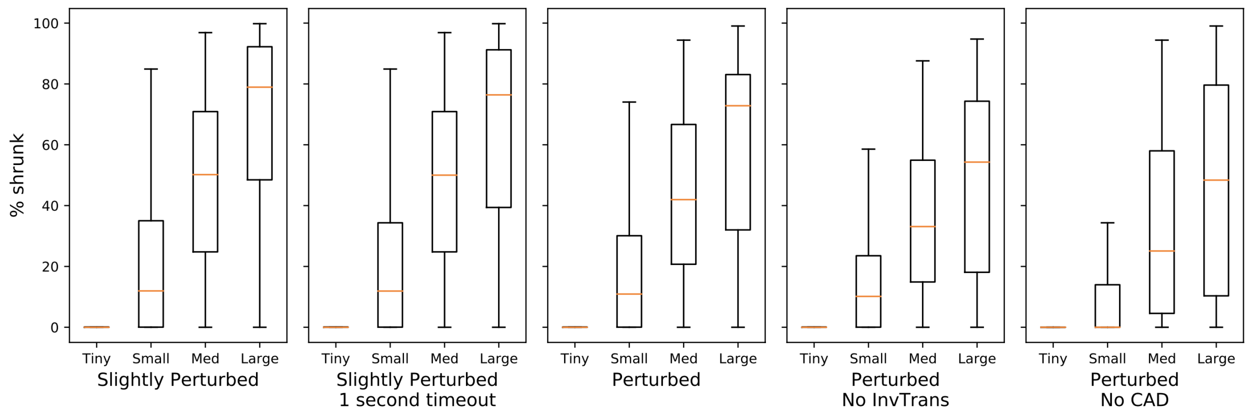
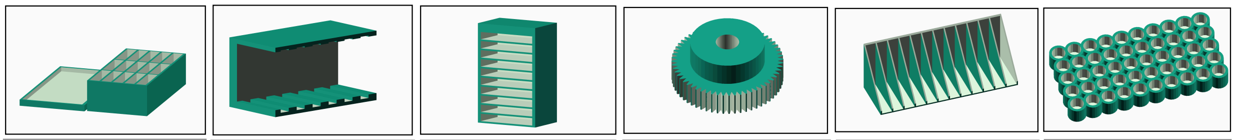


Figure 4.15: Result of running Szalinski on 2,127 Thingiverse examples. Models are grouped by AST size of initial Core λCAD input: 769 were tiny (AST size < 30), 786 small (30 < size < 100), 374 medium (100 < size < 300), and 198 large (300 < size).

4.5.3 Case Studies and Editability

This section discusses three models from the end-to-end evaluation in Section 4.5.1 (a fourth is illustrated in Figure 4.1) and three models from the large scale evaluation in Section 4.5.2.



```
(Union (Difference
  (Cuboid [60, 120, 30])
  (Fold Union
    (Tabulate (i 5) (j 3)
      (Translate [12 * i + 2, 39.3 * j + 2, 2]
        (Cuboid [9.6 37.3 28])))))
 (Fold Difference
  (Map2 Translate
    (List [-67, -2, 0] [-65, 0, 2])
    (List (Cuboid [65, 125, 6])
            (Cuboid [60, 120, 4])))))
```

```
(Difference
  (Cuboid [57, 30, 30])
  (Difference
    (Translate [0, 5, 1.5] (Cuboid [57, 25, 27]))
    (Fold Union
      (Map2 Translate
        (Tabulate (i 7) (j 2) [9 * i, 5, 28 - 26.5 * j])
        (Concat
          (List (Tabulate (i 6) (j 2)
                    (Cuboid [4.5, 25, j + 0.5]))
           (List (Cuboid [3, 25, 0.5])
                   (Cuboid [3, 25, 1.5]))))))))
```

```
(Difference
  (Translate [-57, -3, 3]
    (Cuboid [117, 75, 175]))
  (Fold Union
    (Tabulate (i 10)
      (Translate [-51, -3, 16 * i + 6]
        (Cuboid [105, 58, 13])))))
```

```
(Fold Difference
  (List (Union
      (Cylinder [100, 80, 80])
      (Cylinder [50, 120, 120]))
    (Translate [0, 0, -1] (Cylinder [102, 25, 25]))
    (Fold Union (Tabulate (i 60)
      (Rotate [0, 0, 6 * i]
        (Translate [125, 0, 0]
          (Scale [2.5, 1, 1]
            (Rotate [0, 0, 45]
              (Translate [0, 0, 25]
                (Cuboid [10, 10, 52])))))))))))
```

```
(Fold Union
  (Tabulate (i 12)
    (Translate [0, 13* i, 0]
      (Fold Difference
        (List
          (Cuboid [53.1 14.5 58])
          (Translate [1.5, 1.5, 1.5]
            (Cuboid [51.6, 11.5, 56.6]))
          (Translate [0 0 58]
            (Rotate [0, 45, 0]
             (Cuboid [101.5, 14.5, 100]))))))))
```

```
(Fold Union
  (Tabulate (i 10) (j 5)
    (Translate
        [12.2 * i + 12.2, 12.2 * j + 12.2, 0]
      (Difference
        (Cylinder [13, 7.1, 7.1])
        (Translate [0, 0, 3]
          (Cylinder [11, 5.1, 5.1]))))))
```

Figure 4.16: The first three are examples of end-to-end evaluation where Szalinski ran on the flat CSG output of a mesh decompiler Nandi et al. [2018]. The last three are representative examples that show the usefulness of Szalinski where the flat CSG was generated using OpenSCAD and perturbed to simulate mesh decompilers.

The goal is to highlight some edits made easily possible by Szalinski, which in the flat CSG (and mesh) are nearly impossible. Figure 4.16 shows a rendering of these models and the parametrized λCAD program found by Szalinski. We discuss three categories of edits.

Adding or removing components: consider the gear shown in Figure 4.16. Changing the tooth count in a flat CSG version of this model requires manually computing the position of every tooth and ensuring that the spacing between them is still equal. The λCAD program synthesized by Szalinski makes this modification trivial—it exposes a function (6 × i) for Rotate and the number of teeth (in Tabulate), which can both be easily changed to get a different tooth count. Adding rows or columns of components is also easy in a parametrized model. For example, in the first model in Figure 4.16, another set of compartments can be added by changing the bounds of Tabulate.

Modifying the shape of multiple components: in the last model in Figure 4.16, the cylinders can be all changed to Hexprism by changing it in two places only. These modification in the flat CSGs require changing the shape of each cylinder individually, which is undesirable. Figure 4.1 shows more examples of edits where the shape of the hex-wrench holder can be changed by changing the parameters inferred by Szalinski.

Applying additional affine transformations to components: consider the SD card rack (the second model) in Figure 4.16. This model can be easily customized in the λCAD program to adjust the size of the slots. The λCAD program in the figure shows that in each iteration (in Tabulate), two sizes of Cuboid are removed from the outer box. The dimensions of these can be changed in the function inferred for the Cuboid parameters: (Cuboid [4.5, 25, j + 0.5]) to change the slot size. Similarly, Figure 4.1 showed how an additional rotation can be easily added to Cuboid to make an entirely different model.

Performing these modifications in a flat CSG is tedious and error-prone because they require manually recomputing many parameters for multiple components in the models. Szalinski makes these modifications much easier by exposing different design parameters.

4.5.4 Limitations

Some mesh decompilation tools like InverseCSG synthesize flat CSG programs using enumerative synthesis and random sampling based algorithms like RANSAC Du et al. [2018]. Inferring structure from the output generated by these tools requires equivalence under context using geometric reasoning that Szalinski's prototype currently does not support. InverseCSG provides 50 benchmarks, on all of which we ran Szalinski. The majority of the benchmarks lacked the repetitive structure Szalinski is intended to infer. For one of the models (benchmark 157, a gear), Szalinski was able to infer a `TranslateSpherical` function. However, due to the structure of their outputs, we had to add rewrites like:
`(Difference (Union a b) c)` $\rightsquigarrow$ `(Union (Difference a c) b)` which are unsound without a geometric solver that can check that the intersection of `b` and `c` is empty. We manually applied this rewrite to benchmark-157 but did not add these rewrites to Szalinski's rule database due to their unsoundness.

4.6 Related Work

E-graph based Deductive Program Synthesis E-graphs have been used extensively in superoptimizers Joshi et al. [2002], Tate et al. [2009], Stepp et al. [2011], Bansal and Aiken [2008], and SMT solvers De Moura and Bjørner [2008], de Moura et al. [2015], Detlefs et al. [2005], Torlak and Bodik [2013]. Szalinski's core algorithm is a generalized version of equality saturation Tate et al. [2009]. Integrating linear solvers with compiler optimizers has a long history with tools like Omega Calculator Pugh [1991], Pugh and Wonnacott [1992]. Szalinski's approach of using syntactic rewrites and an arithmetic function solver to modify the E-graph can be considered similar to Simplify Detlefs et al. [2005] which uses an E-graph module for finding equivalent expressions containing uninterpreted functions, and a Simplex module that is used for arithmetic computations.

However, unlike Szalinski, past work does not allow solvers to speculatively add potentially profitable expressions in the E-graph. Inverse transformations allows Szalinski to accomplish this while also mitigating the AC-matching problem for associative and commutative operations like list reordering and regrouping.

2D and 3D Design Synthesis Chapter 3 already covers prior work on inferring high-level CSG from low-level representations like meshes and pixels. This chapter compares

Szalinski with prior work that focuses primarily on loop inference. Ellis et al. Ellis et al. [2018] developed a tool that can automatically generate programs that correspond to hand-drawn images. They first use machine learning to detect primitives in the drawings and then use Sketch Solar-Lezama [2008b] to find loops and conditionals. Szalinski's technique is different from theirs in that they use enumerative search to explore all programs within a given depth (their max AST depth is 3), based on a language grammar, a specification, and a cost, whereas Szalinski uses a rewrite-based synthesis technique where the specification is given as the initial CSG, and Szalinski constructs an E-graph and updates it using semantics preserving rewrites. In order to compare Szalinski with Ellis et al.'s Ellis et al. [2018] tool, we ported their 2D models to 3D and ran Szalinski on them. Szalinski's results had similar loop structure as theirs but further comparison is not possible since their DSL is different. Another line of work Ellis et al. [2019] uses reinforcement learning to synthesize programs for 2D and 3D models. However, the programs inferred by these approaches are much smaller compared to Szalinski.

In computer graphics and vision, symmetry detection Mitra et al. [2013] in 3D shapes is a well studied topic. It can improve performance of geometry processing algorithms. The ability to detect folds and maps in 3D models is more general than symmetry detection because it can find patterns in models that have repetitive structure that is not symmetry. A simple example of this is a union of n cubes increasing in size. In fabrication, Schulz et al. Schulz et al. [2017b] developed algorithms for optimizing parametric CAD models using interpolation methods.

4.7 Conclusions

This chapter addresses the challenge of synthesizing smaller high-level CAD models from the noisy and unstructured outputs of existing triangle mesh to CSG decompilers. We developed *Szalinski*, a prototype tool to synthesize λCAD programs using semantics-preserving rewrites and simple solvers to "reroll loops." By adapting Equality Saturation to the CAD domain, Szalinski can robustly handle common CSG variations exhibited by existing mesh decompilers. Szalinski relies on novel *inverse transformations* to mitigate the AC-matching problem that arises when reordering CAD operations: solvers annotate and merge terms that are only equivalent modulo reordering, then propagate and eliminate such annotations through an E-graph to expose repetitive structure and robustly enable loop rerolling. Inverse transformations are not CAD-specific; we are excited to explore future work investigating how they may be applied in other ordering-sensitive optimization problems, e.g., instruction scheduling Tristan and Leroy [2008, 2009].

We performed a survey of 2,127 real-world CAD models from Thingiverse. The evaluation shows that Szalinski can dramatically shrink many CAD models in seconds.

In future work, it would be interesting to explore richer rewrites for contextual equivalence (Section 4.5.4), more expressive cost functions for capturing richer notions of editability, and connections to interactive CAD editing using direct manipulation tools like Sketch-n-Sketch Chugh et al. [2016].

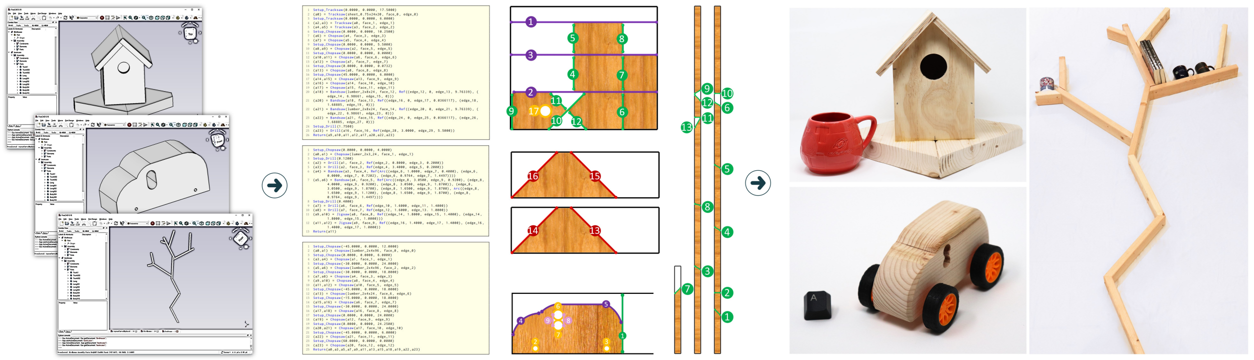
Chapter 5
CARPENTRY COMPILER AND EQUALITY SATURATION

So far we have developed DSLs, compilers, and program synthesizers that operate at the level of high-level designs and meshes. This chapter presents an equality saturation backed optimizing compiler for the domain of subtractive manufacturing — specifically carpentry. The key insight here is that both designs and fabrication plans used in carpentry are programs.

Next-generation manufacturing techniques are revolutionizing the on-demand fabrication of complex custom products. This has spurred important research advances to enable *fabrication-oriented design optimization*. In many applications, however, *fabrication is design-dependent*, defined by a sequence of operations on multiple processes, where the order of operations and choice of hardware can only be optimized for a specific design. This scenario raises unexplored challenges since product design and fabrication instructions must be coupled, even as they are separately optimized.

One of the most influential developments in computer architecture was the introduction of instruction set architectures (ISAs) Amdahl et al. [1964] which define abstract models of computers and serve as an interface between their software and hardware. This abstraction layer enabled the independent development of both hardware and software. This work examines whether similar hardware and software decoupling can be achieved for manufacturing, while focusing The main focus is carpentry design and manufacturing for several reasons. First is application scope. Carpentered items comprise the structures we live in and the furniture they contain, and they are commonly personalized to fit spaces and functions. Second, carpentry provides an appropriate level of complexity for initiating research in this area. It combines multiple processes and assembly constraints within the confines of a bounded problem. Finally, recent advances in mobile robotics allow automated carpentry to occur outside factory floors, making the end-to-end design and fabrication of personalized carpentry not only possible but an exciting and open area of research Lipton et al. [2018].

This chapter introduces Hardware Extensible Languages for Manufacturing (HELM), a system that allows us to represent carpentry designs and fabrication instructions. It takes inspiration from traditional compilers to propose two layers of abstraction, one high-level and process-agnostic (HL-HELM) and the other low-level and process-specific (LL-HELM). HL-HELM is a *design language* related to traditional parametric feature-based CAD languages but focused on subtractive operations that can be mapped to physical woodworking pro-



```
Setup_Tracksaw(0.0000, 0.0000, 17.5000)
(a0) = Tracksaw(sheet_0.75x24x20, face_0, edge_0)
Setup_Tracksaw(0.0000, 0.0000, 6.0000)
(a2,a3) = Tracksaw(a0, face_1, edge_1)
(a4,a5) = Tracksaw(a3, face_2, edge_2)
Setup_Chopsaw(0.0000, 0.0000, 10.2500)
(a6) = Chopsaw(a4, face_3, edge_3)
(a7) = Chopsaw(a5, face_4, edge_4)
Setup_Chopsaw(0.0000, 0.0000, 5.5000)
(a8,a9) = Chopsaw(a2, face_5, edge_5)
Setup_Chopsaw(0.0000, 0.0000, 8.0000)
(a10,a11) = Chopsaw(a6, face_6, edge_6)
(a12) = Chopsaw(a7, face_7, edge_7)
Setup_Chopsaw(0.0000, 0.0000, 0.0732)
(a13) = Chopsaw(a8, face_8, edge_8)
Setup_Chopsaw(45.0000, 0.0000, 6.0000)
(a14,a15) = Chopsaw(a13, face_9, edge_9)
(a16) = Chopsaw(a14, face_10, edge_10)
(a17) = Chopsaw(a15, face_11, edge_11)
(a18) = Bandsaw(lumber_2x8x24, face_12, Ref((edge_12, 0, edge_13, 9.76339), (
        edge_14, 6.98661, edge_15, 0)))
(a20) = Bandsaw(a18, face_13, Ref((edge_16, 0, edge_17, 0.0366117), (edge_18,
        1.68885, edge_19, 0)))
(a21) = Bandsaw(lumber_2x8x24, face_14, Ref((edge_20, 0, edge_21, 9.76339), (
        edge_22, 6.98661, edge_23, 0)))
(a22) = Bandsaw(a21, face_15, Ref((edge_24, 0, edge_25, 0.0366117), (edge_26,
        1.68885, edge_27, 0)))
Setup_Drill(1.7500)
(a23) = Drill(a16, face_16, Ref(edge_28, 3.0000, edge_29, 5.5000))
Return(a9,a10,a11,a12,a17,a20,a22,a23)
```

```
Setup_Chopsaw(0.0000, 0.0000, 4.0000)
(a0,a1) = Chopsaw(lumer_2x3_24, face_1, edge_1)
Setup_Drill(0.1200)
(a2) = Drill(a1, face_2, Ref(edge_2, 0.8000, edge_3, 0.2000))
(a3) = Drill(a2, face_3, Ref(edge_4, 3.4000, edge_5, 0.2000))
(a4) = Bandsaw(a3, face_4, Ref(Arc((edge_6, 1.0000, edge_7, 0.4000), (edge_6,
        0.0000, edge_7, 0.7202), (edge_6, 0.9764, edge_7, 1.4497))))
(a5,a6) = Bandsaw(a4, face_5, Ref(Arc((edge_8, 3.0500, edge_9, 0.9200), (edge_8,
        4.0000, edge_9, 0.9200), (edge_8, 3.0500, edge_9, 1.8700)), (edge_8,
        3.0500, edge_9, 1.8700), (edge_8, 1.6500, edge_9, 1.8700), Arc((edge_8,
        1.6500, edge_9, 1.1200), (edge_8, 1.6500, edge_9, 1.8700), (edge_8,
        0.9764, edge_9, 1.4497))))
Setup_Drill(0.4000)
(a7) = Drill(a6, face_6, Ref(edge_10, 1.6000, edge_11, 1.4000))
(a8) = Drill(a7, face_7, Ref(edge_12, 1.6000, edge_13. 1.0000))
(a9,a10) = Jigsaw(a8, face_8, Ref((edge_14, 1.8000, edge_15, 1.4000), (edge_14,
        1.8000, edge_15, 1.0000)))
(a11,a12) = Jigsaw(a9, face_9, Ref((edge_16, 1.4000, edge_17, 1.4000), (edge_16,
        1.4000, edge_17, 1.0000))
Return(a11)
```

```
Setup_Chopsaw(-45.0000, 0.0000, 12.0000)
(a0,a1) = Chopsaw(lumber_2x4x96, face_0, edge_0)
Setup_Chopsaw(0.0000, 0.0000, 6.0000)
(a3,a4) = Chopsaw(a1, face_1, edge_1)
Setup_Chopsaw(-30.0000, 0.0000, 24.0000)
(a5,a6) = Chopsaw(lumber_2x4x96, face_2, edge_2)
Setup_Chopsaw(-30.0000, 0.0000, 18.0000)
(a7,a8) = Chopsaw(a4, face_3, edge_3)
(a9,a10) = Chopsaw(a8, face_4, edge_4)
(a11,a12) = Chopsaw(a10, face_5, edge_5)
Setup_Chopsaw(-45.0000, 0.0000, 18.0000)
(a13) = Chopsaw(lumber_2x4x24, face_6, edge_6)
Setup_Chopsaw(-15.0000, 0.0000, 18.0000)
(a15,a16) = Chopsaw(a6, face_7, edge_7)
Setup_Chopsaw(-30.0000, 0.0000, 24.0000)
(a17,a18) = Chopsaw(a16, face_8, edge_8)
Setup_Chopsaw(0.0000, 0.0000, 24.0000)
(a19) = Chopsaw(a12, face_9, edge_9)
Setup_Chopsaw(0.0000, 0.0000, 24.2500)
(a20,a21) = Chopsaw(a17, face_10, edge_10)
Setup_Chopsaw(-45.0000, 0.0000, 6.0000)
(a22) = Chopsaw(a21, face_11, edge_11)
Setup_Chopsaw(60.0000, 0.0000, 0.0000)
(a23) = Chopsaw(a20, face_12, edge_12)
Return(a0,a3,a5,a7,a9,a11,a13,a15,a18,a19,a22,a23)
```

Figure 5.1: The HELM compiler converts high-level geometric designs made by users to low-level fabrication instructions that can be directly followed to manufacture parts. The compiler performs multi-objective optimization on the low-level instructions to generate Pareto-optimal candidates.

cesses. LL-HELM is a *fabrication language.* Programs in LL-HELM can be directly followed to manufacture a part, where each operation in a program is drawn from an extensible set of fabrication instructions. This work also propose a new compiler that verifies HL-HELM code and optimizes fabrication instructions (LL-HELM code). Because the target language, LL-HELM, is process-specific, we design an architecture that is *extensible* to new hardware.

In addition to the abstraction and compilation system, a key technical contribution of HELM is a novel optimization algorithm for manufacturing enabled by the proposed pipeline. Cut planning directly affects the precision of manufactured parts, material wastage, and production time. Optimizing multi-process cuts is challenging because it involves a long sequence of interdependent steps with multiple conflicting objectives, and if not done properly, it can cause significant labor overhead. By representing the fabrication process as a program, we can draw on ideas from compiler optimization to find an efficient sequence of operations that meet user specifications by adapting search-based superoptimization techniques based on *E-graphs* Joshi et al. [2002], Tate et al. [2009].[1] E-graphs compactly represent (exponentially) large equivalence classes of terms, support extensibility via simple syntactic rewrites, and enable cooperation of various solvers through a common representation. However, applying E-graphs in the context of fabrication requires addressing several technical challenges: fabrication operations are generally not *linear*,[2] some operations do not map to standard

[1]A clarifying example for the use of E-graphs is arithmetic expression simplification Panchekha et al. [2015]. Equivalences like commutativity and associativity are encoded as rewrite rules; a search engine then explores the space of all possible rewrites in order to minimize the expression's cost.

[2]Here "linear" is meant in the type-theoretic sense Wadler [1990].

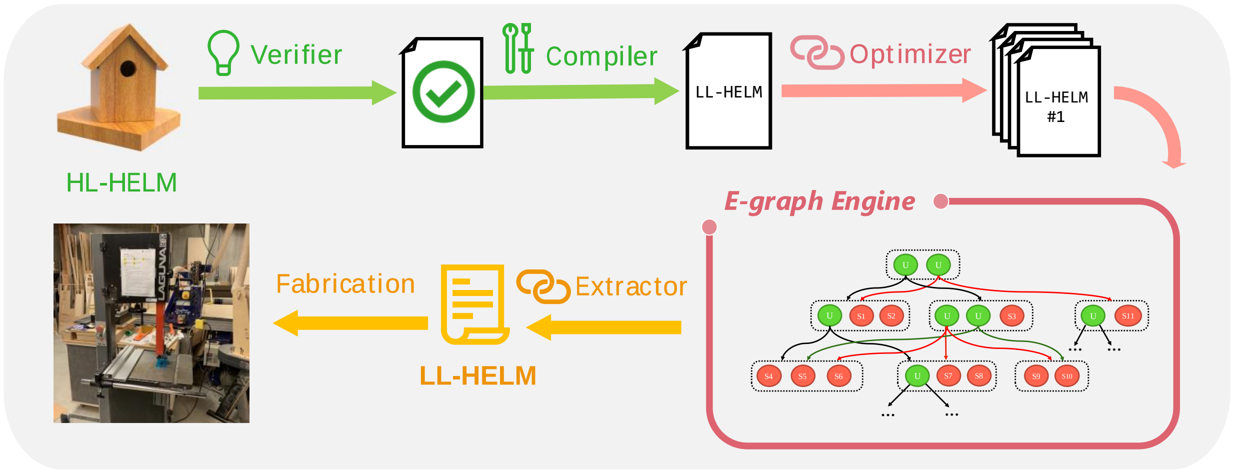


Figure 5.2: System pipeline. The input to HELM is a HL-HELM program designed by a user in the IDE. The verifier first checks if the design is manufacturable. The compiler converts the verified HL-HELM program to a LL-HELM program. Then the various optimizers populate an E-graph by finding various equivalent optimal programs. Finally, the extractor performs a multi-objective optimization to find the most optimal programs from the E-graph.

algebraic operations, some equivalences are difficult to express as syntactic rewrites, and objectives are multiple and conflicting. HELM overcomes these challenges by developing new geometric solvers that communicate with E-graphs, and by extending E-graph extraction to produce a set of Pareto-optimal candidate fabrication plans. The evaluation demonstrates how the resulting fabrication instructions can be optimized to meet different user-specified objectives, such as accuracy, fabrication time, and material cost.

5.1 Overview

Let us now consider the typical process of designing and fabricating a simple wooden part. First, designers consider available materials and fabrication processes and use this information to guide the first draft of their design. The design, typically modeled in a *parametric* CAD system, is then used to iteratively explore possible variations. The designer uses feedback provided by the CAD tool as well as potential simulation plug-ins to iterate on the design. Once satisfied with the resulting configuration, a specific way to fabricate the part must be identified. For example, the fabricator chooses the stock to use for each part, the cutting tools to maximize precision, the order of cuts to minimize the number of setups, or when and how to stack parts to minimize the number of cuts. In a workshop, the designer and fabricator may be the same person; in a corporate setting, they may be different teams in different companies or in different countries.

The above description of the typical pipeline has two important yet conflicting takeaways in the design space and the fabrication space. First, decoupling design and fabrication could advance computational tools that assist each process. On the other hand, it is essential to take fabrication into account during design since it defines the space of what can be physically realized. Mapping free-form designs to a fabrication plan will likely lead to approximations that affect performance or impose unjustifiably high fabrication costs.

5.1.1 Design Philosophy

The proposed architecture accounts for both seemingly conflicting ideas noted above. The goal is to ensure that *design is driven by available fabrication options* while providing abstractions, similar to ISAs, that decouple design and fabrication. This allows advanced algorithms to optimize designs that are fabrication independent and to optimize fabrication that is hardware dependent.

Fabrication-Oriented Design HL-HELM is inspired by feature-based CAD languages, which define a sequence of geometric operations that construct the shape bottom-up. The advantage of this modeling technique is that it defines geometry as programs and has been proven effective. The key difference is that this work defines *fabrication-aware* features. Since carpentry is a subtractive process, the features perform subtractive operations on stock instead of performing standard CSG operations. This ensures that the resulting programs can be effectively verified and compiled to LL-HELM while preserving the process-independent constructive geometry modeling paradigm that designers are accustomed to using.

Fabrication-Independent Design Optimization While it is important to ensure that design is driven by manufacturing realities, this work also seeks abstractions that allow design optimization without the need to compute low-level fabrication details. Two methods can be used to search the design space: interactive exploration and automatic optimization.

For *interactive exploration*, it is important to efficiently verify that a program in HL-HELM is feasible. One option would be to define a dense language, which ensures that all programs in the language are valid, i.e., map to a valid fabrication plan. However, languages constrained to be dense are typically less expressive and intuitive. HELM prioritizes the latter attributes and develop a verifier that can validate a HL-HELM program in real-time. Additionally, HELM has an IDE (Integrated Development Environment) for HL-HELM inspired by modern CAD systems that lets users intuitively explore the design space while ensuring the validity of every design. Using this IDE, user interactions with a 3D model are n ::= *Int* | *Float*   *pt_2* ::= *n n*   *pt_3* ::= *n n n*   *catalog_id* ::= *UID string*

geom::= *Point* pt_2 | *Line* pt_2 pt_2 | *Circle* pt_2 *n* | *Spline* pt_2* | ...

constraint::= *Parallel geom geom* | *Concentric geom geom* | ...

query::= *Query_Face_By_Closest_Point n n n* | *Query_Vertex_By_Closest_Point* | ...

sketch::= *Make_Sketch query geom** *constraint**

design_op::= *Make_Stock n n n* | *Make_Cut id sketch* | *Make_Hole id sketch* | ...

hlhelm::= (*Assign id** *design_op*)*;

face ::= *uid*   *edge* ::= *uid*

setup_op::= *Setup_Chopsaw angle angle offset* | *Setup_Drill diameter* | ...

ref_pt::= *edge offset edge offset*

fab_op::= *Lumber catalog_id*
    | *Sheet catalog_id*
    | *Stack id id*
    | *Unstack id*
    | *Chopsaw id face edge*
    | *Bandsaw id face ref_pt**
    | *Jigsaw id face ref_pt**
    | *Drill id face ref_pt**
    | ...

llhelm::= (*setup_op* | *Assign id** *fab_op*)* *Return* *id**

Figure 5.3: Syntax of HL-HLEM and LL-HELM.

automatically mapped to HL-HELM code, and constraints on valid programs are translated to constraints on user interactions, guiding users in defining valid programs.

In addition to the interactive exploration of the feasible design space provided by the verifier/IDE, HELM's design system uses parametrization as a basis for *automated optimization*. As in standard CAD tools, HELM's feature-based system is *parametric* from construction, defining a search space for physically-based optimization, as is done in the previous work Schulz et al. [2017a].

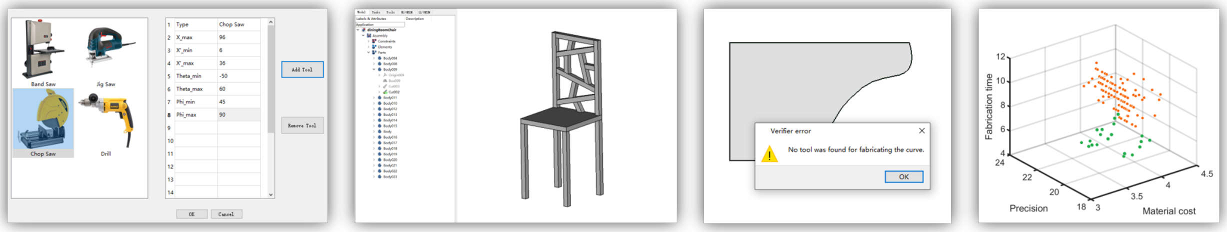
Hardware-Dependent Fabrication Optimization The system's back-end must define a complete list of instructions that can be directly used for fabrication. Therefore, it must define CAM operations for specific types of tools, each with pre-defined capabilities and requirements on the workpieces they can accommodate. It is essential to provide an *extensible* architecture because it would be impossible to define a language that explicitly represents all existing (and emerging) fabrication processes. Extensibility is achieved by establishing a list of verifier rules for each LL-HELM operation and a surjective mapping from HL-HELM to LL-HELM operations. This allows establishing clear guidelines for incorporating new processes into the language.

Finally, the proposed system must automatically generate fabrication plans for a given design. HELM accomplishes this by means of a compiler that can generate LL-HELM code from HL-HELM. The compiler has an optimizer that can handle multiple and conflicting costs, for example, fabrication time, material cost, and accuracy. In summary, this work presents an architecture with the following properties:

- HL-HELM represents subtractive feature-based modeling.
- HL-HELM validity is supported by a verifier and IDE.
- HL-HELM is parametric.
- LL-HELM and the verifier reference the available hardware.
- The full stack is easily extensible to new hardware.
- The compiler performs multi-objective optimization.

5.1.2 Design Processes

Before starting their design, users should import libraries of materials and tools so the compiler has feasible instructions for mapping the designs to specific hardware (Figure 5.4(a)).



1	Type	Chop Saw
2	X_max	96
3	X'_min	6
4	X'_max	36
5	Theta_min	-50
6	Theta_max	60
7	Phi_min	45
8	Phi_max	90
9		
10		
11		
12		
13		
14		

Verifier error

No tool was found for fabricating the curve.

(a) Tool selection (b) Modeling UI (c) Verifier (d) Pareto front exploration

Figure 5.4: Design process: First, users import a library of materials and tools so that the compiler can map design features to fabrication operations. Second, they create a design in an intuitive interface. Third, at each step of the design process, HELM's verifier checks the manufacturability of the design; for example, in (c), the maximal radius of curvature is too big for the part to be fabricable using any of the available processes in the library. Fourth, the compiler generates a set of fabrication plans with different trade-offs. The instructions generated without optimization are shown in orange, and the outputs of HELM (i.e., optimized instructions) are shown in green.

Users then create designs with an intuitive interface that adheres to the same process as standard parametric feature-based CAD systems; in fact, HELM is built as a plug-in for FreeCAD Team [2019]. The key difference is that the allowed features map to subtractive operations that correspond to carpentry operations: get stock, make poly-cuts, and make holes (Figure 5.4(b)). The manufacturability of the designs is checked by the verifier, and users are notified if any features are invalid. As in standard CAD tools, parametric modeling lets users iterate on their designs while satisfying constraints (Figure 5.4(c)). Once designs are finalized, an optimizing compiler generates a set of LL-HELM programs with different trade-offs from which users can choose (Figure 5.4(d)).

5.2 *Language and Compiler*

This section describes the design language, compiler, and fabrication instruction language that were designed based on the considerations and requirements discussed in Section 5.1.

High-level HELM Figure 5.3 (left) shows the grammar for HL-HELM programs. These programs consist of a sequence of assignments that bind *design_ops* to identifiers. *design_ops* are high-level fabrication operations which depend on a set of parameters. The proposed language is inspired by standard feature-based CAD scripting languages FeatureScript [2019], where features map to fabrication operations (e.g., get stock and make cut) as opposed to purely geometric operations (e.g., extrude and loft). As in CAD languages, 2D sketches are used to specify the path of operations and are defined by a set of 2D parametric primitives and constraints. Computationally, a HL-HELM program can be evaluated with an interpreter that runs each assignment in sequence. To run an assignment, the interpreter evaluates the operation to a B-rep (Boundary Representation) using a geometric kernel (OpenCASCADE SAS [2019]) in the context of bindings resulting from previous assignments, referenced using identifiers.

This work also draws ideas from CAD referencing schemes Baba-Ali et al. [2009], Bidarra et al. [2005], using *queries* to reference part of the geometry (e.g., an edge or face) on top of which operations can be defined. This approach allows consistent referencing of parts of the model that, coupled with a direct specification of constraints, allow models to be consistently regenerated after parameter updates. As in modern parametric modeling systems, it lets us define and constrain the ways a model can vary, defining a parameter space that can be used for design optimization Schulz et al. [2017a]. Note that programmers do not need to manually write out complex query parameters since an IDE automatically creates queries when users select a part—i.e., click on a part with the mouse.

Low-level HELM Figure 5.3 (right) shows the syntax of LL-HELM. A LL-HELM program is a sequence of either *setup_op*s or assignments that bind *fab_op*s to identifiers. *fab_op*s are fabrication operations that explicitly reference available hardware and material. These operations include taking a piece of lumber from a material catalog, performing cuts with different tools, and stacking, i.e., placing parts together to allow operations to be applied simultaneously to improve fabrication efficiency. Some fabrication operations require a setup that configures the tool to perform the task, e.g., setting the angles of a chopsaw. A LL-HELM program is concluded by a *Return* statement which returns the resulting parts obtained from fabrication operations.

Unlike HL-HELM, LL-HELM lacks the concept of queries because it is intentionally non-parametric: the compiler finds the optimal fabrication plan for concrete design, as specified by an instance of the parameters. References must be defined to allow the accurate positioning of parts with respect to the cut blade. In LL-HELM, reference points for the cut operation are defined by the intersection of two lines, where each line is specified by an offset from an edge on the part.

There is an explicit surjective mapping from every feature in HL-HELM to an operation in LL-HELM. This mapping is shown in Figure 5.5(b). There are three types of features in HELM's current implementation of HL-HELM which is easily extensible as Section 4 explains. In the figure, green denotes stock allocation, red denotes cuts, and yellow denotes holes.

Tool	Precision	x	y	z	x’	Theta	Phi	R	P	C	I
Jigsaw	low	$(0, \infty)$	$(0, \infty)$	$(0, 1'')$	$(0, \infty)$	(0 , 180)	90	$(1'', \infty)$	T	T	T
Chopsaw	high	$(0, 96'')$	$(0, 6'')$	$(0, 4'')$	$(0, 36'')$	(-50, 60)	(45, 135)	-	F	F	F
Bandsaw	medium	$(0, 26'')$	$(0, 24'')$	$(0, 6'')$	$(0, 13'')$	(0, 180)	90	$(1'', \infty)$	T	T	F
Tracksaw	high	$(0, 96'')$	$(0, 48'')$	$(0, 1'')$	$(0, 36'')$	90	(45, 135)	-	F	F	F
Drill	high	$(0, \infty)$	$(0, \infty)$	$(0, \infty)$	$(0, \infty)$	-	-	d_{bit}	-	-	-

Table 5.1: Process characterization table for all tools in LL-HELM. d_{bit} is the drill-bit diameter. P: Partial, C: Curve, I: Internal.

Process Characterization To generate LL-HELM code, the compiler must understand the capabilities and constraints on each fabrication process, e.g., the maximum depth of a stock that can be set up on a chopsaw. It must also be able to measure the performance of each process in order to optimize fabrication time and accuracy. As part of the architecture, HELM retains for each process the set of constraints and performance measurements in the form of *process characterization*. The process characterization enables the compiler to measure feasibility, fabrication time, and accuracy for a given fab_op in LL-HELM. For example, it uses process characterization to determine that the accuracy of a chopsaw is higher than a bandsaw.

Table 5.1 describes the process characterization for every operation in HELM’s pipeline. The first fours rows summarize the saw operations. Tracksaw and chopsaw are the most precise of all the cutting operations we support, followed by bandsaw and jigsaw. x and y are the maximum lengths the tool can support the respective dimension. z is the maximum height of the part that can be fit under the tool. x' is the maximum distance between the leftmost end of the tool’s platform and the part. In the case of jigsaws, the maximum x and y dimensions are ∞ since the tool does not in any way constrain the part along x or y. Due to the same reason, x’ is also not constrained. The values for the other tools are in the table. The value of z is 1 inch for jigsaw and tracksaw. For chopsaw, it is 4 inches and for bandsaw it is 6 inches. Theta represents the miter angle and phi represents the bevel angle. R is the minimum curvature of a path that the tool can follow. For chopsaw and tracksaw curved cuts are not possible, but for bandsaw and jigsaw the values are shown in the table. Figure 5.5(a) illustrates these parameters. Both jigsaw and bandsaw support partial cuts (P) and curves (C) whereas chopsaw and tracksaw do not. “Internal” (I) indicates whether the tool can be used to make an internal cut on a part. Only jigsaws can be used to perform such cuts.

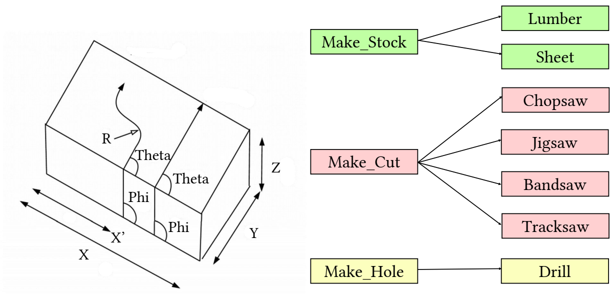


Figure 5.5: (a) Process characterization diagram for saws. (b) Surjective mapping from HL-HELM to LL-HELM in the HELM compiler.

Compiler The compiler from HL-HELM to LL-HELM maps abstract, high-level fabrication operations to concrete, process-specific operations. The first step of this process is to ensure that a valid mapping exists since it is possible to generate HL-HELM programs that do not correspond to feasible instructions. Therefore, every assignment in HL-HELM can be mapped to one or more sequences of assignments and setups in LL-HELM. For example, Make_Cut maps to Setup_Chopsaw followed by Chopsaw, and also to Bandsaw, while Make_Hole maps to Setup_Drill followed by Drill. HELM's verifier sequentially attempts to map each assignment of a HL-HELM program to the possible LL-HELM programs it can be mapped to. It is essentially a simulator to evaluate context using 1) the same geometry kernel used in the front-end, and 2) the process characterization to measure feasibility. If this process can be executed to completion, the HL-HELM program is valid. This can be done interactively and used to provide design feedback in the IDE. If the available hardware changes, this process can automatically verify feasibility and map HL-HELM code to the newly available resources.

Once the compiler generates a valid LL-HELM program, it considers different ways it can be re-written to optimize the fabrication process. This multi-stage process is discussed in Section 5.3.

Extensibility The surjective mapping from HL-HELM to LL-HELM along with the process characterization allows this architecture to be easily extensible to new fabrication processes. Adding a new process involves three steps: 1) adding a new LL-HELM operation and (possibly) setup for the new process, 2) defining the process characterization, 3) defining a mapping from HL-HELM to this process. The third step can be done either by assigning a mapping from an existing design operation or defining a new one. Consider adding a table-saw process. This requires defining the operation and its corresponding setup in LL-HELM

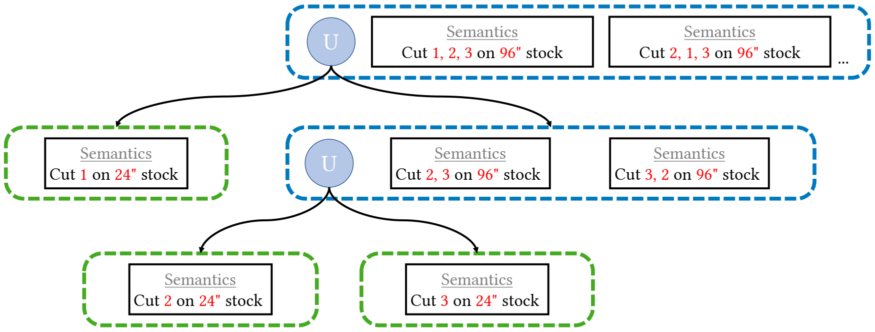


Figure 5.6: An example of an E-graph for a simple design that outputs three 20” long two-by-four parts when the stock library has 96” and 24” stock lumber.

as follows:

$$Tablesaw\ id\ face\ edge,\ Setup\_Tablesaw\ angle\ offset,$$

adding the process characterization for this tool to define the validity constraints and cost functions, and extending the *Make_Cut* operation in HL-HELM to also map to the tablesaw process. Consider another example where a drill press with an arbitrary hole depth is added to the language: *DrillPress id face ref_pt depth*. In this case, the change in HL-HELM involves adding a new operation to allow partial holes: *Make_Partial_Hole id sketch n*, where n is the depth.

5.3 Fabrication Optimization

This section details how HELM's compiler optimizes low-level fabrication plans to provide diverse Pareto-optimal candidates trading off between material cost, fabrication time, and precision.

5.3.1 E-graphs for Fabrication

Chapter 1 provides an overview of E-graphs. Using E-graphs in fabrication requires addressing three technical challenges. First, E-graphs were originally developed for automated theorem proving in structural logics Nelson [1980], where there are no *linearity* constraints on variable reuses. However, HELM's E-graph engine needs to account for linearity in fabrication. For example, after a piece of lumber L is cut into two pieces, the fabrication plan should no longer refer to L since it no longer exists. Second, the fabrication domain requires new *conditional* rewrites, for example, encoding conditions under which cuts can be stacked. Third, since fabrication includes many different and often conflicting objectives, HELM needs to generate several candidates (i.e., Pareto-front candidates) based on user-defined multi-objective cost functions, rather than extracting a single solution from the E-graph bottom-up.

HELM focuses on *stock* and *union* enodes. Stock enodes represent a series of subtractive operations all applied on a single piece of stock, capturing both part layout and per-cut fabrication process selection. Union enodes point to a set of child eclasses, each of which contains stock enodes or (recursively) more union enodes. Each union enode thus represents all the fabrication plans that can be built by selecting representatives from its children. This partitioning into stock and union enodes enables encoding plans that reuse *scrap*, or "offcuts". Each enode in the root eclass represents a set of fully concrete fabrication plans, corresponding either to a particular concrete set of layout and process choices (in the case of a stock enode) or all the recursive combinations of plans from child eclasses (in the case of a union enode). In the E-graph, equivalence is defined as producing identical output, and so all programs that generate the same result will be represented in the same eclass.

Consider the illustration of an E-graph for a simple design that outputs three $20''$ long $2x4$ parts, when two types of stock are available in the library: one is $24''$ long and the other is $96''$ long (Figure 5.6). A sequence of cuts performed on a single stock is represented as a stock enode, e.g., cutting parts 1 and 2 on a $24''$ stock. Since the union of cutting part 2 on a $24''$ stock and part 3 on a $24''$ stock is equivalent to cutting these two parts on a 96" inch stock, these candidates are represented in the same eclass. Even in this simple example, many different programs can be extracted from the E-graph as shown in the figure: all parts on a $96''$ stock, each part on a $24''$ stock, or one part (part 1) on a $24''$ stock and the other two on a $96''$ stock – and within each of these layout strategies many different fabrication process selections are possible.

HELM uses geometric solvers to construct a set of eclasses and enodes in the E-graph, supporting both linearity constraints and conditional rewrites (Sec. 5.3.2) and detail a new method of E-graph extraction which supports the multi-objective optimization requirements for fabrication (Sec. 5.3.3).

5.3.2 E-graph Construction

In general, defining a fabrication plan for making parts in a carpentry project involves the following two major steps: 1) laying out parts on stock lumber and 2) choosing appropriate cutting tools and the order to apply them. Constructing an E-graph that covers many of the possible manufacturing plans is challenging because there are many different ways to assign material, order cuts, and combine multiple tools, *each* of which results in a combinatorial explosion. Combining all of them makes the space of programs even larger. To make the

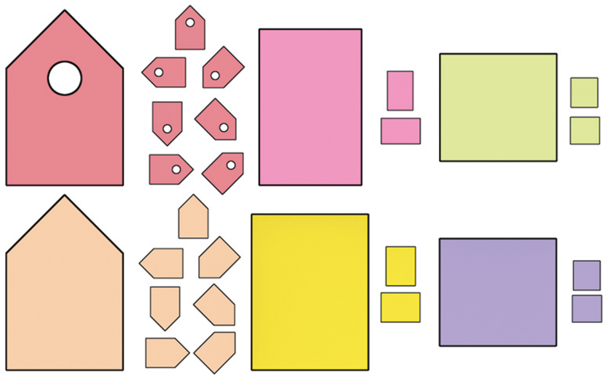


Figure 5.7: 2D shapes for birdhouse and their different orientations.

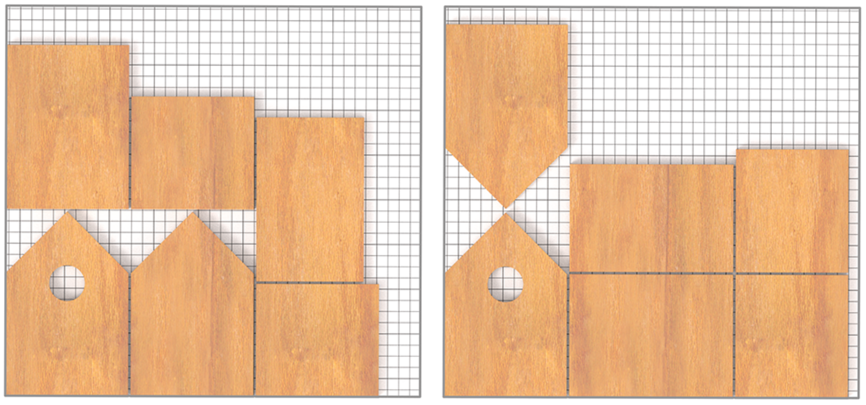


Figure 5.8: Comparison between traditional packing result (left) for minimizing bounding volume and HELM's packing result (right) for maximizing the number of shared edges.

space of programs tractable, HELM uses several pruning strategies that eliminate programs that correspond to unrealistic scenarios and keep only those that are feasible in practical carpentry.

Packing Pieces onto Stock At the first stage of HELM's optimization pipeline, the parts designed by users are assigned to stock lumbers, where orientation and degree-of-freedom are also decided. This work provides a common library of stock lumber that can be readily purchased at home improvement stores; the library can easily be extended with other customized stock. HELM takes as input the bounding boxes of the parts and compares their dimensions with the sizes of the available stock to evaluate the feasibility of an assignment. HELM's prototype implementation includes a library that consists of commonly used materials, and the packing algorithm generates candidates for all stock lumber pieces on which the parts fit. Since many cuts are straight and most parts are polygons, it is possible to minimize the number of cuts by aligning multiple parts so that a single cut can be applied to more than one part (example shown in Figure 5.8). As a result, unlike conventional packing problems which primarily minimize the bounding volume Hopper and Turton [2001], Burke et al. [2006], HELM also minimizes the number of cuts.

Unfortunately, packing problems have been shown to be NP-hard, and it is infeasible to explore the space of all possible packing strategies due to combinatorial explosion. Parts designed in carpentry are usually not arbitrary; HELM therefore targets the packing problem by proposing a simple-yet-efficient algorithm in the cases of 1-DOF (1D packing) and 2-DOFs (2D packing).

Given a set of shapes, e.g., the shapes for the birdhouse shown in Figure 5.7, the goal is to pack them on to a sheet for cutting to maximize the number of aligned edges. To start packing, the algorithm randomly picks an oriented shape and places it on the initial rectangular sheet. This changes the shape of the remaining sheet, as shown in Figure 5.8. To pack the next shape in the remaining sheet, the algorithm picks two edges of the sheet, two edges of the shape, and solves a linear set of constraints to check if the pairs of edges can be aligned. If they cannot be aligned, it continues to pick a different pair of edges from the sheet and the shape. If there is no solution for any pair of edges, the algorithm randomly picks one edge from the sheet and the shape and aligns them to minimize its volume, as is done in standard cutting and packing algorithms Hopper and Turton [2001], Burke et al. [2006]. The packing algorithm takes into account the dimension of the "kerf", i.e., the parts are separated from each other by the width of the saw blade.

This process is repeated for all the shapes in the design to obtain a candidate packing on a stock, and further, HELM's packing algorithm is repeated for all stock pieces in the library. HELM organizes all of the packed results as stock enodes, and constructs E-graphs simultaneously. Since HELM generates many packing strategies for every design, it use a heuristic to prune some of the results. Packing solutions with more aligned edges are better since they require fewer cuts. HELM therefore sort all solutions by the number of aligned edges and keep the top n results.

Defining Cuts on Stock After arranging a set of parts on a piece of stock, we must select the fabrication process for each cut and the order of cuts. Moreover, process-specific setups and references need to be identified under tool constraints and workpiece constraints. For instance, the process characterization of the chopsaw specifies that the maximum thickness of a workpiece is $4''$. These constraints must be considered when selecting tools for each operation. A cut may not be mappable to a fabrication instruction due to violation of constraints. On the other hand, some cuts can be mapped to multiple feasible processes.

Further, a cut can be either across the whole workpiece or only to a certain position (partial). HELM automatically takes all these cases into account to generate a large family of equivalent fabrication plans, which essentially creates more enodes for the eclasses that pointed to from stock enodes, and populates the E-graphs. HELM uses two heuristic pruning strategies.

1. Some measurements are easier to accurately take than others. For example, in the imperial system, distances are more precise and easier to measure if they are integer factors of $[1'', 1/2'', 1/16'']$ as those corresponding to the demarcations of common measuring tapes, and angles when they are integer factors of $[45°, 15°, 1°]$. If HELM finds multiple setups for the same process, it keeps only those that lead to the best measurements and discards others; if any setup involves one of the above measurements then all others are discarded; otherwise, the ones with values closest to some entry in the above lists are kept.

2. HELM prefers complete cuts over partial cuts when possible because partial cuts are difficult to perform, and they tend to be imprecise. For any instruction, if both partial and full cuts are possible, The HELM compiler prunes away the partial cut candidates, and keeps only the full cut candidates. However, if no full-cut solutions can be found, the compiler will use a partial cut solution.

5.3.3 E-graph Extraction

Even with the pruning strategies described in the previous section, the E-graph can have up to $\mathcal{O}(N \times K)$ enodes, where N is the number of e-classes and K is the number of sub-programs in each e-class. The total number of programs that can be generated by combinations of all these sub-programs grows exponentially (i.e., $\mathcal{O}(2^{N\times K})$), so it is important to have efficient ways of exploring this space for extraction.

Objectives HELM produces a set of optimized fabrication plans with respect to the following three objectives.

- *Cost f_c:* Every assignment statement that uses stock, i.e., lumber or sheet is assigned a cost depending on the type of the stock (plywood, two-by-four, two-by-three, etc.).

- *Precision f_p:* The process characterization of a tool provides a precision value. For example, when making straight cuts, a chopsaw is more precise than a bandsaw, which is more precise than a jigsaw. f_p is the product of a tool's precision and the error introduced while making a cut. A standard tape measure or ruler provides divisions of an inch in increments of one-sixteenth. Therefore, the current implementation considers a measurement to have zero error if it is a multiple of one-sixteenth (or better, a whole number). For other measurements, the error is the absolute value of its difference with the closet marking on the tape measure. Due to the modular design of HELM, it is also possible to plug in alternate implementations for error.

- *Time f_t:* Different fabrication processes require different manufacturing time. For example, it is easier to perform a chopsaw cut than a tracksaw cut in practice. Moreover, a program that requires users to change the setup for every cut is worse than a program that reorders and makes multiple cuts with the same setups. HELM's time metric f_t considers the minimization of configuration switching on a single tool.

In summary, given a set of sub-programs constituting a complete fabrication plan, HELM needs to compute all objectives efficiently. f_c is computed by summing up the costs of a program's stock input. f_p is defined as the average value of the sum of all errors (the deviation from the lowest scale) scaled by the precision weight of the tool being used. Both of these objectives are *modular*, i.e., the best representative node for an eclass can be computed directly from its children, and thus are straightforward to compute. However, to compute f_t HELM must measure the benefits of sharing setups across different operations within a program. This requires analyzing and optimizing fully concrete fabrication plans, since considering operations in isolation does not capture the global sharing benefits of a particular ordering of operations. Greedy algorithms are also insufficient for extracting efficient schedules from the E-graph due to linearity: not every pair of statements can be re-grouped. Thus to measure f_t, HELM perform another optimization step that schedules cuts efficiently using a vertex-collapsing-based optimization.

Graph algorithm for measuring time Time can be minimized in two ways: (1) By setup elimination: when two instructions in a program use the same setup, they can be reordered so that a single setup can be used for both instructions, as long as linearity is not violated. (2) By stacking parts together: when two instructions use the same setup *and* also use two separate pieces of lumber that do not depend on each other, the pieces can be stacked together. HELM uses a new graph algorithm to minimize manufacturing time using the above techniques.

For a set of sub-programs, this algorithm builds a dependency graph $\mathcal{G}$ by looking at the input arguments and returned values of each statement. $\mathcal{G}$ is a directed graph and may have multiple connected components, which correspond to different programs. $\mathcal{G}$ corresponds to a valid program only if it has no cycles because a cycle indicates a violation of linearity. If two nodes (a, b) have the same setups, the algorithm *collapses* them and checks that no cycle is introduced in the graph (Figure 5.9). Interestingly, there are two cases that may arise after

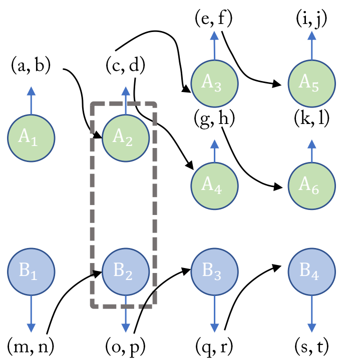


Figure 5.9: Vertex-collapsing algorithm to measure time. Each sub-program is a dependency graph and can be represented by a directed graph. Assume two non-adjacent vertices A_2 and B_2 have the same setups, they can be stacked and cut at the same time if there is no cycle after collapsing.

nodes are collapsed: if (a, b) are adjacent (i.e., there is a directed edge either from a to b or from b to a), collapsing will result in setup elimination; if (a, b) are not adjacent, these two statements can be performed at the same time (for example, stacking them on a chopsaw or parallelizing cross multiple workers). HELM's optimizer uses this graph algorithm to find a program that minimizes time by minimizing the number of setups while respecting *linearity*. This also reorders the sequence of instructions in a program to put two instructions which partially share setups close to each other.

Multi-Objective Optimization For many simplification tasks Panchekha et al. [2015], greedy approaches can be used to extract a program from an E-graph where it is traversed bottom-up and the best enode from each eclass is chosen. This approach does not work for optimizing manufacturing time since it is not an additive metric. Previous work on E-graphs have used constraint solvers Joshi et al. [2002] to extract programs non-greedily, but those approaches are expensive when extracting multiple programs. Further, the three objectives used for E-graph extraction may be conflicting. For example, using more stock would allow simultaneous scheduling of cuts that use the same setups but can increase the cost of lumber. The HELM prototype addresses these problems by using a genetic algorithm for multi-objective optimization.

In multi-objective optimization, genetic algorithms are one of the most common approaches, having been successfully adopted in many fields Zhang and Xing [2017]. HELM uses the NSGA-II Deb et al. [2002] method in HELM's implementation. The NSGA-II algorithm can improve the fitness of a set of candidate solutions to a Pareto front bounded by a multi-objective function. It is an evolutionary process that has selection, mutation, and crossover. The population is classified into a hierarchy of subgroups by diversity metrics for selection. HELM encodes each individual as a tree $\mathcal{T}_i$ which is a subset of the E-graph. A full program can be recovered by traversing $\mathcal{T}_i$ from top to bottom. Since HELM adopts the tree representation, it is difficult to directly use off-the-shelf methods of crossover and mutation. To solve this problem, HELM uses new mutation and crossover operations based on equivalence relations encoded in the E-graph.

In mutation, the algorithm traverses all eclasses in $\mathcal{T}_i$ and mutates their enodes if $rng < p_m$, where rng is the random number generator that uniformly produces a probability in $[0, 1]$ and p_m is the probability of mutation. HELM's algorithm can also mutate an enode to represent it by its argument e-classes. It therefore also randomly expands the e-classes recursively since the leaf nodes of $\mathcal{T}_i$ must only be sub-programs.

In a crossover, the algorithm first randomly selects a pair of individuals $(\mathcal{T}_i, \mathcal{T}_j)$ where $\mathcal{T}_i$ and $\mathcal{T}_j$ have edges to the same e-classes. It switches each pair of same e-classes by exchanging their enodes if $rng < p_c$ where p_c is the probability of crossover.

5.4 Evaluation

This section evaluates HELM against the following criteria: expressiveness of HL-HELM, the quality of the compiler-generated fabrication instructions, how HELM can be used for end-to-end optimizations, and how designs can be physically realized by users following LL-HELM fabrication plans. HELM is implemented in C++ and tested it on a PC with Intel E5 2620 and 64GB RAM.

5.4.1 Expressiveness of HL-Helm

Figure 9 demonstrates HELM's expressiveness by showing examples of a wide range of valid designs made by three experienced woodworkers (with more than three years of experience) using HL-HELM who were trained to use the HELM IDE. The three carpentry experts generated the models in Figure 9 by using an iterative process, with their time split between conceptual exploration and design. These same experts created the physical models shown in Figure 5.1 and filled out a survey relating their experience with the tool and comparing it to conventional CAD systems. Feedback from the woodworkers indicates that while it is easier to produce arbitrary models in standard CAD systems, for carpentry items HELM was faster and more intuitive. This is because HELM allows the designer to keep the fabrication process in mind during the design process.

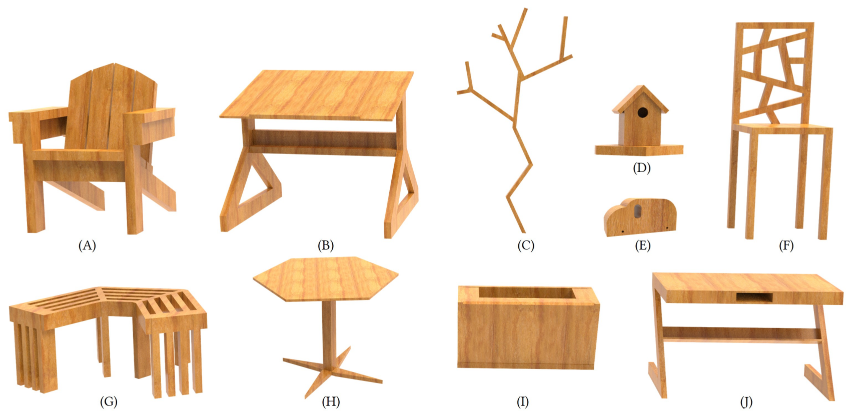


Figure 5.10: A gallery of carpentry designs modeled in HELM. The design time for each model is as follows. (A) Adirondack chair: 3:30 hr; (B) Drafting table: 2:16 hr; (C) Book case: 1:00 hr; D) Bird house: 1:38 hr; E) Toy car: 0:45 hr; F) Dining room chair: 1:20 hr; (G) Bench: 2:02 hr; (H) Coffee table: 0:56 hr; (I) Flower pot: 2:00 hr; (J) Z-table: 1:34 hr.

5.4.2 Optimized Fabrication Instructions

The HELM compiler ran on all of the designs shown in Figure 5.10, apart from 9.E because it is too simple. The results show that the compiler successfully optimizes all the designs sketched by the experts. To evaluate the quality of the optimized results, four recruited woodworking experts came up with fabrication instructions by hand and then computed the cost of their designs. Comparative results are shown in Figure 5.12

In eight out of the nine experiments, the system found solutions that Pareto-dominate the expert fabrication plan. This result validates that the proposed approach is not only a method that can help users with little expertise to find efficient fabrication plans but can also discover solutions that behave better than the ones designed by experienced woodworkers. This is most likely due to the high-dimensionality of the search space and the need to simultaneously consider multiple conflicting objectives, which makes manual exploration challenging. The added benefit of this approach, which is also shown in Figure 5.12, is that it returns not one but multiple solutions with different trade-offs, allowing engineers to pick the one that is more suitable for a specific application.

There was one model for which HELM did not find a solution that Pareto-dominates the expert. It is interesting to note that the expert solution also does not dominate the solutions of the system, but instead indicates a different trade-off that was not found by HELM. This result indicates that while HELM can find good solutions that outperform or match the experts, there are no guarantees that the solutions found are truly Pareto-optimal or that the full Pareto-front is found. Cut planning is a combinatorial problem and, while the use of E-graphs and the pruning strategy make the problem tractable, it is possible that (1) optimal designs are pruned and (2) the genetic algorithm does not discover the full front.

To further illustrate the different solutions and trade-offs, Figure 5.13 shows fabrication plans. In the bookcase, example 9.C, the expert grouped similar cuts on individual pieces of stock and cut them in order from left to right leading to high accuracy at the expense of material cost and time. Solution (A) Pareto-dominates the expert. In this example, HELM was able to significantly reduce the amount of material by using an optimal packing strategy while slightly reducing fabrication time and maintaining the same accuracy. In solution (B), HELM was able to significantly reduce fabrication time at the expense of higher precision error and material cost.

For the flower pot, example 9.I, the differences between the optimized results generated by HELM compared to the expert boils down to sheet packing. HELM made the same tool selection and material choices as the expert. HELM, however, was able to reorder and rearrange cuts to improve fabrication time and accuracy. On the other hand, the expert optimized the utility of unused stock, which was not accounted for in HELM's cost functions.

5.4.3 Design Optimization

Figure 5.11 shows how the parametric nature of HL-HELM is useful for design optimization driven by high-level, fabrication-independent performance metrics. In this example, four design parameters affect the geometry of the bookcase expressed in HL-HELM and the performance metric to be optimized is stability. The figure illustrates different configurations that can be achieved with different parameters and can be measured by the distance of the projection of the center of mass to the convex hull of the contact points. This allows us to optimize designs, which are independent of fabrication, while ensuring the designs before optimization and after optimization are manufacturable using carpentry processes.

5.4.4 Physical Realization from LL-Helm

To evaluate the practicality of HELM's optimizing compiler and languages, we provided the three experts with the LL-HELM code for three models (bookcase, birdhouse, and toy car) which are shown in Figure 5.1. HELM has a user interface (UI) to link the names of variables in LL-HELM programs with correct geometric details, and enable users to interactively visualize them in 3D space. They successfully manufactured all of these designs by following the LL-HELM code step-by-step.

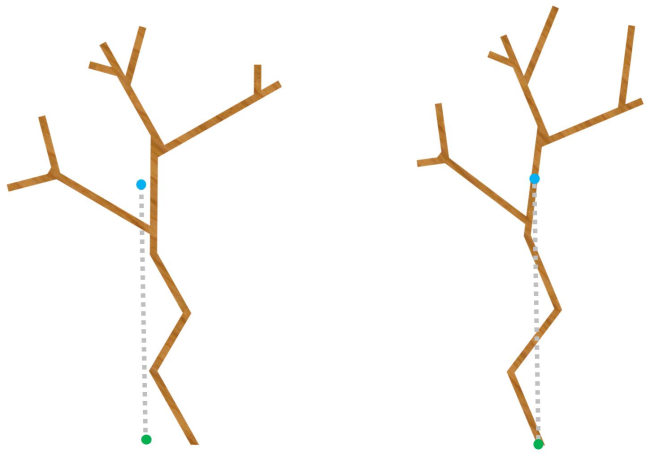


Figure 5.11: Example of design optimization that is enabled by the parametric nature of HL-HELM. The different shapes and corresponding design parameters and performance value (stability) are shown and the optimal one is highlighted (right).

5.4.5 Limitations and Future Work

Developing programming languages techniques for carpentry is a new direction and this work demonstrates the feasibility of this research avenue. However, there are still some limitations that require further investigation.

First, the current HELM prototype does not support shapes involving free-form geometry. Even though these designs can be manufactured using subtractive techniques, additive techniques are usually preferred. Since HELM currently does not support additive methods, such designs would require special treatment. Second, the compiler optimizations are not complete because they do not capture all possible equivalences. As a consequence, the compiler cannot perform optimizations that involve inserting additional cuts, or other temporary operations which may sometimes be useful. Third, the compiler first populates the E-graph with valid programs and then prunes it using heuristics to make the search more tractable. While efficient, this may not always return the optimal fabrication plan. However, as the results show, the instructions generated automatically by HELM can already match or improve upon plans manually developed by human experts. Fourth, the compiler currently uses a fixed-sized kerf for cuts which may make the dimensions of cuts inaccurate. Further, this work uses three simple metrics that were developed with the help of expert carpenters to evaluate fabrication plans. While these metrics can effectively demonstrate the capabilities of the multi-objective optimization pipeline, it would be interesting to investigate richer cost models, for example, to take into account stackability, correlated errors, and

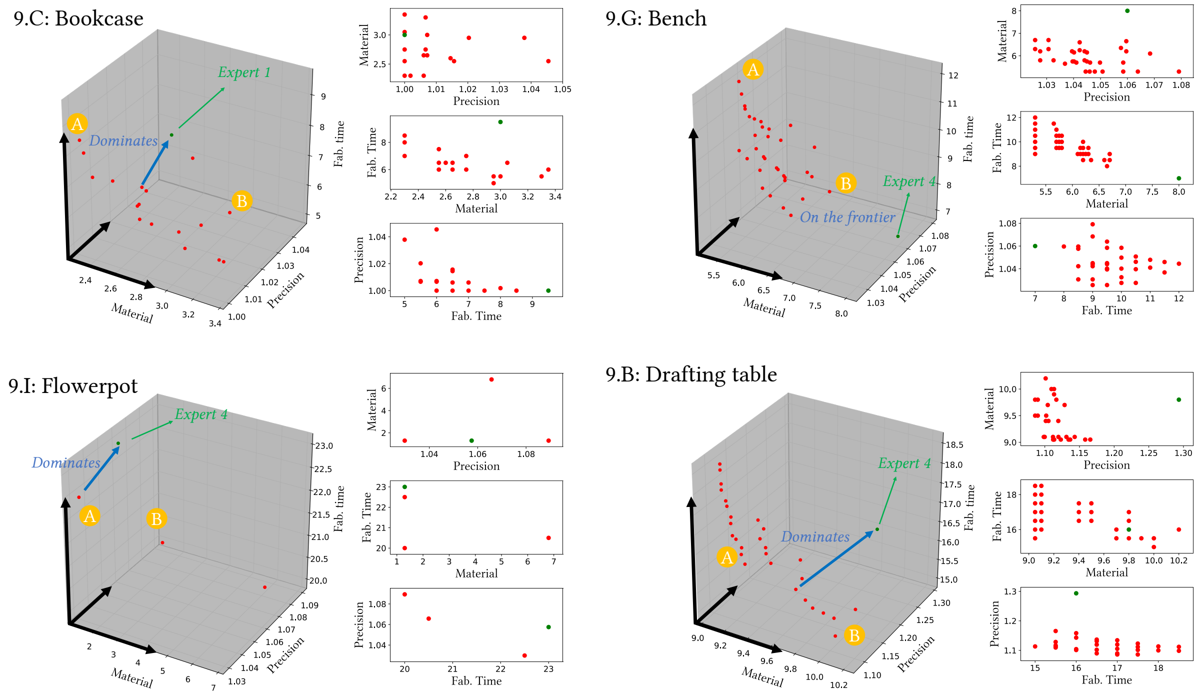


Figure 5.12: Results of the Pareto-fronts discovered by HELM (red) as compared to fabrication instructions hand-written by experts (green). For each example, the plots highlight a point in the discovered front that Pareto-dominates the expert fabrication plan. The 2D projections on the three main axes provide better visualization of the different trade-offs.

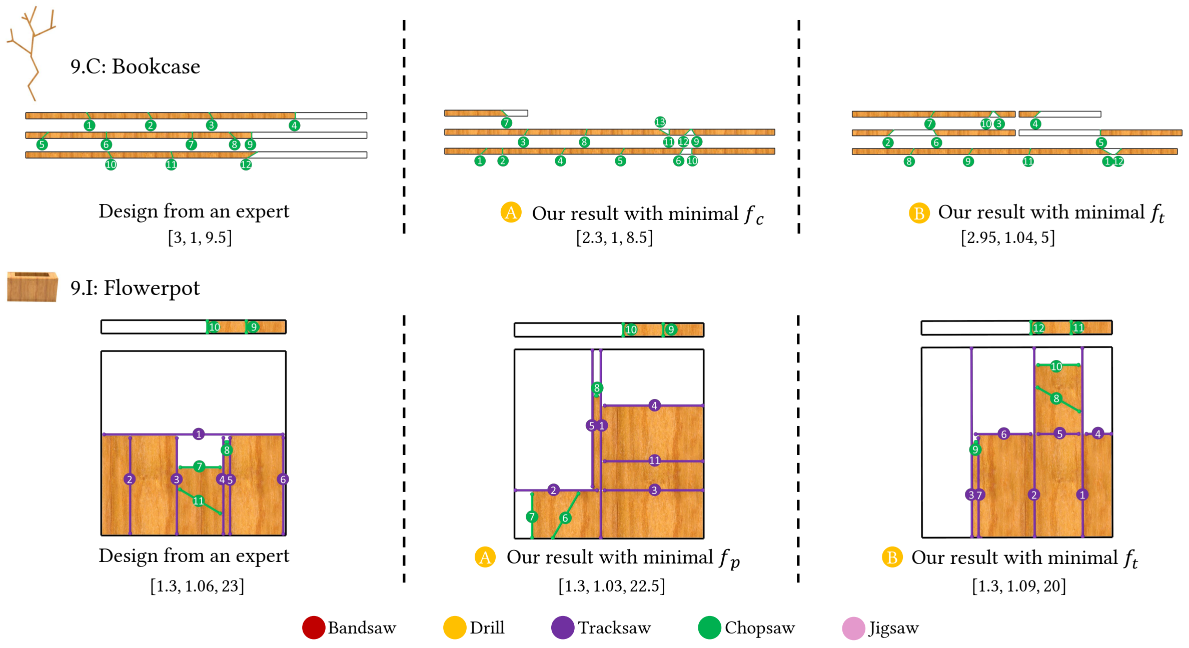


Figure 5.13: The visualization results of auto-generated LL-HELM programs and the fabrication plans hand-written by experts. Colors identify the process and numbers the order of cuts. The costs are shown in the order: f_c, f_p, f_t in square brackets below each figure.

grain-orientation into the precision metric. Finally, while the HELM compiler can optimize for precision, fabrication uncertainty can still affect the outcome. Figure 5.1 (toy car) shows an example of fabrication error due to which the shape of the car's window is different from the original design. Accounting for fabrication error during design is a hard problem even for single-process manufacturing because it depends on available processes. Typically this is handled by having designers predetermine error tolerances which are later verified. Decoupling design from fabrication can let us minimize error, but it still does not let us take the error into account at the design stage, for example, while performing finite element analysis.

5.5 Related Work

This section summarizes existing work in the area of carpentry, design, and optimizing compilers.

Fabrication-Oriented Design Design for fabrication is gaining attention in the computer graphics community Bickel et al. [2018]. Many newly proposed systems guide designers in searching the space of possible designs to both meet user specifications and ensure manufacturability. For example, several works optimize for design appearance Lan et al. [2013], Dong et al. [2010], deformation behavior Bickel et al. [2010], Ma et al. [2017], spinnability Bächer et al. [2017], or buoyancy Wang and Whiting [2016] while ensuring fabricability with an additive process. Other works focus on specific processes, such as interlocking quadrilateral elements Skouras et al. [2015], plush toys Mori and Igarashi [2007], LEGO Luo et al. [2015], or zippables Schüller et al. [2018]. These works all assume that a point in the design space completely determines the fabrication method. In contrast, HELM's approach decouples fabrication from the design specification — a design is created and optimized in HL-HELM, while the fabrication process is expressed and optimized in LL-HELM. This work presents a new compiler that converts designs to fabrication instructions and verifies that a design is manufacturable with the available processes. It optimizes instructions for multiple objectives like precision, time, and material cost and can thus generate different fabrication plans depending on which objective is being optimized for. Thus, in HELM, a single design can generate multiple diverse fabrication plans that can be optimized to meet differing requirements of the manufacturing facility.

Computer-Aided Manufacturing (CAM) Decades of CAM research focused on developing optimal fabrication plans for single specific fabrication processes, such as 5-axis milling Zhao et al. [2018], sheet-metal stretching Konaković et al. [2016], and 3D printing Dai et al. [2018], Alexa et al. [2017]. An important effort to create a multi-process representation was STEP-NC NC [2019], which abstracts away from machine-specific G-code operations to make tool-type-specific machining operations. These operations are interpretable or compilable on different hardware, allowing for inter-machine operations and closed-loop control at the tool-path level Brecher et al. [2006], Xu and Newman [2006]. Extensions to the STEP-NC framework have permitted its expansion from multi-axis milling to other metal-working processes, such as Electrical Discharge Machining (EDM) Sokolov et al. [2006], sheet metal forming Xie and Xu [2006], and 3D printing Um et al. [2017]. However, manual operations are still needed to convert a CAD file to a STEP-NC fabrication plan. More importantly, this task requires expert knowledge to select the fabrication process and verify that geometry is properly mapped to tooling operations. In contrast, HELM is designed on top of process-level abstractions; thus, it is compatible with many different processes. Its optimization framework chooses the process for each part automatically, with no human intervention. There are a few industrial CAM tools Čeli APS [2019], DDX [2019], Solutions [2019] that can be used for carpentry.

Programming Languages for Geometric Modeling and Fabrication Geometric modeling has a long-standing history of using domain-specific languages (DSLs) to describe a sequence of operations that construct geometry. These include early constructive solid geometry (CSG) approaches Laidlaw et al. [1986], modern CAD scripting languages FeatureScript [2019], and many procedural modeling systems Prusinkiewicz et al. [1996], Müller et al. [2006], Schwarz and Müller [2015]. These languages represent design as a process and can be used for optimization as well as inverse design Du et al. [2018], Nandi et al. [2018]. DSLs have also been used for describing fabrication for a single process, such as multi-material 3D printing Vidimče et al. [2013a] and HELM draws on these ideas to define DSLs for *both* design *and* multi-process fabrication. The languages are developed to allow a compiler to efficiently validate a design and optimize the fabrication process.

Design and Fabrication for Carpentry This work is also related to computational design approaches for carpentry and furniture. Fu et al. Fu et al. [2015] suggest using an interlocking structure and Song et al. Song et al. [2017] extend these ideas to designs that can be reconfigured. Umetani et al. Umetani et al. [2012] propose an interactive exploration tool for furniture design, where structural stability is evaluated at interactive rates. Lau et al. Lau et al. [2011] address the problem of converting a manually designed 3D model into parts and connectors, while Li et al. Li et al. [2015] target the foldability problem. These works propose fabrication-oriented design optimization, but they assume that a design uniquely determines a fabrication process. This work builds upon those ideas, defining languages where both design and fabrication can be optimized. In terms of fabrication optimization, packing problems are well studied for material saving. More recently, Koo et al. Koo et al. [2017] investigated this problem and proposed a guided tool for furniture design. The HELM optimizer considers the full fabrication processes, which involves not only material usage but also type and order of operations. This is enabled by treating fabrication as a program and defining a multi-objective optimization solution on top of a data structure (E-graphs) that can represent all equivalent programs.

5.6 Conclusions

This work presents HELM, a system for making high-level, abstract designs and automatically translating them to low-level, optimized fabrication plans. The key insight is that *fabrication plans are programs.* Based on this insight, this work presents new domain-specific languages for high-level (HL-HELM) designs and low-level (LL-HELM) plans, applied and extended compiler techniques to support multi-objective optimization, and demonstrated how these components simultaneously enable fabrication-aware design and optimization while shielding designers from fabrication details.

HELM's compiler from HL-HELM to LL-HELM automatically verifies manufacturability and provides novel optimizations to improve precision, and reduce material cost and manufacturing time. In order to efficiently represent all programs obtained by various optimizations that correspond to a particular fabrication plan, HELM leverages an E-graph data structure from traditional programming languages and compilers. This work demonstrates how to extract Pareto-optimal programs from the E-graph by performing multi-objective optimization.

HELM opens many exciting avenues for future work. The initial prototype provides a solid foundation for exploring interactions between subtractive processes, e.g., carpentry or machining, and additive processes, e.g., 3D printing or welding. Such interactions will enable even more flexibility in generating and optimizing low-level manufacturing plans and further empower designers to take full advantage of the ever-increasing diversity of available fabrication processes. It would also be interesting to exploit HL-HELM to create designs with for-loops, which can be directly unrolled in a pre-processing step, and investigate solutions for supporting recursions. Combining subtractive and additive processes will also enable *error recovery* when a user makes a mistake: for example, if a cut is made too short, a low-level "program patch" could be generated automatically using program synthesis techniques Gulwani et al. [2017] to build the botched part back up and enable resuming execution of the original plan rather than starting over from scratch. Cross-process fabrication plans could also be automatically scheduled for tighter integration between available processes, e.g., using a robotic arm to embed magnets in a part as it is 3D printed or using available processes to construct jigs that make otherwise-infeasible operations possible. Looking further ahead, as more robotic fabrication processes become available, exploring the potential to automatically schedule and optimize human-robot interaction in the fabrication setting will become essential in providing quality, efficiency, and safety in workshops of the future.

As manufacturing processes become increasingly sophisticated, and demand for customization increases, designers, fabricators, and even end-users will need more frameworks like HELM to support an increasingly automated and flexible idea-to-product pipeline.

Chapter 6

RULES INFERENCE FOR AND BY EQUALITY SATURATION

Chapter 4 and Chapter 5 showed two of the *first* uses equality saturation for program synthesis and optimization for CAD and carpentry. While prior work has used equality saturation for compiler optimizations Tate et al. [2009], we showed for the first time, how it can be used as a efficient program synthesizer as well. While Szalinski relied on both syntactic and semantic rewrites (custom solvers that introduced equivalent terms in the E-graph), the HELM compiler relied primarily on domain specific solvers to populate the E-graph. While working on Szalinski and HELM, we observed that one of the key challenges of using E-graphs and equality saturation for program synthesis is coming up with the rewrites rules themselves. This is a challenge not just of equality saturation, but any term-rewriting based tool. Typically these rewrites are written by hand which is tedious and error-prone. To address this challenge, this chapter investigates how rewrites rules used in equality saturation can be automatically synthesized. The applications of rewrite rule synthesis are more general than computational geometry and fabrication — this chapter therefore introduces a fundamental technique for rewrite rule inference and shows how it works on primitive domains like booleans, bitvectors, and rationals.

Several noteworthy projects have developed tool-specific techniques for checking or inferring rules Bansal and Aiken [2006], Menendez and Nagarakatte [2017], Joshi et al. [2002], Singh and Solar-Lezama [2016], but implementing a rewrite system still generally requires domain experts to first manually develop rulesets by trial and error. Such slow, ad hoc, and error-prone approaches hinder design space exploration for new domains and discourage updating existing systems.

This chapter proposes a simple, domain-general approach that uses equality saturation Tate et al. [2009], Willsey et al. [2021] as a rewrite system *on the domain of rewrite rules themselves* to quickly synthesize effective rulesets. In the past, tool-specific techniques to iteratively infer rewrite rules have implicitly adopted a common three-step approach, each constructing or maintaining a set:

1. Enumerate terms from the given domain to build the *term set* T.

2. Select candidate rules from $T \times T$ to build the *candidate set* C.

3. Filter C to select a sound set of useful rules to build the *rule set* R.

We identify and abstract this workflow to provide generic rule inference for user-specified domains.

Our key insight is that *what makes equality saturation successful in rewrite rule application is also useful for rule inference.* Equality saturation can simultaneously prove many pairs of terms equivalent with respect to a given ruleset. We built Ruler, a framework that uses equality saturation to shrink the set T of enumerated terms (lowering candidate *generation* cost) by merging terms equivalent under R, and to shrink the set C of candidate rules (lowering candidate *selection* cost) by removing rules derivable by R. Thus, Ruler uses the set R of rewrite rules to rewrite the next batch of candidate rewrite rules *even as R is being synthesized.*

6.1 Implementing Rewrite Systems

To build a rewrite system for a target domain, programmers must develop a set of rewrite rules and then use a rewrite engine to apply them, e.g., for optimization, synthesis, or verification. Ruler helps automate this process using e-graphs to compactly represent sets of terms and using equality saturation to filter and minimize candidate rules.

Developing Rewrite Rules Within a given domain D, a rewrite rule $\ell \leftrightsquigarrow r$ is a first-order formula consisting of a single equation, where ℓ and r are terms in D and all free variables are $\forall$-quantified. Rewrite rules must be sound: for any substitution σ of their free variables, ℓ and r must have the same semantics, i.e., $[\![\sigma(\ell)]\!]_D = [\![\sigma(r)]\!]_D$.

In many cases, rewrite rules must also be engineered to meet (meta)constraints of the rewrite engine responsible for applying them. For example, classic term rewriting approaches often require special considerations for cyclic (e.g., $(x + y) \rightsquigarrow (y + x)$) or expansive (e.g., $x \rightsquigarrow (x + 0)$) rules Baader and Nipkow [1998]. The choice of rules and their ordering can also affect the quality and performance of the resulting rewrite system Whitfield and Soffa [1997a]. Different ruleset variations may cause a rewrite system to be faster or slower and may be able to derive different sets of equivalences.

To a first approximation, smaller rulesets of more general, less redundant rules are desirable. Having fewer rules speeds up rule-based search since there are fewer patterns to repeatedly match against. Having more general, orthogonal rules also increases a rewrite system's "proving power" by expanding the set of equivalences derivable after a smaller number of rule applications. Avoiding redundancy also aids debugging, making it possible to diagnose a misbehaving rule-based search or optimization by eliminating one rule at a time. Automatic synthesis aims to generate rulesets that are sound and that include nonobvious, profitable rules that even domain experts may overlook for years.[1] Ideally, ruleset synthesis itself should also be fast; rapid rule inference can help programmers explore the design space for rewrite systems in new domains. It can also help with rewrite system maintenance since rulesets must be rechecked and potentially extended whenever any operator for a domain is added, removed, or updated, i.e., when the semantics for the domain evolves.

Applying Rules with Rewrite Engines Given a set of rewrite rules, a rewrite engine is tasked with either optimizing a given term into a "better" equivalent term (e.g., for peepholes McKeeman [1965] or superoptimization Massalin [1987]) or proving two given terms equivalent, i.e., solving the *word problem* Bezem et al. [2003].

Classic term rewriting systems destructively update terms as they are rewritten. This approach is generally fast, but complicates support for cyclic or expansive rules, and makes both rewriting performance and output quality dependent on fine-grained rule orderings. Past work has extensively investigated how to mitigate these challenges by scheduling rules Dershowitz [1982], Knuth and Bendix [1983], Borovanskỳ et al. [1998], Barendregt et al. [1987], special casing cyclic and expansive rules Dershowitz [1987], Bachmair et al. [2000], Eker [2003], Lucas [2001], and efficiently implementing rewrite rule-based search Visser [2001a,b], Clavel et al. [2007b], Kirchner [2015]. Many systems still rely on ad hoc rule orderings and heuristic mitigations developed through trial and error, though recent work Newcomb et al. [2020] has demonstrated how reduction orders Baader and Nipkow [1998] can be automatically synthesized and then used to effectively guide destructive term rewriting systems.

6.2 Ruler's Algorithm

This section describes Ruler, a new equality saturation-based rewrite rule synthesis technique. Like other rule synthesis approaches, Ruler iteratively performs three steps:

1. Enumerate terms into a set T.

2. Search $T \times T$ for a set of candidate equalities C.

3. Choose a useful, valid subset of C to add to the ruleset R.

[1] `https://github.com/halide/Halide/pull/3719`
`https://github.com/uwplse/herbie/issues/261`
`https://github.com/apache/tvm/pull/5974`
`https://github.com/Z3Prover/z3/issues/2575`
`https://github.com/Z3Prover/z3/pull/4663`

```
def ruler (iterations):
  $T$ = empty_egraph()
  $R$ = $\{\}$
  for $i \in [0, \texttt{iterations}]$:
    # add new terms directly to the e-graph representing $T$
    add_terms($T$, $i$)
    loop:
      # combine e-classes in the e-graph representing $T$ that $R$ proves equivalent
      run_rewrites($T$, $R$)
      $C$ = cvec_match($T$)
      if $C = \{\}$:
        break
      # choose_eqs only returns valid candidates by using 'is_valid' internally
      # and it filters out all invalid candidates from $C$
      $R$ = $R \cup$ choose_eqs($R$, $C$)
  return $R$
```

Figure 6.1: Ruler's Core Algorithm. The iterations parameter determines the maximum number of connectives in the terms Ruler will enumerate.

```
# $R$ is the accepted ruleset so far, $C$ is the candidate ruleset.
# Ruler's implementation of choose_eqs is based on a more flexible choose_eqs_n.
def choose_eqs($R$, $C$, $n = \infty$):
   for $step \in [100, 10, 1]$:
       if $step \leq n$:
           $C$ = choose_eqs_n($R$, $C$, $n$, $step$)
   return $C$

# $n$ is the number of rules to choose from $C$, and $step$ is a granularity parameter.
# A larger $step$ size allows you to eliminate redundant rules faster.
def choose_eqs_n($R$, $C$, $n$, $step$):
   # let $K$ be the list of "keepers" which we will return
   $K = []$
   while $C \neq \emptyset$:
      # pick the best $step$ candidate rules from $C$ according to a heuristic
      # that approximates rule "generality", including subsumption.
      $C_{\text{best}}$, $C$ = select($step$, $C$)

      # add the valid ones to $K$
      $K$ = $K \cup \{c \mid c \in C_{\text{best}}.$ is_valid($c$)$\}$

      # remember all the invalid candidates in a global variable $bad$;
      # Ruler uses this to prevent known-invalid candidates from entering $C$ again (not shown)
      $bad$ = $bad \cup \{c \mid c \in C_{\text{best}}. \neg$is_valid($c$)$\}$

      # stop if we have enough rules
      if $|K| \geq n$:
         return $K[0..n]$

      # try to prove terms remaining in $C$ equivalent using rules from $R \cup K$
      $C$ = shrink($R \cup K$, $C$)
   return $K$

def shrink($R$, $C$):
   $E$ = empty_egraph()
   for $(l \leftrightsquigarrow r) \in C$:
      $E$ = add_term($E$, $l$)
      $E$ = add_term($E$, $r$)
   $E$ = run_rewrites($E$, $R$)
   # return the extracted versions of rules from $C$,
   # leaving out anything that was proven equivalent
   return $\{$extract$(E, l) \leftrightsquigarrow$ extract$(E, r) \mid (l \leftrightsquigarrow r) \in C. \neg$equiv$(E, l, r)\}$
```

Figure 6.2: Ruler's implementation of choose_eqs, which aims to minimize the candidate set C by eliminating subsets that the remainder can derive.

Ruler's core insight is that E-graphs and equality saturation can help compactly represent the sets T, C, and R, leading to a faster synthesis procedure that produces smaller rulesets R with greater proving power (Section 6.3.1).

6.2.1 Ruler Overview

Figure 6.1 shows Ruler's core synthesis algorithm, which is parameterized by the following:

- The number of iterations to perform the search for (line 4);
- The language grammar, given in the form of a term enumerator (add_terms, line 6), which takes the number of variables or constants to enumerate over;
- The procedure for validating candidate rules, is_valid (called inside choose_eqs, Figure 6.2 line 20).

These parameters provide flexibility for supporting different domains, making Ruler a rule synthesis framework rather than a single one-size-fits-all tool.

Ruler uses an e-graph to compactly represent the set of terms T. In each iteration, Ruler first extends the set T with additional terms from the target language. Each term $t \in T$ is tagged with a *characteristic vector* (cvec) that stores the result of evaluating t given many different assignments of values to variables. After enumerating terms, Ruler uses equality saturation (run_rewrites) to merge terms in T that can be proved equivalent by the rewrite rules already discovered (in the set R). Next, Ruler computes a set C of candidate rules (cvec_match). It finds pairs $(t_1, t_2) \in T \times T$ where t_1 and t_2 are from distinct eclasses but have matching cvecs and thus are likely to be equivalent. Thanks to run_rewrites, no candidate in C should be derivable from R. However, C is often still large and contains many redundant or invalid candidate rules. Finally, Ruler's choose_eqs procedure picks a valid subset of C to add to R, ideally finding the smallest extension which can establish all equivalences implied by $R \cup C$. Ruler tests candidate rules for validity using a domain-specific is_valid function. This process is repeated until there are no more equivalences to learn between terms in T, at which point Ruler begins another iteration.

We detail each of these phases in the rest of this section.

6.2.2 Enumeration Modulo Equivalence

Rewrite rules encode equivalences between terms, often as relatively small "find and replace" patterns. Thus, a straightforward strategy for finding candidate rules is to find all equivalent pairs of terms up to some maximum size. Unfortunately, the set of terms up to a given size grows exponentially, making complete enumeration impractical for many languages. This challenge may be mitigated by biasing enumeration towards “interesting” terms, e.g. drawn from important workloads, or by avoiding bias and using sampling techniques to explore larger, more diverse terms. Ruler can support both domain-specific prioritization and random sampling via the `add_terms` function. While these heuristics can be very effective, they often risk missing profitable candidates for new classes of inputs or use cases.

Term space explosion can also be mitigated by partitioning terms into equivalence classes and only considering a single, canonical representative from each class. Similar to partial order reduction techniques in model checking Peled [1998] this can make otherwise intractable enumeration feasible. Ruler defaults to this complete enumeration strategy, using an E-graph to compactly represent T and equality saturation to keep T partitioned with respect to equivalences derivable from the rules in R even as they are being discovered.

Enumerating Terms in an E-graph E-graphs are designed to represent large sets of terms efficiently by exploiting sharing and equivalence. For sharing, E-graphs maintain deduplication and maximal reuse of subexpressions via hash-consing. If some term a is already represented in an E-graph, checking membership is constant time and adding it again has no effect. The first time $(a + a)$ is added, a new eclass is introduced with only a single enode, representing the $+$ with both operands pointing to a's eclass. If $((a+a)*(a+a))$ is then added, a new eclass is introduced with only a single enode, representing the $*$ with both operands pointing to $(a + a)$'s eclass. Thus as Ruler adds expressions to T, only the new parts of each added expression increase the size of T in memory.

On iteration i, calling `add_terms(`T`, i)` adds all (exponentially many) terms with i connectives to the E-graph. The first iteration calls `add_terms` with an empty E-graph to add terms with $i = 0$ connectives, thus specifying how many variables and which constants (if any) will be included in the search space. Since these terms are added to an E-graph, deduplication and sharing automatically provide efficient representation, but they do not, by themselves, provide an equivalence reduction to help avoid enumerating over many equivalent terms.

Compacting T using R Ruler's E-graph not only stores the set of terms T, but also an equivalence relation (more specifically, a congruence relation) over those terms. Since the children of an enode are eclasses, a single enode can represent exponentially many equivalent terms. Initially, the E-graph stores no equivalences, i.e., each term is in its own equivalence class.

As the algorithm proceeds, Ruler learns rules and places them in the set R of accepted rules.[2] At the beginning of its inner loop (line 9), Ruler performs equality saturation with

[2] While Figure 6.1 shows R starting empty, the user may instead initialize R with trusted axioms if they choose.

the rules from R. Equality saturation will unify classes of terms in the E-graph that can be proven equivalent with rules from R.

To ensure that run_rewrites only shrinks the term E-graph, Ruler performs this equality saturation on a copy of the E-graph, and then copies the newly learned equalities back to the original E-graph. This avoids polluting the E-graph with terms added during equality saturation.

Ruler's inner loop only terminates once there are no more rules to learn, so the next iteration (add_terms, line 6) *only enumerates over the canonical representatives* from the equivalence classes of terms with respect to R that have been represented up to that point. This compaction of the term space makes complete enumeration possible for non-trivial depths and makes Ruler much more efficient in finding a small set of powerful rules. Section 6.3.3 demonstrates how compaction of T is essential to Ruler's performance.

Since R may contain rules that use partial operators, Ruler's equality saturation implementation only merges eclasses whose cvecs agree in at least one non-null way (see the definition of *match* in Section 6.2.3). For example, consider that $x/x \leftrightsquigarrow 1 \in R$, and both $\frac{a+a}{a+a}$ and $\frac{a-a}{a-a} \in T$. The pattern x/x matches both terms, but equality saturation will not merge $\frac{a-a}{a-a}$ with 1, since $\frac{a-a}{a-a}$ is never defined. On the other hand, $\frac{a+a}{a+a}$ can merge with 1 since the cvecs match.

Prior work Nötzli et al. [2019] on rule inference applies multiple filtering passes to minimize rule sets *after* they are generated. These filters include subsumption order, variable ordering, filtering modulo alpha-renaming, and removing rules in the congruence closure of previously found rules. Ruler eliminates the need for such filtering using equality saturation on the E-graph representing T. Since enumeration takes place over eclasses in T, equivalent terms are "pre-filtered" automatically.

6.2.3 Candidate Rules

Given a set (or in Ruler's case, an E-graph) of terms T, rewrite rule synthesis searches $T \times T$ for pairs of equivalent terms that could potentially be a rule to add to R. The set of candidate rules is denoted C.

The naive procedure for producing candidate rules simply considers every distinct pair:

$$C = \{l \leftrightsquigarrow r \mid l, r \in T.\ l \neq r \land \forall \sigma.\ l[\sigma] = r[\sigma]\}$$

This is prohibitively expensive for two main reasons. First, it will produce many rules that are either in or can be proven by the existing ruleset R. In fact, the naive approach should always produce supersets of C from previous iterations; accepting a candidate rule from C into R would not prevent it from being generated in R in the following iteration. Second, most of the candidates will be unsound, and sending too many unsound candidates to choose_eqs burdens it unnecessarily, since it must search C for valid candidates by invoking the user-supplied is_valid procedure. Ruler's use of an E-graph to represent the term set T addresses both of the these inefficiencies with techniques called *canonical representation* and *characteristic vectors*.

Canonical Representation Consider a situation where $(x + y) \leftrightsquigarrow (y + x) \in R$ and both $(a + b)$ and $(b + a)$ are in T. When selecting terms from which to build a candidate rule, considering both $(a + b)$ and $(b + a)$ would be redundant; any rules derived from one could be derived from the other by composing it with commutativity of $+$. In some rewriting systems, this composition of rewrites cannot be achieved since cyclic rules like commutativity are not permitted. Equality saturation, however, handles and in many cases prefers such compositional rules, since it results in fewer rules to search over the E-graph.

To prevent generating candidate rules which are already derivable by the rules in R, Ruler only considers a single term from each eclass when building candidate rules. When searching for candidate rules, Ruler considers only term pairs (l, r) where $l \neq r$ and both are canonical representatives of eclasses in T. This ensures candidate rules cannot be derived from R; if they could have been, then l and r would have been in the same eclass after the call to run_rewrites.

Characteristic Vectors Canonical representation reduces C from $T \times T$ to $T' \times T'$ where T' is the set of canonical terms from T, but it does not prevent a full $O(n^2)$ search of $T' \times T'$ for valid candidate rules. Ruler employs a technique called *characteristic vectors* (cvecs) to prevent this quadratic search by only considering pairs that are *likely* valid. Ruler associates a characteristic vector v_i with each eclass i. The cvec is the result of evaluating t_i, the canonical term in eclass i, over a set of inputs that serves as a "fingerprint"[3] for the value of that eclass. Stated precisely, let σ_j for $j \in [1, m]$ be a predetermined family of m mappings from variables in T to concrete values, and let eval be the evaluator for the given language. The cvec for eclass i is:

$$v_i = [\mathsf{eval}(\sigma_j, t_i) \mid j \in [1, m]]$$

Ruler computes cvecs incrementally and without redundancy during enumeration using an *eclass analysis* Willsey et al. [2021] to associate a cvec with each eclass; let i be an eclass, t_i its canonical term, and v_i its cvec:

- when $t_i = n$ for a constant n, v_i is populated by copies of n;

[3] Section 6.5 discusses prior work Jia et al. [2019], Bansal and Aiken [2006] that uses "fingerprints" for synthesizing peephole optimizations and graph substitutions.

- when $t_i = f(t_{j_1}, \ldots, t_{j_n})$ for some n-ary operator f from the given language, v_i is computed by mapping f over the cvecs of the subterms: $v_i = \mathsf{map}(f, \mathsf{zip}(v_{j_1}, \ldots, v_{j_n}))$

- when $t_i = x$ for a variable x, v_i is populated by values from the target domain; choosing values to populate the cvecs of variables can be done randomly or with a domain-specific approach (Section 6.3.3 compares two approaches).

To support partial operators (e.g., division), cvecs may have a null value in them to indicate failure to evaluate. We say that cvecs *match* if their non-null values agree in every (and at least one) position, i.e., cvecs $[a_1, \ldots, a_n]$ and $[b_1, \ldots, b_n]$ match iff:

$$\forall i.\ a_i = b_i \lor a_i = \mathsf{null} \lor b_i = \mathsf{null} \qquad \text{and} \qquad \exists i.\ a_i = b_i \land a_i \neq \mathsf{null} \land b_i \neq \mathsf{null}$$

When eclasses in the E-graph representing T merge, they will have matching cvecs, because they have been proven equivalent by valid rules. Ruler aborts if cvecs of merging eclasses do not match; empirically, this helps avoid learning unsound rules even when is_valid is not sound (Section 6.3.3).

Section 6.3 and Section 6.3.2 discuss how cvecs are generated for different domains. Characteristic vectors serve as a filter for validity: if i, j are eclasses and v_i does not match v_j, (using the definition of match from Section 6.2.1) then $t_i \leftrightsquigarrow t_j$ is not valid. This allows Ruler to not consider those pairs when building C:

$$C = \{t_i \leftrightsquigarrow t_j \mid i, j \in \text{eclasses of } T.\ \mathsf{match}(v_i, v_j)\}$$

The cvec_match procedure (called at Figure 6.1, line 10) constructs C by grouping eclasses from T based on their cvecs and then taking pairs of canonical terms from each of those groups.

Validation The candidate set C contains rules that are likely, but not guaranteed, to be valid. The choose_eqs function (discussed in Section 6.2.4) must validate these before returning them by using the user-supplied is_valid function. The soundness of Ruler's output, i.e., whether every rule in R is valid, depends on the soundness of the provided is_valid procedure. Many rule synthesis implementations Singh and Solar-Lezama [2016], Jia et al. [2019] use SMT solvers to perform this validation. Ruler supports arbitrary validation procedures: small domains may use model checking, larger domains may use SMT, and undecidable domains may decide to give up a guarantee of soundness and use a sampling-based validation. Section 6.3.3 compares validation techniques for different domains.

6.2.4 Choosing Rules

After finding a set of candidate rules C, Ruler selects a valid subset of rules from C to add to the rule set R using the `choose_eqs` procedure (Figure 6.1, line 15). As long as `choose_eqs` returns a valid, non-empty subset of C, Ruler's inner loop will terminate: the number of eclasses with matching cvecs (i.e., the subset of T used to compute C) decreases in each iteration since R is repeatedly extended with rules that will cause new merges in `run_rewrites`. Ideally, `choose_eqs` quickly finds a minimal extension of R that enables deriving all equivalences implied by $R \cup C'$ where C' is the valid subset of C. `choose_eqs` also removes invalid candidates from C; if it returns the empty set (i.e., none of the candidates in C are valid), then Ruler's inner loop will terminate in the next iteration due to line 11 in Figure 6.1.

The candidate rules in C are not derivable by R, but many of the candidate rules may be able to derive each other, especially in the context of R. For example, the following candidate set is composed of three rules from the boolean domain, and any two can derive the third:

$$(\hat{}\ \mathsf{x}\ \mathsf{x}) = \mathsf{false} \qquad (\&\ \mathsf{x}\ \mathsf{false}) = \mathsf{false} \qquad (\&\ \mathsf{x}\ \mathsf{false}) = (\hat{}\ \mathsf{x}\ \mathsf{x})$$

An implementation of `choose_eqs` that only returns a single rule $c \in C$ avoids this issue, since adding c to R prevents those rules derivable by $R \cup \{c\}$ from being candidates in the next iteration of the inner loop. However, a single-rule implementation will be slow to learn rules, since it can only learn one at a time (Table 6.1 of our evaluation shows there are sometimes thousands of rules to learn). Additionally, such an implementation has to decide which rule to select, ideally picking the "strongest" rules first. For example, if $a, b \in C$ and $R \cup \{a\}$ can derive b but $R \cup \{b\}$ can not derive a, then selecting b before a would be a mistake, causing the algorithm to incur an additional loop.

Ruler's implementation of `choose_eqs`, shown in Figure 6.2, is parameterized by a value n with default of ∞. At $n = 1$, `choose_eqs` simply returns a single valid candidate from C. For higher n, `choose_eqs` attempts to return a list of up to n valid rules all at once. This can speed up Ruler by requiring fewer trips around its inner loop, but risks returning many rules that can derive each other. To mitigate this, `choose_eqs` tries to not choose rules that can derive each other. In its main loop (line 14), `choose_eqs` uses the `select` function to pick the *step* best rules from C according to a syntactic heuristic.[4] Ruler then validates the selected rules and adds them to a set K of "keeper" rules which it will ultimately return. It then employs the `shrink` procedure (line 34) to eliminate candidates from C that can be derived be $R \cup K$. This works similarly to `run_rewrites` in the Ruler algorithm, but `shrink` works over the remaining *candidate set* C instead of T.

[4] Ruler's syntactic heuristic prefers candidates with the following characteristics (lexicographically): more distinct variables, fewer constants, shorter larger side (between the two terms forming the candidate), shorter smaller side, and fewer distinct operators.

Ruler's `choose_eqs` invokes the inner `choose_eqs_n` procedure with increasingly small step sizes (*step* is defined on line 4). Larger step sizes allow `shrink` to quickly "trim down" C when it contains many candidates. However, a large step also means that `choose_eqs` may admit *step* rules into K at once, some of which may be able to prove each other. Decreasing the step size to 1 eliminates this issue.

Ruler uses $n = \infty$ by default for maximum performance, and Section 6.3.3 measures the effects of this choice on Ruler's performance and output.

6.2.5 Implementation

We implemented Ruler in Rust using the `egg` Willsey et al. [2021] E-graph library for equality saturation. By default Ruler uses Z3 De Moura and Bjørner [2008] for SMT-based validation, although using other validation backends is simple (Section 6.3.3).

Ruler's core consists of under 1,000 lines of code, allowing it to be simple, extensible, and generic over domains. Compared to the rewrite synthesis tool inside the CVC4 solver Barrett et al. [2011], Nötzli et al. [2019], Ruler is an order of magnitude smaller. Since Ruler's core algorithm does not rely on SMT, Ruler can learn rewrite rules over domains unsupported by SMT-LIB Barrett et al. [2016], or for alternative semantics for those domains [5] (Section 6.3.3).

In the following sections, we provide various evaluations of three representative domains on top of Ruler's core. Each domain highlights a verification back-end and cvec generation strategy Ruler supports:

- booleans and bitvector-4: these are small domains which Ruler can efficiently model check and generate sound rules by construction — the cvecs are complete.
- bitvector-32: demonstrates that Ruler supports SMT-based verification for large, non-uniform domains.
- rationals: demonstrates that random sampling is adequate for larger but continuous domains. This domain also showcases Ruler's support for partial operators like division.

The implementation of booleans, bitvectors, and rationals are in approximately 100, 400, and 300 lines, respectively.

6.3 Evaluation

In evaluating Ruler, we are interested in the following research questions:

[5]For example, the Halide Ragan-Kelley et al. [2013] tool uses division semantics where $x/0 = 0$; this is different from the SMT-LIB semantics, but it can easily be encoded using the `ite` operator.

$$\begin{array}{rcl}
\text{var} & ::= & \text{x} \mid \text{y} \mid \text{z} \\
\\
\text{expr} & ::= & \textit{literal} \mid \textit{var} \\
 & \mid & \sim \textit{expr} \\
 & \mid & \textit{expr}\ \&\ \textit{expr} \\
 & \mid & \textit{expr}\ \hat{}\ \textit{expr} \\
 & \mid & (\textit{expr} \mid \textit{expr})
\end{array}$$

(a) Boolean (bool) grammar.

$$\begin{array}{rcl}
\text{var} & ::= & \text{x} \mid \text{y} \mid \text{z} \\
\\
\text{expr} & ::= & \textit{literal} \mid \textit{var} \\
 & \mid & \sim \textit{expr} \mid -\ \textit{expr} \mid \\
 & \mid & \textit{expr} + \textit{expr} \mid \textit{expr} - \textit{expr} \mid \textit{expr} \times \textit{expr} \\
 & \mid & \textit{expr} \ll \textit{expr} \mid \textit{expr} \gg \textit{expr} \\
 & \mid & \textit{expr}\ \&\ \textit{expr} \mid (\textit{expr} \mid \textit{expr})
\end{array}$$

(b) Bitvector grammar used for both bv4 and bv32. $\sim$ is bitwise negation; unary $-$ is two's-complement. We use shift semantics where $a \ll b = a \gg b = 0$ when $b \geq width$.

Figure 6.3: The grammars used by both Ruler and CVC4 in our evaluation.

- *Performance.* Does Ruler synthesize rewrite rules quickly compared to other approaches?
- *Compactness.* Does Ruler synthesize small rulesets?
- *Derivability.* Do Ruler's rulesets derive the rules produced by other methods?
- *End-to-End.* How well do the synthesized rules perform compared to rules generated by experts in a real application?
- *Sensitivity Analysis.* How do the different components of Ruler's core algorithm affect performance and the synthesized rewrite rules?
- *Validation Analysis.* How do different validation strategies affect Ruler's output?

We first evaluate performance, compactness, and derivability (Section 6.3.1) by comparing Ruler against recent work Nötzli et al. [2019] that added rewrite rule synthesis to the CVC4 SMT solver Barrett et al. [2011]. We compose Ruler's rules with Herbie Panchekha et al. [2015] to show an end-to-end evaluation (Section 6.3.2). We then study the effects of different choices in Ruler's search algorithm (Section 6.3.3) and the different validation and cvec generation strategies.

Parameters		Ruler			CVC4			Ruler / CVC4	
Domain	# Conn	Time (s)	# Rules	Drv	Time (s)	# Rules	Drv	Time	Rules
bool	2	0.01	20	1	0.13	53	1	0.06	0.38
bool	3	0.06	28	1	0.82	293	1	0.07	0.10
bv4	2	0.14	49	1	4.47	135	0.98	0.03	0.36
bv4	3	4.30	272	1	372.26	1978	1	0.01	0.14
bv32	2	13.00	46	0.97	18.53	126	0.93	0.70	0.37
bv32	3	630.09	188	0.98	1199.53	1782	0.91	0.53	0.11
								0.04	0.17
								Harmonic Mean	

Table 6.1: Ruler tends to synthesize smaller, more powerful rulesets in less time than CVC4. The table shows synthesis results across domains, and number of variables in the grammar, and maximum term size (in number of connectives, "# Conn"). The domains are booleans, bitvector-4, and bitvector-32. For verification, Ruler uses model checking for booleans and bitvector-4 and Z3 for bitvector-32. The "Drv" column shows the fraction that tool's synthesized ruleset can derive of the other's ruleset; for example, the final row indicates that Ruler's 188 rules derived 98% of CVC4's 1,782 rules, and CVC's rules derived 91% of Ruler's. The final two columns show the ratios of synthesis times and ruleset sizes between the two tools.

6.3.1 Comparison with CVC4

Both Ruler and the CVC4 synthesizer are written in systems programming languages (Rust and C++, respectively), and both take a similar approach to synthesizing rewrite rules: enumerate terms, find valid candidates, select rules and repeat.

At the developers' suggestion, we used CVC4 version 1.8 with –sygus-rr-synth to synthesize rules. We enabled their rule filtering techniques (–sygus-rr-synth-filter-cong, –sygus-rr-synth-filter-match, –sygus-rr-synth-filter-order). We enabled their rule checker (–sygus-rr-synth-check) to verify all synthesized rules. Additionally, we also disabled use of any pre-existing rules from CVC4 to guide the rule synthesis (using –no-sygus-sym-break, –no-sygus-sym-break-dynamic).

Table 6.1 shows the results of the comparison. The following text discusses the results in detail, but, in short, Ruler synthesizes smaller rulesets in less time that have more proving power (Section 6.3.1).

Benchmark Suite

We compare Ruler against CVC4 for booleans, bitvector-4, and bitvector-32. Figure 6.3 shows the grammars. Both Ruler and CVC4 are parameterized by the domain (bool, bv4, or bv32), the number of distinct variables in the grammar, and the size of the synthesized term.[6] All benchmarks were single-threaded and run on an AMD 3900X 3.6GHz processor with 32GB of RAM. Both Ruler and CVC4 were given 3 variables and no constants to start the enumeration.

Derivability

A bigger ruleset is not necessarily a better ruleset. We designed Ruler to minimize ruleset size while not compromising on its capability to prove equalities. We define a metric called the *deriving ratio* to compare two rulesets. Ruleset A has deriving ratio p with respect to ruleset B if set A can derive a fraction p of the rules in B ($A \vDash b$ means rule set A can prove rule b):

$$p = |B_A|/|B| \quad \text{where} \quad B_A = \{b \mid b \in B.\ A \vDash b\}$$

If A and B have deriving ratio of 1 with respect to each other, then they can each derive all of the other's rules.

We use egg's equality saturation procedure to test derivability. To test whether $A \vDash b$ (where $b = b_l \leftrightsquigarrow b_r$) we add b_l and b_r to an empty E-graph, run equality saturation using A, and check to see if the eclasses of b_l and b_r merged. We run egg with 5 iterations of equality saturation. Since this style of proof is bidirectional (egg is trying to rewrite both sides at the same time), derivations of $b_l = b_r$ can be as long as 10 rules from A.

Bitvector and Boolean Implementation

Ruler supports different implementations of the is_valid procedure (Section 6.2.3) for different domains. When the domain is small enough, Ruler can use efficient model checking. For example, there are only $(2^4)^3 = 4096$ assignments of bitvector-4 to three variables. By using cvecs of that length to capture all possibilities, Ruler can guarantee that the cvec_match procedure returns only valid candidate rules, and is_valid need not perform any additional checking. Ruler uses model checking for booleans and bitvector-4, and it uses SMT-backed verification for bitvector-32.

[6] Size is measured in number of connectives, e.g., a has 0, $(a+b)$ has 1, and $(a+(b+c))$ has 2. In CVC4, this is set with the –sygus-abort-size flag.

Results

Table 6.1 shows the results of our comparison with CVC4's rewrite rule synthesis. On average (harmonic mean), Ruler produces 5.8× smaller rulesets 25× faster than CVC4. Ruler and CVC4's results can derive most of each other. On the harder benchmarks (in terms of synthesis times), Ruler's results have a higher derivability ratio; they can prove more of CVC4 rules than vice-versa.

6.3.2 End-To-End Evaluation

How good are Ruler's rewrite rules? Can they be used with existing rewrite-based tools with little additional effort? This section demonstrates how Ruler's output can be plugged directly into an existing rewrite-driven synthesis tool, Herbie.

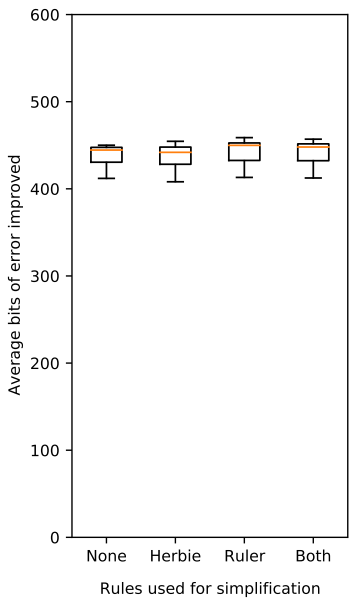
Herbie Panchekha et al. [2015] is a widely-used, open-source tool for automatically improving the accuracy of floating point expressions, with thousands of users and regular yearly releases. Herbie takes as input a numerical expression and returns a more accurate expression. It is implemented in Racket Racket [2021]. Herbie has separate phases for error localization (by sampling), series expansion, regime inference, and simplification, which work together to increase accuracy of numerical programs. The simplification phase uses algebraic rewrites to simplify mathematical expressions, thereby also enabling further accuracy improvements. These are applied using an equality saturation engine. In the past, the set of algebraic rewrites has been the cause of many bugs; of 8 open bugs at the time of this writing, six have been tagged "rules" by the developers Herbie [2021b]. Ruler was able to find rules that addressed one of these issues.

Experimental Setup

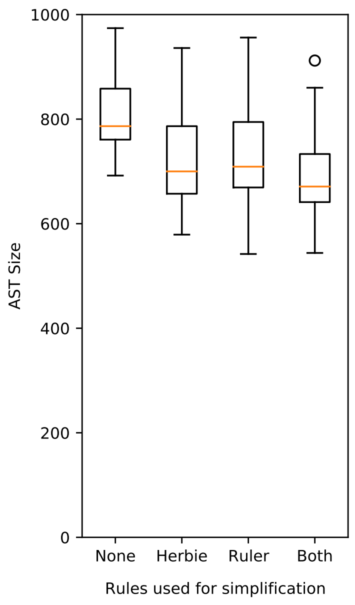
We implemented rationals in Ruler using rational and bigint libraries in Rust Rust [2021a,b]. We then synthesized rewrite rules over rational arithmetic and ran Herbie with the resulting ruleset.

The Herbie benchmark suite has 155 benchmarks; 55 of those are over rationals — i.e., all expressions in these benchmarks consist only of operators: $+, -, \times, /, abs, neg$. At the developers' suggestion, we filtered out 4 of the 55 benchmarks because they repeatedly timed out. We ran all our experiments on the remaining 51 benchmarks under four different configurations:

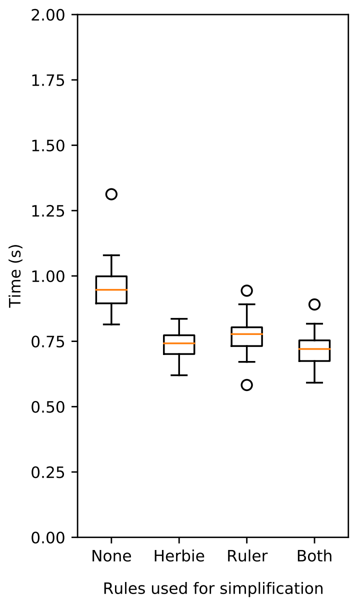
- None: remove all the rational rewrite rules from Herbie's simplification phase. Rational rules are those which consist only of rational operators and no others. Note that all other components of Herbie are left intact, including rules over rational operators



(a) Improvement in average error, Herbie's metric for measuring accuracy (higher is better).



(b) Size of the output AST produced by Herbie (lower is better).



(c) Herbie's running time (lower is better).

Figure 6.4: Comparing Herbie results between four configurations. Each boxplot represents the results from 30 seeds, where each data point is obtained by summing the value (average error, AST size, time) over all 51 benchmarks. The columns dictate what rational rules Herbie has access to: either none, its default rules, only Ruler's rules, or both. Herbie's rational rules reduce AST size and speed up simplification without reducing accuracy, and Ruler's rules perform similarly (with or without Herbie's rules).

combined with other operators, and rules entirely over other operators. None is the baseline.

- Herbie: no changes to Herbie, simply run it on the 51 benchmarks.
- Ruler: replace Herbie's rational rules with output of Ruler.
- Both: run Herbie with both Ruler's rational rules and the original Herbie rational rules.

Herbie has a default timeout of 180 seconds for each benchmark. It has a node limit of 5000 in its underlying equality saturation engine, i.e., it stops applying the simplification rules once the E-graph has 5000 enodes. We ran our experiments with three settings — (1) using the defaults, (2) increasing the timeout to 1000 seconds, and the node-limit to 10,000 to account for the addition of extra rules to Herbie's ruleset, and (3) decreasing the node limit to 2500 — we found that our results were stable and robust across all three settings. Figure 6.4 shows the results for the default setting. For all four configuration (None, Herbie, Ruler, Both), we ran Herbie for 30 seeds (because Herbie's error localization relies on random sampling).

We used Ruler to synthesize rational rules of depth 2 with 3 variables using random testing for validation ("rational" under Table 6.2).[7] Ruler learned 50 rules in 18 seconds, all of which were proven sound with an SMT post-pass. Four rules were expansive — i.e., rules like $(a \leftrightsquigarrow (a \times 1))$ whose LHS is only a variable. We removed these expansive rules from the ruleset as per the recommendation of the Herbie developers. Herbie's rules are uni-directional — we therefore expanded our rules for compatibility, ultimately leading to 76 uni-directional Ruler rules.

Discussion

The Herbie simplifier uses equality saturation to find smaller, equivalent programs. The simplifier itself does not directly improve accuracy; rather, it generates more candidates that are then used in the other accuracy improving components of Herbie. While ideally, Herbie would return a more accurate *and* smaller output, Herbie's ultimate goal is to find more accurate expressions, even if it sacrifices AST size. Herbie's original ruleset has been developed over the past 6 years by numerical methods experts to effectively accomplish this goal. Any change to these rules must therefore ensure that it does not make Herbie's result less accurate.

[7] For rationals, the add_terms implementation enumerates terms by depth rather than number connectives, since that matches the structure of Herbie's existing rules.

Figure 6.4 shows the results of running Herbie with rules synthesized by Ruler. Each box-plot corresponds to one of the four configurations. The baseline (None) and Herbie in Figure 6.4's accuracy and AST size plots highlight the significance of rational rewrites in Herbie — these expert-written rules reduce AST size without reducing accuracy. The plots for Ruler show that running Herbie with only Ruler's rational rules has almost the same effect on accuracy and AST size as Herbie's original, expert written ruleset. The plot for Both shows that running Herbie together with Ruler's rules further reduces AST size, still without affecting accuracy. The timing plots show that adding Ruler's rules to Herbie does not make it slower. The baseline timing is slower than the rest because removing all rational simplification rules causes Herbie's other components take much longer to find the same results.

In summary, Ruler's rational rewrite rules can be easily integrated into Herbie, and they perform as well as expert-written rules without incurring any additional overhead.

Derivability Herbie's original rational ruleset consisted of 52 rational rules. Ruler synthesized 76 uni-directional rational rules (50 bidirectional rules). We compared the two rulesets for proving power, by deriving each with the other using the approach described in Section 6.3.1. We found that Herbie's ruleset was able to derive 42 out of the 50 Ruler rules. It failed to derive the remaining 8. Ruler on the other hand, was able to derive all 52 rules from Herbie. We highlight two of the 8 Ruler rules that Herbie's ruleset failed to derive that concern multiplications interaction with absolute value: ($|a \times b| \leftrightsquigarrow |a| \times |b|$), and ($|a \times a| \leftrightsquigarrow a \times a$).

Fixing a Herbie Bug The above two rules found by Ruler helped the Herbie team address a GitHub issue Herbie [2021a]. In many cases, Herbie may generate large, complex outputs without improving accuracy, which makes the program unreadable and hard to debug. This is often due to lack of appropriate rules for expression simplification. The issue raised by a user (Herbie [2021a]) was in fact due to the missing rule ($|x| \times |x| \leftrightsquigarrow x \times x$). The two rules above, can together, accomplish the effect of this rule, thereby solving the issue. We submitted these two rules to the Herbie developers and they added them to their ruleset.

6.3.3 Sensitivity Studies: Analyzing Ruler Framework Parameters

As a rewrite rule inference *framework*, Ruler provides parameters that can be varied to support different user-specified domains. In particular, the "learning rate" parameter n for choose_eqs and the choice of validation method present potential tradeoffs in terms of synthesis time, ruleset quality, and soundness.

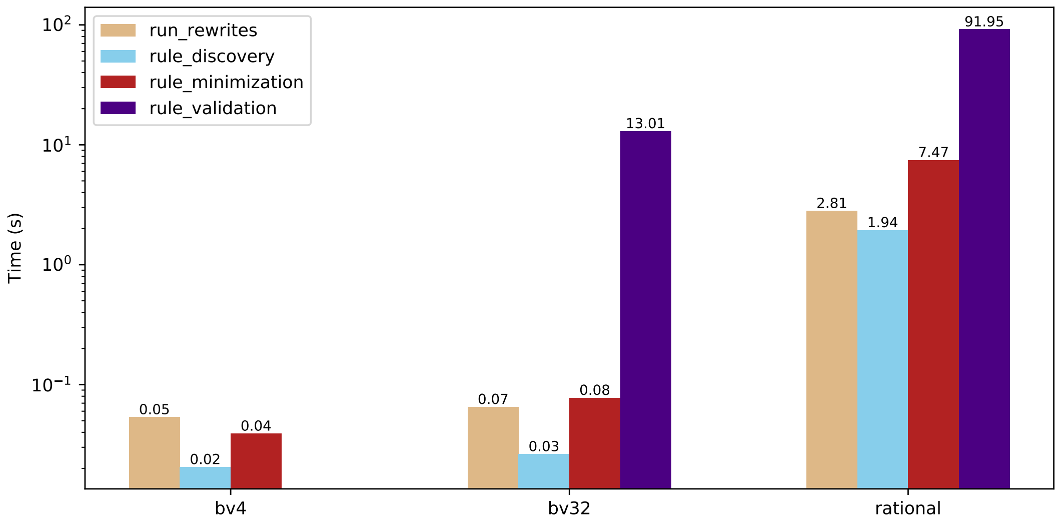


Figure 6.5: Search Profile. Time spent (in log scale) by Ruler in the various phases of its algorithm for bitvector-4, bitvector-32, and rationals. Most of the time is spent in rule validation (when applicable), then minimization. Notice that bitvector-4 does not have an explicit validation time because the rules are correct by construction — the cvecs are complete.

In this section we evaluate how varying these parameters affects three representative domain categories: (1) small domains like bitvector-4 where exhaustive model checking by complete cvecs is feasible, (2) large domains like bitvector-32 with non-uniform behavior that typically require constraint solving for validation, and (3) infinite domains like rationals with uniform behavior where fuzzing may be sufficient for validation. We additionally profile Ruler's search to see the impact of `run_rewrites` and compare cvec generation strategies.

Profiling Ruler Search, Varying `choose_eqs`, and Ablating `run_rewrites`

To help guide our study of Ruler's search, we first profiled how much time Ruler spent in each phase across our representative domains. Figure 6.5 plots the results using model checking for bitvector-4 and SMT for both bitvector-32 and rationals. `run_rewrites` (Figure 6.1) shows time compacting the term space with learned rules, `rule_discovery` (`cvec_match`, Figure 6.1) shows time discovering candidate rules, `rule_minimization` (`choose_eqs`, Figure 6.2) shows time selecting and minimizing rules, and `rule_validation` (`is_valid`, Figure 6.2) shows time validating rules. We ran each experiment in this section 10 times, for two iterations of Ruler, and plot mean values.[8]

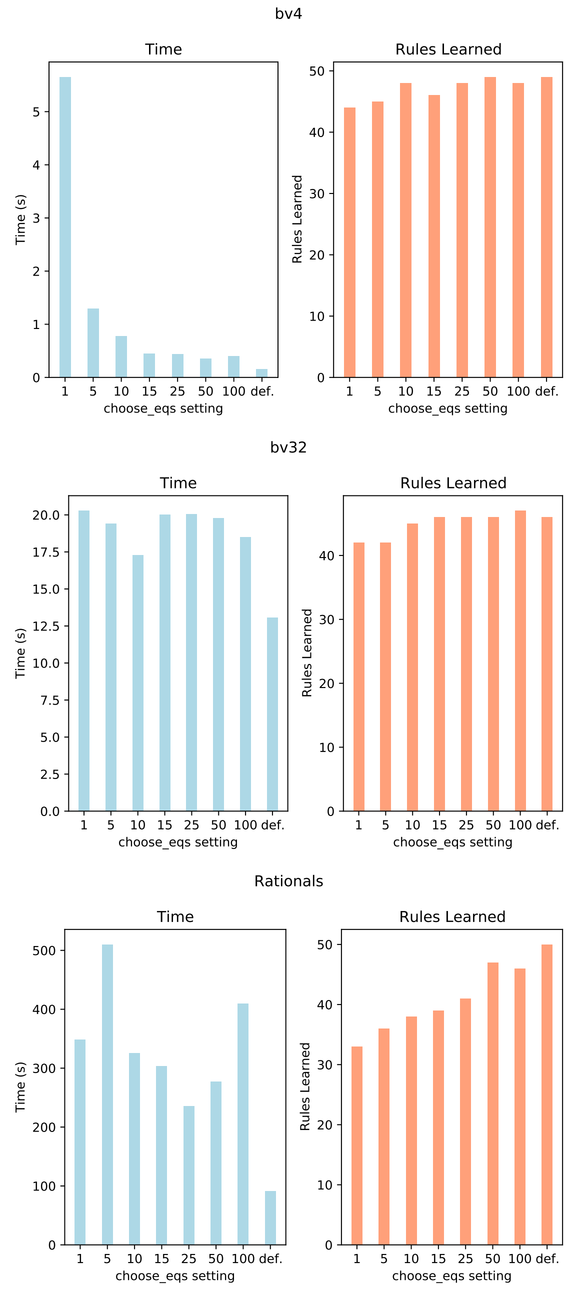
After validation (for applicable domains), Ruler spends most of its search time minimizing candidate rules. This is expected because `choose_eqs` minimizes the set of candidates C by invoking equality saturation and $|C|$ can reach roughly 10^6. Ruler uses a "learning rate" parameter n in `choose_eqs` to control how aggressively it tries to minimize rules (line 27, Figure 6.2). When $n = 1$, Ruler selects only a single "best" rule, requiring more rounds of selecting and shrinking candidate rules. By default, $n = \infty$, which causes Ruler to select a minimized version of that iteration's entire candidate ruleset.

Figure 6.6a shows how varying n affects overall search time and the resulting ruleset size: more aggressive minimization at $n = 1$ is slower but produces smaller rulesets relative to the default $n = \infty$. This is expected: since the set of candidate rules C is typically small compared to the entire term E-graph T, it is more efficient to iteratively shrink C than to repeatedly shrink T with only a few additional rules each time. These rulesets all generally had equivalent inter-derivability (Section 6.3), with the minimum ratio of 0.92 due to heuristics.

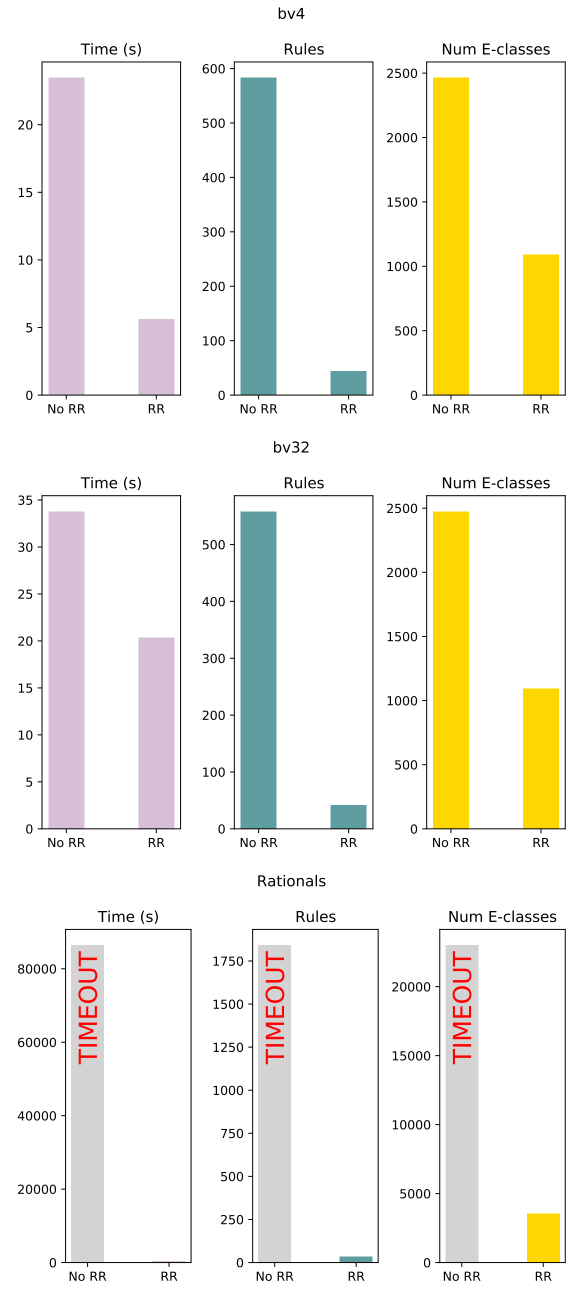
To understand how much shrinking the term E-graph T impacts search, we also compared running Ruler with and without `run_rewrites`. We set $n = 1$ for minimization as that setting relies the most on `run_rewrites`.[9] As Figure 6.6b shows, `run_rewrites` significantly improves

[8] For rationals, Ruler iterations correspond to expression depth, while for bitvector-4 and bitvector-32 it corresponds to number of connectives. Relative standard deviation across all experiments was always below 0.033.

[9] We also conducted this experiment with the default $n = \infty$ and found less pronounced results as



(a) Comparison of choose_eqs across values of n. def corresponds to $n = \infty$, Ruler's default configuration. Larger n values are generally faster but produce slightly more rules.



(b) For choose_eqs with $n = 1$, run_rewrites is essential for both speed and synthesizing small rulesets. "No RR" is Ruler without run_rewrites. For rationals, with "No RR" did not complete in 24 hours; with RR, it completed in 350 seconds.

Figure 6.6: Comparison of Ruler's performance across variations of the search algorithm.

search time and ruleset size while simultaneously requiring less space to store T.

Sensitivity Analysis for Validation Methods

Figure 6.5 shows that Ruler spends most of its search time in rule validation when using SMT. To investigate the relative performance and soundness of other validation methods, we compared various strategies for constructing cvecs and applying increasing levels of fuzzing during rule synthesis across our representative domains.

Table 6.2 shows that fuzzing can be used to synthesize surprisingly sound rulesets, with only a single configuration (bitvector-4, C = 343, random = 10) producing any unsound rules. This is because equality saturation tends to "amplify" the unsoundness of invalid rules. Similar to inadvertently proving `False` in an SMT solver, unsound rules in equality saturation quickly lead to attempted merges of distinct constants or eclasses with incompatible cvecs. Ruler detects such unsound merge attempts and exits immediately after reporting an error to the user along with the rule that triggered the bogus merge (which may or may not be the rule ultimately responsible for introducing unsoundness in the E-graph). These "equality saturation soundiness" crashes are indicated by "—" entries in Table 6.2. For the sole configuration that found unsound results without crashing, we reran the experiment with modestly increased resource limits and Ruler was able to detect the unsoundness without SMT. Despite this encouraging result, we emphasize that fuzzing alone cannot guarantee soundness in general.

For small domains like bitvector-4 with 3 variables, Ruler can employ exhaustive cvecs to quickly synthesize small, sound rulesets. For larger domains like bitvector-32, exhaustive cvecs are infeasible: even for 2 variables they would require cvecs of length $(2^{32})^2$. Larger domains like bitvector-32 with subtly non-uniform behavior especially require verification or good input sampling since, e.g., even if $x > 0$ it is possible to have $x * x = 0$.

To mitigate this challenge, Ruler allows cvecs to be randomly sampled (R), or populated by taking the Cartesian product (C) of sets of user-specified "interesting values".For example, the Ruler bitvector domains use values around 0, 1, MIN, MAX, rationals uses 0, 1, 2, -1, -2, $\frac{1}{2}, \ldots$.

We found that for uniform domains like rationals, using small cvecs with some random testing is sufficient for generating sound rules. For rationals, the low variability in the number of rules learned across the different configurations is an artifact of Ruler's minimization heuristics and cvec_matching — grouping eclasses based on cvecs (Section 6.2.3) to determine which terms are matched to become potential rewrite rule candidates.

For bitvector-32, we found that seeding cvecs with interesting constants was more effective

expected.

4-bit Bitvector

cvec		random: 0		random: 10		random: 100		random: 1000		SMT	
C	343	—	—	49/2	0.1s	49/-	0.1s	49/-	0.1s	49/-	1.1s
C	1331	49/-	0.1s	49/-	0.1s	49/-	0.1s	49/-	0.1s	49/-	1.0s
C	4096	49/-	0.2s	49/-	0.2s	49/-	0.2s	49/-	0.2s	49/-	1.1s
R	343	49/-	0.1s	49/-	0.1s	49/-	0.1s	49/-	0.1s	49/-	1.0s
R	1331	49/-	0.1s	49/-	0.1s	49/-	0.1s	49/-	0.1s	49/-	1.0s
R	4096	49/-	0.2s	49/-	0.2s	49/-	0.2s	49/-	0.2s	49/-	1.1s

32-bit Bitvector

C	343	—	—	—	—	—	—	—	—	46/-	13.6s
C	1331	46/-	0.1s	46/-	0.1s	46/-	0.1s	46/-	0.1s	47/-	13.2s
C	6859	46/-	0.2s	46/-	0.2s	46/-	0.2s	46/-	0.2s	47/-	13.3s
R	343	—	—	—	—	—	—	—	—	37/-	7.6s
R	1331	—	—	—	—	—	—	—	—	37/-	7.7s
R	6859	—	—	—	—	—	—	—	—	37/-	7.8s

Rational

cell format: #sound/#unsound time

C	27	—	—	50/-	16.8s	50/-	20.6s	47/-	55.3s	50/-	122.4s
C	125	46/-	33.3s	46/-	32.7s	46/-	32.8s	46/-	34.3s	46/-	38.2s
C	729	52/-	342.3s	52/-	341.6s	52/-	357.2s	49/-	341.3s	52/-	349.5s
R	27	50/-	16.4s	50/-	16.5s	46/-	16.0s	48/-	17.4s	50/-	19.5s
R	125	49/-	64.0s	49/-	65.2s	48/-	65.1s	48/-	64.2s	49/-	68.5s
R	729	50/-	413.6s	49/-	311.5s	49/-	308.4s	50/-	414.0s	49/-	316.7s

Table 6.2: Comparing cvec generation and validation strategies. First column shows cvec generation approach and length: C = Cartesian product of hand-picked values, R = randomly sampled values. Middle columns correspond to validation by random testing over varying number of samples, and the last column is for SMT based validation. We check the rules for soundness with a separate, SMT-based post-pass and report the number of sound/unsound rules and the synthesis time in seconds. A dashed cell indicates that Ruler detected unsoundness and crashed.

than random cvecs— due to the nonuniform nature of larger bitvectors, naively sampling random cvec values was insufficient for uncovering all the edge cases during rule validation, but when unsound rules were added, Ruler crashed due to "equality saturation soundiness" violations.

Handling Domain Updates

An important potential application of automatic rewrite rule synthesis is in helping programmers explore the design space for rewrite systems in new domains or during maintenance of existing systems to handle updates when a domain's semantics evolves. To simulate such a scenario, we took inspiration from the recent change in Halide's semantics[10] to define $x/0 = 0$, and similarly changed the implementation of division for the rationals domain to make the operator total.

Under the original rationals semantics where division by zero is undefined, Ruler learns 50 rules in roughly 123 seconds with SMT validation and 47 equivalent rules in roughly 21 seconds fuzzing 100 random values for validation (Table 6.2).

Making division total for the rationals domain in Ruler required changing a single line in the rationals interpreter. After this change, we used fuzzing with 100 random values to synthesize 47 rules in 18 seconds. Since fuzzing is potentially unsound, we also extended SMT support in our modified version of rationals with an additional 12 line change. We then synthesized 47 rules in 59 seconds using SMT validation and checked that both the new fuzzing-inferred and new SMT-inferred rulesets could each completely derive the other (the rulesets were identical).

Comparing the rulesets between the original and updated division semantics revealed expected differences, e.g., only the original rulesets contained $x/x \leftrightsquigarrow 1$ and only the updated rulesets contained $x/0 \leftrightsquigarrow 0$. Checking derivability between the old and new rulesets identified 5 additional rules that were incompatible between the semantics, shedding additional light on the consequences of the change to division semantics.

6.4 Limitations and Future Work

Like any synthesis tool, Ruler uses limits, caps, and heuristics to achieve practical performance. Of particular note are the heuristics in `choose_eqs` — `select` for scoring candidate rules, and the *step* size used to determine the number of rules to process at a time. `select` uses syntactic, size-based heuristics to approximate richer concepts like subsumption. As Section 6.3.3 showed, these heuristics can affect the size of the ruleset, though not significantly.

[10] https://github.com/halide/Halide/pull/4439

While Ruler's equality saturation approach eliminates α-equivalent rules, it does not eliminate α-equivalent terms from the enumeration set T. Doing so could significantly reduce the enumeration space and increase performance.

Ruler admits other implementations of term enumeration via `add_terms` (Section 6.2), but our default prototype implementation only explores complete enumeration. Stochastic enumeration, potentially based on characteristics of a workload, could further improve scalability. Ruler could also seed the initial term E-graph used for enumeration with expressions drawn from interesting workloads, e.g., benchmark suites or traces from users. This seeding could both speed up rule inference and improve the effectiveness of generated rules. Enumeration could also be limited to a semantically meaningful language subset; for example, it is possible to learn a subset of rules over reals by learning rules over rationals — the latter should be faster (as rationals have fewer operators), and the rational rules learned should remain sound when lifted to operating over reals.

Ruler already provides limited support for partial operators like `div` by allowing the interpreter to return a null value (Section 6.2). Ruler does not, however, infer the *conditions* that ensure that partial operators succeed. Furthermore, some rewrites are total but still depend on some condition being met: for example $|x| \leftrightsquigarrow x$ only when x is non-negative. An extension to Ruler could possibly to infer such side conditions based on "near cvec matches" where only a few entries differ between two eclasses's cvecs, building on prior work Menendez and Nagarakatte [2017] that infers preconditions for peephole optimizations.

6.5 Related Work

This section discusses prior work on rewrite rule synthesis. Most of the work in this area focuses on domain-specific rewrite synthesis tools, unlike Ruler, which is a domain-general framework for synthesizing rules, given a grammar and interpreter. We also focus on other framework-based approaches for rule synthesis and compare Ruler against them.

6.5.1 Rule synthesis for SMT solvers

Pre-processing for SMT solvers and related tools often involves term rewriting. Past work has attempted to automatically generate rules for such rewrites. Recent work from Nötzli et al. [2019] is the most relevant to Ruler. They present a partially-automated approach for enumerating rewrite rules for SMT solvers. We provide a comparison with Ruler in Section 6.3. The main commonality between our work and theirs is the use of sampling to detect new equivalences; this is similar to cvec matching in Ruler. Ruler's approach is unique in its use of equality saturation to shrink both the candidate rules and the set of enumerated terms. Nötzli et al. [2019] apply filtering strategies like subsumption, canonical variable ordering, and semantic equivalence; their term enumeration is based on Syntax-Guided Synthesis (SyGuS). Their tool can be configured to use an initial set of rules from cvc4 Barrett et al. [2011] to help guide their search for new rules. Ruler currently synthesizes generalized rules from scratch. Like Nötzli et al. [2019], Ruler generates rules that do not guarantee a reduction order, since it synthesizes rules like commutativity and associativity. To mitigate exponential blowups when using such rules, one approach is limiting the application of these rules Willsey [2021]. Chapter 4 has also demonstrated the use of inverse transformations to mitigate the AC-matching problem. Newcomb et al. [2020] have recently explored how to infer termination orders for non-terminating rules, which could be interesting future work for Ruler as well.

SWAPPER Singh and Solar-Lezama [2016] is a tool for automatically generating formula simplifiers using machine learning and constraint-based synthesis. SWAPPER finds candidate patterns for rules by applying machine learning on a corpus of formulae. Conditions in SWAPPER are inferred by first enumerating all possible expressions from a predicate language. Then, the right hand side of the rule is synthesized by fixing the predicate and using Sketch Solar-Lezama [2008b]. Romano and Engler [2013] infer reduction rules to simplify expressions before passing them to a solver, thereby reducing the number of queries sent to, and subsequently time spent, in the solver. The rules are generated by symbolic program evaluation, and validated using a theorem prover. Nadel [2014] used a combination of constant propagation and equivalence propagation to speed up bit-vector rewriting in various solvers. Several other papers Niemetz et al. [2018], Hansen [2012] propose algorithms and tools for automatically generating rules for bit-vectors. Newcomb et al. [2020] recently used program synthesis and formal verification to improve the rules in Halide. They focus only on integer rules and rely on mining specific workloads to identify candidates for rewrites. Preliminary experiments in Ruler indicate that supporting integers and even floats is achievable with random sampling or other validation approaches.

6.5.2 Instruction Selection and Graph Substitutions

Several tools have been proposed to automatically synthesize rewrite rules for instruction selectors. Buchwald et al. [2018] propose a hybrid approach called “iterative CEGIS”, combining enumeration with counter-example guided inductive synthesis (CEGIS) to speed up the synthesis of a rule library. As the authors describe in the paper, their tool does not support division, and they also do not infer any rules over floats, since the SMT solvers they rely on are not suitable for these domains. Ruler can be used to infer rules for domains not supported by SMT or that have different semantics because its core algorithm does not rely on SMT — it uses SMT for verifying rules for domains that are supported, but Section 6.3.3 shows that it is straightforward to use other validation techniques, or even change the semantics of the language and get a new set of rewrites. Dias and Ramsey [2010] proposed a heuristic search technique for automatic generation of instruction selectors given a machine description. Their work uses algebraic laws to rewrite expressions to expand the space of expressions computable in a machine.

TASO Jia et al. [2019] is a recent tool that automatically infers graph substitutions for optimizing graph-based deep learning computations. TASO automatically generates rules and verifies them using Z3 De Moura and Bjørner [2008]. To generate the candidates, TASO enumerates expressions from a grammar up to a certain depth and applies random testing to find equivalences, similar to Ruler. To verify the rules, TASO uses a set of axioms that express operator properties in first order logic. The axioms are used to prove that the generated rules for graph substitution are correct. TASO uses subsumption to eliminate rules that are direct special cases of other rules.

6.5.3 Theory Exploration

QuickSpec Claessen et al. [2010] is a tool that automatically infers specifications for Haskell programs from tests in the form of algebraic equations. Their approach is similar to Ruler in the sense that they too use tests to find potential equivalences between enumerated terms and filter out equations that are derivable from others.

Equations generated by QuickSpec have been used in an inductive theorem prover called HipSpec Claessen et al. [2013] to prove other properties about Haskell programs and also integrated with Isabelle/HOL Johansson et al. [2014]. TheSy Singher and Itzhaky [2021] uses a symbolic equivalence technique for theory exploration to generate valid axioms for algebraic data types (ADTs). TheSy also uses E-graphs (specifically the `egg` library) to find equivalences and filter out redundant axioms via term rewriting. Compared to other tools Johansson et al. [2014], TheSy typically found fewer, more powerful axioms. Using Ruler for theory exploration Johansson et al. [2010], especially for ADTs would be an interesting experiment in the future.

6.5.4 Peephole Optimizations

The Denali Joshi et al. [2002] superoptimizer first showed how to use E-graphs for optimizing programs by applying rewrite rules. Tate et al. [2009] first introduced equality saturation, generalizing some of the ideas in Denali to optimize programs with complex constructs like loops, and conditionals. Since then, multiple tools have used and further generalized equality saturation as a technique for program synthesis, optimization, and verification Nandi et al. [2020], Wang et al. [2020], Panchekha et al. [2015], Wu et al. [2019b], Premtoon et al. [2020], Stepp et al. [2011]. All these tools rely on the implicit assumption that the rewrite rules will be provided to the tool. These rulesets are typically written by a programmer and therefore can have errors or may not be complete. Several tools have automated peephole optimization generation Davidson and Fraser [2004], Bansal and Aiken [2006], Menendez and Nagarakatte [2017]. Bansal and Aiken [2006] presented a tool for automatically inferring peephole optimizations using superoptimization, using exhaustive enumeration to generate terms up to a certain depth, and leveraging canonicalization to reduce the search space. They use fingerprints to detect equivalences by grouping possibly equivalent terms together based on their evaluation on a few assignments. Grouping likely equivalent terms can eliminate many invalid candidate rules from even being generated in the first place. Ruler's use of cvecs is similar to the idea of fingerprints.

Several other papers Sharma et al. [2015], Schkufza et al. [2014] have extended and/or use STOKE for synthesizing superoptimizations. Alive-Infer Menendez and Nagarakatte [2017] is a tool for automatically generating pre-conditions for peephole optimizations for LLVM. Alive-Infer works in three stages: first, it generates positive and negative examples whose validity is checked using an SMT solver. It then uses a predicate enumeration technique to learn predicates, which are used as preconditions. Finally, it uses a boolean formula learner to generate a precondition. Menendez et al. [2016] also developed Alive-FP, a tool that automatically verifies peephole optimizations involving floating point computations. Recently, Lopes et al. [2021] published Alive2, which provides bounded, fully automatic translation validation, while handling undefined behaviour.

6.6 Conclusion

This chapter presented a new technique for automatic rewrite rule inference using equality saturation. We identified three key steps in rule inference and proposed Ruler, an equality saturation-based framework that can be used to infer rule-based optimizations for diverse domains.

Ruler's key insight is that equality saturation makes each of the three steps of rule inference more efficient. We implemented rule synthesis in Ruler for booleans, bitvectors, and rationals. We compared Ruler against a state-of-the-art rule inference tool in CVC4; Ruler generates significantly smaller rulesets much faster. We presented a case study showing how Ruler infers rules for complex domains like rationals. Our end-to-end results show that Ruler-synthesized rules can replace and even surpass those generated by domain experts over several years.

We hope that this work energizes the community around equality saturation and incites further exciting research into equality saturation for rewrite rule synthesis.

Chapter 7

FUTURE WORK AND CONCLUSIONS

Let's revisit the thesis statement from Chapter 1

Viewing the computational fabrication pipeline as a compiler enables the use of (1) modern PL theory to guide the systematic development and formal reasoning of this pipeline, and (2) reverse compilation and compiler optimization techniques to build novel program synthesizers and reliable compilers.

Chapter 2 validates (1), Chapters 3, 4, 5 validate (2). In Chapter 2, we saw how to formalize polygon meshes and CAD languages using denotational semantics. We developed a compiler from our functional programming language, λCAD, to mesh and provided a sketch of proof of correctness. By doing so, we laid the foundations for further work in the area of verified compilation for 3D geometry. This formal foundations and reasoning has several advantages including enabling the development of further tools to make computational geometry more usable, reliable, and accurate.

Equipped with the formal foundations, we then moved on to Chapters 3 and 4 and explored program synthesis and compiler optimization for geometric programs and fabrication. We developed new, general techniques to reverse engineer high-level λCAD programs from low-level triangle meshes, thereby making thousands of mesh models shared in online repositories Thingiverse [2019], GrabCAD [2019] significantly more editable / readable. Chapter 5 explored optimizing compilers for carpentry. We developed a compiler that generates a Pareto-front of optimal instruction sets for carpentered objects thereby reducing manufacturing time, material cost, and error.

The techniques in this thesis are applicable beyond fabrication and geometry and have already been used in several other completed Yang et al. [2021], VanHattum et al. [2021], Smith et al. [2021] and ongoing projects in other domains. Chapter 6 on automatic rewrite rule inference developed as a natural next step and has already found application in numerous domains. The main takeaways for readers of this thesis are that PL and compiler techniques can (and should!) be applied in "non-traditional" domains to develop better understanding and therefore, improved and more reliable systems for different domains. Over the years, I have encountered skeptics who have questioned this approach but every project I have done in this style has provided me with new insights, and led to novel ideas and new projects.

Finally, it is not just the other domains that benefit from PL ideas. As we have seen in this thesis, many novel PL ideas and tools with wide applications also emerge as a consequence.

7.1 Limitations

The work in this thesis did not account for every aspect of computational fabrication. For example, the DSLs we built do not account for factors like *tolerance* — designs made for manufacturing require tolerance so that they can be fabricated by different machines / techniques / companies without affecting the functionality. Similarly, there are fabrication-level properties that ultimately affect the final outcome (e.g., appearance, strength, stability) even though they are traditionally not represented at the design level. Our design DSLs could be extended to account for such properties allowing them to represent both design and manufacturing constraints using similar abstractions. Recent work from our group is already addressing these challenges for the domain of carpentry Zhao et al. [2021]. Finally, humans play a significant role in fabrication — many finer-grained decisions regarding precision, orientation of the model, offsets, etc., are critical in manufacturing and are currently not modeled in our DSLs. While Chapter 5 was a first attempt at addressing some of these challenges, I hope future work (see Section 7.2) will pursue this direction further by exploring how humans can be part of the synthesis and compilation loops.

7.2 Future Work

There are far too many ideas remaining to be explored in the space of PL/compilers for fabrication. Some directions that I hope will get explored in the future, either by me or others are summarized below. You can read more about several additional directions in our SNAPL'17 paper Nandi et al. [2017].

- *Parallel toolpaths.* The primary use of extrusion-based desktop 3D printing is *rapid* prototyping. Unfortunately, in practice, they are *slow*, often taking days to print some models. A simple yet effective solution would be to add more extruders that can simultaneously print parts of a large model. In fact, many 3D printers are equipped with multiple toolheads Flashforge [2020], Ultimaker [2020], LeapFrog [2020], BNC3D [2020] with varying configurations, and some are modular Snapmaker [2020], allowing the extruder to be swapped by a cutter or a plotter. However, these multi-head printers are intended for multi-material or multi-color printing. Some have been used for "ditto printing" Repetier [2020], where two identical objects are simultaneously printed. Little prior work on desktop printers has demonstrated the use of multiple toolheads to simultaneously print the *same* object. This is partly due to the fact that while there has been a lot of recent research on optimizing single head, the space of algorithms and design decisions for multiple heads has not been explored as much. It would be interesting to extend the use of multi-head desktop 3D printers to support parallel printing. Multi-head printers have varying configurations—the most widely available ones have two connected extruders at a fixed distance that move in unison, others may have three connected extruders, or extruders that are independent in one or more axes. An early prototype we built Carton et al. [2021] shows that even for the "simple" configuration with two extruders that are fixed distance apart, parallel printing can have high impact. The challenge is to automatically generate machine instructions (G-code) for a variety of different printer configurations. A potential approach would be for the parallel toolpath compiler to be parametrized over the printer configuration, i.e. it should generate machine instructions (G-code) specific to a printer and that does not violate the constraints accompanying the printer. The G-code syntax and semantics also depends on the printer (and its firmware) and therefore the compiler must generate instructions from the right language.

- *Combining Reincarnate and Szalinski in an E-graph.* Chapters 3 and 4 showed two separate decompilation "phases" for synthesizing high-level λCAD from low-level triangle meshes. The second phase used equality saturation to infer loops from the output of the first phase. Performing these two decompilations as two separate passes simplifies the problem and has shown promising results. A consequence of this separation, however, is that the second phase's search and therefore its output is constrained by the output of the first phase. Performing both phases of decompilation together in an E-graph may mitigate this problem by "keeping more programs around" and therefore enabling the synthesis tool to find more diverse solutions. This will involve substantial engineering — a geometry kernel similar to the one we developed for Reincarnate will have to be used for the "dynamic rewrites" to find an equivalent λCAD from a mesh.

- *Szalinski 2.0 aka Synthesizing generalized CAD from multiple flat CADs.* Szalinski's success raises the question of whether this technique can be extended to automatically generate a generalized program from multiple traces. Szalinski can be viewed as a tool for finding a program from a single trace. What can it accomplish given multiple traces? Would it be possible to learn library functions or infer a general parametrized implementation of a program such that each trace is a concrete instantiation of that function? How would it compare against other existing techniques for inferring templates Ellis et al. [2020], Kakarla et al. [2020]?

- *Software engineering practices for CAD/CAM.* While implementing λCAD, Reincarnate, Szalinski, and HELM, we made several design decisions that were aimed at making the code easy to test, robust to rounding errors, and easy to maintain. The majority of our implementation is in OCaml. There are several core geometric components that are relied upon by the rest of the tools. To account for such dependencies, for λCAD, we used a full functorial design and functorized all modules over a Geometry module. We also implemented several number system backends that were helpful for debugging and identifying rounding errors.

For testing, we wrote our own infrastructure and quickly realized that writing good tests for CAD/CAM is challenging—almost all geometric computations are over reals and it is tedious to manually write good tests that cover all the edge cases. Further, complex geometric functions (e.g. triangle-triangle intersection) rely on more fundamental geometric functions (e.g. line-line intersection) which means that a bug in one of the fundamental algorithms can lead to an error higher up in the pipeline. Floating point related rounding errors only makes this problem worse. Techniques like property-based testing Claessen and Hughes [2019] can make the development of geometry processing libraries less tedious and provide insights on how to specify geometric operations used in CAD/CAM frameworks.

- *Interactive theorem proving and type systems.* Formal verification using interactive theorem provers like Coq and Lean has been massively successful Wilcox et al. [2015], Mullen et al. [2016], Jang et al. [2012], Tatlock and Lerner [2010], Weitz et al. [2016], Sergey et al. [2018], Leroy [2006], Chlipala [2010]. Verified implementations of compilers, distributed systems, and web browsers now exist are becoming increasingly mainstream Ringer et al. [2019]. Little prior work focuses on formal guarantees for computational geometry kernels. This is challenging because geometric operations generally on reals which are approximated using floats that are hard to reason about; however a formally verified geometry kernel can improve the reliability of CAD/CAM pipelines and numerous other tools relying on geometry kernels. We hope that our work on λCAD (Chapter 2), together with other recent work on formalizing geometry Sherman et al. [2019] and verifying floating point optimizations in verified compilers Becker et al. [2019] leads to further investigation in this direction, perhaps using a more mechanized approach.

- *Beyond CSGs and Meshes.* All the tools in this thesis use CSGs and meshes as the core data structures for representing 3D solids. In many commercial CAD tools, B-reps, splines, and signed distance functions (SDFs) are the underlying representations. These data structures are more effective than meshes for representing curves like spheres and cylinders. In fact, meshes contain a subset of the information contained using these representations. Our semantics and compiler proof will not be directly applicable to B-reps or splines. It would be very interesting to consider how to extend/update our semantics to be able to reason about these more advanced data structures. Recent work Sherman et al. [2019] can be used as an inspiration for this. Further, our high-level synthesis algorithm supports decompilation of mesh-based models into a higher-level representations like CSG. Extending it to support decompilation of B-reps and SDFs to CSGs and perhaps other CAD representations (or translating between them!) would make Reincarnate and Szalinski applicable to many more CAD packages and models. This will be quite challenging — these non-mesh representations are complex and contain much more information which might be hard to navigate. On the other hand, using the additional information in these formats can also be used to inform the synthesis algorithm in clever ways thereby making it faster and more scalable.

- *Machine learning based tools.* Recently, several tools have used machine learning for 2D primitive detection Ellis et al. [2018], automatically finding CSG trees from 2D and 3D images Sharma et al. [2017], and generating programs with loops from datasets of 3D models Tian et al. [2019]. Their goals align with the goals of Reincarnate and Szalinski but their approach uses neural networks. It would be interesting to compare our tools with the ML-based tools. As a preliminary experiment, we have used CNNs for shape detection on a synthetic dataset that we generated and found that while primitive matching is easier to accomplish, complex shapes composed of two or more primitives can be difficult to detect.

- *Human-Computer Interaction (HCI).* Tools like Reincarnate and Szalinski have direct implications from a HCI perspective. These tools are intended to make the CAD/CAM experience easier for end-users by providing a low-barrier to entry. Their goal is to facilitate CAD modeling for everyone. It would be interesting to evaluate the extent to which these tools reach this goal. We have already performed some preliminary user studies to compare the benefits of Reincarnate with those of mesh editing tools like Blender Blender [2018] and Meshmixer Meshmixer [2018] and found that customizing pre-made models is much easier in Reincarnate but would like to perform a more rigorous analysis of their usefulness and usability. Developing tools like Sketch-N-Sketch Chugh et al. [2016] would also make the ideas in this thesis more usable for new users and casual makers.

BIBLIOGRAPHY

C. Alcock, N. Hudson, and P. K. Chilana. Barriers to using, customizing, and printing 3d designs on thingiverse. In *Proceedings of the 19th International Conference on Supporting Group Work*, GROUP '16, pages 195–199, New York, NY, USA, 2016. ACM. ISBN 978-1-4503-4276-6. doi: 10.1145/2957276.2957301. URL `http://doi.acm.org/10.1145/2957276.2957301`.

M. Alexa, K. Hildebrand, and S. Lefebvre. Optimal discrete slicing. *ACM Trans. Graph.*, 36 (1), Jan. 2017. ISSN 0730-0301. doi: 10.1145/2999536. URL `http://doi.acm.org/10.1145/2999536`.

R. Alur, L. D'Antoni, S. Gulwani, D. Kini, and M. Viswanathan. Automated grading of dfa constructions. In *Proceedings of the Twenty-Third International Joint Conference on Artificial Intelligence*, IJCAI '13, pages 1976–1982. AAAI Press, 2013. ISBN 978-1-57735-633-2. URL `http://dl.acm.org/citation.cfm?id=2540128.2540412`.

G. M. Amdahl, G. A. Blaauw, and F. P. Brooks. Architecture of the ibm system/360. *IBM J. Res. Dev.*, 8(2):87–101, Apr. 1964. ISSN 0018-8646. doi: 10.1147/rd.82.0087. URL `http://dx.doi.org/10.1147/rd.82.0087`.

M. Artin. *Algebra*. Pearson Prentice Hall, 2011. ISBN 9780132413770. URL `https://books.google.com/books?id=S6GSAgAAQBAJ`.

F. Baader and T. Nipkow. *Term Rewriting and All That*. Cambridge University Press, 1998. doi: 10.1017/CBO9781139172752.

M. Baba-Ali, D. Marcheix, and X. Skapin. A method to improve matching process by shape characteristics in parametric systems. *Computer-Aided Design and Applications*, 6(3): 341–350, 2009.

M. Bächer, B. Bickel, E. Whiting, and O. Sorkine-Hornung. Spin-it: Optimizing moment of inertia for spinnable objects. *Commun. ACM*, 60(8):92–99, July 2017. ISSN 0001-0782. doi: 10.1145/3068766. URL `http://doi.acm.org/10.1145/3068766`.

L. Bachmair, I. Ramakrishnan, A. Tiwari, and L. Vigneron. Congruence closure modulo associativity and commutativity. In *International Workshop on Frontiers of Combining Systems*, pages 245–259. Springer, 2000.

M. S. Baldwin, G. R. Hayes, O. L. Haimson, J. Mankoff, and S. E. Hudson. The tangible desktop: A multimodal approach to nonvisual computing. *ACM Trans. Access. Comput.*, 10(3):9:1–9:28, Aug. 2017. ISSN 1936-7228. doi: 10.1145/3075222. URL `http://doi.acm.org/10.1145/3075222`.

N. Banovic, R. L. Franz, K. N. Truong, J. Mankoff, and A. K. Dey. Uncovering information needs for independent spatial learning for users who are visually impaired. In *Proceedings of the 15th International ACM SIGACCESS Conference on Computers and Accessibility*, ASSETS '13, pages 24:1–24:8, New York, NY, USA, 2013. ACM. ISBN 978-1-4503-2405-2. doi: 10.1145/2513383.2513445. URL `http://doi.acm.org/10.1145/2513383.2513445`.

S. Bansal and A. Aiken. Automatic generation of peephole superoptimizers. In *Proceedings of the 12th International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS XII, page 394–403, New York, NY, USA, 2006. Association for Computing Machinery. ISBN 1595934510. doi: 10.1145/1168857.1168906. URL `https://doi.org/10.1145/1168857.1168906`.

S. Bansal and A. Aiken. Binary translation using peephole superoptimizers. In *Proceedings of the 8th USENIX Conference on Operating Systems Design and Implementation*, OSDI'08, pages 177–192, Berkeley, CA, USA, 2008. USENIX Association. URL `http://dl.acm.org/citation.cfm?id=1855741.1855754`.

H. P. Barendregt, M. C. van Eekelen, J. R. Glauert, J. R. Kennaway, M. J. Plasmeijer, and M. R. Sleep. Term graph rewriting. In *International conference on parallel architectures and languages Europe*, pages 141–158. Springer, 1987.

C. Barrett, C. L. Conway, M. Deters, L. Hadarean, D. Jovanović, T. King, A. Reynolds, and C. Tinelli. Cvc4. In *Proceedings of the 23rd International Conference on Computer Aided Verification*, CAV'11, page 171–177, Berlin, Heidelberg, 2011. Springer-Verlag. ISBN 9783642221095.

C. Barrett, P. Fontaine, and C. Tinelli. The Satisfiability Modulo Theories Library (SMT-LIB). `www.SMT-LIB.org`, 2016.

A. Bauer. Efficient computation with dedekind reals. 2008.

H. Becker, E. Darulova, M. O. Myreen, and Z. Tatlock. Icing: Supporting fast-math style optimizations in a verified compiler. In I. Dillig and S. Tasiran, editors, *Computer Aided Verification - 31st International Conference, CAV 2019, New York City, NY, USA, July 15-18, 2019, Proceedings, Part II*, volume 11562 of *Lecture Notes in Computer Science*, pages 155–173. Springer, 2019. doi: 10.1007/978-3-030-25543-5_10. URL `https://doi.org/10.1007/978-3-030-25543-5_10`.

W. Belkhir and A. Giorgetti. Lazy ac-pattern matching for rewriting. *Electronic Proceedings in Theoretical Computer Science*, 82:37–51, Apr 2012. ISSN 2075-2180. doi: 10.4204/eptcs.82.3. URL `http://dx.doi.org/10.4204/EPTCS.82.3`.

M. Bezem, J. Klop, E. Barendsen, R. de Vrijer, and Terese. *Term Rewriting Systems*. Cambridge Tracts in Theoretical Computer Science. Cambridge University Press, 2003. ISBN 9780521391153. URL `https://books.google.ca/books?id=oe3QKzhFEBAC`.

B. Bickel, M. Bächer, M. A. Otaduy, H. R. Lee, H. Pfister, M. Gross, and W. Matusik. Design and fabrication of materials with desired deformation behavior. *ACM Trans. Graph.*, 29 (4):63:1–63:10, July 2010. ISSN 0730-0301. doi: 10.1145/1778765.1778800. URL `http://doi.acm.org/10.1145/1778765.1778800`.

B. Bickel, P. Cignoni, L. Malomo, and N. Pietroni. State of the art on stylized fabrication. *Computer Graphics Forum*, 37(6):325–342, 2018. doi: 10.1111/cgf.13327.

R. Bidarra, P. J. Nyirenda, and W. F. Bronsvoort. A feature-based solution to the persistent naming problem. *Computer-Aided Design and Applications*, 2(1-4):517–526, 2005.

Blender. Blender, 2018. `https://www.blender.org/`.

BNC3D. BNC3D SIGMAX R19, 2020. `https://www.bcn3d.com/bcn3d-sigmax-r19/`.

J. Bornholt. Program Synthesis Explained, 2019. `https://www.cs.utexas.edu/~bornholt/post/synthesis-explained.html`.

P. Borovanskỳ, C. Kirchner, H. Kirchner, P.-E. Moreau, and C. Ringeissen. An overview of elan. *Electronic Notes in Theoretical Computer Science*, 15:55–70, 1998.

D. Brannan, D. Brannan, M. Esplen, and J. Gray. *Geometry*. Cambridge University Press, 1999. ISBN 9780521597876. URL `https://books.google.ca/books?id=q49lhAzXTFEC`.

C. Brecher, M. Vitr, and J. Wolf. Closed-loop capp/cam/cnc process chain based on step and step-nc inspection tasks. *International Journal of Computer Integrated Manufacturing*, 19 (6):570–580, 2006.

S. Buchwald, A. Fried, and S. Hack. Synthesizing an instruction selection rule library from semantic specifications. In *Proceedings of the 2018 International Symposium on Code Generation and Optimization*, CGO 2018, page 300–313, New York, NY, USA, 2018.

Association for Computing Machinery. ISBN 9781450356176. doi: 10.1145/3168821. URL `https://doi.org/10.1145/3168821`.

E. Burke, R. Hellier, G. Kendall, and G. Whitwell. A new bottom-left-fill heuristic algorithm for the two-dimensional irregular packing problem. *Operations Research*, 54(3):587–601, 2006. doi: 10.1287/opre.1060.0293. URL `https://doi.org/10.1287/opre.1060.0293`.

M. Carton, C. Nandi, A. Anderson, H. Zhao, E. Darulova, D. Grossman, J. Lipton, A. Schulz, and Z. Tatlock. A roadmap towards parallel printing for desktop 3d printers. In *Proceedings of the 32nd Annual International Solid Freeform Fabrication Symposium*, Virtual, 2021.

CGAL. CGAL, 2018. `https://www.cgal.org`.

CGAL. Marshall, 2019. `https://github.com/andrejbauer/marshall`.

X. A. Chen, S. Coros, J. Mankoff, and S. E. Hudson. Encore: 3d printed augmentation of everyday objects with printed-over, affixed and interlocked attachments. In *Special Interest Group on Computer Graphics and Interactive Techniques Conference, SIGGRAPH '15, Los Angeles, CA, USA, August 9-13, 2015, Posters Proceedings*, page 3:1, 2015. doi: 10.1145/2787626.2787650. URL `http://doi.acm.org/10.1145/2787626.2787650`.

X. A. Chen, J. Kim, J. Mankoff, T. Grossman, S. Coros, and S. E. Hudson. Reprise: A design tool for specifying, generating, and customizing 3d printable adaptations on everyday objects. In *Proceedings of the 29th Annual Symposium on User Interface Software and Technology, UIST 2016, Tokyo, Japan, October 16-19, 2016*, pages 29–39, 2016. doi: 10.1145/2984511.2984512. URL `http://doi.acm.org/10.1145/2984511.2984512`.

A. Chlipala. A verified compiler for an impure functional language. In *Proceedings of the 37th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '10, page 93–106, New York, NY, USA, 2010. Association for Computing Machinery. ISBN 9781605584799. doi: 10.1145/1706299.1706312. URL `https://doi.org/10.1145/1706299.1706312`.

R. Chugh, B. Hempel, M. Spradlin, and J. Albers. Programmatic and direct manipulation, together at last. In *Proceedings of the 37th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '16, pages 341–354, New York, NY, USA, 2016. ACM. ISBN 978-1-4503-4261-2. doi: 10.1145/2908080.2908103. URL `http://doi.acm.org/10.1145/2908080.2908103`.

K. Claessen and J. Hughes. Quick check, 2019. `http://www.cse.chalmers.se/~rjmh/QuickCheck/`.

K. Claessen, N. Smallbone, and J. Hughes. Quickspec: Guessing formal specifications using testing. In G. Fraser and A. Gargantini, editors, *Tests and Proofs*, pages 6–21, Berlin, Heidelberg, 2010. Springer Berlin Heidelberg. ISBN 978-3-642-13977-2.

K. Claessen, M. Johansson, D. Rosén, and N. Smallbone. Automating inductive proofs using theory exploration. In M. P. Bonacina, editor, *Automated Deduction – CADE-24*, pages 392–406, Berlin, Heidelberg, 2013. Springer Berlin Heidelberg. ISBN 978-3-642-38574-2.

M. Clavel, F. Durán, S. Eker, P. Lincoln, N. Martí-Oliet, J. Meseguer, and C. Talcott. *All About Maude - a High-performance Logical Framework: How to Specify, Program and Verify Systems in Rewriting Logic*. Springer-Verlag, Berlin, Heidelberg, 2007a. ISBN 3-540-71940-7, 978-3-540-71940-3.

M. Clavel, F. Durán, S. Eker, P. Lincoln, N. Martí-Oliet, J. Meseguer, and C. Talcott. *All About Maude-A High-Performance Logical Framework: How to Specify, Program, and Verify Systems in Rewriting Logic*, volume 4350. Springer, 2007b.

Customizable. Thingiverse customizable, 2019. `https://www.thingiverse.com/customizable`.

C. Dai, C. C. L. Wang, C. Wu, S. Lefebvre, G. Fang, and Y.-J. Liu. Support-free volume printing by multi-axis motion. *ACM Trans. Graph.*, 37(4):134:1–134:14, July 2018. ISSN 0730-0301. doi: 10.1145/3197517.3201342. URL `http://doi.acm.org/10.1145/3197517.3201342`.

J. W. Davidson and C. W. Fraser. Automatic generation of peephole optimizations. *SIGPLAN Not.*, 39(4):104–111, Apr. 2004. ISSN 0362-1340. doi: 10.1145/989393.989407. URL `https://doi.org/10.1145/989393.989407`.

DDX. EasyWOOD, CAD/CAM software for 5 axis woodworking, nesting true shape — DDX. `http://www.ddxgroup.com/en/software/easywood`, 2019.

M. de Berg. *Computational Geometry: Algorithms and Applications*. Springer, 1997. ISBN 9783540612704. URL `https://books.google.com/books?id=_vAxRFQcNA8C`.

L. De Moura and N. Bjørner. Z3: An efficient smt solver. In *Proceedings of the Theory and Practice of Software, 14th International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, TACAS'08/ETAPS'08, pages 337–340, Berlin, Heidelberg, 2008. Springer-Verlag. ISBN 3-540-78799-2, 978-3-540-78799-0. URL `http://dl.acm.org/citation.cfm?id=1792734.1792766`.

L. de Moura, S. Kong, J. Avigad, F. van Doorn, and J. von Raumer. The lean theorem prover (system description). In A. P. Felty and A. Middeldorp, editors, *Automated Deduction - CADE-25*, pages 378–388, Cham, 2015. Springer International Publishing. ISBN 978-3-319-21401-6.

K. Deb, A. Pratap, S. Agarwal, and T. Meyarivan. A fast and elitist multiobjective genetic algorithm: Nsga-ii. *IEEE Transactions on Evolutionary Computation*, 6(2):182–197, April 2002. ISSN 1089-778X. doi: 10.1109/4235.996017.

P. Delfs, M. T̈ows, and H.-J. Schmid. Optimized build orientation of additive manufactured parts for improved surface quality and build time. *Additive Manufacturing*, 12(Part B): 314 – 320, 2016. ISSN 2214-8604. doi: https://doi.org/10.1016/j.addma.2016.06.003. URL `http://www.sciencedirect.com/science/article/pii/S2214860416301142`. Special Issue on Modeling & Simulation for Additive Manufacturing.

J. Demmel and Y. Hida. Fast and accurate floating point summation with application to computational geometry. *Numerical Algorithms*, 37(1):101–112, Dec 2004. ISSN 1572-9265. doi: 10.1023/B:NUMA.0000049458.99541.38. URL `https://doi.org/10.1023/B:NUMA.0000049458.99541.38`.

N. Dershowitz. Orderings for term-rewriting systems. *Theoretical computer science*, 17(3): 279–301, 1982.

N. Dershowitz. Termination of rewriting. *Journal of symbolic computation*, 3(1-2):69–115, 1987.

D. Detlefs, G. Nelson, and J. B. Saxe. Simplify: A theorem prover for program checking. *J. ACM*, 52(3):365–473, May 2005. ISSN 0004-5411. doi: 10.1145/1066100.1066102. URL `http://doi.acm.org/10.1145/1066100.1066102`.

J. a. Dias and N. Ramsey. Automatically generating instruction selectors using declarative machine descriptions. In *Proceedings of the 37th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '10, page 403–416, New York, NY, USA, 2010. Association for Computing Machinery. ISBN 9781605584799. doi: 10.1145/1706299.1706346. URL `https://doi.org/10.1145/1706299.1706346`.

Y. Dong, J. Wang, F. Pellacini, X. Tong, and B. Guo. Fabricating spatially-varying subsurface scattering. *ACM Trans. Graph.*, 29(4):62:1–62:10, July 2010. ISSN 0730-0301. doi: 10.1145/1778765.1778799. URL `http://doi.acm.org/10.1145/1778765.1778799`.

T. Du, J. Priya Inala, Y. Pu, A. Spielberg, A. Schulz, D. Rus, A. Solar-Lezama, and W. Matusik. Inversecsg: automatic conversion of 3d models to csg trees. pages 1–16, 12 2018. doi: 10.1145/3272127.3275006.

J. Dumas, A. Lu, S. Lefebvre, J. Wu, and C. Dick. By-example synthesis of structurally sound patterns. *ACM Trans. Graph.*, 34(4):137:1–137:12, July 2015. ISSN 0730-0301. doi: 10.1145/2766984. URL `http://doi.acm.org/10.1145/2766984`.

A. Edalat and A. Lieutier. Foundation of a computable solid modelling. *Theoretical Computer Science*, 284(2):319 – 345, 2002. ISSN 0304-3975. doi: https://doi.org/10.1016/S0304-3975(01)00091-3. URL `http://www.sciencedirect.com/science/article/pii/S0304397501000913`.

S. Eker. Associative-commutative rewriting on large terms. In *International Conference on Rewriting Techniques and Applications*, pages 14–29. Springer, 2003.

K. Ellis, D. Ritchie, A. Solar-Lezama, and J. B. Tenenbaum. Learning to infer graphics programs from hand-drawn images, 2018. URL `https://openreview.net/forum?id=H1DJFybC-`.

K. Ellis, M. I. Nye, Y. Pu, F. Sosa, J. B. Tenenbaum, and A. Solar-Lezama. Write, execute, assess: Program synthesis with a repl. In *NeurIPS*, 2019.

K. Ellis, C. Wong, M. Nye, M. Sable-Meyer, L. Cary, L. Morales, L. Hewitt, A. Solar-Lezama, and J. B. Tenenbaum. Dreamcoder: Growing generalizable, interpretable knowledge with wake-sleep bayesian program learning, 2020.

FeatureScript. Weclome to FeatureScript, 2019. `https://cad.onshape.com/FsDoc/`.

Flashforge. Flashforge creator dual extruder 3d printer, 2020. `https://flashforge-usa.com/products/creator-dual-extrusion-3d-printer-certified-refurbished`.

M. Friedrich, P.-A. Fayolle, T. Gabor, and C. Linnhoff-Popien. Optimizing evolutionary csg tree extraction. In *Proceedings of the Genetic and Evolutionary Computation Conference*, GECCO '19, pages 1183–1191, New York, NY, USA, 2019. ACM. ISBN 978-1-4503-6111-8. doi: 10.1145/3321707.3321771. URL `http://doi.acm.org/10.1145/3321707.3321771`.

C.-W. Fu, P. Song, X. Yan, L. W. Yang, P. K. Jayaraman, and D. Cohen-Or. Computational interlocking furniture assembly. *ACM Trans. Graph.*, 34(4):91:1–91:11, July 2015. ISSN 0730-0301. doi: 10.1145/2766892. URL `http://doi.acm.org/10.1145/2766892`.

S. Galjaard, S. Hofman, and S. Ren. *New Opportunities to Optimize Structural Designs in Metal by Using Additive Manufacturing*, pages 79–93. Springer International Publishing, Cham, 2015. ISBN 978-3-319-11418-7. doi: 10.1007/978-3-319-11418-7_6. URL `http://dx.doi.org/10.1007/978-3-319-11418-7_6`.

Geomagic Design X. Geomagic Design X, 2018. `https://www.3dsystems.com/software/geomagic-design-x`.

D. Goldberg. What every computer scientist should know about floating-point arithmetic. *ACM Computing Surveys*, 23(1):5–48, Mar. 1991. URL `http://doi.acm.org/10.1145/103162.103163`.

GrabCAD. The largest online community of professional designers, engineers, manufacturers, and students, 2019. `https://grabcad.com/`.

T. Grimm. *User's Guide to Rapid Prototyping*. Society of Manufacturing Engineers, 2004. ISBN 9780872636972.

S. Gulwani, O. Polozov, and R. Singh. Program synthesis. *Foundations and Trends in Programming Languages*, 4(1-2):1–119, 2017. ISSN 2325-1107. doi: 10.1561/2500000010. URL `http://dx.doi.org/10.1561/2500000010`.

A. Guo, J. Kim, X. A. Chen, T. Yeh, S. E. Hudson, J. Mankoff, and J. P. Bigham. Facade: Auto-generating tactile interfaces to appliances. In *Proceedings of the 2017 CHI Conference on Human Factors in Computing Systems*, CHI '17, pages 5826–5838, New York, NY, USA, 2017. ACM. ISBN 978-1-4503-4655-9. doi: 10.1145/3025453.3025845. URL `http://doi.acm.org/10.1145/3025453.3025845`.

T. A. Hansen. *A constraint solver and its application to machine code test generation.* PhD thesis, Melbourne, Australia, 2012.

Herbie. Herbie can generate more-complex expressions that aren't more precise, 2021a. `https://github.com/uwplse/herbie/issues/261#issuecomment-680896733`.

Herbie. Optimize floating-point expressions for accuracy, 2021b. `https://github.com/uwplse/herbie/issues`.

C. M. Hoffmann, J. E. Hopcroft, and M. S. Karasick. Towards implementing robust geometric computations. In *Proceedings of the Fourth Annual Symposium on Computational Geometry*, SCG '88, pages 106–117, New York, NY, USA, 1988. ACM. ISBN 0-89791-270-5. doi: 10.1145/73393.73405. URL `http://doi.acm.org/10.1145/73393.73405`.

M. Hofmann, J. Burke, J. Pearlman, G. Fiedler, A. Hess, J. Schull, S. E. Hudson, and J. Mankoff. Clinical and maker perspectives on the design of assistive technology with rapid prototyping technologies. In *Proceedings of the 18th International ACM SIGACCESS Conference on Computers and Accessibility*, ASSETS '16, pages 251–256, New York, NY, USA, 2016a. ACM. ISBN 978-1-4503-4124-0. doi: 10.1145/2982142.2982181. URL `http://doi.acm.org/10.1145/2982142.2982181`.

M. Hofmann, J. Harris, S. E. Hudson, and J. Mankoff. Helping hands: Requirements for a prototyping methodology for upper-limb prosthetics users. In *Proceedings of the 2016 CHI Conference on Human Factors in Computing Systems*, CHI '16, pages 1769–1780, New York, NY, USA, 2016b. ACM. ISBN 978-1-4503-3362-7. doi: 10.1145/2858036.2858340. URL `http://doi.acm.org/10.1145/2858036.2858340`.

M. K. Hofmann. Making connections: Modular 3d printing for designing assistive attachments to prosthetic devices. In *Proceedings of the 17th International ACM SIGACCESS Conference on Computers and Accessibility*, ASSETS '15, pages 353–354, New York, NY, USA, 2015. ACM. ISBN 978-1-4503-3400-6. doi: 10.1145/2700648.2811323. URL `http://doi.acm.org/10.1145/2700648.2811323`.

E. Hopper and B. Turton. An empirical investigation of meta-heuristic and heuristic algorithms for a 2d packing problem. *European Journal of Operational Research*, 128(1):34 – 57, 2001. ISSN 0377-2217. doi: https://doi.org/10.1016/S0377-2217(99)00357-4. URL `http://www.sciencedirect.com/science/article/pii/S0377221799003574`.

K. Hormann and A. Agathos. The point in polygon problem for arbitrary polygons. *Comput. Geom. Theory Appl.*, 20(3):131–144, Nov. 2001. ISSN 0925-7721. doi: 10.1016/S0925-7721(01)00012-8. URL `http://dx.doi.org/10.1016/S0925-7721(01)00012-8`.

N. Hudson, C. Alcock, and P. K. Chilana. Understanding newcomers to 3d printing: Motivations, workflows, and barriers of casual makers. In *Proceedings of the 2016 CHI Conference on Human Factors in Computing Systems*, CHI '16, pages 384–396, New York, NY, USA, 2016. ACM. ISBN 978-1-4503-3362-7. doi: 10.1145/2858036.2858266. URL `http://doi.acm.org/10.1145/2858036.2858266`.

ImplicitCAD. Powerful, Open-Source, Programmatic CAD, 2019. `http://www.implicitcad.org/`.

D. Jang, Z. Tatlock, and S. Lerner. Establishing browser security guarantees through formal shim verification. In T. Kohno, editor, *Proceedings of the 21th USENIX Security Symposium, Bellevue, WA, USA, August 8-10, 2012*, pages 113–128. USENIX

Association, 2012. URL `https://www.usenix.org/conference/usenixsecurity12/technical-sessions/presentation/jang`.

S. Jha, S. Gulwani, S. A. Seshia, and A. Tiwari. Oracle-guided component-based program synthesis. In *Proceedings of the 32Nd ACM/IEEE International Conference on Software Engineering - Volume 1*, ICSE '10, pages 215–224, New York, NY, USA, 2010. ACM. ISBN 978-1-60558-719-6. doi: 10.1145/1806799.1806833. URL `http://doi.acm.org/10.1145/1806799.1806833`.

Z. Jia, O. Padon, J. Thomas, T. Warszawski, M. Zaharia, and A. Aiken. Taso: Optimizing deep learning computation with automatic generation of graph substitutions. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles*, SOSP '19, page 47–62, New York, NY, USA, 2019. Association for Computing Machinery. ISBN 9781450368735. doi: 10.1145/3341301.3359630. URL `https://doi.org/10.1145/3341301.3359630`.

M. Johansson, L. Dixon, and A. Bundy. Conjecture synthesis for inductive theories. *Journal of Automated Reasoning*, 47:251–289, 2010.

M. Johansson, D. Rosén, N. Smallbone, and K. Claessen. Hipster: Integrating theory exploration in a proof assistant. In S. M. Watt, J. H. Davenport, A. P. Sexton, P. Sojka, and J. Urban, editors, *Intelligent Computer Mathematics*, pages 108–122, Cham, 2014. Springer International Publishing. ISBN 978-3-319-08434-3.

R. Joshi, G. Nelson, and K. Randall. Denali: A goal-directed superoptimizer. *SIGPLAN Not.*, 37(5):304–314, May 2002. ISSN 0362-1340. doi: 10.1145/543552.512566. URL `http://doi.acm.org/10.1145/543552.512566`.

S. K. R. Kakarla, A. Tang, R. Beckett, K. Jayaraman, T. Millstein, Y. Tamir, and G. Varghese. Finding network misconfigurations by automatic template inference. In *17th USENIX Symposium on Networked Systems Design and Implementation (NSDI 20)*, pages 999–1013, Santa Clara, CA, Feb. 2020. USENIX Association. ISBN 978-1-939133-13-7. URL `https://www.usenix.org/conference/nsdi20/presentation/kakarla`.

J. Kim, A. Guo, T. Yeh, S. E. Hudson, and J. Mankoff. Understanding uncertainty in measurement and accommodating its impact in 3d modeling and printing. In *Proceedings of the 2017 Conference on Designing Interactive Systems, DIS '17, Edinburgh, United Kingdom, June 10-14, 2017*, pages 1067–1078, 2017. doi: 10.1145/3064663.3064690. URL `http://doi.acm.org/10.1145/3064663.3064690`.

P. Kim. *Rigid Body Dynamics for Beginners: Euler Angles & Quaternions*. CreateSpace Independent Publishing Platform, 2013. ISBN 9781493598205. URL `https://books.google.com/books?id=bJEengEACAAJ`.

H. Kirchner. Rewriting strategies and strategic rewrite programs. In *Logic, Rewriting, and Concurrency*, pages 380–403. Springer, 2015.

H. Kirchner and P.-E. Moreau. Promoting rewriting to a programming language: A compiler for non-deterministic rewrite programs in associative-commutative theories. *J. Funct. Program.*, 11(2):207–251, Mar. 2001. ISSN 0956-7968. URL `http://dl.acm.org/citation.cfm?id=968486.968488`.

D. E. Knuth and P. B. Bendix. Simple word problems in universal algebras. In *Automation of Reasoning*, pages 342–376. Springer, 1983.

M. Konaković, K. Crane, B. Deng, S. Bouaziz, D. Piker, and M. Pauly. Beyond developable: Computational design and fabrication with auxetic materials. *ACM Trans. Graph.*, 35 (4):89:1–89:11, July 2016. ISSN 0730-0301. doi: 10.1145/2897824.2925944. URL `http://doi.acm.org/10.1145/2897824.2925944`.

B. Koo, J. Hergel, S. Lefebvre, and N. J. Mitra. Towards zero-waste furniture design. *IEEE Transactions on Visualization and Computer Graphics*, 23(12):2627–2640, Dec 2017. ISSN 1077-2626. doi: 10.1109/TVCG.2016.2633519.

V. Krishnamurthy and M. Levoy. Fitting smooth surfaces to dense polygon meshes. In *Proceedings of the 23rd Annual Conference on Computer Graphics and Interactive Techniques*, SIGGRAPH '96, pages 313–324, New York, NY, USA, 1996. ACM. ISBN 0-89791-746-4. doi: 10.1145/237170.237270. URL `http://doi.acm.org/10.1145/237170.237270`.

D. H. Laidlaw, W. B. Trumbore, and J. F. Hughes. Constructive solid geometry for polyhedral objects. In *Proceedings of the 13th Annual Conference on Computer Graphics and Interactive Techniques*, SIGGRAPH '86, pages 161–170, New York, NY, USA, 1986. ACM. ISBN 0-89791-196-2. doi: 10.1145/15922.15904. URL `http://doi.acm.org/10.1145/15922.15904`.

Y. Lan, Y. Dong, F. Pellacini, and X. Tong. Bi-scale appearance fabrication. *ACM Trans. Graph.*, 32(4):145:1–145:12, July 2013. ISSN 0730-0301. doi: 10.1145/2461912.2461989. URL `http://doi.acm.org/10.1145/2461912.2461989`.

M. Lau, A. Ohgawara, J. Mitani, and T. Igarashi. Converting 3d furniture models to fabricatable parts and connectors. In *ACM SIGGRAPH 2011 Papers*, SIGGRAPH '11, pages 85:1–85:6, New York, NY, USA, 2011. ACM. ISBN 978-1-4503-0943-1. doi: 10.1145/1964921.1964980. URL `http://doi.acm.org/10.1145/1964921.1964980`.

LeapFrog. Bolt pro, 2020. `https://www.lpfrg.com/products/leapfrog-bolt-pro/`.

X. Leroy. Formal certification of a compiler back-end or: Programming a compiler with a proof assistant. *SIGPLAN Not.*, 41(1):42–54, Jan. 2006. ISSN 0362-1340. doi: 10.1145/1111320.1111042. URL `https://doi.org/10.1145/1111320.1111042`.

H. Li, R. Hu, I. Alhashim, and H. Zhang. Foldabilizing furniture. *ACM Trans. Graph.*, 34 (4):90:1–90:12, July 2015. ISSN 0730-0301. doi: 10.1145/2766912. URL `http://doi.acm.org/10.1145/2766912`.

J. I. Lipton, A. Schulz, A. Spielberg, L. H. Trueba, W. Matusik, and D. Rus. Robot assisted carpentry for mass customization. In *2018 IEEE International Conference on Robotics and Automation (ICRA)*, pages 1–8, Brisbane, QLD, Australia, May 2018. IEEE. doi: 10.1109/ICRA.2018.8460736.

N. P. Lopes, J. Lee, C.-K. Hur, Z. Liu, and J. Regehr. Alive2: Bounded translation validation for llvm. 2021.

S. Lucas. Termination of rewriting with strategy annotations. In *International Conference on Logic for Programming Artificial Intelligence and Reasoning*, pages 669–684. Springer, 2001.

S.-J. Luo, Y. Yue, C.-K. Huang, Y.-H. Chung, S. Imai, T. Nishita, and B.-Y. Chen. Legolization: Optimizing lego designs. *ACM Trans. Graph.*, 34(6):222:1–222:12, Oct. 2015. ISSN 0730-0301. doi: 10.1145/2816795.2818091. URL `http://doi.acm.org/10.1145/2816795.2818091`.

L.-K. Ma, Y. Zhang, Y. Liu, K. Zhou, and X. Tong. Computational design and fabrication of soft pneumatic objects with desired deformations. *ACM Transactions on Graphics (TOG)*, 36(6):239, 2017.

H. Massalin. Superoptimizer: A look at the smallest program. In *Proceedings of the Second International Conference on Architectual Support for Programming Languages and Operating Systems*, ASPLOS II, page 122–126, Washington, DC, USA, 1987. IEEE Computer Society Press. ISBN 0818608056. doi: 10.1145/36206.36194. URL `https://doi.org/10.1145/36206.36194`.

J. McCann, L. Albaugh, V. Narayanan, A. Grow, W. Matusik, J. Mankoff, and J. K. Hodgins. A compiler for 3d machine knitting. *ACM Trans. Graph.*, 35(4):49:1–49:11, 2016. doi: 10.1145/2897824.2925940. URL `http://doi.acm.org/10.1145/2897824.2925940`.

J. McCrae, N. Umetani, and K. Singh. Flatfitfab: Interactive modeling with planar sections. In *Proceedings of the 27th Annual ACM Symposium on User Interface Software and Tech-*

nology, UIST '14, pages 13–22, New York, NY, USA, 2014. ACM. ISBN 978-1-4503-3069-5. doi: 10.1145/2642918.2647388. URL `http://doi.acm.org/10.1145/2642918.2647388`.

W. M. McKeeman. Peephole optimization. *Commun. ACM*, 8(7):443–444, July 1965. ISSN 0001-0782. doi: 10.1145/364995.365000. URL `https://doi.org/10.1145/364995.365000`.

D. Menendez and S. Nagarakatte. Alive-infer: Data-driven precondition inference for peephole optimizations in llvm. In *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI 2017, page 49–63, New York, NY, USA, 2017. Association for Computing Machinery. ISBN 9781450349888. doi: 10.1145/3062341.3062372. URL `https://doi.org/10.1145/3062341.3062372`.

D. Menendez, S. Nagarakatte, and A. Gupta. Alive-fp: Automated verification of floating point based peephole optimizations in LLVM. In X. Rival, editor, *Static Analysis - 23rd International Symposium, SAS 2016, Edinburgh, UK, September 8-10, 2016, Proceedings*, volume 9837 of *Lecture Notes in Computer Science*, pages 317–337. Springer, 2016. doi: 10.1007/978-3-662-53413-7_16. URL `https://doi.org/10.1007/978-3-662-53413-7_16`.

A. Meshmixer. Autodesk. Meshmixer, 2018. `http://www.meshmixer.com/`.

N. J. Mitra, M. Pauly, M. Wand, and D. Ceylan. Symmetry in 3d geometry: Extraction and applications. *Comput. Graph. Forum*, 32(6):1–23, Sept. 2013. ISSN 0167-7055. doi: 10.1111/cgf.12010. URL `http://dx.doi.org/10.1111/cgf.12010`.

P. Moon and D. Spencer. *Field theory handbook: including coordinate systems, differential equations, and their solutions*. Springer-Verlag, 1988. ISBN 9780387027326. URL `https://books.google.com/books?id=EDnvAAAAMAAJ`.

Y. Mori and T. Igarashi. Plushie: An interactive design system for plush toys. *ACM Trans. Graph.*, 26(3), July 2007. ISSN 0730-0301. doi: 10.1145/1276377.1276433. URL `http://doi.acm.org/10.1145/1276377.1276433`.

S. Mueller, S. Im, S. Gurevich, A. Teibrich, L. Pfisterer, F. Guimbretière, and P. Baudisch. Wireprint: 3d printed previews for fast prototyping. In *Proceedings of the 27th Annual ACM Symposium on User Interface Software and Technology*, UIST '14, pages 273–280, New York, NY, USA, 2014a. ACM. ISBN 978-1-4503-3069-5. doi: 10.1145/2642918.2647359. URL `http://doi.acm.org/10.1145/2642918.2647359`.

S. Mueller, T. Mohr, K. Guenther, J. Frohnhofen, and P. Baudisch. fabrickation: Fast 3d printing of functional objects by integrating construction kit building blocks. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, CHI '14, pages 3827–3834, New York, NY, USA, 2014b. ACM. ISBN 978-1-4503-2473-1. doi: 10.1145/2556288.2557005. URL `http://doi.acm.org/10.1145/2556288.2557005`.

E. Mullen, D. Zuniga, Z. Tatlock, and D. Grossman. Verified peephole optimizations for compcert. In *Proceedings of the 37th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '16, page 448–461, New York, NY, USA, 2016. Association for Computing Machinery. ISBN 9781450342612. doi: 10.1145/2908080.2908109. URL `https://doi.org/10.1145/2908080.2908109`.

P. Müller, P. Wonka, S. Haegler, A. Ulmer, and L. Van Gool. Procedural modeling of buildings. *ACM Trans. Graph.*, 25(3):614–623, July 2006. ISSN 0730-0301. doi: 10.1145/1141911.1141931. URL `http://doi.acm.org/10.1145/1141911.1141931`.

J. R. Munkers. Topology, 2000.

A. Nadel. Bit-vector rewriting with automatic rule generation. In *Proceedings of the 16th International Conference on Computer Aided Verification - Volume 8559*, page 663–679, Berlin, Heidelberg, 2014. Springer-Verlag. ISBN 9783319088662. doi: 10.1007/978-3-319-08867-9_44. URL `https://doi.org/10.1007/978-3-319-08867-9_44`.

C. Nandi, A. Caspi, D. Grossman, and Z. Tatlock. Programming Language Tools and Techniques for 3D Printing. In B. S. Lerner, R. Bodík, and S. Krishnamurthi, editors, *2nd Summit on Advances in Programming Languages (SNAPL 2017)*, volume 71 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 10:1–10:12, Dagstuhl, Germany, 2017. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik. ISBN 978-3-95977-032-3. doi: 10.4230/LIPIcs.SNAPL.2017.10. URL `http://drops.dagstuhl.de/opus/volltexte/2017/7122`.

C. Nandi, J. R. Wilcox, P. Panchekha, T. Blau, D. Grossman, and Z. Tatlock. Functional programming for compiling and decompiling computer-aided design. volume 2, pages 99:1–99:31, New York, NY, USA, July 2018. ACM. doi: 10.1145/3236794. URL `http://doi.acm.org/10.1145/3236794`.

C. Nandi, M. Willsey, A. Anderson, J. R. Wilcox, E. Darulova, D. Grossman, and Z. Tatlock. Synthesizing structured CAD models with equality saturation and inverse transformations. In *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI 2020, page 31–44, New York, NY, USA, 2020. Association for Computing Machinery. ISBN 9781450376136. doi: 10.1145/3385412.3386012. URL `https://doi.org/10.1145/3385412.3386012`.

C. Nandi, M. Willsey, A. Zhu, Y. R. Wang, B. Saiki, A. Anderson, A. Schulz, D. Grossman, and Z. Tatlock. Rewrite rule inference using equality saturation. *Proceedings of the ACM on Programming Languages*, (OOPSLA), 2021.

S. NC. Step-nc, 2019. `http://www.step-nc.org/index.htm`.

C. G. Nelson. *Techniques for Program Verification*. PhD thesis, Stanford, CA, USA, 1980. AAI8011683.

J. L. Newcomb, S. Johnson, S. Kamil, A. Adams, and R. Bodik. Verifying and improving halide's term rewriting system with program synthesis. *Proceedings of the ACM on Programming Languages*, (OOPSLA), 2020.

A. Niemetz, M. Preiner, A. Reynolds, C. Barrett, and C. Tinelli. Solving quantified bit-vectors using invertibility conditions. In H. Chockler and G. Weissenbacher, editors, *Computer Aided Verification*, pages 236–255, Cham, 2018. Springer International Publishing. ISBN 978-3-319-96142-2.

A. Nötzli, A. Reynolds, H. Barbosa, A. Niemetz, M. Preiner, C. Barrett, and C. Tinelli. Syntax-guided rewrite rule enumeration for smt solvers. In M. Janota and I. Lynce, editors, *Theory and Applications of Satisfiability Testing – SAT 2019*, pages 279–297, Cham, 2019. Springer International Publishing. ISBN 978-3-030-24258-9.

OFF. OFF files, 2018. `http://www.geomview.org/docs/html/OFF.html`.

OpenScad. OpenScad. The Programmers Solid 3D CAD Modeller, 2019. `http://www.openscad.org/`.

P. Panchekha, A. Sanchez-Stern, J. R. Wilcox, and Z. Tatlock. Automatically improving accuracy for floating point expressions. In *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '15, pages 1–11, New York, NY, USA, 2015. ACM. ISBN 978-1-4503-3468-6. doi: 10.1145/2737924.2737959. URL `http://doi.acm.org/10.1145/2737924.2737959`.

A. Pasko, V. Adzhiev, A. Sourin, and V. Savchenko. Function representation in geometric modeling: concepts, implementation and applications. *The Visual Computer*, 11(8):429–446, Aug 1995. ISSN 1432-2315. doi: 10.1007/BF02464333. URL `https://doi.org/10.1007/BF02464333`.

D. Peled. Ten years of partial order reduction. In *International Conference on Computer Aided Verification*, pages 17–28. Springer, 1998.

P. M. Phothilimthana, T. Jelvis, R. Shah, N. Totla, S. Chasins, and R. Bodik. Chlorophyll: Synthesis-aided compiler for low-power spatial architectures. In *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '14, pages 396–407, New York, NY, USA, 2014. ACM. ISBN 978-1-4503-2784-8. doi: 10.1145/2594291.2594339. URL `http://doi.acm.org/10.1145/2594291.2594339`.

P. M. Phothilimthana, A. Thakur, R. Bodik, and D. Dhurjati. Scaling up superoptimization. *SIGPLAN Not.*, 51(4):297–310, Mar. 2016. ISSN 0362-1340. doi: 10.1145/2954679.2872387. URL `http://doi.acm.org/10.1145/2954679.2872387`.

Powershape. Powershape, 2018. `https://www.autodesk.com/products/powershape/overview`.

V. Premtoon, J. Koppel, and A. Solar-Lezama. Semantic code search via equational reasoning. In *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI 2020, page 1066–1082, New York, NY, USA, 2020. Association for Computing Machinery. ISBN 9781450376136. doi: 10.1145/3385412.3386001. URL `https://doi.org/10.1145/3385412.3386001`.

P. Prusinkiewicz, M. Hammel, J. Hanan, and R. Mech. L-systems: from the theory to visual models of plants. In *Proceedings of the 2nd CSIRO Symposium on Computational Challenges in Life Sciences*, volume 3, pages 1–32. Citeseer, 1996.

W. Pugh. The omega test: a fast and practical integer programming algorithm for dependence analysis. In *Proceedings Supercomputing '91, Albuquerque, NM, USA, November 18-22, 1991*, pages 4–13, 1991. doi: 10.1145/125826.125848. URL `https://doi.org/10.1145/125826.125848`.

W. Pugh and D. Wonnacott. Eliminating false data dependences using the omega test. In *Proceedings of the ACM SIGPLAN'92 Conference on Programming Language Design and Implementation (PLDI), San Francisco, California, USA, June 17-19, 1992*, pages 140–151, 1992. doi: 10.1145/143095.143129. URL `https://doi.org/10.1145/143095.143129`.

Racket. Racket, the Programming Language, 2021. `https://racket-lang.org/`.

J. Ragan-Kelley, C. Barnes, A. Adams, S. Paris, F. Durand, and S. Amarasinghe. Halide: A language and compiler for optimizing parallelism, locality, and recomputation in image processing pipelines. In *Proceedings of the 34th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '13, page 519–530, New York, NY, USA, 2013. Association for Computing Machinery. ISBN 9781450320146. doi: 10.1145/2491956.2462176. URL `https://doi.org/10.1145/2491956.2462176`.

Repetier. Ditto printing, 2020. `https://forum.repetier.com/discussion/4140/ditto-printing-mixing-extruder`.

Rhinoceros. Rhinoceros, 2018. `https://www.rhino3d.com/`.

T. Ringer, K. Palmskog, I. Sergey, M. Gligoric, and Z. Tatlock. QED at large: A survey of engineering of formally verified software. *Foundations and Trends in Programming Languages*, 5(2-3):102–281, 2019. doi: 10.1561/2500000045. URL `https://doi.org/10.1561/2500000045`.

A. Romano and D. Engler. Expression reduction from programs in a symbolic binary executor. In E. Bartocci and C. R. Ramakrishnan, editors, *Model Checking Software*, pages 301–319, Berlin, Heidelberg, 2013. Springer Berlin Heidelberg. ISBN 978-3-642-39176-7.

C. Ronse. Regular open or closed sets, 1990.

Rust. Rust bigInt Library, 2021a. `https://docs.rs/num-bigint/0.4.0/num_bigint/struct.BigInt.html`.

Rust. Rust Rational Library, 2021b. `https://docs.rs/num-rational/0.4.0/num_rational/struct.Ratio.html`.

O. C. SAS. OPEN CASCADE, 2019. `https://www.opencascade.org`.

E. Schkufza, R. Sharma, and A. Aiken. Stochastic superoptimization. *SIGPLAN Not.*, 48 (4):305–316, Mar. 2013. ISSN 0362-1340. doi: 10.1145/2499368.2451150. URL `http://doi.acm.org/10.1145/2499368.2451150`.

E. Schkufza, R. Sharma, and A. Aiken. Stochastic optimization of floating-point programs with tunable precision. In *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '14, page 53–64, New York, NY, USA, 2014. Association for Computing Machinery. ISBN 9781450327848. doi: 10.1145/2594291.2594302. URL `https://doi.org/10.1145/2594291.2594302`.

R. Schnabel, R. Wahl, and R. Klein. Efficient RANSAC for Point-Cloud Shape Detection. *Computer Graphics Forum*, 2007. ISSN 1467-8659. doi: 10.1111/j.1467-8659.2007.01016.x.

C. Schüller, R. Poranne, and O. Sorkine-Hornung. Shape representation by zippables. *ACM Trans. Graph.*, 37(4):78:1–78:13, July 2018. ISSN 0730-0301. doi: 10.1145/3197517.3201347. URL `http://doi.acm.org/10.1145/3197517.3201347`.

A. Schulz, A. Shamir, D. I. W. Levin, P. Sitthi-amorn, and W. Matusik. Design and fabrication by example. *ACM Trans. Graph.*, 33(4):62:1–62:11, July 2014. ISSN 0730-0301. doi: 10.1145/2601097.2601127. URL `http://doi.acm.org/10.1145/2601097.2601127`.

A. Schulz, A. Shamir, I. Baran, D. I. W. Levin, P. Sitthi-Amorn, and W. Matusik. Retrieval on parametric shape collections. *ACM Transactions on Graphics*, 36(1):11:1–11:14, Jan. 2017a. ISSN 0730-0301.

A. Schulz, J. Xu, B. Zhu, C. Zheng, E. Grinspun, and W. Matusik. Interactive design space exploration and optimization for cad models. *ACM Trans. Graph.*, 36(4):157:1–157:14, July 2017b. ISSN 0730-0301. doi: 10.1145/3072959.3073688. URL `http://doi.acm.org/10.1145/3072959.3073688`.

M. Schwarz and P. Müller. Advanced procedural modeling of architecture. *ACM Trans. Graph.*, 34(4):107:1–107:12, July 2015. ISSN 0730-0301. doi: 10.1145/2766956. URL `http://doi.acm.org/10.1145/2766956`.

I. Sergey, J. R. Wilcox, and Z. Tatlock. Programming and proving with distributed protocols. *Proceedings of the ACM on Programming Languages*, 2(POPL):28:1–28:30, 2018. doi: 10.1145/3158116. URL `https://doi.org/10.1145/3158116`.

G. Sharma, R. Goyal, D. Liu, E. Kalogerakis, and S. Maji. Csgnet: Neural shape parser for constructive solid geometry. *CoRR*, abs/1712.08290, 2017. URL `http://arxiv.org/abs/1712.08290`.

R. Sharma, E. Schkufza, B. Churchill, and A. Aiken. Conditionally correct superoptimization. *SIGPLAN Not.*, 50(10):147–162, Oct. 2015. ISSN 0362-1340. doi: 10.1145/2858965.2814278. URL `https://doi.org/10.1145/2858965.2814278`.

B. Sherman, J. Michel, and M. Carbin. Sound and robust solid modeling via exact real arithmetic and continuity. *Proc. ACM Program. Lang.*, 3(ICFP):99:1–99:29, July 2019. ISSN 2475-1421. doi: 10.1145/3341703. URL `http://doi.acm.org/10.1145/3341703`.

J. Shewchuk. Adaptive precision floating-point arithmetic and fast robust geometric predicates. 18:305–363, 10 1997.

R. Singh and A. Solar-Lezama. Swapper: A framework for automatic generation of formula simplifiers based on conditional rewrite rules. In *Proceedings of the 16th Conference on*

Formal Methods in Computer-Aided Design, FMCAD '16, page 185–192, Austin, Texas, 2016. FMCAD Inc. ISBN 9780983567868.

E. Singher and S. Itzhaky. Theory exploration powered by deductive synthesis. In A. Silva and K. R. M. Leino, editors, *Computer Aided Verification*, pages 125–148, Cham, 2021. Springer International Publishing. ISBN 978-3-030-81688-9.

SketchUp. SketchUp, 2018. `http://www.sketchup.com/`.

M. Skouras, S. Coros, E. Grinspun, and B. Thomaszewski. Interactive surface design with interlocking elements. *ACM Trans. Graph.*, 34(6):224:1–224:7, Oct. 2015. ISSN 0730-0301. doi: 10.1145/2816795.2818128. URL `http://doi.acm.org/10.1145/2816795.2818128`.

P. Smid. *CNC Programming Handbook: A Comprehensive Guide to Practical CNC Programming*. Industrial Press, 2003. ISBN 9780831131586. URL `https://books.google.com/books?id=JNnQ8r5merMC`.

G. H. Smith, A. Liu, S. Lyubomirsky, S. Davidson, J. McMahan, M. Taylor, L. Ceze, and Z. Tatlock. Pure tensor program rewriting via access patterns (representation pearl). *arXiv preprint arXiv:2105.09377*, 2021.

Snapmaker. Modular 3-in-1 3d printers, 2020. `https://snapmaker.com/platform/?gclid=EAIaIQobChMIp7-KlZPy5wIVbRitBh3H-QvvEAAYASABEgJk_fD_BwE`.

A. Sokolov, J. Richard, V. Nguyen, I. Stroud, W. Maeder, and P. Xirouchakis. Algorithms and an extended step-nc-compliant data model for wire electro discharge machining based on 3d representations. *International Journal of Computer Integrated Manufacturing*, 19 (6):603–613, 2006.

A. Solar-Lezama. *Program Synthesis by Sketching*. PhD thesis, University of California, Berkeley, 9 2008a.

A. Solar-Lezama. *Program Synthesis by Sketching*. PhD thesis, Berkeley, CA, USA, 2008b. AAI3353225.

Solidworks. Solidworks, 2018. `http://www.solidworks.com/`.

R. Solutions. woodCAD—CAM. `https://www.rsasolutions.com/products/woodcadcam/`, 2019.

P. Song, C.-W. Fu, Y. Jin, H. Xu, L. Liu, P.-A. Heng, and D. Cohen-Or. Reconfigurable interlocking furniture. *ACM Trans. Graph.*, 36(6):174:1–174:14, Nov. 2017. ISSN 0730-0301. doi: 10.1145/3130800.3130803. URL `http://doi.acm.org/10.1145/3130800.3130803`.

SpaceClaim. SpaceClaim, 2018. `http://www.spaceclaim.com/en/Solutions/ReverseEngineering.aspx`.

O. Stava, J. Vanek, B. Benes, N. Carr, and R. Měch. Stress relief: Improving structural strength of 3d printable objects. *ACM Trans. Graph.*, 31(4):48:1–48:11, July 2012. ISSN 0730-0301. doi: 10.1145/2185520.2185544. URL `http://doi.acm.org/10.1145/2185520.2185544`.

M. Stepp, R. Tate, and S. Lerner. Equality-based translation validator for llvm. In G. Gopalakrishnan and S. Qadeer, editors, *Computer Aided Verification*, pages 737–742, Berlin, Heidelberg, 2011. Springer Berlin Heidelberg. ISBN 978-3-642-22110-1.

I. E. Sutherland. Sketch pad a man-machine graphical communication system. In *Proceedings of the SHARE Design Automation Workshop*, DAC '64, pages 6.329–6.346, New York, NY, USA, 1964. ACM. doi: 10.1145/800265.810742. URL `http://doi.acm.org/10.1145/800265.810742`.

R. Tate, M. Stepp, Z. Tatlock, and S. Lerner. Equality saturation: A new approach to optimization. In *Proceedings of the 36th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '09, pages 264–276, New York, NY, USA, 2009. ACM. ISBN 978-1-60558-379-2. doi: 10.1145/1480881.1480915. URL `http://doi.acm.org/10.1145/1480881.1480915`.

Z. Tatlock and S. Lerner. Bringing extensibility to verified compilers. In B. G. Zorn and A. Aiken, editors, *Proceedings of the 2010 ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2010, Toronto, Ontario, Canada, June 5-10, 2010*, pages 111–121. ACM, 2010. doi: 10.1145/1806596.1806611. URL `https://doi.org/10.1145/1806596.1806611`.

T. F. Team. FreeCAD Your own 3D parametric modeler, 2019. `https://www.freecadweb.org/`.

A. Teibrich, S. Mueller, F. Guimbretière, R. Kovacs, S. Neubert, and P. Baudisch. Patching physical objects. In *Proceedings of the 28th Annual ACM Symposium on User Interface Software & Technology*, UIST '15, pages 83–91, New York, NY, USA, 2015. ACM. ISBN 978-1-4503-3779-3. doi: 10.1145/2807442.2807467. URL `http://doi.acm.org/10.1145/2807442.2807467`.

E. The Future. Enabling The Future, 2018. `http://enablingthefuture.org`.

Thingiverse. Hexagonal candle holder, 2018a. `https://www.thingiverse.com/thing:756968`.

Thingiverse. Welcome To Customizer, 2018b. `https://www.thingiverse.com/customizer`.

Thingiverse. Ultimate 22 hex-wrench holder, 2018c. `https://www.thingiverse.com/thing:1752602`.

Thingiverse. Thingiverse, 2019. `https://www.thingiverse.com/`.

Y. Tian, A. Luo, X. Sun, K. Ellis, W. T. Freeman, J. B. Tenenbaum, and J. Wu. Learning to infer and execute 3d shape programs. In *International Conference on Learning Representations*, 2019.

E. Torlak and R. Bodik. Growing solver-aided languages with rosette. In *Proceedings of the 2013 ACM International Symposium on New Ideas, New Paradigms, and Reflections on Programming & Software*, Onward! 2013, page 135–152, New York, NY, USA, 2013. Association for Computing Machinery. ISBN 9781450324724. doi: 10.1145/2509578.2509586. URL `https://doi.org/10.1145/2509578.2509586`.

S.-A.-A. Touati and D. Barthou. On the decidability of phase ordering problem in optimizing compilation. In *Proceedings of the 3rd Conference on Computing Frontiers*, CF '06, pages 147–156, New York, NY, USA, 2006. ACM. ISBN 1-59593-302-6. doi: 10.1145/1128022.1128042. URL `http://doi.acm.org/10.1145/1128022.1128042`.

J.-B. Tristan and X. Leroy. Formal verification of translation validators: A case study on instruction scheduling optimizations. *SIGPLAN Not.*, 43(1):17–27, Jan. 2008. ISSN 0362-1340. doi: 10.1145/1328897.1328444. URL `http://doi.acm.org/10.1145/1328897.1328444`.

J.-B. Tristan and X. Leroy. Verified validation of lazy code motion. *SIGPLAN Not.*, 44 (6):316–326, June 2009. ISSN 0362-1340. doi: 10.1145/1543135.1542512. URL `http://doi.acm.org/10.1145/1543135.1542512`.

A. Udupa, A. Raghavan, J. V. Deshmukh, S. Mador-Haim, M. M. Martin, and R. Alur. Transit: Specifying protocols with concolic snippets. *SIGPLAN Not.*, 48(6):287–296, June 2013. ISSN 0362-1340. doi: 10.1145/2499370.2462174. URL `http://doi.acm.org/10.1145/2499370.2462174`.

Ultimaker. Reliable 3d printers that simply work for you, 2020. `https://ultimaker.com/en/resources/52867-dual-extrusion`.

J. Um, M. Rauch, J.-Y. Hascoët, and I. Stroud. Step-nc compliant process planning of additive manufacturing: remanufacturing. *The International Journal of Advanced Manufacturing Technology*, 88(5-8):1215–1230, 2017.

N. Umetani and R. Schmidt. Cross-sectional structural analysis for 3d printing optimization. In *SIGGRAPH Asia 2013 Technical Briefs*, SA '13, pages 5:1–5:4, New York, NY, USA, 2013. ACM. ISBN 978-1-4503-2629-2. doi: 10.1145/2542355.2542361. URL `http://doi.acm.org/10.1145/2542355.2542361`.

N. Umetani, T. Igarashi, and N. J. Mitra. Guided exploration of physically valid shapes for furniture design. *ACM Trans. Graph.*, 31(4):86:1–86:11, July 2012. ISSN 0730-0301. doi: 10.1145/2185520.2185582. URL `http://doi.acm.org/10.1145/2185520.2185582`.

A. VanHattum, R. Nigam, V. T. Lee, J. Bornholt, and A. Sampson. Vectorization for digital signal processors via equality saturation. In *Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS 2021, page 874–886, New York, NY, USA, 2021. Association for Computing Machinery. ISBN 9781450383172. doi: 10.1145/3445814.3446707. URL `https://doi.org/10.1145/3445814.3446707`.

Čeli APS. Woodwork for Inventor - Furniture design software. `https://www.woodworkforinventor.com`, 2019.

K. Vidimče, S.-P. Wang, J. Ragan-Kelley, and W. Matusik. Openfab: A programmable pipeline for multi-material fabrication. *ACM Trans. Graph.*, 32(4):136:1–136:12, July 2013a. ISSN 0730-0301. doi: 10.1145/2461912.2461993. URL `http://doi.acm.org/10.1145/2461912.2461993`.

K. Vidimče, S.-P. Wang, J. Ragan-Kelley, and W. Matusik. Openfab: A programmable pipeline for multi-material fabrication. *ACM Trans. Graph.*, 32(4):136:1–136:12, July 2013b. ISSN 0730-0301. doi: 10.1145/2461912.2461993. URL `http://doi.acm.org/10.1145/2461912.2461993`.

E. Visser. Stratego: A language for program transformation based on rewriting strategies system description of stratego 0.5. In *International Conference on Rewriting Techniques and Applications*, pages 357–361. Springer, 2001a.

E. Visser. A survey of rewriting strategies in program transformation systems. *Electronic Notes in Theoretical Computer Science*, 57(2), 2001b.

P. Wadler. Linear types can change the world! In *PROGRAMMING CONCEPTS AND METHODS*. North, 1990.

C. Wang, A. Cheung, and R. Bodik. Synthesizing highly expressive sql queries from input-output examples. *SIGPLAN Not.*, 52(6):452–466, June 2017. ISSN 0362-1340. doi: 10.1145/3140587.3062365. URL `http://doi.acm.org/10.1145/3140587.3062365`.

L. Wang and E. Whiting. Buoyancy optimization for computational fabrication. *Computer Graphics Forum*, 35(2):49–58, 2016. doi: 10.1111/cgf.12810.

Y. R. Wang, S. Hutchison, J. Leang, B. Howe, and D. Suciu. SPORES: Sum-product optimization via relational equality saturation for large scale linear algebra. VLDB Endowment, 2020.

K. Weitz, D. Woos, E. Torlak, M. D. Ernst, A. Krishnamurthy, and Z. Tatlock. Scalable verification of border gateway protocol configurations with an SMT solver. In E. Visser and Y. Smaragdakis, editors, *Proceedings of the 2016 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2016, part of SPLASH 2016, Amsterdam, The Netherlands, October 30 - November 4, 2016*, pages 765–780. ACM, 2016. doi: 10.1145/2983990.2984012. URL `https://doi.org/10.1145/2983990.2984012`.

D. Whitfield and M. L. Soffa. An approach to ordering optimizing transformations. *SIGPLAN Not.*, 25(3):137–146, Feb. 1990. ISSN 0362-1340. doi: 10.1145/99164.99179. URL `http://doi.acm.org/10.1145/99164.99179`.

D. L. Whitfield and M. L. Soffa. An approach for exploring code improving transformations. *ACM Trans. Program. Lang. Syst.*, 19(6):1053–1084, Nov. 1997a. ISSN 0164-0925. doi: 10.1145/267959.267960. URL `https://doi.org/10.1145/267959.267960`.

D. L. Whitfield and M. L. Soffa. An approach for exploring code improving transformations. *ACM Trans. Program. Lang. Syst.*, 19(6):1053–1084, Nov. 1997b. ISSN 0164-0925. doi: 10.1145/267959.267960. URL `http://doi.acm.org/10.1145/267959.267960`.

J. R. Wilcox, D. Woos, P. Panchekha, Z. Tatlock, X. Wang, M. D. Ernst, and T. E. Anderson. Verdi: a framework for implementing and formally verifying distributed systems. In D. Grove and S. Blackburn, editors, *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation, Portland, OR, USA, June 15-17, 2015*, pages 357–368. ACM, 2015. doi: 10.1145/2737924.2737958. URL `https://doi.org/10.1145/2737924.2737958`.

M. Willsey. egg documentation, 2021. `https://docs.rs/egg/0.6.0/egg/struct.BackoffScheduler.html`.

M. Willsey, C. Nandi, Y. R. Wang, O. Flatt, Z. Tatlock, and P. Panchekha. egg: Fast and extensible equality saturation. *Proceedings of the ACM on Programming Languages*, 5 (POPL), 2021. doi: 10.1145/3434304. URL `https://doi.org/10.1145/3434304`.

C. Wu, H. Zhao, C. Nandi, J. I. Lipton, Z. Tatlock, and A. Schulz. Carpentry compiler. *ACM Trans. Graph.*, 38(6):195:1–195:14, Nov. 2019a. ISSN 0730-0301. doi: 10.1145/3355089.3356518. URL `http://doi.acm.org/10.1145/3355089.3356518`.

C. Wu, H. Zhao, C. Nandi, J. I. Lipton, Z. Tatlock, and A. Schulz. Carpentry compiler. volume 38, pages 195:1–195:14, 2019b. doi: 10.1145/3355089.3356518. URL `https://doi.org/10.1145/3355089.3356518`.

S. Xie and X. Xu. A step-compliant process planning system for sheet metal parts. *International Journal of Computer Integrated Manufacturing*, 19(6):627–638, 2006.

X. W. Xu and S. T. Newman. Making cnc machine tools more open, interoperable and intelligent - a review of the technologies. *Computers in Industry*, 57(2):141–152, 2006. doi: 10.1016/j.compind.2005.06.002.

Y. Yang, P. M. Phothilimtha, Y. R. Wang, M. Willsey, S. Roy, and J. Pienaar. Equality saturation for tensor graph superoptimization. In *Proceedings of Machine Learning and Systems*, 2021.

C.-K. Yap and V. Sharma. Robust geometric computation. In *Encyclopedia of Algorithms*, 2008.

J. Zhang and L. Xing. A survey of multiobjective evolutionary algorithms. In *2017 IEEE International Conference on Computational Science and Engineering (CSE) and IEEE International Conference on Embedded and Ubiquitous Computing (EUC)*, volume 1, pages 93–100, July 2017. doi: 10.1109/CSE-EUC.2017.27.

H. Zhao, H. Zhang, S. Xin, Y. Deng, C. Tu, W. Wang, D. Cohen-Or, and B. Chen. Dscarver: Decompose-and-spiral-carve for subtractive manufacturing. *ACM Trans. Graph.*, 37(4): 137:1–137:14, July 2018. ISSN 0730-0301. doi: 10.1145/3197517.3201338. URL `http://doi.acm.org/10.1145/3197517.3201338`.

H. Zhao, A. Zhu, M. Willsey, C. Nandi, Z. Tatlock, J. Solomon, and A. Schulz. Co-optimization of design and fabrication plans for carpentry, 2021.

Q. Zhou, J. Panetta, and D. Zorin. Worst-case structural analysis. *ACM Trans. Graph.*, 32(4):137:1–137:12, July 2013. ISSN 0730-0301. doi: 10.1145/2461912.2461967. URL `http://doi.acm.org/10.1145/2461912.2461967`.